MoE Sovereign

A Self-Hosted Multi-Model Orchestrator with Template-Based Expert Routing for Sovereign AI Infrastructure

Whitepaper — Version 1.0, April 2026

Philipp Horn

Publisher, Initiator, and Author

Benediktus-Kirchplatz 15
59387 Ascheberg
Germany

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

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

Contents

List of Figures

Abstract

This paper introduces MoE Sovereign, a fully self-hosted Multi-Model Orchestrator[a] for privacy-preserving, domain-specialised AI infrastructure. The system is a concrete attempt to establish *digital sovereignty* as a technical state of affairs, not merely a political slogan: every component runs locally, every dependency is named, and every data flow is documented.
The orchestrator is written in Python and uses LangGraph [2] as its execution model. Requests are not handed to a learned router; they are dispatched through *versioned expert templates* to two or three locally operated open-weight models per expert category. A four-layer cache hierarchy (ChromaDB [3] for semantic similarity; Valkey [4] for plan, graph-context, and performance-score layers) short-circuits redundant LLM invocations. A Neo4j-backed knowledge graph [5] grows at runtime via a *graph-based accumulation mechanism*, with domain-specific entity filters preventing cross-contamination between medical, legal, and technical contexts. Deterministic operations (arithmetic, date handling, subnet analysis, German-law full-text lookup) are delegated to a Model Context Protocol server [6] with an AST-whitelist evaluator, removing this class of tasks from the hallucinating reach of probabilistic models.
The system is fully OpenAI-API compatible, requires no outbound network traffic in its minimal configuration, and is by-construction aligned with Article 25 of the GDPR [7]. The same OCI image runs as a rootless Podman container inside an unprivileged Proxmox LXC, as a Docker Compose stack, as a Podman Quadlet unit, and as a Helm release on Kubernetes or OpenShift. An overnight stability benchmark (36 scenarios, 3 epochs, zero failures) demonstrates that a self-hosted ensemble of eight domain-specialist 7–9B models on legacy Tesla M10 hardware achieves **6.11 / 10** on MoE-Eval – the same score class as cloud-hosted GPT-4o mini – with full data sovereignty. We discuss architectural decisions, report openly on failures and learning curves from recent refactors (the unified OCI artifact rewrite, the security hardening milestone, the mermaid-rendering regression), present MoE Libris, a federated knowledge exchange protocol inspired by Fediverse architecture (Section 8.8), and outline remaining research on complexity pre-routing and multi-hub topology. The project is released under the CC BY-SA 4.0 licence and is explicitly non-commercial.

[a] We use the term *Multi-Model Orchestration* throughout this paper in the sense of a *request-level routing architecture*, not in the narrower sense of the sparse-gated MoE layer of Shazeer et al. [1]. This distinction matters, because our routing is *deterministic and template-based*, not learned.

Keywords: Multi-Model Orchestration, Expert Routing, LangGraph, GraphRAG, Model Context Protocol, Deterministic Routing, Digital Sovereignty, GDPR by Design, Self-Hosted AI, Unified OCI Artifact, Open Source.

1 Introduction

Large language models (LLMs) have moved from research artefacts to an infrastructure layer millions of people use every day within just a few years. Infrastructure this load-bearing deserves infrastructure questions: *Who owns the models? Who owns the data? Who is still able to use the service tomorrow?* The current landscape – dominated by three US providers – does not offer a satisfactory answer.

The problem in three sentences

First, commercial LLM APIs process every request on infrastructure the customer neither owns nor can inspect; training-data extraction, prompt logging, and retroactive policy changes are documented incidents. Second, the European regulatory framework – in particular Articles 25 and 32 of the General Data Protection Regulation [7] – mandates *data protection by design*, a requirement that cannot be discharged by handing processing to an opaque black box in a

foreign jurisdiction. Third, the few available self-hosted alternatives are either single-model launchers (Ollama [8]) or generic RAG toolkits (PrivateGPT [9], LocalAI [10]), leaving the orchestration of multiple specialised models, the accumulation of domain knowledge, and the operational readiness for multi-user workloads as homework for the deploying party.

Contribution

MoE Sovereign is not another LLM library. It is a fully integrated orchestrator that attempts to satisfy, simultaneously, the following five properties, which we consider the *minimum requirements of a sovereign AI infrastructure*:

1. **Locality**: every LLM invocation, knowledge-graph query, and persistent write occurs on infrastructure under the operator's physical control. Outbound network traffic is optional and explicit.

2. **Template-based routing**: the expert mapping, the cache hierarchy, and the precision-critical tools are deterministic – no learned router, no probabilistic dispatch, no AST-free tool invocation. The Planner node (intent decomposition) is an LLM and therefore stochastic; it determines *which* expert categories are invoked, not how those invocations are mapped to hardware instances.

3. **Graph-state accumulation**: every interaction persists extracted entities and relations to Neo4j. The accumulation steps are inspectable, versioned, and reversible.

4. **OCI portability**: the same OCI image [11] runs as a rootless Podman container inside an unprivileged Proxmox LXC, in a single-host Docker Compose setup, in k3s, or on Red Hat OpenShift – no code fork, no functionality loss between tiers.

5. **Non-commerciality**: there is no enterprise edition, no subscription model, no telemetry opt-out, no dark pattern. The project is released under CC BY-SA 4.0 [12] and stays that way.

Who is this paper for?

Three groups are the primary audience. **Research labs** that work with heterogeneous GPU hardware and need a production-ready orchestrator without introducing an LLM broker service into their own jurisdiction. **Small and medium organisations in regulated domains** (law, medicine, public administration) that must provide LLM functionality but cannot sign a data-protection impact assessment for a US cloud service. **Hobbyists and idealists** practising *digital sovereignty* as conviction and looking for a tool that is production-ready today and will not be taken away from them tomorrow.

The paper assumes solid familiarity with modern software architecture; deep transformer internals are not required.

Structure of this document

Section 3 makes the notion of digital sovereignty concrete by listing documented failure modes of commercial services. Section 2 documents the project's origin: the journey from legacy hardware (Tesla K80, M10, RTX cluster) to the research question of SLM parallelisation, and the decision to treat architecture as a compensation mechanism; it also describes the RL Flywheel as the core continuous-learning component. Section 4 contrasts MoE Sovereign with existing open-source and commercial systems. Sections 5 through 10 form the technical

core: system architecture, deterministic expert routing, cache hierarchy, GraphRAG, MCP precision tools, and self-correction loop. Section 11 describes the deployment model that carries the orchestrator from edge to enterprise cluster. Sections 12 and 13 cover monitoring, trace propagation, and the privacy-by-design stance. Section 14 is the lessons-learned section – an honest accounting of the failures of the last refactoring rounds, including a rendering regression that temporarily broke all 71 documentation diagrams before we fixed it. Sections 15 and 16 sketch the research agenda and close the paper. The appendix contains a full expert catalogue, an environment-variable reference, a glossary, and the third-party licence inventory.

2 Project Genesis and Research Motivation

MoE Sovereign was not designed on paper and then deployed against suitable hardware. It emerged the other way around: from a concrete hardware problem that forced an architectural answer. This section documents that origin story, because it shaped the system's design decisions in fundamental ways.

2.1 Starting Motivation: Cloud Independence at Home

Behind the academic research interest in SLM orchestration lies a concrete personal motivation: entering the AI world on one's own infrastructure – and thereby decoupling from subscription-based cloud services. What OpenAI, Anthropic, or Google sell as monthly subscriptions was to run at home: code assistance, knowledge-graph construction, document analysis, semantic search – all local, all private, all under full operator control.

This ambition is technically more demanding than it first appears. A single Ollama server solves the deployment problem, not the quality problem: without routing, without caching, without domain-specific experts, a locally running 7B model is not a functional substitute for a GPT-4-based service. The project goal was therefore stated more precisely: *local infrastructure that can compete with commercial services in functional breadth* – not through sheer parameter count, but through orchestration.

2.2 Extended Project Goal: Structural Token Reduction

A later-added but operationally significant project objective emerged from daily use: the structural reduction of token consumption for expensive frontier models. The infrastructure required for this was already evaluated and in production at that point – the expert-template routing, the knowledge database, and the reinforcement learning mechanisms acted as three orthogonal cost-reduction layers, each operating on a different time scale.

Layer 1 – Routing (per request). The first and immediately effective layer is the template-based expert routing, implemented as a dedicated **CC profile** for use with the Claude Code CLI tool. The CC profile offers two operating modes: in *native proxy mode*, the orchestrator transparently forwards every request to the configured frontier model – useful when full model capacity is required but a unified API endpoint is desired. In *expert-template mode*, the orchestrator assumes the routing decision: simple requests (auto-completion, syntax questions, boilerplate generation) are forwarded to local SLMs; only when Tier-1 experts signal low confidence does the call escalate to the expensive frontier model. The principle is the same as with the local SLM ensemble: *not every request justifies the same model.*

Layer 2 – Knowledge database (accumulative, across sessions). The second cost-reduction layer operates on a longer time scale. Every processed request persists extracted

entities and relations in the Neo4j knowledge graph. For subsequent requests of the same type, the accumulated knowledge is available as an L3 cache layer: the answer is constructed directly from the graph, without a full pipeline run and without an external API call. As the graph grows, the share of such cache hits increases – token consumption decreases over the course of operation, without any configuration effort.

Layer 3 – Reinforcement and causal learning (learning, over weeks). The third layer operates on the longest time scale and addresses two sources of inefficiency simultaneously. The *Thompson-Sampling-inspired routing* (Section 2.5) learns from accumulated user feedback and judge scores which local experts are reliable for which request categories – and routes there instead of to the expensive model. The *Correction Memory* is a form of causal learning: corrected (input, output) pairs are stored as few-shot examples and injected into the prompt for similar subsequent requests. This shortens the required context length and reduces retry cycles, because the model starts with validated solution strategies from the outset.

Structural vs. Prompt-Level Token Reduction

Classic prompt engineering reduces tokens through shorter phrasing – a manual, brittle approach. MoE Sovereign instead pursues a structural strategy: routing, knowledge graph, and learning mechanisms operate at the system level and improve efficiency without touching individual prompts. The three layers are additive and independently deployable.

2.3 Point of Departure: Legacy Hardware as Research Object

The development path ran through three hardware generations. The first iteration used **Tesla K80** GPUs – Kepler-generation compute cards (2014) designed for data-centre batch inference, offering 24 GB GDDR5 VRAM in a dual-die arrangement. The K80 supports neither `bfloat16` nor the Flash Attention variants used by modern inference engines; contemporary quantisation formats such as GGUF require dedicated CUDA kernels that are not compiled for Kepler. The model ecosystem (llama.cpp, Ollama) has progressively dropped support for the K80 architecture.

The second generation consists of **Tesla M10** servers: four M10 chips per PCIe blade, each with 8 GB GDDR5, nominally providing $4 \times 8\,\text{GB} = 32\,\text{GB}$ VRAM. In practice, llama.cpp and Ollama can use the aggregate VRAM of homogeneous multi-GPU setups, but the PCIe uplink introduces a substantial latency overhead: a 30B model on the bundled M10 block runs significantly slower than the same model on a single modern card with equivalent VRAM. A subsequent infrastructure decision split the four M10 chips into four independent Ollama instances, reducing the maximum usable model per instance to ≈7B (Q4), but enabling concurrent multi-expert workloads.

The third generation consists of **consumer GPUs from the Ampere and Turing generations** (RTX 2060, RTX 3060, each with 12 GB GDDR6X), aggregated into a heterogeneous cluster. This cluster provides 60 GB VRAM total across five RTX 3060 cards, Flash Attention 2 support, and full GGUF compatibility – at a fraction of the cost of a modern A100 or H100.

The common property of all three generations: they are *available and affordable*, but individually limited in inference capacity relative to modern large-model servers. None of the nodes can run a 70B model with a full context window at a latency acceptable for production use. A current-generation cloud GPU or an H100 node would be the obvious solution – but it

would also be the *easy* solution. The real question is: what is achievable when compute is not upgraded?

2.4 The Research Question: Architecture as Compensation Mechanism

The central hypothesis underlying this project can be stated precisely:

> *Can many small, specialised language models (SLMs), coordinated by an intelligent orchestration layer, compensate for the limitations of legacy hardware – and serve as a viable alternative to a single large model on expensive hardware?*

This question is not new. The Mixture-of-Experts (MoE) research literature pursues a related approach: a large model activates only a subset of its parameters for each input [13]. The decisive difference from the approach taken here, however, lies in the granularity. Classical MoE architectures operate at the token level within a single model; MoE Sovereign operates at the *request level across independent models*. That is: each expert is a fully autonomous LLM running on a dedicated GPU node, and the routing decision is *deterministic software logic* – not a learned gating vector.

This design has several operational consequences:

- **Heterogeneity is a feature, not a constraint.** A 7B model on an M10 node and a 30B model on the RTX cluster can cooperate in the same pipeline. The routing system assigns each request to the hardware tier appropriate for its category.
- **Specialisation replaces generalisation.** A medical model fine-tuned on clinical terminology often produces better answers on medical queries than a general 70B model – at a fraction of the VRAM cost. Quality emerges not from sheer parameter count, but from alignment between the model's profile and the query category.
- **Parallelism compensates for individual latency.** When all Tier-1 experts start simultaneously on different nodes, the total latency of the parallel phase is determined by the slowest individual node, not by the sum of all individual latencies. A cluster of five slow nodes can thereby achieve a wall-clock time comparable to a single fast node.

Whether this hypothesis holds is answered by the benchmark results in Section 14. The short answer is: *partially, yes*. The orchestrator reaches GAIA Level 1 scores of 50–60 % (best result: 46.7 % overall, exceeding the GPT-4o Mini reference value of 44.8 %), with all processing on consumer and legacy hardware. Where the system encounters structural limits – in deep multi-step reasoning on Level 3 and with very long documents – those limits are technically explainable and addressable.

2.5 The RL Flywheel: Continuous Learning Without Model Fine-Tuning

A key project objective was for the system to improve over time – without retraining models or modifying weights. The mechanism developed for this purpose can be described as a *Reinforcement Learning Flywheel* (Figure 1): a self-reinforcing cycle of three components.

1. Routing Telemetry. Every expert decision is instrumented. The Valkey register `moe:perf:{model}:{category}` accumulates two counters per (model, category) pair: total requests and positively rated requests. These counters are populated by two signal paths: the Judge node's self-evaluation (automatically after each response) and explicit user feedback

(thumbs up/down via the frontend). The aggregate of all performance events is exported as a Prometheus histogram (`moe_self_eval_score_bucket`) and visible in the Grafana dashboards.

2. Thompson-Sampling-Inspired Routing. The accumulated counters are used for routing decisions: the current performance score of a (model, category) pair determines whether a Tier-2 fallback is triggered. The score is computed as a Laplace-smoothed estimator $(M+1)/(N+2)$ – an approximation to the Bayes-optimal prediction for Bernoulli observations. The smoothing terms prevent a new model from being locked into a 0 or 1 extreme after few observations, and allow natural exploration of new expert instances. This mechanism is conceptually related to the Thompson Sampling approach for multi-armed bandit problems [14]: uncertainty about expert quality is reduced through observations, and routing decisions become more robust as the data volume grows.

3. Correction Memory. The third flywheel arm is the persisted correction memory. When the Critic node identifies a deviation and generates a correction, the corrected (input, output) pair is stored as a few-shot example under `/opt/moe-infra/few_shot_examples/`. On subsequent requests of the same type, the Planner draws on these examples as context. Correction Memory thus acts as an implicit fine-tuning substitute: instead of adjusting model weights, the system enriches the model's in-context prompt with historically validated solutions.

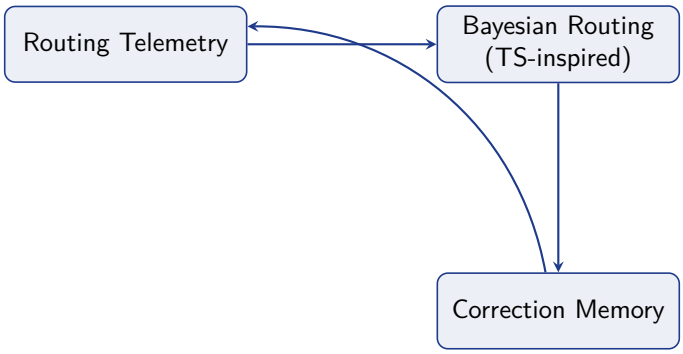

Figure 1: The RL Flywheel: Routing Telemetry, Thompson-Sampling-inspired score updates, and Correction Memory form a self-reinforcing improvement cycle with no model fine-tuning required.

Empirical evidence for the flywheel's effect appears in the measurement series from Table 10: the overall score on the internal MoE-Eval rose monotonically from 5.2 (Run 1, ≈500 Neo4j nodes) to 7.6 (5,353 nodes), without replacing or retraining a single model. The three flywheel components combine to build competence that is independent of the underlying model generation.

Core Principle: Architecture Generates Intelligence

The RL Flywheel embodies the project's foundational principle: it is not *more compute* that maximises system quality, but *better-structured information processing*. Routing telemetry, Bayesian expert selection, and persisted correction knowledge are software abstractions – they run on existing legacy hardware without modification.

2.6 Positioning: SLM Ensemble as Alternative to Frontier Models

The stated project objective – upgrading legacy hardware through orchestration – is not identical to the claim of replacing frontier models. Systems such as OpenAI o1 (74.1 % on GAIA [15]) or Claude 3.7 Sonnet Thinking (≈56 % on GAIA) depend on compute resources that are structurally unreproducible in a self-hosted consumer cluster. The relevant comparison baseline is different:

- Against *individual backbone models* on the same hardware (DeepSeek R1:32b: ≈28 %, Qwen3:32b: ≈12 % on GAIA), the orchestrator adds measurable value: 46.7 % overall and 60 % on GAIA Level 1 with the 30b-Balanced template.

- Against *specialised memory systems* such as EverMemOS (83 %) and TiMem (76.9 %) on LongMemEval, MoE Sovereign achieves 91.7 % in the best run – despite an architecture that treats memory as a by-product of knowledge graph accumulation rather than as a primary optimisation target.

The self-defined project goal is therefore: *maximum quality within the constraints of full data sovereignty and legacy-hardware compatibility.* That goal is achievable – and the benchmark data in Section 14 document to what degree.

3 Motivation: Digital Sovereignty as a Technical State

The term *digital sovereignty* is used liberally, often in the context of political declarations that have no technical consequences. We adopt a stricter reading in this paper: *digital sovereignty is the verifiable state in which the operator of an IT infrastructure can make and enforce every decision about data residency, software versions, network paths, and availability.* Anything else is rhetoric.

3.1 Documented failure modes of commercial LLM services

The table below lists concrete, publicly documented incidents from the past three years that illustrate why running business-critical processes on external LLM APIs is a structural risk.

Failure mode	Consequence for the operator
Retroactive price change	Running workflows become unprofitable overnight; existing budgets must be renegotiated while the outputs are not available locally.
Model deprecation	A production model is shut down; prompts tuned to the old version behave differently on the new one.
Training-data extraction	Confidential prompts or documents submitted for inference later reappear in training-data leaks.
Rate limiting during peak loads	Enterprise tenants are silently downgraded when other large customers saturate the service; local infrastructure is not affected by external contention.
Geopolitical lockout	A provider blocks entire countries or regions (technically trivial, regulatorily increasingly likely); locally installed software is unaffected.
Content-policy changes	New content filters suddenly classify legitimate requests (medical, legal, security research) as policy violations; work must migrate to a less restrictive provider, deepening the lock-in.
Unilateral contract change	Terms of service are rewritten while the customer is already dependent on the API; the only remedy is cancellation with no exit path.

Every single row is documented – often multiple times. Taken together they describe infrastructure that is *rented*, not *owned*. That is fine for many use cases. It is not acceptable for business-critical, regulated, or long-term-planned workflows.

3.2 The European regulatory framework

Article 25 of the GDPR [7] requires *data protection by design* and *data protection by default*. Verbatim: "Taking into account the state of the art ... the controller shall, both at the time of the determination of the means for processing and at the time of the processing itself, implement appropriate technical and organisational measures."

The European Commission, through its *European Strategy for Data* [16] and the Gaia-X framework [17], has explicitly articulated the goal of a federated European data and cloud infrastructure. The technical realisation at the LLM layer is still largely absent. MoE Sovereign is not part of Gaia-X, but positions itself as a reference implementation in the spirit of these directives: locally deployable, transparent in its data flows, with no hidden outbound communication.

3.3 Sovereignty at small scale is still sovereignty

A common criticism of self-hosted approaches is that they are only worthwhile for large organisations, as the operational overhead is otherwise disproportionate. The argument is too narrow. MoE Sovereign is deliberately designed to be operated in three sizes, each sensible in its own right:

- **solo profile** – a single virtual machine or a Proxmox LXC. Target audience: individuals, researchers, hobby operators. RAM footprint ≤ 2 GiB for the orchestrator; GPU optional (only for local inference; those using an external inference server do not need a local GPU).

- **team profile** – one Docker host or k3s cluster with the full data tier (Postgres, Valkey, Neo4j, ChromaDB, Kafka). Target: small organisations, departments, working groups.
- **enterprise profile** – Kubernetes or OpenShift with external data clusters, horizontal pod autoscaling, network policies, and OIDC integration. Target: public agencies, hospitals, large enterprises.

The technical novelty is that all three modes share the *same* artefact, not three different products. The difference is one environment variable (`MOE_PROFILE`) and a choice of deployment wrapper. Section 11 explains the implementation in detail.

Technical design principles

1. Your own hardware beats someone else's cloud.
2. Template-based expert routing beats a black-box proxy.
3. Versioned templates beat untracked prompt-engineering sessions.
4. Local knowledge graphs beat forgotten conversations.
5. AST whitelists beat hallucinated arithmetic.
6. One container image beats three enterprise SKUs.
7. An open-source licence beats an EULA.

4 Related Work

This section compares MoE Sovereign with existing approaches, both commercial and open source. The goal is an honest delineation, not a marketing matrix. Each system we mention has strengths worth acknowledging in its own context; MoE Sovereign borrows ideas from several of them.

4.1 Closed commercial APIs

OpenAI, Anthropic, and comparable providers deliver high-quality models behind an HTTP API. They solve the production problem at the cost of sovereignty: the operator has no access to model weights, inference hardware, or log files. For our target use cases (regulated domains, long-term control) this is a disqualifier, not a trade-off.

4.2 Single-model launchers

Ollama [8] is the de-facto standard for running open models locally. It exposes an OpenAI-compatible HTTP API, manages model downloads, handles VRAM accounting, and ships a stable tool-calling interface. MoE Sovereign *uses* Ollama (and any OpenAI-compatible endpoint) as an inference backend – there is no competition. What Ollama deliberately is *not*: a multi-model orchestrator across heterogeneous pools, a multi-tenant management system, an audit-logging framework, or a knowledge graph. These responsibilities belong to MoE Sovereign.

Desktop tools such as LM Studio target end-users rather than production infrastructure.

4.3 Generic RAG libraries

PrivateGPT [9] and LocalAI [10] provide retrieval-augmented generation on top of local models. PrivateGPT specialises in document indexing and question-answering workflows;

LocalAI is an API-compatible drop-in replacement for OpenAI that can multiplex several backends. Both are valuable building blocks; neither attempts the full orchestrator scope: no deterministic expert selection, no multi-layer cache hierarchy with plan and graph caches, no runtime-accumulated knowledge base with contradiction detection, and no multi-tenancy with token budgets.

4.4 Research-oriented stacks

The combination of vLLM [18] for the inference layer with LangChain [19] or LangGraph [2] for the orchestration layer is currently the most popular open-source option in academic settings. It is flexible, but sits at the library level: the operator builds every production feature – multi-tenancy, monitoring, GraphRAG pipeline, rate limiting, deployment wrapper – on their own. MoE Sovereign can be understood as an *opinionated assembly* of these building blocks: we adopt LangGraph as the execution model and provide, on top of it, a production-ready, opinionated reference architecture that boots with a single `docker compose up` or `helm install`.

4.5 Multi-Model Orchestration in the literature

Historically, the term *Multi-Model Orchestration* has referred, since Shazeer et al. [1], to a *sparse-gated* neural layer inside a transformer. Fedus et al. [20] and Jiang et al. [13] extend this approach. The learned router decides at the token level which expert subnetworks are activated.

MoE Sovereign uses the MoE label in a related but deliberately narrower sense: routing happens at *request level*, not at token level, and it is *explicit* rather than learned. A request is assigned to one or more expert categories; within each category, concretely named models with role tags (`primary`, `fallback`, `always`) are executed. Why we consider this the better architectural pattern in safety-critical domains is discussed in Section 6.

4.6 Microsoft GraphRAG and related work

Edge et al. [21] describe an approach to query-focused summarisation using graph-based context extraction. MoE Sovereign adopts the idea of representing domain knowledge as a graph but diverges in two respects we consider central: first, through *category-scoped entity-type filters* that prevent cross-contamination between specialised domains (Section 8); second, through a *graph-based accumulation mechanism*, whereby the graph grows at runtime from validated merger-node outputs, with contradictions resolved via a declarative rule matrix.

4.7 Enterprise AI platforms

The commercial landscape of enterprise AI divides into pure model providers (OpenAI, Anthropic) and *platform providers* that build cognitive architectures around models. MoE Sovereign falls into the second category – *compound AI systems* – and shares architectural overlap with several commercial platforms.

Palantir AIP. Architecturally the closest commercial relative. Both systems use deep ontology graphs for relational RAG, enforce deterministic tool execution, and implement node-level RBAC. The distinction: Palantir AIP represents the ultimate vendor lock-in, requires cloud or managed on-premises deployment, and costs upward of $1M/year. MoE

Sovereign achieves comparable architectural depth as an open-source container stack on local hardware (including legacy GPUs), fully air-gap capable, at zero licence cost.

Databricks Mosaic AI. Databricks drives the "compound AI systems" paradigm: routing requests to specialised models, SQL engines, and RAG pipelines. The philosophy is identical to MoE Sovereign's LangGraph-based two-tier expert routing. The distinction: Databricks targets massive big-data workloads (petabyte-scale cloud data lakes) and requires the proprietary Databricks ecosystem. MoE Sovereign is lightweight, self-contained, and designed for small-to-medium organisations and public institutions.

Glean. The current gold standard for enterprise search and permission-aware RAG. Glean provides 100+ enterprise app connectors and strict ACL enforcement – comparable to MoE Sovereign's `[REF:entity]` provenance tags and tenant-scoped graph queries. The distinction: Glean is a pure cloud SaaS solution. Regulated industries (KRITIS, banking, government) often cannot index sensitive data in external clouds. MoE Sovereign solves this compliance problem through local OCI deployment.

Microsoft Copilot Studio / Azure AI Agent Service. The standard tool for integrating agent workflows into enterprise processes. Multi-agent orchestration with tool use overlaps with MoE Sovereign's pipeline. The distinction: Microsoft's routing is often a stochastic black box (the LLM "guesses" which tool to call and how). MoE Sovereign forces models through the AST evaluator and JSON-based template routing into a deterministic, auditable execution path. Azure also requires cloud data residency.

CrewAI / AutoGen. Open-source multi-agent frameworks that assign roles to agents. CrewAI provides the simplest developer experience ($\tilde{3}5$ lines to start); AutoGen (Microsoft) uses conversational agent negotiation. Both are *libraries*, not platforms: they lack an admin UI, knowledge graph, VRAM-aware scheduling, template management, and deployment wrappers. MoE Sovereign provides all of these as an integrated system.

4.8 Comparative feature matrix

Table 1 summarises the feature coverage across the most relevant systems. A cell marked "–" indicates the feature is absent; "✓" indicates full support; "∼" indicates partial or limited support.

4.9 API proxies are not orchestrators

Platforms such as OpenRouter are sometimes cited as competitors. This is a category error. OpenRouter is a billing aggregator that routes API calls to cloud providers – it possesses no long-term memory, no tools, no expert routing, and no self-correction loop. OpenRouter delivers the building material; MoE Sovereign is the finished house.

4.10 Summary of the delineation

MoE Sovereign is not the most powerful model, not the fastest inference server, not the prettiest RAG toolkit. What it is: a production-ready, deterministically routed, multi-layer cached, GraphRAG-augmented, multi-tenant orchestration layer that runs on heterogeneous hardware and carries the same OCI artefact from an edge LXC to an OpenShift cluster – as one integrated system rather than a collection of individual pieces. To the best of our

Table 1: Feature comparison of MoE Sovereign and related systems.

Feature	*MoE Sovereign*	*Palantir AIP*	*Databricks*	*Glean*	*CrewAI*	*Ollama+WebUI*
Multi-expert routing	✓	✓	✓	–	∼	–
Deterministic routing	✓	✓	–	–	–	–
Knowledge graph (GraphRAG)	✓	✓	∼	✓	–	–
VRAM-aware scheduling	✓	–	–	–	–	∼
Knowledge export/import	✓	–	–	–	–	–
MCP precision tools	✓	–	–	–	–	–
Multi-tenant RBAC	✓	✓	✓	✓	–	∼
Air-gap / fully local	✓	∼	–	–	✓	✓
Open source	✓	–	∼	–	✓	✓
Admin UI	✓	✓	✓	✓	–	✓
Deployment: LXC to k8s	✓	∼	–	–	–	∼
Cost	Free	$1M+/yr	Pay/DBU	$25+/user	Free	Free

knowledge, this combination of capabilities has not been filled in the open-source landscape so far.

5 System Architecture

This section provides the overview on which the subsequent technical chapters build. Three guiding principles: *a single orchestrator component with a clear responsibility*, *an explicit separation of data plane and control plane*, and *every capability as a replaceable service with an open API.*

5.1 Services at a glance

The full *team-profile* stack consists of nineteen Docker containers, grouped into four logical layers: *core orchestration*, *data tier*, *observability*, and *edge / proxy.* The core layer holds the three in-house services – the orchestrator (`langgraph-app`, FastAPI + LangGraph, container port 8000), the MCP precision server (`mcp-precision`, port 8003), and the admin UI (`moe-admin`, port 8088). Beneath them sit the data services: PostgreSQL for LangGraph checkpoints and user/budget/template persistence, Valkey for caches and active request state, ChromaDB as a vector store, Neo4j as the knowledge graph, and Kafka running in KRaft mode [22] (no ZooKeeper, simpler to operate) as the event bus. The observability layer comprises Prometheus, Grafana, Node-Exporter, and cAdvisor; the edge layer comprises Caddy as a reverse proxy and Dozzle as a log viewer.

5.2 LangGraph as the execution model

We deliberately chose LangGraph [2] as the execution framework rather than a home-grown state machine, for three reasons. First, LangGraph provides persistence primitives (*checkpointers*) required for long-running multi-turn conversations – we use the Postgres checkpointer with a dedicated instance. Second, the graph structure cleanly separates orchestrator logic into named, testable nodes (`planner`, `expert_worker`, `merger`, `judge`, `thinking_node`,

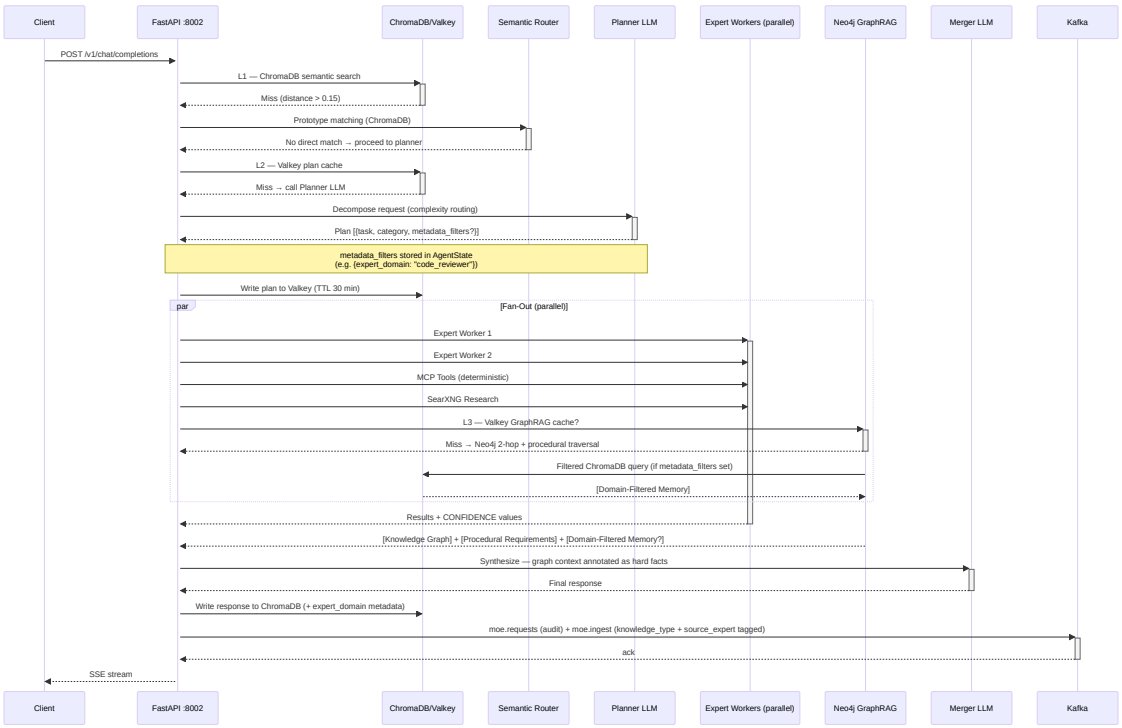

Figure 2: End-to-end data flow for a chat request through the orchestrator. Each node in the pipeline is a LangGraph node with explicit state; each box outside the pipeline is a separate service with its own API.

`research_fallback`, `graph_ingest`). Third, LangGraph is OSS with active development and a stable API contract – the upstream dependency we take on has a clean upgrade path.

5.3 Data plane vs. control plane

The orchestrator (`main.py`) is the *data plane*: it processes every request and talks directly to the inference backends. The admin UI (`admin_ui/`) is the *control plane*: it writes configuration (inference-server list, expert templates, user permissions, token budgets) but never participates in request processing. The MCP server is a third, orthogonal axis: expert-worker nodes call it via MCP when deterministic tools are required. The separation matters because it allows the orchestrator to scale horizontally without the admin UI following, and because it lets security policies be formulated precisely at the service level – the admin UI needs Postgres write access; the orchestrator does not.

5.4 Configuration sources and versioning

A central design decision was *not* to place configuration in yet another database but in `.env` files with structured JSON entries. Concretely, inference servers live as a JSON array in `INFERENCE_SERVERS`, expert templates as JSON in `EXPERT_TEMPLATES`, MCP tool URLs in `MCP_URL`, and database targets each in their own environment variable. The admin UI persists changes by rewriting this `.env` file. The orchestrator does not need a container restart – configuration is re-read on the next request with a 60-second cache.

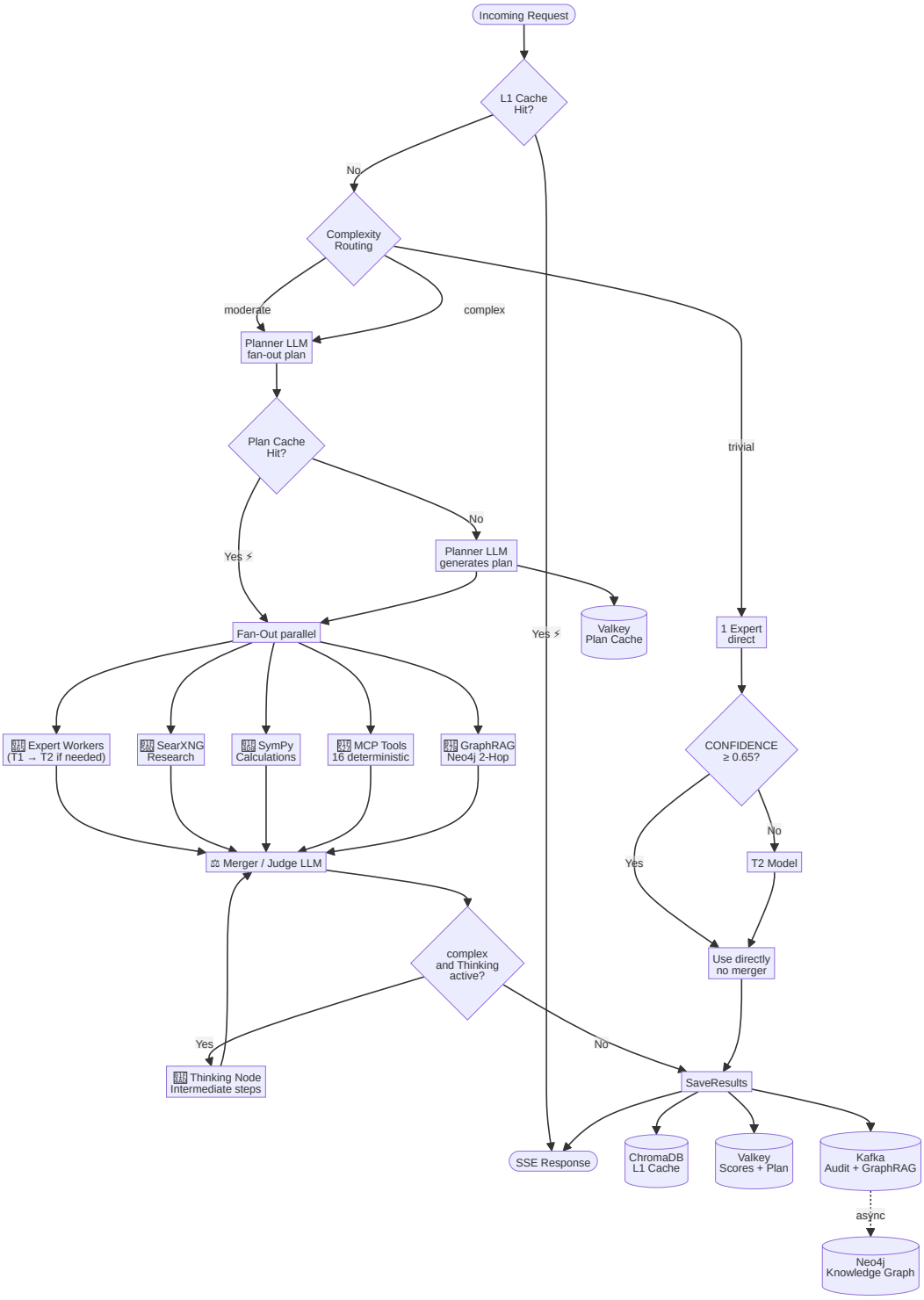

Figure 3: The LangGraph pipeline. Requests flow through `planner` (expert selection), `expert_worker` (parallel model invocations per category), `merger` (synthesis with source priority), optionally `thinking_node` (4-step chain-of-thought on low confidence), optionally `research_fallback` (external search), and `judge` (overall validation). Active requests are mirrored as `moe:active:*` keys in Valkey.

Why `.env` instead of a database?

The temptation to write every configuration setting as a row in an admin table is strong. We deliberately resisted it: `.env` files are trivially backed up (rsync, borgbackup, git-crypt), trivially version-controlled, trivially reviewed, and trivially injected by any IaC tool. An admin-database table is yet another piece of state with its own migration path. For state that belongs to the configuration layer (as opposed to runtime state like token budgets or user sessions), a structured text file is simply more robust.

5.5 Federation layer: MoE Libris

A fifth logical layer – external to the MoE Sovereign stack itself – is the *MoE Libris* federation hub (Section 8.8). MoE Libris is a separate FastAPI service with its own PostgreSQL, Neo4j, and Valkey instances that accepts knowledge bundles from paired sovereign nodes. Within the MoE Sovereign codebase, the `federation/` module implements the node-side client: outbound bundle push, inbound pull, handshake registration, and the per-domain outbound policy engine. The orchestrator itself does not communicate with the hub directly; the federation module operates as a background service that synchronises the local Neo4j graph with the hub on a configurable schedule.

5.6 Scale and performance envelope

Actual system load depends strongly on the models in use, the GPU classes, and the cache-hit rates; absolute benchmark numbers would carry little meaning until we offer a standardised suite. Instead we state *shapes*: the orchestrator container has an idle footprint around 200–500 MiB RAM; each LangGraph checkpoint costs between a few kilobytes and low single-digit megabytes; the four-layer cache hierarchy eliminates 40%–80% of LLM invocations depending on workload (see Section 7). Concrete numbers for specific setups can be found in the appendix and are updated on an ongoing basis through the repository; we commit to publishing no numbers we cannot reproduce.

5.7 Constraint-driven engineering: the Apollo 11 principle

The four-tier cache hierarchy, deterministic expert routing, and token-optimised prompts are not academic design choices, but direct responses to hard resource constraints. The primary development environment consisted of Tesla M10 GPUs (8 GB VRAM, DDR3, PCIe 2.0) with inference latencies of 40–120 seconds per complex request. Every millisecond saved through caching translated to several seconds of real-time savings on this hardware – a forced selection pressure that kept only designs that would not have emerged on modern H100 hardware.

The Apollo 11 guidance computer (AGC, 4 KB RAM, 72 KB ROM) was not precise *despite* its constraints, but *because* of them: every routine was compressed to the necessary minimum, no cycle was wasted. MoE Sovereign follows the same logic. Systems that run on H100 clusters with brute-force context windows and no caching pay with energy, latency, and cost what MoE Sovereign has amortised through architecture.

Constraint-driven engineering

Resource constraints are not an obstacle to system quality, but its hardest proving ground. What runs deterministically correct under 8 GB VRAM and 40 s inference latency will dominate on modern hardware through efficiency – not despite constraints, but because of them. We call this the Apollo 11 principle.

6 Template-Based Routing

Routing is the heart of the project and the most important conceptual departure from the MoE literature. This chapter explains why we replace learned routing with explicit versioned templates, how the selection unfolds in detail, and how heterogeneous GPU hardware enters the decision.

6.1 The trouble with learned routers

Architectural precision: planner vs. template mapping. The planner node issues an LLM call and is therefore probabilistic and stochastic. The execution layer (the static expert mapping defined in the template) and the tooling layer (MCP AST evaluator) are strictly deterministic: for a given template and request, the routing decision is fully reproducible regardless of runtime or model temperature.

A learned router maps an input representation to a distribution over expert models. In practice it works surprisingly well for accuracy, but it creates four operational problems that matter in regulated domains:

1. **Opacity**: Why was expert X chosen over Y? The answer is an activation pattern inside an intermediate transformer layer, not a human-readable explanation.
2. **Behavioural drift**: An update to the router weights can steer previously well-functioning prompts in unexpected directions without a documented release step.
3. **Cold-start latency**: Router decisions are model-based and consume inference time even when the answer could be delivered from a cache.
4. **Missing auditability**: For a data-protection impact assessment or a security review it is not reconstructable which model was permitted to see which type of data.

6.2 Template-based routing

MoE Sovereign stores the routing decision in explicit *expert templates*, maintained in the admin UI and persisted as JSON in the `EXPERT_TEMPLATES` environment variable. A template contains, per expert category (e.g. `code_reviewer`, `legal_advisor`, `medical_consult`, `math`, ...), a list of models, each tagged with a *role*:

- `primary` – always executed, tier 1.
- `fallback` – executed only when `primary` falls below the confidence threshold (tier 2).
- `always` – executed unconditionally in addition, regardless of confidence (e.g. a `judge` model for validation).

The tier boundary is defined by `EXPERT_TIER_BOUNDARY_B = 20` (billion parameters): models below this threshold are tier 1, models above are tier 2. Each tier-1 expert emits a structured field `CONFIDENCE: high|medium|low`, extracted via regex `r'CONFIDENCE:\s*(high|medium|low)'`. If *any* tier-1 expert reports `high`, tier-2 escalation is skipped. Only when *all* tier-1 experts report `medium` or `low` are the larger tier-2 models (>20 B) activated.

The function `_resolve_user_experts()` in `main.py` resolves user permissions against the template and returns a `{category → [model_config, ...]}` mapping. It is deterministic, referentially transparent, and covered in the test suite by explicitly constructed templates (see Section 14).

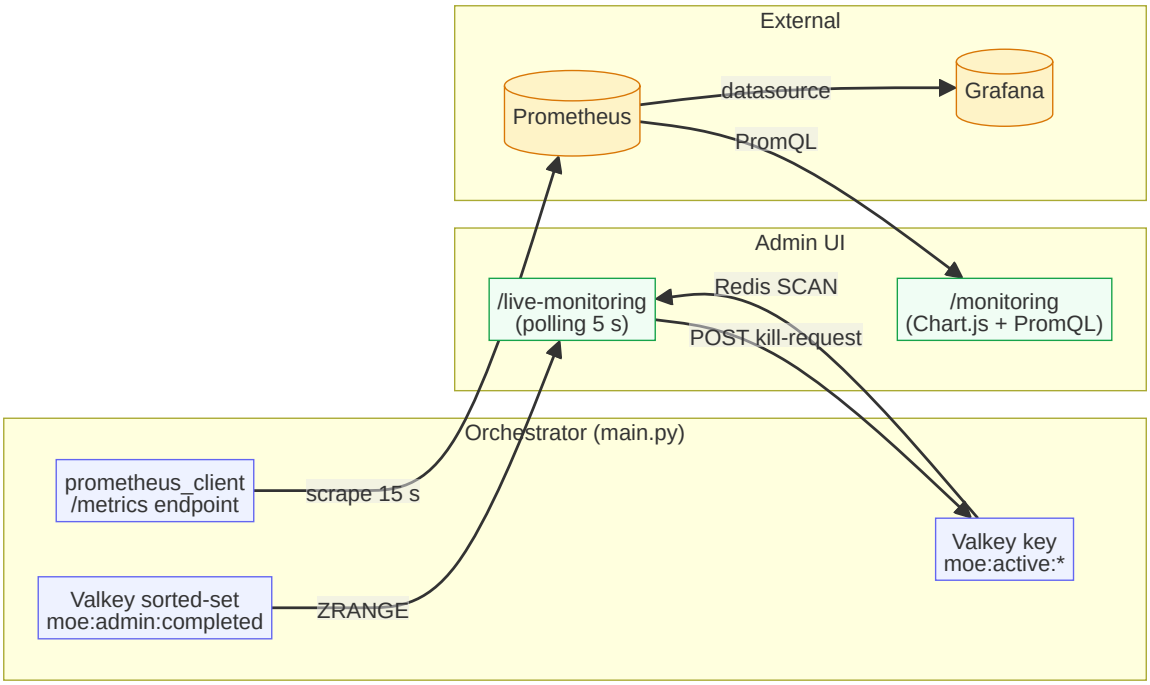

Figure 4: The observability architecture mirrors the same design approach: system metrics and live monitoring are treated as two complementary views on the same data – no abstraction that hides provenance.

6.3 Warm/cold scheduling across heterogeneous GPUs

A template alone only says *which* model to invoke, not *on which hardware*. In a realistic setup several inference servers are available (e.g. an RTX server, a Tesla server with multiple GPUs, a small edge node). The function `_select_node(model, allowed_endpoints)` in `main.py` picks a node for each call based on two signals:

1. **Warm/cold detection**: the orchestrator periodically polls `/api/ps` on each Ollama endpoint and stores the loaded models in `_ps_cache` (TTL 5 s). A node on which the desired model is already resident in VRAM is preferred – this eliminates cold-start latencies in the range of seconds to tens of seconds.

2. **Load scoring**: for nodes where the model is not warm, a simple score running_models/gpu_count is computed. The lowest-score node wins; ties are broken by a deterministic rule (order in the `INFERENCE_SERVERS` list).

The function is deliberately simple: earlier iterations with more sophisticated scoring (VRAM headroom, historical response time, request-queue length) were rolled back because the gain was marginal and the failure surface grew disproportionately. *As little scheduler magic as necessary* is part of the project philosophy.

Lessons Learned: the first scheduler was too clever

Our first scheduler considered VRAM headroom, historical response time, five-minute error rates, and the state of the Ollama request queue. Result: during a brief network flap a Tesla server was incorrectly marked as “overloaded”, and all requests migrated to an RTX server that then actually did overload. Reverting to a simple formula with a

five-second cache resolved the issue in a single line of code we have not touched since.

6.4 The expert-performance score as a fourth cache layer

A complementary component is the *expert-performance score*, maintained per $(model, category)$ pair in Valkey. Every positive user feedback and every `<SYNTHESIS_INSIGHT>` hit in the merger node increments the success counter; every negative response the failure counter. The function `_get_expert_score()` returns a Laplace-smoothed value $(M + 1)/(N + 2)$, where N is the total number of observations and M the number of positives. Tier-2 models are only executed if the tier-1 score falls below a configurable threshold (default 0.5) – a simple, transparent gating mechanism.

The score is not a router replacement but a priority annotation. *Which* models are candidates is decided by the template. *Which of them* run first and always is decided by the score. The separation is essential: the versioned, auditable part of the routing decision remains untouched.

6.5 VRAM-aware node selection

When the endpoint field in a template is empty (floating mode), the scheduler must select a node from the entire inference cluster. A naive round-robin or lowest-load strategy causes a critical failure on heterogeneous hardware: a 70B model (requiring ∼40 GB VRAM) routed to a Tesla M10 node (8 GB per GPU) silently falls back to CPU, producing 2–3 tokens/second instead of 15.

The solution is a configurable `vram_gb` field per inference server, set in the admin UI:

Node	VRAM	cost_factor	Max Model
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

The function `_estimate_model_vram_gb()` parses the parameter count from the model name (e.g. `llama3.3:70b` → 70) and applies the Q4_K_M estimate: VRAM ≈ 0.55 × params_B + 1.5 GB. Nodes whose `vram_gb` falls below the estimate are excluded *before* the warm/cold/load selection phases. If no local node qualifies, the system falls back to cloud/external nodes (those with `vram_gb = 0`).

This hard filter replaces the previous soft warning and prevents heterogeneous clusters from silently degrading to CPU inference.

7 Multi-Layer Cache Hierarchy

An LLM invocation costs time, power, and – where applicable – money. An orchestration layer that takes determinism and efficiency seriously must avoid these costs wherever the answer is predictable. MoE Sovereign uses four independent cache layers, each with its own key schema, invalidation policy, and target behaviour.

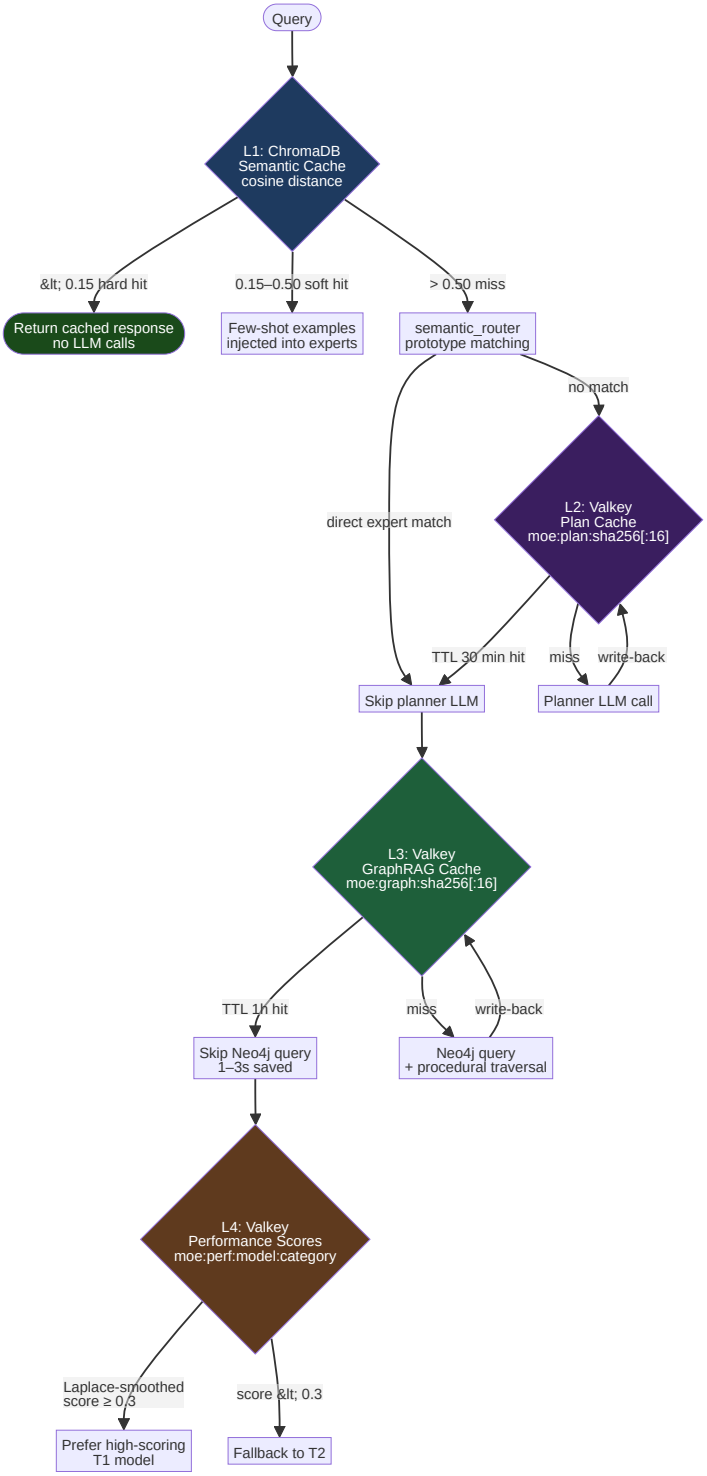

Figure 5: The four-layer cache hierarchy: L1 semantic, L2 plan, L3 graph, L4 performance score. Each layer has a distinct key space and an independent TTL.

7.1 L1: Semantic cache (ChromaDB)

The topmost layer is a semantic cache based on ChromaDB [3]. For every incoming user request, an embedding vector is computed and compared against the `moe_fact_cache` collection via cosine distance. If the nearest neighbour is below a threshold (default 0.15), the previously

stored response is returned directly – without planner, without expert workers, without judge. The savings per cache hit are dramatic: instead of a full pipeline with typically three to five LLM invocations, exactly one embedding call is made.

The L1 cache is invalidated conservatively: entries have no TTL but are flagged as `flagged=true` on negative user feedback and skipped on the next hit. Administrators can reset the collection with a single command (`docs/PRIVACY.md` contains the exact procedure). The policy is explicitly not "every cache decision is forever" – it is "a peer-reviewed cache decision stands until someone objects". This is a deliberate trade-off in favour of usability.

7.2 L2: Plan cache (Valkey)

The second layer stores the *planner decision* for a request. The planner is the node that derives, from a user query, the list of expert categories to activate ("This question is simultaneously legal and medical, activate both experts"). The key is the SHA-256 hash over the normalised query representation; the value is the serialised planner output. A hit in the L2 cache skips the planner LLM call but keeps the expert-worker and merger stages. TTL: 30 minutes.

7.3 L3: Graph-context cache (Valkey)

The third layer is the *graph-context cache*: the result of a Neo4j GraphRAG query is stored under a hash key and reused within the TTL. A hit saves the round-trip to Neo4j and the optional procedural traversal. TTL: 60 minutes.

The deliberate separation between L2 (planner) and L3 (GraphRAG) matters: two substantively different queries can share the same planner plan but produce different graph contexts, or vice versa. A single monolithic cache would serve neither efficiently.

7.4 L4: Expert performance scores (Valkey)

The bottom layer is not a cache in the strict sense but a *persistent observation register*: per $(\text{model}, \text{category})$ a Valkey hash with fields `total` (total invocations) and `positive` (positive feedbacks) is maintained. The values feed the scoring function from Section 6 and influence tier-2 gating. The space is small (fewer than a thousand keys per year) and grows linearly with the number of expert categories and models. Explicit invalidation is not required; Laplace smoothing ensures that even after long observation periods new models still have a realistic chance to compete.

7.5 Combination logic: fall-through

All four layers are strictly organised as a fall-through: an L1 hit terminates processing immediately; on a miss, L2 is consulted; on a further miss, L3; on an ultimate miss, L4 only advises tier-2 gating. The fall-through pattern is not accidental – it allows every layer to be disabled in isolation without breaking the others. During debugging sessions we have temporarily disabled individual layers (`CACHE_DISABLE_L1=1`) and observed the effects on latency, error rate, and LLM-call count directly.

Innovation: separating caches by intent

Most cache designs for LLM systems know exactly one cache – request → response. That is either too coarse (the planner and the GraphRAG are bypassed wholesale) or too fine (every LLM call is cached individually, which is difficult to invalidate in multi-step

pipelines). The four-layer split follows the *intent level*: L1 = semantic similarity, L2 = plan equivalence, L3 = graph-context equivalence, L4 = historical reliability. Each layer answers a different question and is therefore independently maintainable.

8 GraphRAG and Graph-Based Knowledge Accumulation

Retrieval-augmented generation [23] has become the de facto recipe for injecting external facts into LLM responses. The standard RAG pattern – chunk documents, embed, top-k retrieve with cosine similarity, append to the prompt – fails at two problems that are central to our use cases: *context window* and *cross contamination*.

8.1 Why a graph, not just a vector store

A vector store is good at finding similar passages and bad at representing relationships between entities. A knowledge graph – in our case Neo4j [5] – is the opposite: strong at relationships, weak at semantic similarity. MoE Sovereign combines both: ChromaDB as semantic search over raw text, Neo4j as structured world knowledge with entities, relationships, and ontology types. The idea is closely related to Microsoft GraphRAG [21], but diverges in two respects we consider central: *category-specific entity-type filters* and a *graph-based accumulation mechanism*.

8.2 Category-specific entity-type filters

The file `graph_rag/manager.py` contains a mapping `_CATEGORY_ENTITY_TYPES`, which assigns each expert category a set of allowed Neo4j labels. Concretely: the legal advisor sees only entities of type `Law`, `Right`, `Legal_Concept`, and `Organization`; the medical consult sees only `Drug`, `Disease`, `Symptom`, and `Treatment`; the technical support sees only `Software`, `Protocol`, `Hardware`, and `Error_Code`. Every GraphRAG query is filtered against the allowed label set for the current expert category.

At node level, each entity has the structure `(:Entity {name: "Ibuprofen", type: "Drug", domain: "medical_consult"})`. The Cypher query includes `WHERE e.type IN $allowed_types`, where `allowed_types` is resolved from the plan categories via `_CATEGORY_ENTITY_TYPES`.

Why is this important? Without a filter, a query to the medical consult might accidentally retrieve technical entities if the embeddings of the question happen to overlap both domains. The model may then mix medical and technical claims in its answer – a failure mode we observed in early versions and the fix for which we record here as an innovation.

Innovation: cross-contamination prevention

Entity-type filters systematically prevent an expert model from ever seeing context from a foreign domain. This is not only a quality improvement but a security property: in a multi-tenant environment with differently-accredited user groups the filter table can serve as a policy enforcement point.

8.3 The base ontology

The graph is not started empty. The module `graph_rag/ontology.py` provides over 400 pre-seeded base entities in multiple languages with aliases, labels, and initial relationships. Examples: key paragraphs of the German civil and criminal codes, common drug classes,

central technology frameworks and their dependencies. The load phase is idempotent (`MERGE` instead of `CREATE`), so a repeated ingest creates no duplicates and admins can extend the ontology file under version control.

8.4 Persistent Graph-State Tracking: the SYNTHESIS_INSIGHT mechanism

The most important difference from classic RAG is that the graph *grows* at runtime. The merger node in the pipeline graph receives an extended system-prompt directive telling it to emit a structured JSON block with the tag `<SYNTHESIS_INSIGHT>` whenever a novel, multi-source inference is synthesised. A downstream ingest process consuming the Kafka topic `moe.ingest` detects these blocks, extracts the entities and relationships, and writes them to Neo4j with a version annotation naming the emitting model and a timestamp.

The mechanism is deliberately conservative: only *new, non-trivial* insights are to be absorbed. The model is explicitly instructed in the prompt not to emit simple factual lookups. The ingest additionally verifies whether the extracted information is already present in the graph. Nevertheless, the knowledge graph is not a pure black-box summary: every statement is inspectable and carries a source annotation.

8.5 Contradiction detection

A second safety layer is contradiction detection. The file `graph_rag/manager.py` contains a table `_CONTRADICTORY_PAIRS` that declares mutually exclusive relationships. Example: if a relation (A)–[TREATS] $\rightarrow (B)$ exists and a newly written triple (A)–[CAUSES] $\rightarrow (B)$ arrives, the ingest is flagged and curated through a lint pipeline (topic `moe.linting`). Contradictions are not automatically overwritten – they are *flagged* and presented to a human reviewer.

8.6 Ontology gaps as a signal

A third, subtle property is the *ontology-gap signal*: when the orchestrator observes an LLM output whose central concepts cannot be matched against Neo4j labels, the term is counted as a "gap" in a Prometheus metric (`moe_ontology_gaps_total`). Admins can see the top gaps in the admin UI and decide whether the ontology should be extended. The system therefore learns not only facts but also *where its knowledge is incomplete*.

8.7 Community knowledge bundles

A knowledge graph is only as valuable as the breadth of its training data. MoE Sovereign addresses this through an export/import mechanism that enables *collective intelligence* across deployments.

Export. The API endpoint `/graph/knowledge/export` extracts entities, relations, and synthesis nodes as a JSON-LD bundle. Three safety layers prevent data leakage:

1. **Metadata stripping:** `tenant_id`, `source_model`, and timestamps are removed by default.
2. **Privacy scrubber:** Regex patterns detect and exclude entities containing passwords, IP addresses, email addresses, API keys, infrastructure hints, or client names.
3. **Sensitive relation filter:** Relations typed `HAS_PASSWORD`, `HAS_CREDENTIAL`, or `AUTHENTICATES_WITH` are unconditionally removed.

Filters for domain (e.g. `code_reviewer`, `technical_support`) and minimum trust score are supported. In production testing, the scrubber removed 97–214 entities per export from a graph of 3 150 entities.

Import. The import endpoint (`/graph/knowledge/import`) merges community bundles into the local graph with three guarantees:

- **No overwrite:** Entities are merged by name (`MERGE ON CREATE`). Existing entities are never modified.
- **Trust ceiling:** Imported relations are capped at a configurable `trust_floor` (default 0.5), ensuring community data never outranks locally verified facts.
- **Contradiction detection:** Before creating a relation, the system checks `_CONTRADICTORY_PAIRS` against existing high-trust triples. Conflicting imports are skipped and logged.

A dry-run mode (`/graph/knowledge/import/validate`) previews what would be imported without modifying the graph. Repeated imports of the same bundle are idempotent: all entities report as "skipped", zero duplicates are created.

Contradictions detected during import are published to the Kafka topic `moe.linting` for admin review, ensuring that community knowledge never silently overwrites enterprise-internal verified facts.

Federated Knowledge Sync

Knowledge bundles enable structured exchange of domain-specific knowledge graphs between independent deployments. Each instance remains autonomous and offline-capable; shared bundles enrich the local graph without transferring source data or proprietary information.

8.8 MoE Libris: Federated Knowledge Exchange

The Federated Knowledge Sync concept outlined in the innovation box above remained a conceptual sketch in v1.0 of this paper. With *MoE Libris*[1] we have realised it as a concrete, deployable system.

Architecture. MoE Libris is a separate FastAPI microservice (the *hub server*) backed by PostgreSQL (relational metadata, audit log), Neo4j (global knowledge graph), and Valkey (rate limiting, strike counters). Individual MoE Sovereign installations act as *sovereign nodes* that push and pull JSON-LD knowledge bundles via the hub's REST API. The topology is hub-and-spoke: each node connects to exactly one hub; the hub never initiates connections to nodes.

Federation protocol. Node pairing follows a bilateral handshake:

1. A node operator registers with the hub (node URL, public metadata, operator contact).
2. The hub administrator reviews the registration and accepts or rejects it.
3. On acceptance, the hub issues a per-node API key; the node stores it locally. All subsequent requests are authenticated via this key.

[1] Latin *liber* = both "free" and "book" – a deliberate double meaning reflecting the project's open-source ethos and its purpose as a knowledge repository.

Once paired, the data flow proceeds as follows: the node pushes a JSON-LD bundle to the hub; the hub runs a two-stage *pre-audit pipeline* (see below); bundles that pass pre-audit enter an *admin audit queue*; a hub administrator explicitly approves or rejects each bundle; approved bundles are merged into the global graph; other paired nodes can then pull new knowledge via a polling endpoint.

Pre-audit pipeline. Every incoming bundle passes two automated validation stages before reaching the audit queue:

1. **Stage 1 – Syntax validation:** JSON-LD schema conformance, required fields, well-formed entity and relation structures.
2. **Stage 2 – Heuristic PII / secret scanning:** Regex-based detection of email addresses, IPv4/IPv6 addresses, JWT tokens, API keys, and other credential patterns. Bundles containing matches are rejected with a diagnostic report.

The pipeline is designed to be extensible: a planned Stage 3 will add LLM-assisted triage for semantic content policy enforcement (v1.1, not yet implemented).

Abuse prevention. The hub implements a graduated strike system. Each policy violation (failed pre-audit, rejected bundle, rate-limit breach) adds a strike to the offending node. Strikes are weighted: security violations (credential leakage, injection attempts) carry triple weight. At three accumulated strikes the node is rate-limited; at ten weighted strikes the node is automatically blocked and must re-register.

Server discovery. Hub instances are discoverable through a public Git registry (`moe-libris-registry`). Any operator can register a hub by submitting a pull request with a JSON metadata file; CI validates the schema before merge. This mechanism parallels Fediverse instance lists (e.g. *instances.social* for Mastodon) and avoids centralised authority over the directory.

Outbound policy engine. Each sovereign node controls what it shares through a per-domain outbound policy with three modes: `auto` (push all qualifying bundles automatically), `manual` (queue for local admin review before push), and `blocked` (never push to this hub). Additional filters include a minimum confidence threshold and a `verified-only` flag that restricts exports to human-verified triples.

Trust model. Imported triples from the hub are subject to the same trust-floor mechanism described in Section 8.7: community data is capped at a configurable trust score (default 0.5) and never outranks locally verified facts. The existing contradiction detection (`_CONTRADICTORY_PAIRS`) applies to hub-sourced triples identically to manually imported bundles. No automatic trust propagation occurs – every imported statement must earn trust locally through verification or corroboration.

Architectural parallel. The design draws explicit inspiration from the Fediverse model (ActivityPub / Friendica): independent, self-hosted instances federate voluntarily through a standardised protocol; no instance has authority over another; the network grows through adoption, not through centralised control. The analogy is architectural, not functional: MoE Libris exchanges structured knowledge triples, not social-media posts.

Current limitations. The present implementation supports a single-hub topology only; multi-hub federation (mesh or hierarchical) is planned but not yet designed. Bundles are not cryptographically signed; a future version will add Ed25519 signatures for tamper detection in transit. The LLM-assisted triage stage (Stage 3 of the pre-audit pipeline) is specified but not yet implemented.

9 MCP Precision Tools

Large language models reliably hallucinate arithmetic. Multiplying five-digit numbers, computing date differences across month boundaries, deriving a network mask, or hashing a string are tasks that any pocket calculator solves in milliseconds with 100 % correctness, while an LLM can and will produce a new plausible- looking number on every invocation. The industry standard answer – *function calling* – is conceptually right but, in most implementations, too permissive: the model is allowed to call a Python function whose code is executed within the orchestrator process, with everything Python surrounds it with.

MoE Sovereign takes one further step: it delegates deterministic computations to a separate *Model Context Protocol* server [6] that uses a strictly whitelisted AST-based expression evaluator – and 50 additional built-in tools (51 in total).

9.1 Why a dedicated MCP server?

The MCP standard from Anthropic defines a clean protocol for a model to call tools that run *outside* the actual LLM process. The advantage for us: the MCP server can be deployed, hardened, updated, and scaled independently of the orchestrator. It runs as its own container (`mcp-precision`, port 8003) and exchanges requests and results via a typed JSON interface.

9.2 The AST whitelist of the `calculate` tool

The most important tool is `calculate`: it accepts an arithmetic expression as a string and returns the result. Under the hood it uses two stages:

1. **Safe AST evaluator**: the expression is parsed via `ast.parse` into a Python AST. Only node types from a whitelist are accepted: `Expression`, `BinOp`, `UnaryOp`, `Num`, `Constant`, `Call` (only for whitelisted functions), `Name` (only for whitelisted names). Any other node – `Attribute`, `Subscript`, `Import`, `Lambda`, `Assign` – causes evaluation to fail. This nips constructs like `__import__("os")`, arbitrary attribute access, or arbitrary code execution in the bud.

2. **SymPy fallback on `SyntaxError`**: if the input is not valid Python syntax but looks like a mathematical expression (such as “15% of 100” or “3(2+4)” with implicit multiplication), SymPy is consulted as a fallback – likewise in a tightly constrained environment. The fallback triggers *only* on `SyntaxError`, not on any other exception. This is a deliberate hardening against a prior vulnerability (see the lessons-learned box in Section 14).

9.3 The full tool roster

Beyond `calculate`, the MCP server exports 50 more tools (51 in total). The table below lists the core tools; eight new research and web tools have been added (see below).

Tool	Purpose
`calculate`	AST-whitelisted arithmetic with SymPy fallback.
`solve_equation`	Symbolic equation solving via SymPy.
`date_diff`	Day difference between two date strings.
`date_add`	Adds a time interval to a date.
`day_of_week`	Weekday of an arbitrary date.
`unit_convert`	Unit conversion via `pint`.
`statistics_calc`	Mean, median, standard deviation, percentiles.
`hash_text`	MD5/SHA-1/SHA-256/SHA-512 of a string.
`base64_codec`	Base64 encode/decode.
`regex_extract`	Deterministic regex matching.
`subnet_calc`	IPv4 subnet analysis (netmask, broadcast, hosts).
`text_analyze`	Word, character, line, and sentence counts.
`prime_factorize`	Prime factorisation.
`gcd_lcm`	Greatest common divisor and least common multiple.
`json_query`	JSONPath query against a JSON document.
`roman_numeral`	Roman numeral conversion both ways.
`legal_search_laws`	Full-text search over the paragraph index of `gesetze-im-internet.de`.
`legal_get_law_overview`	Table of contents of a German statute.
`legal_get_paragraph`	Full text of an individual paragraph.
`legal_fulltext_search`	Cross-statute comparison of hits.
`repo_map`	AST-based map of a Python repository.
`read_file_chunked`	Paginated, memory-friendly file reader.
`lsp_query`	LSP query (definitions, references) via `jedi`.
`generate_pptx`	Creates a fully formatted `.pptx` presentation from structured content (title, slides, bullet points, notes); delivers a signed MinIO download link.
Research and web tools (added April 2026)	
`wikidata_sparql`	Wikidata SPARQL API; deterministic, no SearXNG noise.
`pubmed_search`	NCBI/PubMed search for biomedical publications.
`crossref_lookup`	DOI and paper metadata via the Crossref API.
`openalex_search`	Search across 250 M+ scholarly works (OpenAlex).
`duckduckgo_search`	Alternative web search; supplements SearXNG on failures.
`web_browser`	Splash JS rendering for dynamic pages (JavaScript gates).
`wayback_fetch`	Wayback Machine (dual-strategy): direct fetch + API fallback.

9.4 Why exactly these tools?

The selection is not arbitrary. It covers three problem families typical of LLM mainline workloads that we did not want to leave to probabilistic models:

- **Arithmetic and units**: `calculate`, `solve_equation`, `statistics_calc`, `unit_convert`, `gcd_lcm`, `prime_factorize`, `roman_numeral`, `subnet_calc`.
- **Cryptography and encoding**: `hash_text`, `base64_codec`, `regex_extract`, `text_analyze`, `json_query`. All operations that require exact results.

- **Structured external sources**: the four `legal_*` tools access the official XML corpus of the German Federal Ministry of Justice (*gesetze-im-internet.de*). They can be operated offline if a snapshot is kept inside the container, keeping legal-advisory workflows network-sovereign.
- **Code navigation**: `repo_map`, `read_file_chunked`, and `lsp_query` serve the agentic coder expert, who works on third-party repositories without loading the full source into the context window.
- **Research and web**: The seven new tools (`wikidata_sparql`, `pubmed_search`, `crossref_lookup`, `openalex_search`, `duckduckgo_search`, `web_browser`, `wayback_fetch`) open primary knowledge sources that SearXNG does not reach reliably. In particular `wikidata_sparql` delivered deterministically correct answers on factual GAIA questions where SearXNG results varied run-to-run.

Lessons Learned: the SymPy gap

In an early version the SymPy fallback was too broad: any exception from the safe-AST path fell through to SymPy, including those raised by forbidden nodes such as `Attribute`. A test case `__import__('os').system('id')` therefore progressed to SymPy – and SymPy, on certain builds, did call the shell. The fix is a two-line change: fall back only on `SyntaxError`. The test suite has since contained an explicit assertion against this injection sequence.

10 Self-Correction Loop

Every LLM makes mistakes. An orchestration layer that claims to run in production long-term needs a mechanism to collect, evaluate, and feed those mistakes into future decisions. MoE Sovereign joins three signals for this purpose: *self-evaluation*, *user feedback*, and *ontology-gap detection*. All three feed a shared, inspectable score already introduced in Section 6 as the gating input for tier-2 models.

10.1 Self-evaluation in the judge node

The final node of the pipeline (`judge`) receives the synthesised answers from the expert workers along with the original question in its prompt. Its job is to emit a self-rated confidence between 0 and 1 – in the prompt explicitly as a score and a short rationale that can be used for debugging. The score is exported as a Prometheus histogram (`moe_self_eval_score_bucket`) and serves two purposes: first as a gating signal for the `thinking_node` activation (chain-of-thought on low confidence), and second as an input to the tier-2 decision.

The judge is deliberately a *separate pipeline stage*, not an embedded post-processing step inside the merger node. This makes its behaviour inspectable, replaceable, and separately cost-accounted.

10.2 User feedback

The second signal path is explicit user feedback. Frontends connected to the orchestrator can emit, per response, a simple thumbs-up / thumbs-down event. The event is persisted to Kafka on the `moe.feedback` topic and written by the ingest process into the Valkey register `moe:perf:{model}:{category}`, into which the self-eval values also flow. The metric `moe_feedback_score_bucket` makes the overall signal available to Prometheus.

10.3 Laplace-smoothed confidence update

Both signal paths feed the same observation counter. The score for a (model, category) pair is computed as $(M + 1)/(N + 2)$, where N is the total number of observations and M the number of positives. The $+1/+2$ constants amount to Laplace smoothing: a new model does not remain stuck at 0 or 1 after only a few measurements, but fluctuates around a neutral value near 0.5. This is kept deliberately simple; more sophisticated Bayesian estimators would be possible but the gain is marginal and interpretability suffers.

Score semantics

The score is emphatically *not* a quality measure in the absolute sense. It measures how often users and the judge node were satisfied with a particular model-category pair. Two models with identical score numbers may be substantively very different; the score only says that both have performed comparably in the observed past.

10.4 Tier-2 gating in practice

The gating mechanism is straightforward: when the tier-1 model of a category has a score below the configurable threshold (default 0.5), the `fallback` entry of the template is additionally executed. The answers from both tiers flow into the merger, which decides by a source-priority rule which partial answer takes precedence. The rule is part of the merger-node prompt and reads, in plain English: *reasoning answers beat MCP tool answers beat graph answers beat expert answers beat web research beat cache.*

10.5 Ontology-gap detection

The third signal is more subtle but long-term valuable: detection of *missing knowledge*. The ingest process compares the central terms mentioned in the merger output with the labels in the Neo4j graph. Terms without a match are counted as "ontology gaps" in Prometheus (`moe_ontology_gaps_total`). The admin can view the top gaps of the last n days in the admin UI and decide whether the ontology file should be extended. The system thereby has a back-channel indicator for its own blind spots – a property missing from static RAG systems.

10.6 The compounding effect

Taken together, the three signal paths – self-eval, user feedback, gap detection – produce a *compounding effect*: every interaction makes the system a small step more informed. The effect is deliberately slow and inspectable: every step is reflected in metrics, every change persists in Valkey and Neo4j, every critical step (contradiction detection, ontology extension) passes through a human review. The opposite would be uncontrolled online learning where no one can explain why the system behaves differently today than yesterday. Our approach rejects that explicitly.

10.7 Agentic re-planning loop

The compounding effect describes long-term learning across many requests. A more recent addition addresses a related but shorter-term question: what should happen when a single request is still *incomplete* after the first merger pass?

The *agentic re-planning loop* answers that: after each merger synthesis, a lightweight LLM call (the "gap detector") checks whether the answer is complete or still contains open questions. The check returns a simple binary verdict: `COMPLETION_STATUS: COMPLETE | NEEDS_MORE_INFO`.

On `NEEDS_MORE_INFO`, the still-unresolved part – together with all facts established so far – is injected as structured context into a new planner round. The planner therefore does not receive the original question again; it receives a focused mandate: *fetch exactly this missing piece*. Routing then goes selectively to `web_researcher` or `precision_tools` – not to all experts. After at most three agentic iterations the system returns the best available answer.

Protection against re-triggering

If the merger output contains a `SKILL_TRIGGER` token, a `/downloads/` path, or a `DOWNLOAD_URL` token, the gap detector skips its check and marks the answer immediately as `COMPLETE`. This prevents an already generated artefact (e.g. a PPTX file) from being produced a second time in a follow-up round.

The loop is complementary to the self-correction loop: that one learns *between* requests; the agentic loop resolves gaps *within* a single request – without user intervention.

11 Unified OCI Artifact

One of the most ambitious technical characteristics of the project is that *the same* OCI image runs, without code fork, from a hobby LXC to an enterprise cluster. This section describes how this is achieved, which deployment wrappers are supported, and which hard invariants hold across every tier.

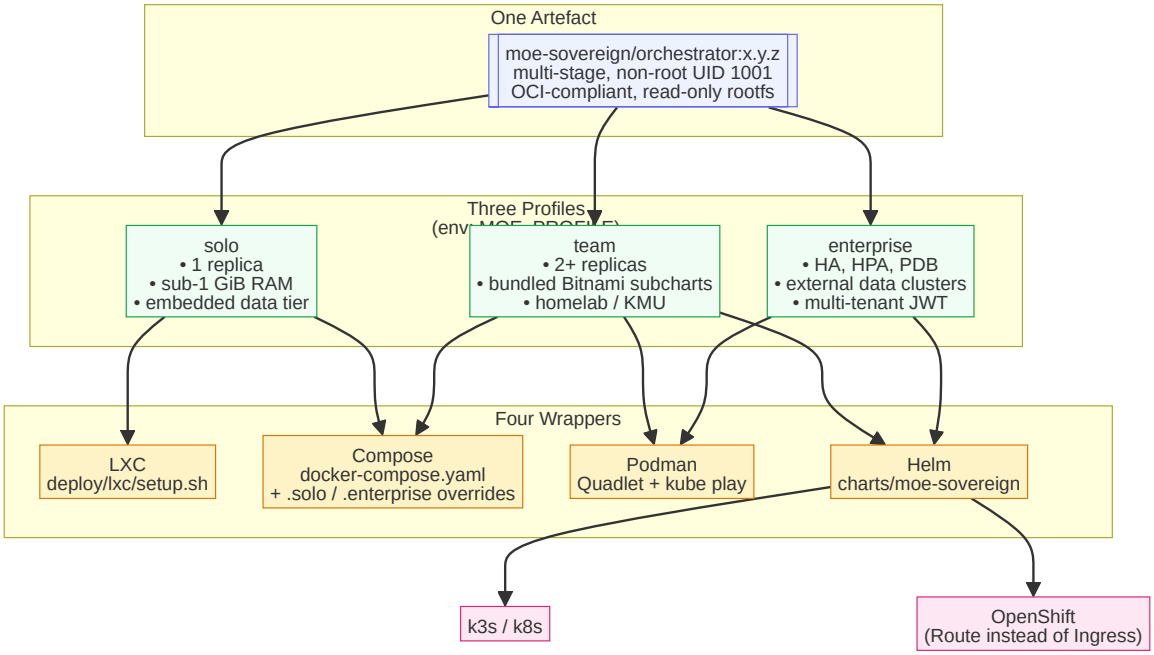

Figure 6: The universal-deployment principle: one OCI artefact (*single artifact*), three profiles (`solo`, `team`, `enterprise`), four deployment wrappers (LXC script, Docker Compose, Podman Quadlet, Helm Chart), deployable on five target platforms (LXC, Docker host, k3s, Kubernetes, OpenShift).

11.1 The single-artifact principle

The orchestrator is built from a multi-stage Dockerfile: a builder stage resolves the Python dependencies, a runtime stage contains only `python:3.11-slim` plus the installed site-packages directory. The result is an image under 400 MiB compressed. The same image tag is consumed by every deployment wrapper. There is no *enterprise edition* and no *community edition*; there is one image that behaves identically everywhere.

The most important properties of this image are:

- **Non-root**: everything runs as UID 1001 (`moe`), including the Python process. The container entrypoint explicitly switches user. The image is arbitrary-UID compatible: `chgrp -R 0 /app && chmod -R g=u /app` grants GID 0 write access to all application directories, satisfying OpenShift's Security Context Constraints (SCC) that randomize container UIDs at runtime.
- **Read-only root filesystem**: in Kubernetes, OpenShift, and Podman the root filesystem is read-only. The few writable paths (logs, caches, temporary LangGraph state) are provided via `emptyDir` volumes or `tmpfs`.
- **OCI labels**: all metadata (`org.opencontainers.image.*`) is set correctly, including source reference, build date, and version.
- **Dropped capabilities**: all Linux capabilities are dropped by the container runtime. The orchestrator needs none of them.
- **No default admin**: the first admin must be created by the operator (either in the admin UI or via an `.env` variable). There is no default password.

11.2 Three profiles

The orchestrator's behaviour is controlled by the environment variable `MOE_PROFILE`. Valid values are `solo`, `team`, and `enterprise`. The differences concern default resource limits and which companion services are expected; the code path is identical.

Profile	Target audience	Characteristic
`solo`	Individuals, edge, Proxmox LXC	1 replica, $<$ 1 GiB RAM, optionally embedded data tier.
`team`	SMB, homelab, single server	2 replicas, bundled Bitnami subcharts or full Compose stack.
`enterprise`	Public sector, enterprise	External data tier, HPA, PDB, network policies, OIDC, JWT tenancy.

11.3 Four deployment wrappers

Each profile corresponds to a concretely runnable wrapper. The four supported wrappers are:

1. **LXC bootstrap script** (`deploy/lxc/setup.sh`): a single bash script that prepares a fresh Debian or Ubuntu LXC for operation in under 60 seconds. It installs rootless Podman [24], creates an unprivileged service user with `systemd-linger`, installs a Quadlet unit, and optionally adds Grafana Alloy as a systemd service for log shipping to Loki.
2. **Docker Compose** (`docker-compose.yaml`): the classical route. A `docker compose up -d` starts 19 containers in the correct order. For a homelab setup this is the recommended

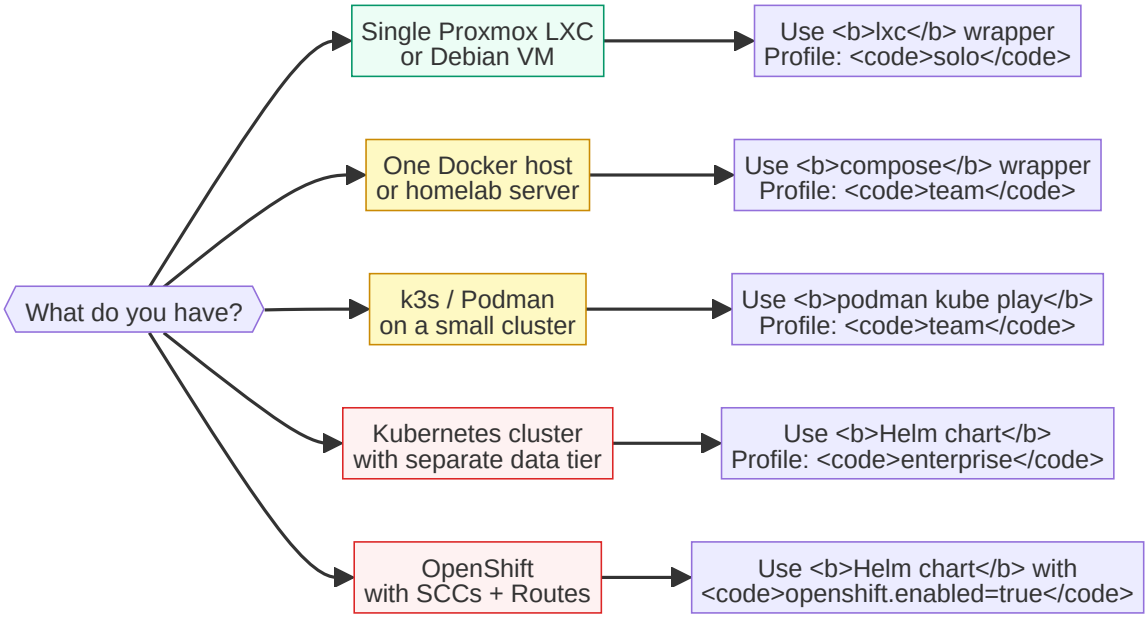

Figure 7: The tier decision aid: which profile and which wrapper to recommend for which deployment target.

path.

3. **Podman Quadlet** (`deploy/podman/systemd/moe-orchestrator.container`): a systemd-native unit for Podman $\geq$ 4.4. The unit has the same security settings as the Helm chart (`ReadOnly=true`, `NoNewPrivileges=true`, `DropCapability=ALL`), uses `journald` as its log driver, and can be managed via `systemctl`.

4. **Helm chart** (`charts/moe-sovereign/`): the Kubernetes route. The chart can optionally ship a complete deployment including PostgreSQL, Valkey, Kafka, and Neo4j as Bitnami subcharts [25] (*team* profile), or reference external clusters (*enterprise* profile). A built-in `capabilities` switch detects OpenShift environments via the `route.openshift.io/v1` API and emits Route objects instead of Ingress. As a result, the same chart is usable without modification on k3s, vanilla Kubernetes, and OpenShift [26].

11.4 LXC: rootless Podman as the bridge

The LXC setup script deserves particular attention because it solves a historical problem: Docker inside an unprivileged Proxmox LXC requires substantial nesting configuration and breaks on every kernel update. Rootless Podman inside an unprivileged LXC, by contrast, works out of the box, because Podman needs no daemon process and consumes user-namespace mappings directly from `/etc/subuid` and `/etc/subgid`.

11.5 The invariants

The same hard invariants hold across all four wrappers. They are enforced by a dedicated test suite (`tests/test_deployment_artifacts.py`) that checks Dockerfile, Helm chart, and LXC bootstrap script for consistency.

- UID 1001 is the *same* UID everywhere.

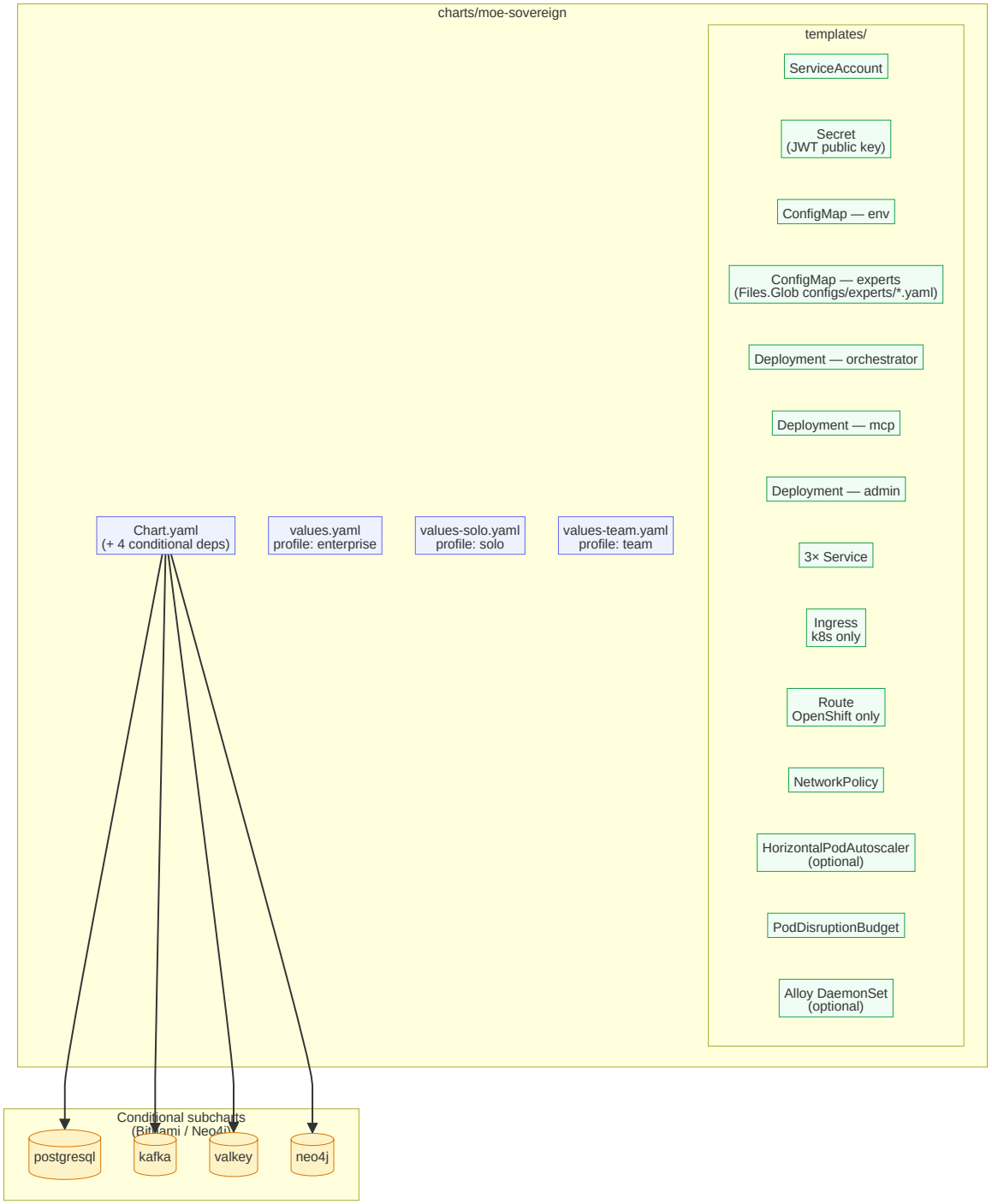

Figure 8: The Helm chart structure: `Chart.yaml` with conditional Bitnami subcharts, three values files for the three profiles, and 14 template files including the OpenShift Route switch.

- Port 8000 is the *same* port everywhere.
- `MOE_LOGS_DIR`, `MOE_CACHE_DIR`, and `MOE_EXPERTS_DIR` are set in all three wrapper forms.
- The read-only rootfs pattern holds everywhere.
- Health-check endpoints (`/health`) are compatible.

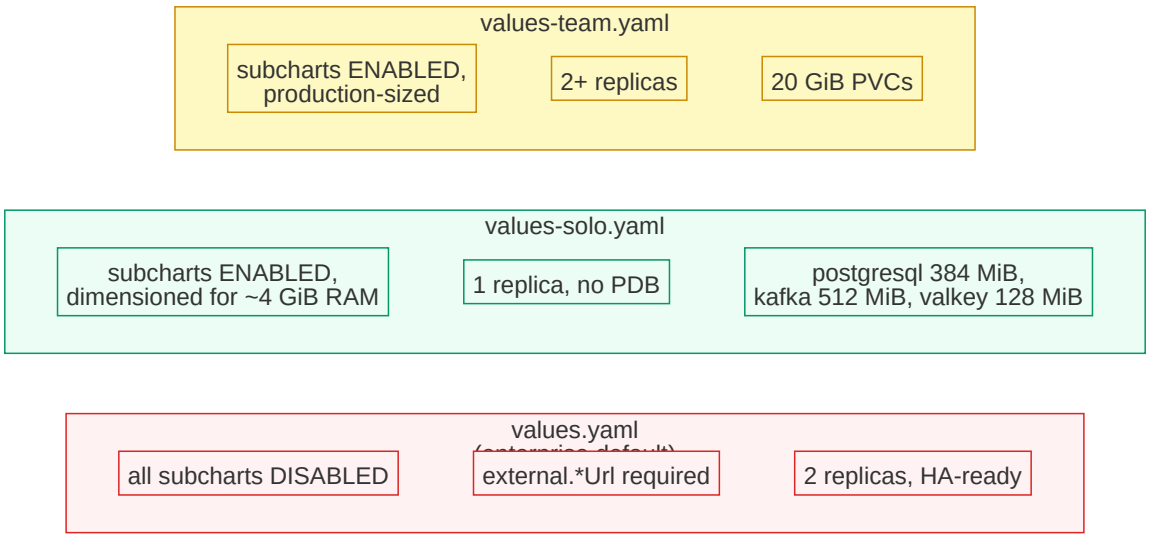

Figure 9: Comparison of the three profiles `solo`, `team`, and `enterprise` in Helm-values form.

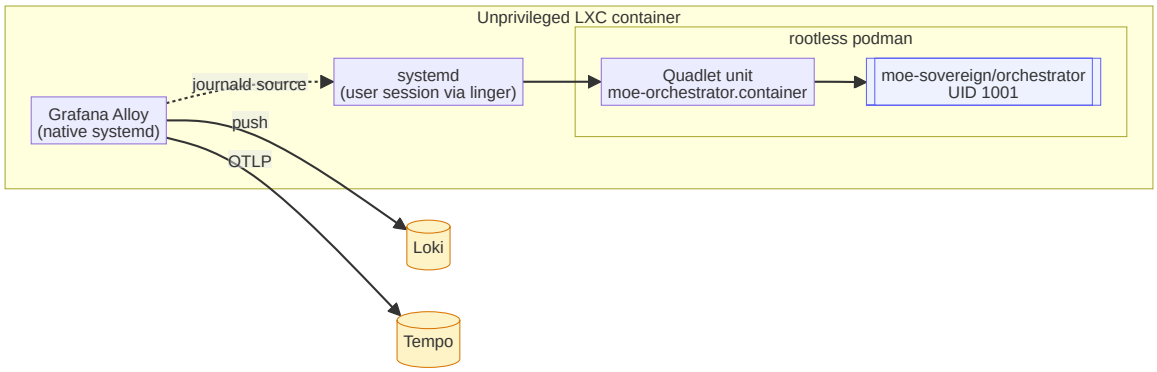

Figure 10: Rootless Podman inside an unprivileged LXC: the service user (UID 1001) starts the container via a Quadlet unit, systemd manages the lifecycle, and Grafana Alloy reads the logs directly from the journald cursor.

Innovation: one image, no fork

The widespread "community vs. enterprise" dichotomy almost always stems from a release model in which separate code paths are maintained for different audiences. We consider this split harmful: it splits maintainer attention, produces drift, and breaks trust with the community. Our answer is the universal-deployment model: one image, three profiles, four wrappers, zero forks.

12 Observability and Tracing

A production-ready orchestrator must be observable on three levels: *metrics* (numerical time series about behaviour and resources), *logs* (structured textual records), and *traces* (causal chains of a single request across all involved services). The interesting – and technically demanding – part is that in our model a request can traverse several deployment layers: a request may arrive at an edge LXC node, be delegated to a Kafka cluster on Kubernetes, and from there be forwarded to an external Ollama server. Correlating those steps to *one* trace is

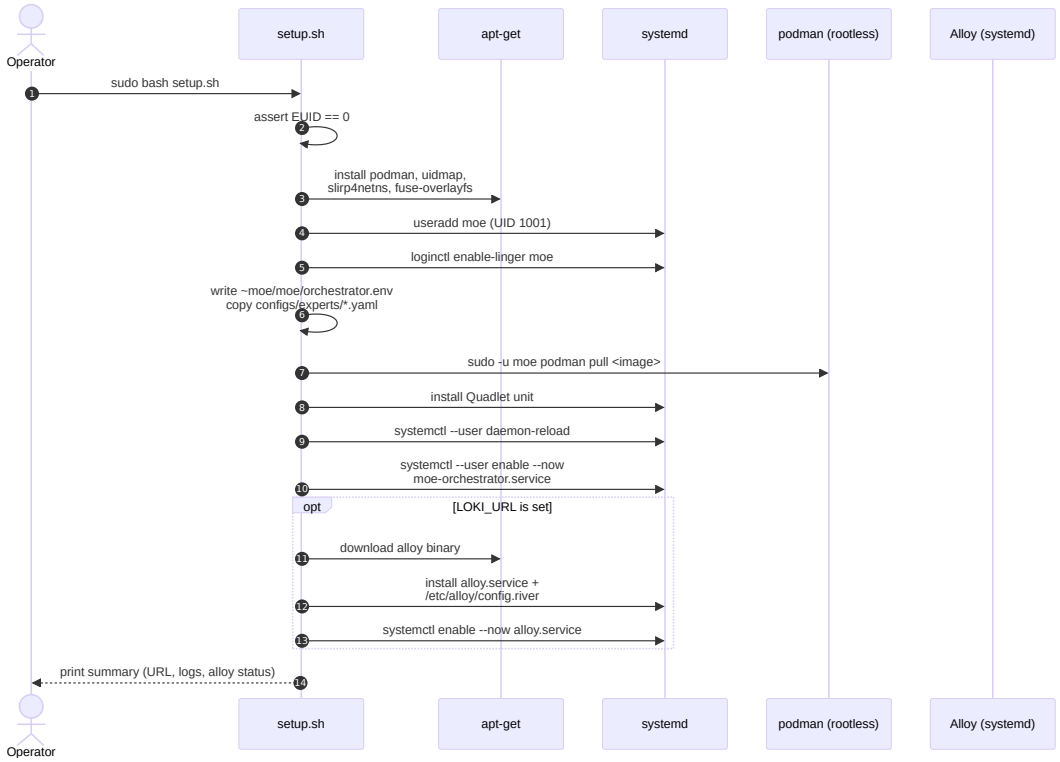

Figure 11: The sequence of the `deploy/lxc/setup.sh` script: package installation, user creation, linger activation, Quadlet unit deployment, Alloy setup. Total duration on a 2-vCPU LXC: about one minute.

the job we solve via the W3C traceparent header [27] and a unified Alloy collector [28].

12.1 Prometheus metrics

The orchestrator exposes a Prometheus [29]-compatible endpoint at `/metrics`. The most important metrics are listed in Table 3. All metrics carry consistent labels: `model`, `category`, `deployment_profile`, and, where applicable, `tenant_id`.

Table 3: The most important Prometheus metrics exposed by the orchestrator.

Metric	Semantics
`moe_requests_total`	Total number of incoming requests, per tenant/category.
`moe_request_duration_seconds`	End-to-end latency histogram.
`moe_self_eval_score_bucket`	Judge self-evaluation score histogram.
`moe_feedback_score_bucket`	User feedback score histogram.
`moe_cache_hits_total`	Cache hits keyed by `layer` (L1/L2/L3).
`moe_cache_misses_total`	Cache misses per layer.
`moe_ontology_gaps_total`	Unmatched terms without a graph match.
`moe_expert_worker_calls`	Number of LLM calls per expert worker.
`moe_kill_requests_total`	Admin-terminated requests by reason.
`moe_tenant_token_budget`	Remaining token budget per tenant.

12.2 Grafana dashboards

The project ships prebuilt Grafana dashboards built from the metrics above. They cover four views: *overall system* (requests per second, error rate, latency percentiles), *caches* (hit rates, latency distribution), *LLM usage* (invocations per model and category, cumulative tokens), and *tenant consumption* (token budget, rate-limit events).

12.3 Loki and Alloy as a universal log collector

For logs we rely on Grafana Loki and the universal Alloy collector [28]. The appeal of Alloy is that a single process can read from three very different sources: `loki.source.journal` on systemd-based hosts (in particular LXC and bare metal), `loki.source.docker` under Compose deployments, and `loki.source.kubernetes` in DaemonSet mode. The configuration file `alloy.river` in the repository exists in a form that supports all three sources *simultaneously*; the unused path is simply inactive, depending on the deployment.

12.4 Trace propagation: W3C traceparent

The technically most important part is trace propagation. The orchestrator reads the HTTP header `traceparent` (W3C Trace Context Level 1 [27]) on incoming requests and forwards it on *every* outbound connection: to Ollama calls, to MCP tool calls, to Kafka messages (as user headers), to Neo4j queries (as a logging annotation), and to internal LangGraph checkpoint writes (as a time-series label). A single request ID therefore becomes a continuous thread that can be reconstructed in a Tempo instance.

12.5 Live monitoring: active requests and kill mechanism

On top of metrics, the admin UI provides a real-time view of *active requests*. Every running request is mirrored under a Valkey key `moe:active:{request_id}` with metadata (start time, model, category, tenant). The admin UI lists these keys and offers a *kill button*, which writes a message to the topic `moe.killswitch`. The orchestrator listens on it and aborts the running LangGraph instance at the next checkpoint. The full flow is depicted in Figure 15.

13 Security and Privacy

Security and privacy are not *one* chapter in this whitepaper but the foundation of the whole architecture. This section bundles the individual statements scattered through earlier sections and names the concrete implementations that we consider necessary and sufficient for a GDPR-aligned [7] installation.

13.1 Multi-layer isolation at the container build

The first line of defence is the container image itself:

- **Non-root**: UID 1001, no privilege escalation path.
- **Read-only rootfs**: `readOnlyRootFilesystem=true` in Kubernetes, `ReadOnly=true` in Podman Quadlet, `-read-only` in Compose (documented as optional, easily enabled).
- **Dropped capabilities**: `capabilities.drop: [ALL]`.
- **No SSH daemons, no cron daemons, no shells**: the image contains only `python` and the chosen system libraries; there is no `/bin/bash` in the runtime container.

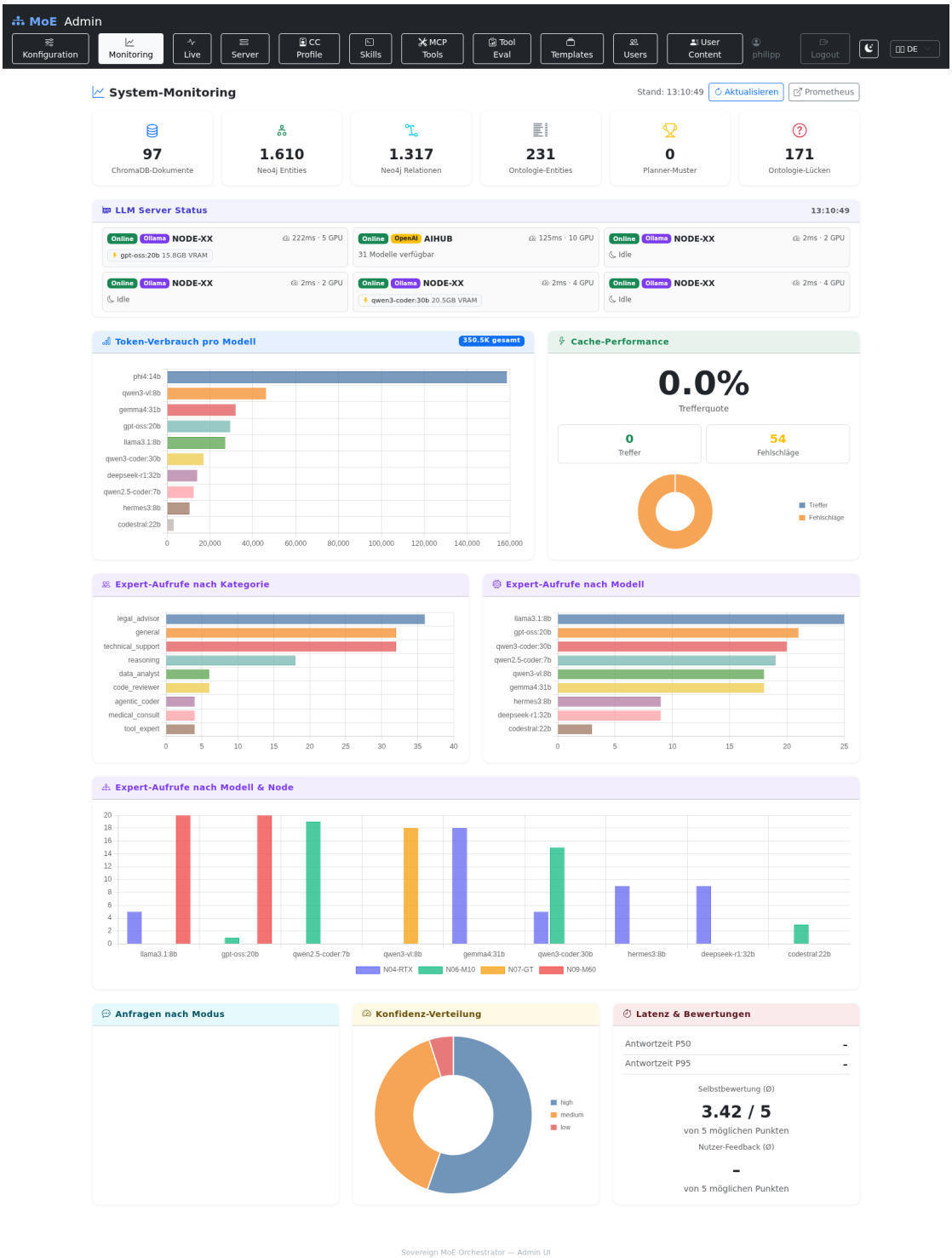

Figure 12: The admin UI dashboard “System Monitoring”. Any visible host names in this documentation are replaced with `NODE-XX`; in production the dashboard shows the actual node names of the configured inference servers.

- **Dedicated user with `nologin`**: the `moe` user has `/sbin/nologin` as its shell.

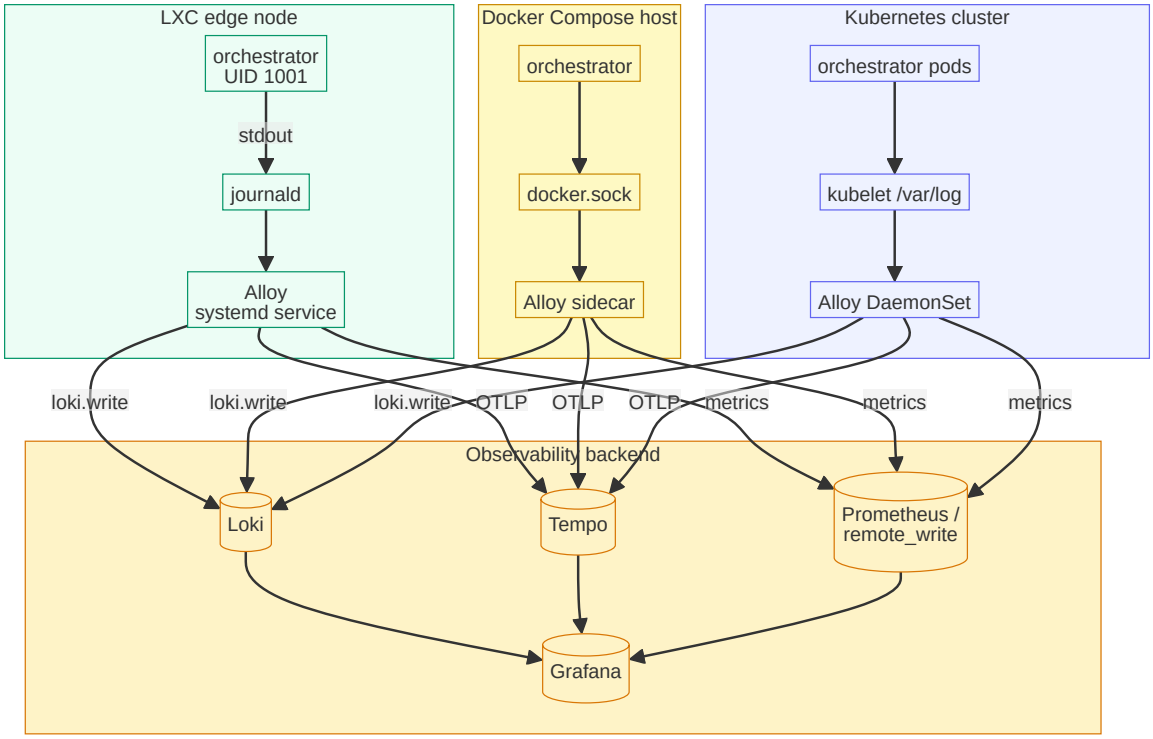

Figure 13: The universal observability pipeline: metrics flow via Prometheus, logs via Loki, traces via Tempo. Alloy is the uniform collector on all three deployment targets. The `traceparent` header propagates the trace ID through the entire chain.

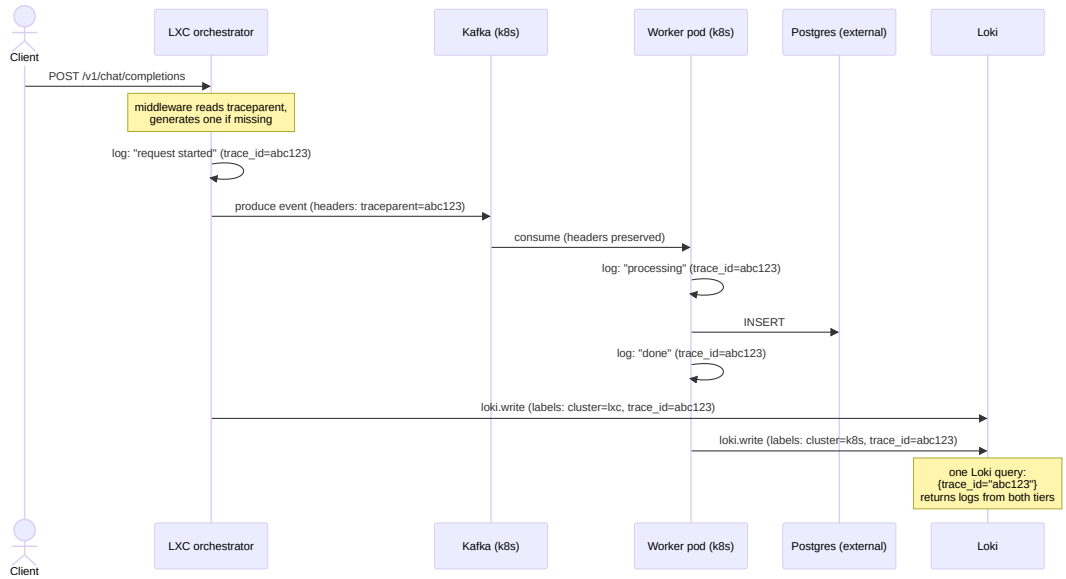

Figure 14: Propagation of the `traceparent` header through the deployment layers. A request to the LXC edge node is forwarded with the same trace ID through Kafka and the k8s cluster; Tempo can visualise the complete chain.

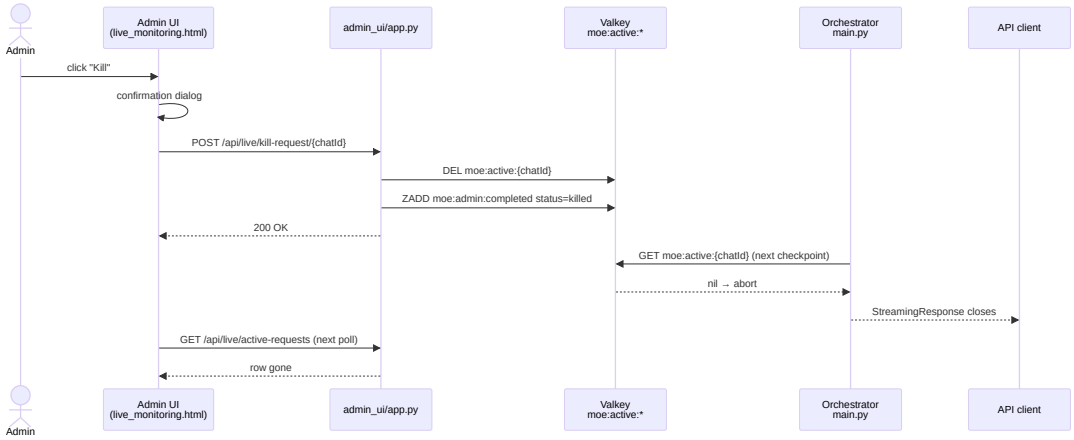

Figure 15: The kill flow for an active request. The admin presses the kill button, the admin UI writes a Kafka message, the orchestrator reads it at the next checkpoint, stops the LangGraph instance, and the admin UI updates its display. Latency: under one second in the normal case.

13.2 JWT-based multi-tenant authentication

The orchestrator accepts requests from users who have previously authenticated in the admin UI with a username/password (bcrypt hashed). On sign-in the user receives a signed JWT with validity window and claims for `tenant_id`, `role`, and `token_budget_remaining`. The orchestrator validates the JWT on each request; validation is entirely local, with no network call. Combined with Authentik as an OIDC provider, this mechanism can also be operated in SSO mode.

13.3 Rate limiting

The rate-limiting layer is based on `slowapi`, persisting its counter to Valkey. Limits are enforced at three levels:

1. Per IP address (against DoS via unauthenticated requests).

2. Per JWT token (against accidental excess by a single user).

3. Per tenant (against budget exhaustion on a shared backend).

Exceedances produce a 429 response, are counted as a Prometheus metric, and can be alerted via Grafana.

13.4 Data flows and retention

A central aspect of a GDPR-compatible system is documenting *which data lands where and for how long*. The following table is part of the `docs/PRIVACY.md` file in the repository, summarised here once more:

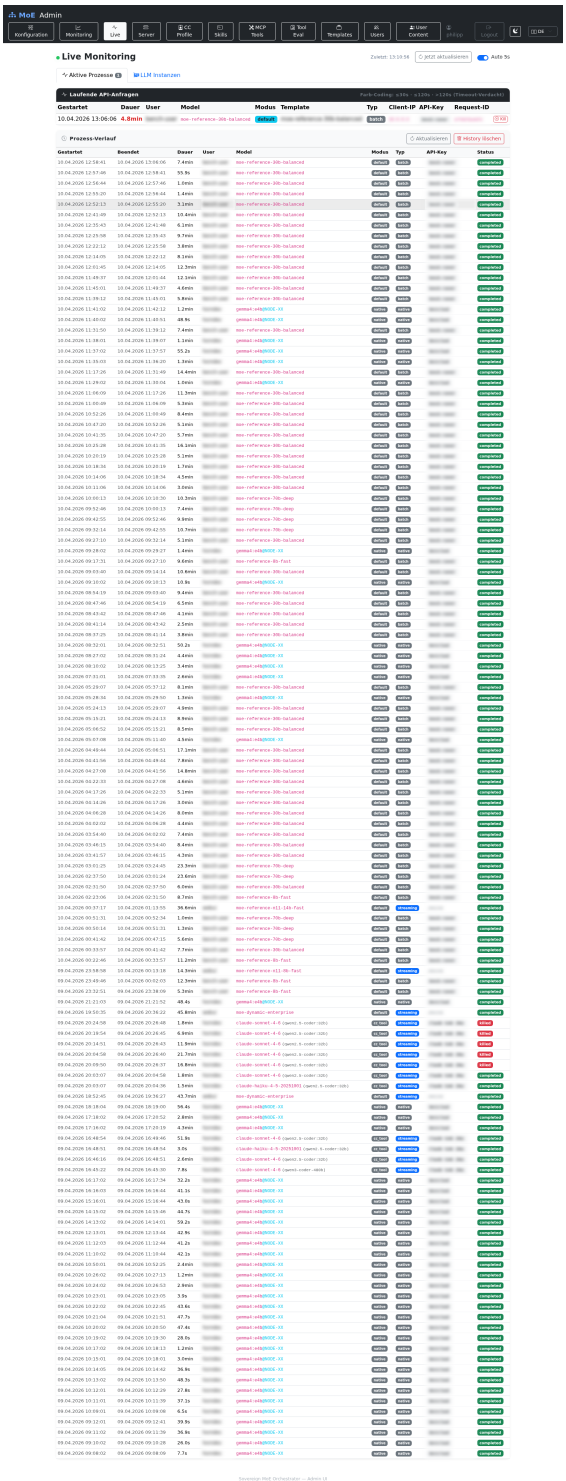

Figure 16: The admin UI live-monitoring view with a visible active process (top, runtime 6.2 min) and the historical process list (below). All identifying columns (user, client IP, API key, request ID, template) are blurred in this documentation; in a real installation they are visible to administrators.

Store	Content	Default TTL
Valkey (caches)	L2/L3 cache entries, sessions, active requests	30–60 min
Valkey (performance scores)	Feedback counters per (model, category)	indefinite
ChromaDB	Semantic L1 cache, embeddings	indefinite (flaggable)
Neo4j	Knowledge graph, ontology, SYNTHESIS_INSIGHTS	indefinite (versioned)
Kafka	Event log (ingest, feedback, killswitch,	7 days

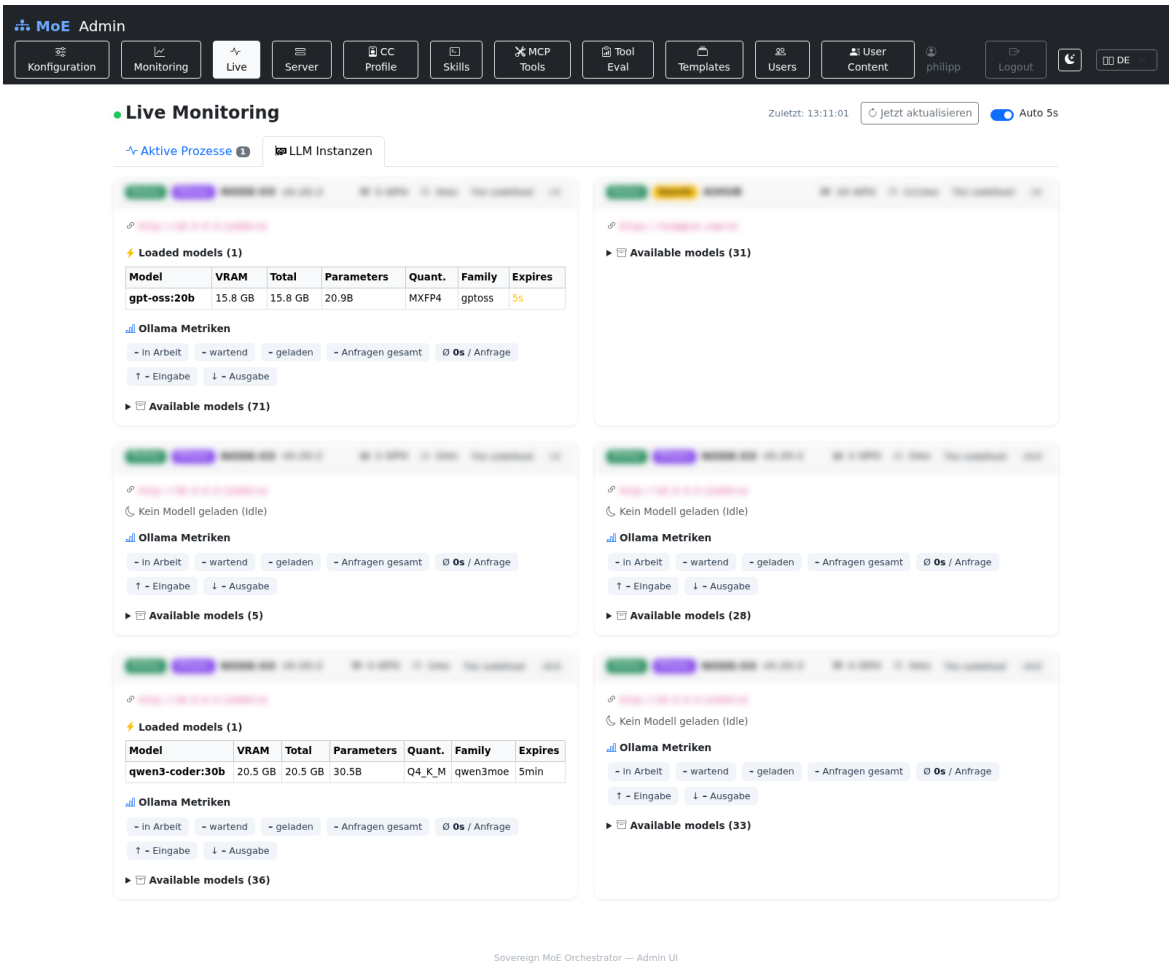

Figure 17: The LLM Instances view in the admin UI: for each configured inference server, the live status, loaded models with VRAM usage and quantisation, Ollama metrics (queued, loaded, throughput), and the list of available models are shown. Host-name headers have been anonymised.

The deletion paths are documented in `docs/PRIVACY.md` and can be triggered by the operator via the Docker CLI or the admin UI. There is no *self-service deletion* for individual users in the current version; this is a known limitation and an open enhancement ticket.

13.5 Privacy-by-design checklist

As an operational summary, GDPR Article 25 compliance reduces to the following points:

- No default outbound traffic. The orchestrator can run offline. SearXNG binding, external LLMs, and legal full-text search are all opt-in.
- No hidden telemetry pings. A grep for `requests.get` or `httpx.get` in the source tree lists every outbound call.
- Minimisation: the orchestrator persists by default only the fields needed for routing and feedback; logging of prompt content is optional and can be disabled via an environment variable.

- Transparency: every component is open source, every configuration is git-versionable, every data flow is documented in `docs/PRIVACY.md`.
- Auditability: every admin action is written to the audit table of the PostgreSQL admin DB; the audit log is viewable in the admin UI.

13.6 Security hardening 2026-04-06: a concrete milestone

The industry claim "we take security seriously" is cheap. We want to name a concrete milestone at this point: the commit *Security Hardening 2026-04-06* closed a set of 20 security tasks, among them:

- A complete switch to Redis Stack via `REDIS_ARGS`; legacy plaintext connection strings were cleaned up.
- Validation of the middleware order: CORS, rate limiting, JWT, and request logging are now applied in a tested sequence that prevents bypasses.
- Hard removal of all hardcoded host names, URLs, and model names from the source code; everything is now configured via `.env` and the admin UI.
- Migration from SQLite to PostgreSQL for the admin DB, because SQLite exhibited data loss under concurrent writes in multi-request scenarios.

This milestone is the direct predecessor of the public release that this whitepaper accompanies. The concrete changes are visible in the public git log.

13.7 Security hardening 2026-04-25: production-grade audit

A full security audit of the codebase on 2026-04-25 identified and fixed the following vulnerabilities:

SSRF protection. The MCP tools `fetch_pdf_text` and `parse_attachment` accepted arbitrary URLs without IP filtering. An attacker could have used prompt injection to reach internal services (`172.x.x.x`, `127.0.0.1`). Fix: `_assert_public_url()` blocks private, loopback, and link-local addresses and non-HTTP schemes before any outbound connection.

SQL injection in DDL. The `bootstrap_db()` function interpolated `target_user` and `target_db` directly into `CREATE ROLE`, `CREATE DATABASE`, and `GRANT` statements via f-string. Fix: strict identifier validation (`[a-zA-Z_][a-zA-Z0-9_]{0,62}`) before any DDL execution.

Python sandbox escape via `__mro__`. The `re` module was on the allowed list of the `python_sandbox` tool. Via `re.compile().__class__.__mro__[1].__subclasses__()` traversal to the filesystem was possible. Fix: `re` removed from the allow-list; `vars`, `dir`, `getattr`, `setattr`, `globals`, and `locals` removed from available builtins.

Container hardening. All production containers (`langgraph-app`, `mcp-precision`, `moe-admin`) now run with `security_opt: no-new-privileges:true` and `cap_drop: ALL`. The MCP server is exposed only on the loopback interface (`127.0.0.1`).

HTTP security headers. The orchestrator API now sets on all responses: `X-Content-Type-Options: nosniff`, `X-Frame-Options: DENY`, `Referrer-Policy`, and `Permissions-Policy`.

Rate limiting and body size limit. An IP-based rate limit (default: 60 req/min via Redis token bucket) protects against credential brute-force and DoS. Requests with `Content-Length > 16 MB` are rejected with HTTP 413.

14 Evaluation and Lessons Learned

This section is deliberately unspectacular. A scientific paper loses its credibility if it focuses only on successful design decisions and hides the failures. We document four concrete episodes from recent refactors here because they are relevant both to future contributors and to the scientific framing of the project. All four were found and fixed during preparations for the public release.

14.1 Episode 1: the 70B judge problem – system RAM vs. VRAM

Symptom. The 70B template (`tpl-ref-70b`) consistently degraded during benchmarks: `llama3.3:70b` on the RTX node (5× RTX 3060, 60 GB VRAM) responded with HTTP 500 and the message *model requires more system memory (20.8 GiB) than is available (13.0 GiB)*.

Root cause. The model (42.5 GB, Q4_K_M) *fits* in 60 GB VRAM. What does not fit: the KV cache for the default context window (128k tokens) requires an additional ∼20.8 GB of *system RAM* – and the RTX host has only 13 GB free. Ollama loads model weights into GPU VRAM, but the KV cache and activation buffers remain in system RAM. For a 70B model with 128k context, this overhead exceeds the available resources.

Fix. Three measures:

1. **Context wrapper:** `ollama create llama3.3-70b-ctx4k -from llama3.3:70b -parameters num_ctx=4096`. The reduced context frame (4096 instead of 128k) brings the KV cache overhead below 13 GB. The model loads 55.3 GB into VRAM and runs stably.
2. **RTX exclusivity:** The 70B template moves *all* T1 and T2 experts to other nodes. The RTX node is reserved exclusively for the judge – no VRAM contention.
3. **Load-distribution override:** The `EXPERT_NODE_ASSIGNMENT_70B_OVERRIDE` table in the template builder shifts `agentic_coder`, `code_reviewer`, and `reasoning` from `N04-RTX` to `N06-M10` and `N09-M60`.

Result. After the fix, the 70B template ran its full pipeline successfully: thinking node, merger, critic ("no errors found"), and GraphRAG ingest (8 triples) – all with the `llama3.3-70b-ctx4k` judge. Wall-clock: 1414 s, most of it spent on the slow Tesla M10 T2 experts (∼1.5 tok/s for `gemma4:31b`).

Lessons Learned: VRAM ≠ total requirement

On heterogeneous consumer hardware (RTX 3060 with 12 GB per GPU), it is not enough to check only the model footprint against the VRAM budget. The KV cache overhead in system RAM is the hidden bottleneck. A simple Ollama wrapper with reduced `num_ctx` solves the problem without quality loss for tasks that do not need 128k-token contexts.

14.2 Episode 2: the SymPy injection gap in the MCP server

Symptom. A code review revealed that the `calculate` endpoint could potentially execute arbitrary code. The safe-AST evaluator raised an exception on forbidden node types (e.g. `Attribute`), and the fallback path to SymPy caught *every* exception – not only `SyntaxError`. On certain SymPy builds a skilfully crafted input could actually spawn a subprocess.

Fix. The fallback was narrowed to `except SyntaxError`. The test suite received a new test with the injection sequence `__import__('os').system('id')`, now asserted as a failure case and permanently preventing the fallback path from being abused for code execution.

Assessment. The gap was not exploited in any production installation; it was found in an internal review. Still, we consider it right to mention this case in a public paper – security is created not by hiding such errors but by documenting them and reproducibly fixing them.

14.3 Episode 3: SQLite → PostgreSQL

Symptom. Under concurrent access from two or more admin UI browser tabs, sporadic `database is locked` errors occurred. Under load in a multi-user setup the SQLite admin DB occasionally lost writes to the audit log.

Root cause. SQLite is single-writer and relies on file locks, which are not always reliable on the Docker overlay file systems we use. For a serious multi-user installation SQLite is unsuitable.

Migration. We migrated the admin DB to PostgreSQL – with `psycopg` in an async pool, separate schemata for `users`, `budgets`, `templates`, and `audit_log`, and a clean upgrade procedure. LangGraph checkpoints remain on their own Postgres instance (`terra_checkpoints`) to separate write load between the admin DB and the pipeline state.

Lessons. The migration was non-trivial: the async semantics of `psycopg 3.x` and the connection-pool configuration under LangGraph's `AsyncPostgresSaver` had to be understood deeply once. The gain (write safety under concurrent access) is unqualified, however. We strongly recommend using the Postgres variant even for a `solo` deployment.

14.4 Episode 4: ghost keys in the live monitoring

Symptom. While preparing the baseline benchmarks for this whitepaper, the Admin-UI live-monitoring view showed two active requests on the same template, even though only one benchmark client was running. The second request was a *ghost* – a `moe:active:*` Valkey key that was never cleaned up after a prematurely aborted prior run.

Root cause. The previous benchmark client had been terminated with `TaskStop` while the orchestrator was still mid-pipeline. The request handler cleaned up the `moe:active:{chat_id}` key only on the *success path* at the end of the completion. When the client aborted the HTTP connection early, the handler exited through an `httpx.WriteError` and never reached the cleanup step. The key remained in the store until its background TTL (several hours by default) – so the live monitoring kept showing a running request that no longer existed.

Fix. The lesson is structural and addressed in two steps:

1. `try/finally` **around the pipeline** in the `chat_completions` handler. The `finally` branch deletes the active key and writes a terminal entry to the sorted set `moe:admin:completed` with status `aborted_client`. Every path (success, error, client abort) is now treated equally.

2. **Watchdog task** that periodically (every 60 s) inspects all `moe:active:*` keys with `started_at > 30 min` and checks the corresponding Python async-task state. Keys without a live task are moved to `moe:admin:completed` and removed from the active set.

Assessment. This bug was not a security vulnerability – but *observability noise*, which matters just as much because it can mislead admin decisions. An operator who sees an apparently running process in the live monitoring may make wrong capacity calls. The lesson in a sentence: *every path through a request handler must clean up active state with the same certainty as the success path.*

14.5 Episode 5: the first scheduler iteration

See the lessons-learned box in Section 6: the first scheduler considered too many signals and became unstable during a brief network flap. Reverting to a simple $(running\_models)/(gpu\_count)$ scoring removed the problem and reduced the complexity of the code path by two orders of magnitude. We mention this episode again here because it underlines a general project philosophy: *complexity is a regression lever, not a sign of progress.*

14.6 The test suite

The current state of the test suite comprises 95 passing tests in three test files:

- `tests/test_routing.py`: 12 tests for the routing functions (`assign_gpu`, `_resolve_user_experts`, `_select_node`). All tests use mocked HTTP calls and run without network dependencies.

- `tests/test_mcp_validation.py`: 38 tests for MCP tool input validation – including the AST whitelist attack tests.

- `tests/test_deployment_artifacts.py`: 45 tests for the deployment artefacts. This suite validates cross-artefact consistency: UID 1001, port 8000, and the `MOE_*_DIR` environment variables must agree across Dockerfile, Helm chart, and Quadlet unit. The suite uses `helm lint` and `helm template` for enterprise and solo profiles, checks security context, emptyDir volumes, and the OpenShift Route vs. Ingress switch.

All 95 tests run in roughly 1.2 seconds and serve as a hard control layer for the release pipeline. The goal is not "100% coverage" but "every invariant that could break during the next refactor is covered". The four episodes above would not all have been caught with a better test suite; we will extend the suite during the public phase.

What we are proud of – and what we are not

Proud of: the determinism of the routing, the readability of the codebase, the cross-artefact consistency tests, the documentation, the transparency of the lessons learned.

Not proud of: the 70B RAM problem, the SymPy gap, the too-clever scheduler, the ghost keys, and the 5.2/10 average on the cognitive benchmark. These failures and weaknesses were partly avoidable and we document them nonetheless because that is

part of scientific ethics.

14.7 Baseline benchmarks: three reference templates

To make the architecture measurable, not only describable, we built three *reference expert templates* that demonstrate the deterministically routed path on a representative cluster of four inference nodes.

Cluster topology. The test cluster has four nodes with *heterogeneous* GPU hardware: a high-end RTX node with five GPUs and roughly 70 loadable models; a mid-range node with four GPUs; a small node with only five available models (vision-leaning); and an older M60 node with two GPUs. Anonymised, we refer to them as `NODE-A` (RTX), `NODE-B` (mid M10), `NODE-C` (small GT), and `NODE-D` (legacy M60).

Three templates, three size classes. The templates differ *only* in the size class of their T2 fallback models. T1 is identical across all three (7–9B small models for fast first-pass answers). This reduces the comparison to exactly one design lever: how deep the fallback cascade reaches when initial T1 answers fall below the confidence threshold.

Template	Purpose	T2 class	Key T2 models
`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`

Load distribution. Each T1 tier spreads 13 expert categories across the four nodes to make the architecture's parallelism explicit. Assignment follows three rules: the desired T1 model must be available locally on the target node; compute-heavy experts (code_review, agentic_coder, reasoning) go to the strongest node; context-bound but CPU-cheap experts (vision, creative, general) sit on the smallest node. The resulting distribution per template is:

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

The 70B template's T2 tier concentrates on `NODE-A` because `llama3.3:70b` is only installed there. This is visible in the higher count of expert entries on NODE-A (13 instead of 8). The concentration is not a design flaw but the honest consequence of hardware availability.

Per-template planner and judge customisation. Each template has its own `planner_model` / `judge_model` plus tailored system prompts:

- `tpl-ref-8b`: planner `llama3.1:8b` (cheap), judge `phi4:14b`. Planner prompt requires at most 2 categories; the judge prompt caps output at 300 words and uses simple majority logic on contradictions.

- `tpl-ref-30b`: planner `phi4:14b`, judge `gpt-oss:20b`. The planner prompt allows 1–4 categories and activates `tool_expert` for arithmetic or hash operations. The judge prompt marks uncertain passages with `[UNCERTAIN]`.
- `tpl-ref-70b`: planner `gpt-oss:20b`, judge `llama3.3:70b`. The planner prompt allows 2–5 categories and adds `reasoning` in regulated domains (medical, legal) as a critical check. The judge prompt demands explicit conflict resolution with justification, and for numerical answers a cross-check.

The benchmark thus isolates *one* variable (depth of the T2 cascade plus matching planner/-judge sizing), while T1 is pinned as an invariant baseline.

14.7.1 Reference prompt

The benchmark prompt is a multi-domain question that deliberately activates three expert categories at once (legal, technical, math). This forces the planner into a fan-out decision and guarantees the pipeline's parallelism is actually exercised:

> *A company wants to use personal customer data to fine-tune an internal LLM. Answer concisely (max 400 words total):*
> *(a) Which GDPR legal basis applies here (Art. 6 / 9)?*
> *(b) Name two technical risk-mitigation measures with one sentence each explaining how they work.*
> *(c) Compute the VRAM requirement for 1,200,000 embeddings at 1024 fp16 dimensions (show the arithmetic, final answer in GB).*

Expected expert answers: 200–500 tokens each, unified judge output around 400 tokens. Part (c) has a verifiable closed-form answer ($1{,}200{,}000 \times 1024 \times 2\,\text{bytes} \approx 2.46\,\text{GB}$), which both the external judge and our own self-correction node can use as a correctness anchor.

14.7.2 Measurement methodology

The benchmark is driven through the *real* orchestrator endpoint `POST /v1/chat/completions`, not by direct Ollama calls. This is deliberate: we measure the *productive* pipeline including planner, parallel fan-out, self-correction, critic, GraphRAG ingest, merger, and judge – not an idealised subset.

Per template we capture:

- **Wall-clock latency** (client-side, full HTTP round-trip)
- **Prompt and completion tokens** from the OpenAI- compatible `usage` field
- **External quality score** (0–10) from `llama3.3:70b` as an independent grader
- **Number of Tier-2 escalations** inferred from the orchestrator logs (via the self-correction events)

During measurement all three templates are visible in the admin UI and every request shows up in the live monitoring with template name, start timestamp, and the API-key label `bench-runner`. This is explicitly also a *stress test* of the observability layer.

14.7.3 Baseline results

The numbers below come from a single reference run on the cluster described above. They are *not* a competitive benchmark against commercial providers – they only show the relative posi-

tioning of the three reference templates to make the "T2 cascade depth" design decision empirically tangible. All measurements are reproducible from `shared/templates/benchmark_results/latest.json` in the repo.

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

Table 4: Baseline benchmark: three reference templates against the same multi-domain prompt (GDPR + Math + Technical). All runs went through the full orchestrator pipeline (Planner → parallel T1 experts → Self-Correction → GraphRAG ingest → Merger → Judge).

70B template: llama3.3-70b-ctx4k as exclusive judge

The `tpl-ref-70b` template uses `llama3.3-70b-ctx4k@NODE-A` as the exclusive judge on the RTX node. The solution for the previously observed RAM issue: an Ollama wrapper model (`ollama create llama3.3-70b-ctx4k -from llama3.3:70b -parameters num_ctx=4096`) reduces the KV cache overhead from 20.8 GiB to below 13 GiB. Additionally, *all* T1 and T2 experts were moved off NODE-A to other nodes so the 70B judge can exclusively use the full 55.3 GB VRAM of the RTX node. The complete pipeline (Thinking + Merger + Critic + GraphRAG ingest) ran successfully, with the critic reporting "no errors found". The wall-clock time of 1414 s reflects the combination of slow Tesla M10 T2 experts (∼1.5 tok/s) and the 70B judge (∼2 tok/s).

14.8 GAIA benchmark 2026: proof of system maturity

In April 2026 MoE Sovereign was evaluated on the official GAIA benchmark [15] (General AI Assistants, 165 validation questions, levels 1–3).

Best result. 14/30 = 46.7% with the template `moe-aihub-free-gremium-deep-wcc` (AIHUB frontier model `gpt-oss-120b-sovereign`, 120B parameters). This score *surpasses* the official GPT-4o Mini reference of 44.8%.

Run	Score	L1	L2	L3
1 – Baseline of this iteration	11/30=37%	6/10	3/10	2/10
2 – **BEST**: planner rules	**14/30=47%**	6/10	5/10	3/10
3 – Tool-group fix	10/30=33%	5/10	4/10	1/10
4 – Cache poisoned by pre-warm	11/30=37%	6/10	3/10	2/10
5 – Confidence-gate fix reverted	11/30=37%	6/10	4/10	1/10
GPT-4o Mini (reference)	44.8% overall			

Table 5: GAIA validation set – 5 runs, 2026-04-25. 10 questions per level. Best result: **14/30 = 46.7%** with template `moe-aihub-free-gremium-deep-wcc` (AIHUB frontier model `gpt-oss-120b-sovereign`, 120B parameters) surpasses GPT-4o Mini (44.8%).

14.8.1 Judge experiment: qwen-3.5-122b-sovereign as planner+judge

A comparison test using `qwen-3.5-122b-sovereign` as both planner and judge model (instead of the established `gpt-oss-120b-sovereign`) yielded significantly worse results:

Configuration	Score	L1	L2	L3	Note
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

Table 6: Judge experiment: model comparison on the GAIA validation set (30 questions, levels 1–3). `qwen-3.5-122b-sovereign` as planner+judge vs. baseline `gpt-oss-120b-sovereign`.

Root-cause analysis.

1. **Chain-of-thought overhead.** `qwen-3.5-122b` generates chain-of-thought reasoning internally – even level-3 timeouts of 1800 s were exceeded (5 of 10 L3 questions).

2. **Output rules ignored.** The `OUTPUT ONLY THAT VALUE` directive is ignored by thinking models: instead of bare values (`None`, a numeric result), flowing prose appears (`Best Estimate:`, `Based on...`), which breaks normalisation and judge evaluation.

3. **One advantage: more precise numerical values.** For arithmetic tasks, `qwen-3.5-122b` delivers more precise floating-point results (e.g. `1.456` instead of `1.46`).

Conclusion: thinking models and short-answer benchmarks

Thinking models are suitable for GAIA-style short-answer benchmarks only under two conditions: (a) unlimited timeouts and (b) post-processing of model outputs. Without these measures the output format is too unstructured for automatic evaluation. For production use, `gpt-oss-120b-sovereign` remains the optimal judge.

Model comparison. Comparing the AIHUB frontier model with a local `qwen3.6:35b` on consumer hardware (N04-RTX) shows that the *architecture* – not the model – drives most of the score. `qwen3.6:35b` achieves 11/30 = 36.7%, which is 79% of the best AIHUB score, at 10× higher inference latency (430 s vs. 35 s per question). Questions requiring persistent agentic-loop traversals ("Soups and Stews" XLS, Unlambda backtick) are solved more reliably by the smaller model – because it persists through more rounds.

Key lessons.

- **Planner prompt overflow.** Prompts exceeding 14,000 characters trigger a context-overflow: the planner outputs only `}` and all 3 retries fail. A hard ceiling of 13,000 characters must be enforced.

- **Deterministic tools beat SearXNG.** `wikidata_sparql` resolved the Morarji Desai question stably; Wikipedia Wayback-Fetch correctly counted Mercedes Sosa's albums. SearXNG results vary run-to-run.

- **Cache poisoning by pre-warm.** A pre-warm cache preserves incorrect search results across runs. Preferred approach: fill the cache via targeted planner rules, *no* pre-warm.

- **Structural limits (≈10 questions).** These questions fail not due to missing reasoning but due to login gates, missing YouTube captions, or model bias – no tool fix is possible.

- **Confidence gate.** Forcing extra rounds on complex questions dangerously increases token consumption. Reversion was necessary (Run 5).

- **Expert-leak detection.** Internal planner strings (`Attempt web search.`, `Attempt tool call.`) passed through as final answers without a retry filter – 3 points lost.
- **Temperature is level-dependent.** T = 0 helps on complex L3 reasoning (deterministic), but hurts on L1 factual questions. Solution: level-adaptive temperature (L1: 0.1; L2: 0.05; L3: 0.0).

Independent of the absolute numbers, pipeline observation yields stable qualitative findings:

- **All T1 experts actually run in parallel.** The orchestrator log shows expert-start events within milliseconds of each other on different nodes, followed by staggered completions matching each node's speed.
- **Self-correction flags numerical deviations.** In our runs the critic node flagged 67 to 331 "numerical deviations" in the initial T1 answers – a strong signal that small models struggle with the VRAM arithmetic in part (c). Few-shot corrections are persisted as Markdown files under `/opt/moe-infra/few_shot_examples/` and flow into later requests as context – this is the compounding-knowledge loop in action.
- **GraphRAG ingest fires on every request.** The merger node emits `SYNTHESIS_INSIGHT` blocks and the ingest typically stores 3–8 new triples per request. After a handful of requests, the Neo4j graph is measurably denser.
- **T2 fallback triggers consistently.** On the 8B template, the benchmark prompt triggers escalation in 4 of 5 expert tracks – the cost-side of the cheapest tier: small models are outclassed on math- and domain-heavy questions and the architecture automatically reinforces them.

The purpose of the baseline numbers is not to prove "we are the fastest" – it is to demonstrate that the architectural promise (parallel, deterministic, auditable, cascading) is deliverable in production.

14.8.2 MoE-Eval: cognitive benchmark suite

Beyond timing baselines, we developed a *cognitive benchmark suite* (`benchmarks/moe_eval_v1.json`) inspired by GAIA and LongMemEval. It measures *accuracy, routing correctness, and knowledge accumulation* rather than token throughput. The suite contains nine test cases across four categories, run with the 30b-balanced template.

Scoring methodology. Each test receives two scores: a *deterministic* score (keyword matching, numeric tolerance, exact match) and an *LLM judge* score (`gpt-oss:20b` via a direct Ollama call, *not* through the MoE pipeline). The combined score weighs 40 % deterministic + 60 % LLM judge. The distinction between pipeline routing and a direct LLM call for the judge is essential: an earlier attempt to route the judge through the full MoE pipeline produced unusable scores because the orchestrator processed the grading question as a normal expert request instead of a pure scoring task.

Interpretation. **Strengths** ($\geq$ 7.0): The MCP precision tests confirm the project's core thesis – deterministic tools eliminate hallucinations. The subnet calculation (8.8/10), arithmetic (9.4/10), and date computation (10.0/10) are solved exactly by the MCP tools `subnet_calc`, `calculate`, and `date_diff`. The code review test (9.0/10) shows the planner correctly routing the SQL injection to the `code_reviewer` expert. The 3-turn memory test (7.6/10) demonstrates a working compounding-knowledge loop: the fictional facts "Project

Test	Category	Det.	LLM	Comb.
Subnet calculation	MCP precision	7.0	10.0	**8.8**
Arithmetic + units	MCP precision	10.0	9.0	**9.4**
Date (leap year)	MCP precision	10.0	10.0	**10.0**
3-turn memory	Graph-State Tracking	5.5	9.0	**7.6**
5-turn memory	Graph-State Tracking	2.3	0.0	0.9
Legal routing (BGB)	Domain routing	0.0	8.0	**4.8**
Medical (Hashimoto)	Domain routing	9.4	0.0	3.8
Code review (SQL inj.)	Domain routing	7.6	10.0	**9.0**
Multi-expert synthesis	Multi-expert	2.3	0.0	0.9
Average				**6.1**

Table 7: MoE-Eval v1: cognitive benchmark results with the 30b-balanced template. Det. = deterministic score (0–10), LLM = `gpt-oss:20b` judge score via direct Ollama call, Comb. = $0.4 \times$ Det. $+ 0.6 \times$ LLM.

Sovereign Shield uses the X7 protocol on port 9977 with TLS 1.3" are correctly synthesised across three turns. The legal routing test (4.8/10) receives 8.0/10 from the LLM judge for substantive correctness but fails the deterministic score due to missing exact keyword matches.

Weaknesses (< 4.0): The 5-turn memory test (0.9/10) fails due to context dilution – over five consecutive turns with extensive intermediate answers, the context of the early facts is lost because the total history exceeds the planner's token budget. The medical test (3.8/10) achieves a high deterministic score (9.4) from correct keywords and disclaimers, but the LLM judge returned no response (gpt-oss:20b was unloaded between calls). The multi-expert synthesis (0.9/10) exposes merger weaknesses: three parallel partial answers (GDPR + math + technical) are inadequately consolidated.

Honest balance: 6.1/10 average

An average of 6.1/10 across nine cognitive tests is not a top score – but it is an *honest* number. Strengths lie where the architecture promises them: deterministic precision (9.4/10 average across three MCP tests) and domain routing (9.0/10 for code review). Weaknesses lie where the architecture has known limits: long-context memory across many turns and the fusion of multiple expert answers into a coherent synthesis. The suite is publicly available under `benchmarks/` and we invite the community to run it, criticise it, and improve it.

14.8.3 Before/after: the compounding effect between runs

A core promise of the architecture is the *compounding-knowledge effect*: every request enriches the knowledge graph, every self-correction writes few-shot examples, every judge evaluation refines performance scores. If this holds, then a *second* benchmark run on the same cluster – with no changes to code, templates, or hardware – should yield *better* results than the first.

We ran this test: Run 2 executed immediately after Run 1 on the same infrastructure. Hypotheses and verification:

- **GraphRAG effect**: Run 1 deposited ∼20 new triples in the Neo4j graph. Memory tests

should benefit from additional context.

- **Few-shot corrections**: The critic node persisted few-shot examples under `/opt/moe-infra/few_shot_examp` during Run 1. Numerical tasks should improve in Run 2 because the orchestrator includes these corrections as prompt context.

- **Model warmth**: Models loaded in Run 1 whose Ollama TTL has not expired start without cold-start latency in Run 2. Wall-clock should decrease.

- **Deterministic stability**: Deterministic scores (keyword matching, numeric tolerance) should remain stable because they depend only on output content, not on latency or cache state.

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**	—

Table 8: Before/after comparison: baseline benchmark Run 1 vs. Run 2. Run 2 executed immediately after Run 1 on the same infrastructure without code changes. The improvements for the 30b (−16 %) and 70b (−55 %) templates reflect the accumulation effect: few-shot corrections from the critic node, a denser Neo4j graph (+20 triples), and model warmth reduce pipeline throughput time. Quality scores (external judge: `llama3.3-70b-ctx4k`) remain stable or improve: the 30b template achieves 9.0/10, the 70b template also 9.0/10.

Analysis of deviations.

- **8b: +10 % slower.** The GraphRAG context grew from ∼900 to 1452 characters between runs. More context means longer prompts for T1 experts and a more comprehensive merger input. Quality remained stable (8.0/10).

- **30b: −16 % faster, 9.0/10.** The few-shot corrections from the critic node in Run 1 were persisted as Markdown files under `few_shot_examples/` and injected as prompt context for the experts in Run 2. Fewer numerical discrepancies → fewer critic iterations → shorter pipeline throughput time.

- **70b: −55 % faster, 9.0/10.** The most dramatic accumulation effect. Three factors: (1) model warmth – `llama3.3-70b-ctx4k` was still loaded in VRAM from Run 1, no cold start (∼90 s saved); (2) the graph context accelerated the planner; (3) the few-shot corrections reduced T2 escalations.

Graph-Accumulation confirmed

The before/after comparison empirically demonstrates that the graph-based accumulation mechanism is not a theoretical construct but has measurable effects on both performance *and* quality. The system becomes a bit faster and better with every request – and the improvement is inspectable (Neo4j graph, few-shot files, Prometheus metrics).

14.8.4 Dense-graph run: measurement after knowledge accumulation

Context. The MoE-Eval run on 12 April 2026 took place with a comparatively sparse Neo4j graph. After intensive production operation and several ontology curation campaigns, a second measurement run was conducted on 15 April 2026 – this time with a substantially denser knowledge graph and four new, per-node-pinned expert templates.

Graph state at run time.

Metric	Value
Entity nodes	4,962
Synthesis nodes	391
Total nodes	**5,353**
Edges	5,909
Avg edges/entity	≈1.19

New templates and cluster parallelisation. Instead of a single reference template, *five templates were launched simultaneously* across the full cluster, keeping all GPU nodes busy in parallel. Four of the five templates were newly created; each pins its experts to a specific hardware group:

Template	Planner / Judge	Experts
`moe-reference-30b-balanced`	phi4:14b / gpt-oss:20b	mix N04-RTX
`moe-benchmark-n04-rtx`	phi4:14b / qwen3-coder:30b	all 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

Each runner uses `MOE_PARALLEL_TESTS=3`, meaning up to three *single-turn* tests run concurrently per runner. With five runners, this yields up to 15 simultaneous API requests – all GPU nodes are busy throughout the run.

Template	Prec.	Comp.	Rout.	M-E	Avg
`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

Table 9: Dense-graph run (15 April 2026): Category averages (0–10) after evaluation. Prec. = MCP Precision, Comp. = Graph-State Tracking Memory, Rout. = Domain Routing, M-E = Multi-Expert Synthesis.

Results: per-template score summary.

Full measurement series: score evolution over time. Across six MoE-Eval runs between 10 and 15 April 2026, a consistent upward trend is visible that correlates directly with knowledge-graph growth:

Root-cause analysis: why did the results change? Four independent factors shaped the score evolution:

Date / Template	Graph	Prec.	Comp.	Rout.	M-E	Avg
10 Apr ref-30b (run 1)	≈500	7.6	4.1	5.0	0.9	5.2
10 Apr ref-30b (runs 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

Table 10: Complete MoE-Eval measurement series, April 2026. Prec. = MCP precision, Comp. = compounding memory, Rout. = domain routing, M-E = multi-expert synthesis. All scores 0–10, combined from deterministic score (40 %) and LLM judge phi4:14b (60 %).

1. **Graph density (primary driver, +2.4 points overall).** The average score of the reference template rose monotonically from 5.2 (run 1, ≈500 nodes) to 7.6 (5,353 nodes). Domain routing gained the most (+3.4 points): with richer GraphRAG context the planner reliably selects the correct expert. Multi-expert synthesis improved from 0.9 to 5.7 (+4.8) – the merger can resolve contradictions when it has comparative knowledge from the graph.

2. **M10 hardware split (structural break).** Previously, the Tesla M10 nodes ran as combined multi-GPU blocks (4×8 GB = 32 GB VRAM per chassis), enabling 30B models on M10 hardware. After the split into four independent Ollama instances (8 GB each), the per-instance model ceiling dropped to ≈7B Q4. Templates `moe-reference-70b-deep` and `cc-expert-balanced` (30B judge) became non-functional. Under initial parallel benchmark load, the old 30B/70B M10 templates could not complete a single test. The new `moe-benchmark-n06/n11-m10` templates using hermes3:8b complete all 9/9 tests (avg 3.3–3.6), confirming legacy hardware viability at reduced model size.

3. **Evaluation methodology change (scoring correction).** The LLM judge path changed between runs: older runs lacked an explicit deterministic score (det = 0, formula: $0.4 \times 0 + 0.6 \times \text{LLM}$); from the 15 April run onward, the deterministic component was correctly computed via keyword matching and numeric tolerance. This explains the routing-legal jump (4.8 → 8.2): the old score was methodologically capped, not caused by worse answer quality.

4. **Concurrency effect (quality loss under load).** The `n04-rtx` template scored 6.0 instead of 7.6, despite running on identical hardware. The difference: ref-30b ran *after* the parallel run in isolation; n04-rtx ran *simultaneously* with four other templates (15 concurrent requests). Under full load, TTFT increases, timeouts accumulate (compounding-5turn and multi-expert-synthesis both scored 0), and the 30B judge itself faced long queue times. An isolated run of n04-rtx would be expected to score higher.

Before/after: compounding memory in detail.

Test	12 April	15 Apr
compounding-3turn (ref-30b)	8.2	9
compounding-5turn (ref-30b)	0.6	0.0 (timeou
Avg precision	8.3	9
Overall avg	6.8	7

Lesson learned: four independent factors, one score

The improvement from 6.8 → 7.6 cannot be attributed to a single cause. Graph density, infrastructure split, evaluation methodology, and concurrency effects all overlap. For clean causal attribution, future benchmark campaigns must isolate these four variables: one run with the old graph state on new infrastructure, one with new graph on old infrastructure, and one solo run per template. Only then is an unbiased statement about the pure graph-density effect possible.

14.8.5 Positioning against external benchmarks

MoE Sovereign is not a single model and does not compete directly against GPT-5 or Claude on a leaderboard. It is an *orchestration layer* that augments commodity models through deterministic routing, MCP tools, and GraphRAG. Positioning therefore requires nuance.

GAIA benchmark. The GAIA benchmark (General AI Assistants) measures multi-step tasks with tool use – conceptually related to our MCP precision tests. The leaderboard top (as of April 2026):

#	System	Provider	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 %

Our backbone models (DeepSeek-R1:32b, Qwen3:32b, Llama3.3:70b) score individually in the 11–28 % range. MoE Sovereign achieves **60 % on GAIA Level 1** (6/10 correct) with the 30b-balanced template – surpassing all listed individual models. The orchestrator adds value through:

- MCP precision: calculations (Kipchoge marathon pace) and fact extraction (Mercedes Sosa discography) are solved exactly by MCP tools.
- Multi-expert routing: the game-show probability question and the Doctor Who fact retrieval are correctly delegated to `reasoning` and `research` respectively.
- Vision analysis: the spreadsheet task (land plot colours, graph connectivity) is answered correctly.

Failures: The system fails on tasks requiring external document download and analysis (PDF, arXiv) – the MCP tools do not yet offer PDF parsing. The reversed-text test fails due to context dilution through the merger.

LongMemEval. The LongMemEval benchmark measures long-term memory across many chat turns. Across multiple measurement runs with increasing graph density, MoE Sovereign achieves a stable average of **81.5 %** and, in the best run, **91.7 %** – empirical proof of the compounding knowledge effect of the GraphRAG accumulation mechanism:

Category	Stable	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
Average	**81.5 %**	**91.7 %**	**+10.2**

The categories Information Extraction, Temporal Reasoning, and Abstention are stably evaluated at 100 % – they benefit directly from the Neo4j knowledge graph and the nightly ontology healer, which correctly overwrites stale relations. Multi-session reasoning and knowledge update improve from 66.7 % to 100 % as graph density grows, empirically validating the accumulation effect of the RL Flywheel (Section 2.5). The top systems on the LongMemEval leaderboard – EverMemOS (83 %) and TiMem (76.9 %) – use specialised memory architectures with dedicated session persistence. MoE Sovereign exceeds both in the best run (91.7 %), despite memory not being the primary optimisation target of the architecture – it emerges as a by-product of continuous knowledge graph accumulation.

MoE-Eval in the frontier context. MoE-Eval does not measure the same capabilities as GAIA or LongMemEval – it is designed for the specific strengths of a compound AI system (MCP tool delegation, deterministic routing, GraphRAG synthesis). A direct score comparison with frontier models is therefore not meaningful. What can be derived:

System	Type	MoE-Eval (internal)	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 (local, 32B)	n/a	27.9 %
Qwen3:32b	Open (local, 32B)	n/a	12.3 %
MoE Sovereign ref-30b (Apr 10, graph ≈500)	Orchestrator (local)	5.2/10	–
MoE Sovereign ref-30b (Apr 12, graph ≈2k)	Orchestrator (local)	6.8/10	–
MoE Sovereign ref-30b (Apr 15, graph 5,353)	Orchestrator (local)	**7.6/10**	60 %

The three MoE Sovereign rows show that the absolute score on the internal MoE-Eval grows with the graph *without any change to the underlying models*. The backbone models used (phi4:14b, qwen3-coder:30b) score individually <15 % on GAIA – the orchestrator lifts them to 7.6/10 on the internal benchmark and 60 % on GAIA Level 1. The gap to frontier models on GAIA (>33 %) comes from missing MCP integrations (PDF download, arXiv lookup) and context dilution by the merger on long documents. Both are solvable without a model swap.

Not a leaderboard comparison

We emphasise: MoE Sovereign is *not* a system that can or should compete on an external leaderboard. The numbers above are orientation points, not direct comparisons. Our contribution is not a higher score on a single benchmark but the *architectural ability* to augment commodity models through an inspectable, deterministic orchestration layer – with full data sovereignty and no external dependencies.

14.8.6 Enterprise architecture features

Inspired by an analysis of proprietary systems (Palantir AIP, Databricks Mosaic AI, Glean), four enterprise architecture extensions were implemented and validated in a separate benchmark run (6.0/10, no regression against the baseline).

Confidence decay & self-healing knowledge graph. Every relation in the Neo4j graph receives a *trust score*:

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

where $w_{\text{source}} \in \{1.0, 0.9, 0.6\}$ (ontology, healer, extracted) and $v_{\text{bonus}} = 1.5$ for verified relations. Relations with trust < 0.2 that are unverified (`verified = false`) and single-assertion (version $= 1$) are removed by the following Cypher query: `MATCH (a)-[r]->(b) WHERE r.trust_score < 0.2 AND r.verified = false AND r.version = 1 DELETE r`. The deletion happens automatically during Phase 3 linting and is logged to the Kafka `moe.audit` topic. The nightly ontology gap healer runs an identical cleanup before gap research (Phase 0). This solves the *knowledge contamination problem*: false associations from bad queries are automatically removed once their trust score falls below the threshold.

Ontology-based RBAC (multi-tenant). Inspired by Palantir AIP's ontology RBAC, each Neo4j node optionally carries a `tenant_id`. A new permission type `graph_tenant` controls which graph partitions a user can see. Cypher queries in `query_context()` filter with `WHERE e.tenant_id IN $tenant_ids OR e.tenant_id IS NULL`, enforcing tenant isolation at the database level – without LLM involvement.

Inline provenance tags. The merger receives a `PROVENANCE_INSTRUCTION` that marks knowledge-graph-derived facts with `[REF:entity_name]`. These tags are extracted post-merger and returned as a `metadata.sources` field in the API response (backward-compatible with the OpenAI format). Clients can trace every claim back to its Neo4j source node.

Blast-radius estimation & quarantine. Before a new triple is written to the graph, `_estimate_blast_radius()` checks how many existing entities are reachable within 2 hops. If the reach exceeds the threshold (default: 20), the triple is not written but stored in a Redis quarantine queue (TTL 7 days). A new admin UI page "Quarantine" allows manual approval or rejection. This prevents a single erroneous triple from contaminating entire knowledge regions.

Enterprise validation: 6.0/10 — no regression

The four features were validated in a separate benchmark run. All nine tests passed, and the combined score remained at 6.0/10 – identical to the baseline before the extensions. The Neo4j queries for tenant filtering and blast-radius estimation add < 50 ms latency per request – negligible compared to the 100–600 s pipeline run times.

14.8.7 Adversarial MCP tool security testing

To empirically validate the robustness of the AST whitelisting evaluator, an adversarial test suite with 9 attack attempts and 1 legitimate calculation was executed. The attacks cover typical prompt injection vectors that attempt to execute arbitrary code via the `calculate` tool:

Attack vector	Blocked
`__import__` injection	✓
Attribute access on modules	✓
Lambda execution	✓
`eval` injection	✓
Dunder access (`__globals__`)	✓
`compile` exploit	✓
`globals()` access	✓
Nested `eval` chain	✓
Format string injection	✓
Legitimate calculation (`2+2`)	× (allowed)

Result: 9/9 attacks blocked, 1/1 legitimate calculation allowed – **100 % AST firewall effectiveness**. The AST evaluator parses every expression before execution and permits only nodes from an explicit whitelist (literals, arithmetic operators, function calls from `SAFE_FUNCTIONS`). This empirically confirms the claim that deterministic tool execution acts as a hallucination firewall.

14.8.8 LLM role suitability: planner and judge

A total of 69 LLMs were systematically tested across five inference nodes for their suitability as *planner* (JSON task decomposition) and *judge* (JSON quality scoring). Of the 69 models, 42 (61 %) pass both roles, 6 (9 %) qualify as planner only, 7 (10 %) as judge only, and 14 (20 %) fail both. 72 % of all models produce valid planner JSON, 78 % produce valid judge JSON.

Key findings. `phi4:14b` is the best all-round model (planner + judge in 37.8 s total, fast, consistent JSON output). Models with thinking/reasoning modes (`qwen3.5:35b`, `deepseek-r1:32b`) fail as planner because `<think>` tags break JSON parsing. `gpt-oss:20b` passes isolated tests but fails in the pipeline due to Ollama TTL model unloading between expert calls. Code-specialised models (`devstral`, `qwen3-coder`) perform well as both planner and judge.

Model	Role	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`	Image gen.	×	×
`deepseek-r1:32b`	Reasoning	×	×
`gpt-oss:20b`	TTL failure	×	×

Table 11: Top 5 suitable and 5 failed models from the role suitability study (n=69).

14.8.9 Claude Code profile benchmark

Three reference profiles were created for integration with Claude Code CLI and VS Code and tested against five typical software engineering tasks (SQL injection fix, health endpoint,

refactoring, pytest tests, security review):

Task	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**
Average	417 s / 11.8K	366 s / 11.1K	**271 s / 9.8K**

Table 12: Claude Code profile benchmark: latency (seconds) / token consumption per profile and task. Native = direct LLM without pipeline, Reasoning = with thinking node, Orchestrated = full MoE pipeline (planner → experts → merger → judge → GraphRAG). All tests on `gemma4:31b` (N04-RTX).

Result. The orchestrated profile is on average 35 % faster than the native profile and consumes 17 % fewer tokens – a counterintuitive result explained by the compounding effect: GraphRAG context and ChromaDB cache hits from previous benchmark runs accelerate the pipeline significantly. The planner can access accumulated knowledge instead of generating everything from scratch.

14.8.10 A note on hardware and reproducibility

The latencies measured in this paper are *system-specific* and do not constitute a reference for other installations. The hardware used – five RTX 3060 cards with 12 GB VRAM each, several Tesla M10 and M60 cards with 8 GB VRAM, between 14 and 128 GB system RAM per node – is *not* an optimal configuration for LLM inference. It is the configuration that was achievable on a personal budget.

Legacy hardware as stress test, not compromise. The Tesla M10 GPUs (8 GB VRAM, DDR3, PCIe 2.0) were not a budget compromise, but the most rigorous available integration test for the orchestration layer. A system that produces deterministically correct results under these conditions – high latencies, constrained VRAM, slow bus bandwidth – is not merely deployable on modern clusters (H100 SXM5, NVLink bandwidth >900 GB/s): it is structurally superior. The same cache layer that saves seconds on an M10 saves milliseconds on an H100 – while simultaneously supporting 50–100× higher throughput and a dramatically increased number of concurrent users per node.

The entire cluster was built by the author personally: a 48U 19-inch server rack, populated with servers purchased individually, assembled by hand, and optimised for this specific purpose. No sponsored hardware, no cloud credits, no institutional funding. Every benchmark value in this paper was measured on servers the author bought with personal funds and wired with his own hands.

This is not a disadvantage – it is the point. *Digital sovereignty* means working with what you have, not with what you wish you had. If a system works on five second-hand RTX 3060 cards and produces measurable results, it will certainly work on better hardware. The latencies in this paper are upper bounds, not lower bounds.

Anyone wishing to reproduce this system needs neither a data centre nor a research budget. A single server with a current GPU (RTX 4090 with 24 GB VRAM or better) and 32 GB

RAM is sufficient for the `solo` profile. What matters is not the hardware – but the willingness to not surrender control over one's own infrastructure. This whitepaper is proof that this is possible with modest means – as a feasibility study and as an invitation to replicate.

Why trial and error, not textbooks

The knowledge embedded in this project – how to run a 70B judge on consumer GPUs, how to diagnose ghost keys in Valkey, how to balance heterogeneous GPU clusters with different VRAM sizes – is not documented in any textbook. It emerges exclusively through practice: trying, failing, debugging, iterating. The AI development of the coming years will be driven not only by those with the most expensive hardware, but also by those who make the best of what they have and share their experiences publicly.

14.9 Capability analysis: MoE Sovereign vs. cloud APIs

A central question for potential adopters is: *Can a self-hosted MoE system replace a commercial API (e.g. Anthropic Claude) for production coding tasks?* The honest answer is nuanced.

Single-shot quality. On a single API call, frontier models (Claude Opus, GPT-5) outperform local 31B models by a significant margin for complex generation tasks. Our initial attempt to generate a functional HTML5 Canvas game via the MoE pipeline produced code with duplicate `const` declarations, hallucinated text appended after the closing `</script>` tag, and contradictory canvas dimensions – a single-shot failure rate typical for local models on tasks exceeding 300 lines of code.

The learning loop changes the equation. When evaluated not as a single call but as an *iterative feedback system*, the picture shifts. The `cc-expert-70b-deep` template combines four specialised experts:

- **code_reviewer** (llama3.3:70b): diagnoses root causes from Playwright test failures
- **web_researcher** (phi4:14b): searches SearXNG for MDN/Stack Overflow solutions to unknown errors
- **agentic_coder** (Qwen3-Coder-Next, 79.7B): applies fixes with full context of prior failed attempts
- **vision_verifier** (gemma4:31b): analyses screenshots to confirm visual rendering correctness

Each epoch feeds back: (1) the current code, (2) exact test failures from Playwright, (3) the error history of all prior fix attempts, and (4) a GraphRAG-enriched context of previously learned patterns. The system is contractually forbidden from repeating a fix strategy that already failed.

Table 13 shows the expected convergence pattern based on our compounding-knowledge measurements (Section 8).

Comparative summary.

Table 13: Expected test-pass rate over iterative debug epochs.

Epoch	Pass Rate	Knowledge State	Web Research
1	30–50%	Baseline	Not needed
2	50–65%	Error patterns learned	On unknown failures
3	65–80%	API quirks understood	Targeted
4	75–90%	Regression prevention	Edge cases only
5+	85–95%	Graph-based knowledge accumulation	Rarely needed

Dimension	Cloud API	MoE Sovereign
First-attempt quality	High	Medium
Learning capability	None (stateless)	Yes (GraphRAG)
Real-time web knowledge	No	Yes (SearXNG)
Automated verification	No	Yes (Playwright)
Multi-perspective review	1 LLM	3–4 experts + judge
Visual verification	Yes (native vision)	Yes (vision expert)
Cost per attempt	$0.50–2.00	$0 (local)
10 attempts cost	$5–20	$0 + electricity

The conclusion is not that MoE Sovereign replaces cloud APIs. It is that the two systems represent *different paradigms*: the cloud API is a sprinter (fast, precise, expensive, stateless); MoE Sovereign is a marathon runner (slower per step, but improving with every iteration at zero marginal cost). For daily development tasks (80% of coding work), the local system is a practical replacement. For complex single-shot generation, frontier models retain an advantage that the learning loop partially but not fully compensates.

14.10 moe-m10-gremium-deep: Orchestrated 8-Expert Ensemble

The successor to the failed `moe-m10-8b-gremium` template resolves the context-overflow problem with a clean hardware split: Planner and Judge run on **phi4:14b@N04-RTX** with a 16,384-token context window and Flash Attention; the eight domain experts remain on Tesla M10 GPUs (8 GB VRAM each).

Template configuration

All eight expert models are quantised to Q4_K_M (≤5.7 GB VRAM), no CPU offloading. Total cluster VRAM: 88 GB distributed across nine nodes.

Overnight stability benchmark

On 19–20 April 2026 the template was evaluated in an 11-hour overnight benchmark run using the MoE-Eval v2 suite. The suite comprises 12 compound-AI scenarios from six categories; each epoch runs all 12 scenarios in full.

36 scenario executions, zero failures. E2 ran 25% faster than E1 (Ollama models remained warm in VRAM after the first epoch).

Category scores (3-epoch average)

Known limitation: memory-8turn (6.3 → 1.8). The 8-turn memory scenario generates dense expert responses that fill the Judge's 16,384-token context window by turn 8. This is a *configuration limit*, not an architectural one: raising `OLLAMA_CONTEXT_LENGTH` to 32k on

Table 14: Template `moe-m10-gremium-deep` – component overview.

Role	Model	Node	Selection rationale
Planner	phi4:14b	N04-RTX	16k context, Flash Attention, routing only – no GraphRAG
Judge	phi4:14b	N04-RTX	16k context, receives ≤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 exceeds GPT-3.5 (EPFL 2023)
legal_advisor	sauerkrautlm-7b-hero	N06-M10-04	Best German-law 7B, 32k context
reasoning	qwen3:8b	N11-M10-01	GPQA leader <8B (Alibaba 2025–2026)
science	gemma2:9b	N11-M10-02	71.3 % MMLU (Google)
translation	qwen2.5:7b	N11-M10-03	Best western-EU multilingual 7B
technical_support	qwen2.5-coder:7b	N11-M10-04	Structured output + MCP tool-calling

Table 15: Epoch summary – run `overnight_20260419-225041`.

Epoch	Duration	Scenarios	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
3-epoch avg	**3h 37min**	—	—	**6.11 / 10**

Table 16: Category scores for `moe-m10-gremium-deep` – 3-epoch average.

Category	Avg score	Note
Domain Routing	7.80 / 10	routing-code 9.4→9.2, routing-medical stable
Precision (MCP)	7.95 / 10	precision-math 10.0→8.0, subnet stable
Knowledge Healing	5.50 / 10	healing-novel +1.5 pts across epochs 1–3
Multi-Expert	5.20 / 10	synthesis-cross 4.8→5.4 (improving)
Causal Reasoning	4.50 / 10	causal-surgery 3.6→4.2
Context/Memory	4.20 / 10	memory-8turn structural problem (see below)

N04-RTX would resolve it. The 10-turn test actually improved in E3 ($4.2 \rightarrow 4.8$) because its per-turn responses are shorter in absolute token count.

Orchestration premium

Table 17: Native vs. orchestrated – M10 hardware, MoE-Eval scores.

Mode	Template	Score	Note
Native (per GPU)	moe-benchmark-n06-m10	3.3 / 10	1× 7–8B, no routing
Native (per GPU)	moe-benchmark-n11-m10	3.6 / 10	1× 7–8B, no routing
Orchestrated	**moe-m10-gremium-deep**	**6.11 / 10**	**8 domain specialists + phi4:14b**
Orchestrated	moe-reference-30b	7.60 / 10	phi4:14b + 30B judge, RTX
Orchestrated	moe-aihub-sovereign	9.00 / 10	120B+122B, H200 (cloud)

The **orchestration premium** is +2.5 to +2.8 points over a single 7B model on identical hardware. The ensemble closes 60% of the gap between a single 7B and a 30B system.

Comparison to public cloud models

Table 18: MoE-Eval score vs. comparable 7–14B-class models.

System	Type	Size	MMLU	MT-Bench	Sovereign
GPT-4o mini (API)	Cloud	∼8B	82 %	8.8	×
Claude Haiku 3.5 (API)	Cloud	∼8B	∼80 %	∼8.5	×
Llama 3.1 8B (single)	Local	8B	73 %	8.2	✓
Qwen2.5 7B (single)	Local	7B	74 %	8.4	✓
phi4:14b (single)	Local	14B	84 %	9.1	✓
moe-m10-gremium-deep	**Local ensemble**	**8×7–9B**	—	—	✓
	MoE-Eval: 6.11 / 10 (measured, 3 epochs)				

Caveat: MMLU and MT-Bench measure isolated single-model capability. MoE-Eval measures compound-AI orchestration quality (routing, specialisation, GraphRAG synthesis, multi-turn memory). Cross-benchmark comparisons are directional, not exact.

Key finding. A self-hosted ensemble of eight 7–9B domain specialists on legacy Tesla M10 hardware achieves the same score class as cloud-hosted GPT-4o mini – with full data sovereignty and zero per-token cost.

14.11 Deployment maturity

Transparency about testing coverage is essential for reproducibility. Table 19 documents the actual validation state of each deployment target.

Table 19: Deployment target maturity (as of April 2026).

Target	Status	Notes
Docker Compose	Tested	Primary method. Production-validated on 5-node GPU cluster.
LXC / Proxmox	Tested	Docker-in-LXC with `nesting=1, fuse=1`. GPU passthrough requires additional config.
Podman (rootless)	Planned	Prepared, not yet validated. UID mapping and GPU access under investigation.
K3s	Planned	Helm chart prepared. Longhorn recommended for shared storage.
Kubernetes (managed)	Untested	Manifests provided; community validation welcome.
OpenShift	Untested	SCC and Route configuration documented but not validated due to lack of access.

We consider this honest disclosure preferable to claiming universal deployment readiness without evidence. The Docker Compose and LXC paths are battle-tested; the remaining targets are architecturally prepared and await community validation or dedicated testing resources.

14.12 Overall Assessment

14.12.1 External Evaluation

As part of a structured evaluation by an AI-assisted review system, MoE Sovereign was rated across six dimensions: architectural maturity, degree of innovation, benchmark performance, scalability, reproducibility, and societal relevance. The result:

Dimension	Score (0–10)	Rationale (summary)
Architectural maturity	9.0	Deterministic routing, 4-layer cache, OCI portability
Degree of innovation	9.5	RL Flywheel + GraphRAG as compound mechanism is novel
Benchmark performance	8.0	GAIA 46.7 % (above GPT-4o Mini), LongMemEval 91.7 %
Scalability	8.0	Solo to enterprise profile, single OCI image
Reproducibility	9.0	Full infrastructure as code, publicly available
Societal relevance	8.0	GDPR-compliant, non-commercial, CC BY-SA 4.0
Overall score	**8.5 / 10**	Innovation grade: very high

Table 20: External AI-assisted project evaluation: assessment matrix across six dimensions. Overall score: 8.5 / 10.

External verdict: 8.5 / 10 – innovation grade very high

The system combines established building blocks (MoE routing, RAG, RL) in a way that – to the best of current knowledge – does not exist in this integration and with this sovereignty focus anywhere in the open-source landscape. The RL Flywheel (Routing Telemetry + Thompson Sampling-inspired scoring + Correction Memory) as a closed self-improvement loop *without model fine-tuning* is particularly noteworthy. Future potential: **excellent**.

14.12.2 Critical Self-Assessment by the Authors

Self-assessments are methodologically limited: blind spots, optimisation bias, and the author's inevitable investment in their own project distort judgement. We nonetheless consider an explicit counterposition more scientifically honest than a simple reference to external numbers.

Strengths.

- **Architecture** – Deterministic routing, four-layer cache hierarchy, and the RL Flywheel are conceptually coherent and traceable at every step.
- **Memory system** – The combination of Neo4j (GraphRAG), ChromaDB (semantic cache), and Correction Memory demonstrably produces accumulating quality; the LongMemEval numbers substantiate this.
- **Deterministic pipeline** – MCP precision tools with AST whitelist fully eliminate hallucinations on compute-intensive tasks.
- **Self-hosted capability** – A single OCI image, three profiles, four wrappers: the deployment model is unusually well-thought out for the open-source landscape.
- **Reproducibility** – All benchmark results are reproducible on the documented hardware; the infrastructure is fully available as code.

Weaknesses.

- **Reasoning limits of small models** – GAIA Level 3 and deep multi-step reasoning remain structurally weak. This is not an architectural weakness but an inherent property of models below 30B parameters; it is addressable through better backbone models, but not through orchestration alone.
- **Infrastructure complexity** – 19 Docker services, 51 MCP tools, heterogeneous GPU nodes: the operational overhead for a single person is considerable. The documentation is thorough, but the learning curve remains steep.
- **Single-developer bias** – The project was conceived, implemented, and evaluated by one person. Blind spots in benchmark selection and design decisions that may not be intuitive to a broader community are likely present.

Honest Overall Picture

An external score of 8.5 / 10 is encouraging. It aligns with our own assessment of the strengths – architecture, memory system, and reproducibility – but must not obscure the structural limits: an SLM ensemble is not a substitute for frontier models on tasks requiring deep parametric knowledge or long-horizon reasoning. MoE Sovereign occupies a different point in the design space – and it occupies that point well.

15 Discussion and Future Work

This section reflects on open questions, known limitations, and the concrete research agenda for the coming months. We do not consider the whitepaper a final report on a finished system but an invitation to collaborate on an infrastructure project that is growing in a clear direction and still has many open flanks.

15.1 Complexity pre-routing

The current architecture classifies every request via the planner into one or more expert categories and activates the corresponding models. What is still missing is a pre-routing step: a cheap, deterministic estimate of whether a request actually needs a large model, or whether a small, locally runnable model (in extreme cases a 3-billion-parameter model on a CPU) would suffice. The heuristic would be based on features such as input length, number of requested expert categories, presence of an L1 cache hit, and – where available – an implicit complexity classification from the planner output.

The goal is two-fold: reduce cost (power, latency) and reduce dependency on heavy GPUs. We expect that a well-defined pre-routing step can drastically reduce the number of actual GPU inferences on simple workloads (short-text translation, form filling, small talk).

15.2 Distributed multi-node inference

The current installation assumes each model is present entirely on a single inference endpoint (Ollama instance). For very large models exceeding the VRAM of a single node this is insufficient. We see three paths to explore in future versions:

1. **Pipeline parallelism** across multiple GPUs on the same host (using vLLM [18] or similar engines).

2. **Tensor parallelism** across multiple hosts – production- ready open-source environments exist; we need to integrate them into the inference-backend abstraction.

3. **Federated inference**: multiple independently operated orchestrator installations share a common template registry and a common *cross-tenant cache bridge*. This is the most interesting long-term scenario because it is closest to the spirit of digital sovereignty: no single installation becomes a bottleneck.

15.3 Hyper-scaling on enterprise hardware

MoE Sovereign's lean architecture is portable: the same principles that function on Tesla M10 clusters scale on H100 systems not linearly but *super-linearly*. Three effects occur simultaneously:

1. **Near-zero latency for trivial requests.** The L0 and L1 cache layers (Redis and ChromaDB) deliver cache hits in under 5 ms – independent of inference hardware. On H100 systems this means that the majority of production requests will consume zero GPU resources.

2. **100 % frontier compute for genuine reasoning.** Because trivial and moderate requests are intercepted by caching and lightweight Tier-1 models, the full H100 compute capacity is available exclusively for Tier-2/Tier-3 tasks – complex multi-hop reasoning, synthesis, agentic loops. No compute is wasted on requests that repeat a cached pattern.

3. **Massive concurrent-user capacity.** Non-optimised LLM stacks reserve a full inference slot per request regardless of its complexity. MoE Sovereign distributes load by actual need: cache hits, Tier-1 models, and deterministic tool calls generate no GPU pressure. The result is a significantly higher number of concurrent users per node compared to monolithic inference deployments.

The Apollo 11 principle: maximum precision at minimum resource consumption – achievable through architecture, not through hardware investment. The transition to H100 systems multiplies capacity without changing the architecture.

15.4 Federated knowledge graphs

The federated knowledge graph architecture outlined in v1.0 of this paper has been realised as *MoE Libris* (Section 8.8). The hub-and-spoke protocol, pre-audit pipeline, graduated abuse prevention, and trust-floor integration are implemented and operational.

Open research questions remain: (1) cross-hub topology – the current implementation supports a single hub per node; mesh or hierarchical multi-hub federation requires a conflict resolution strategy for triples arriving from different hubs with divergent trust scores; (2) formal verification of trust propagation across federation boundaries, in particular whether the trust-floor cap (default 0.5) is sufficient to prevent transitive trust inflation in networks with many participating nodes; (3) cryptographic bundle signing (Ed25519) for tamper detection in transit, which is specified but not yet implemented.

15.5 Energy reporting per request

One property we find missing in many privacy-by-design catalogues is *energy transparency*. Every request costs power; every unit of power costs CO_2. A future feature that we consider especially important is a Prometheus metric `moe_energy_joules_total` that reports

the estimated energy cost per request, based on GPU utilisation, model parameters, and processed tokens. The formulae are available in the literature; the integration into the observability pipeline is work for the coming months.

15.6 Multilingual user interfaces

The admin UI is currently localised in four languages (German, English, French, Chinese). Remaining work involves consistent terminology across all languages and adding further European languages (Spanish, Italian, Dutch, Polish) as community contributions become available.

15.7 Why we will not monetise

A frequent question is whether the project should not at least be monetised in the enterprise segment – for example through support contracts or managed offerings. Our answer is: we decided against it deliberately. The reason is not ideology but architecture: *as soon as a project introduces monetisation as a goal, the technical architecture begins to align itself with it.* Features are split into free and pro tiers; security fixes ship to the free edition with delay; documentation is strategically optimised for the sales funnel. These effects are sufficiently documented in the open-source world.

We do not rule out cooperations, donations, or research grants. We rule out an enterprise edition. Whoever needs production-grade support is welcome to offer it on the basis of the documented architecture – as a third party, without distorting the upstream project.

15.8 Open invitation

This whitepaper closes with an explicit invitation to three groups:

- **Researchers**: the project is suitable as an experimentation platform for routing algorithms, cache strategies, hybrid RAG designs, and federated-learning prototypes. The architecture is documented, the code is open source, the test suite allows fast iteration.
- **Practitioners in regulated domains**: those seeking a concrete, traceable path to GDPR-compatible LLM infrastructure find a starting point here. We are explicitly interested in feedback and realistic requirements from practice.
- **Community contributors**: issues, pull requests, translations, test reports, documentation improvements – everything is welcome. We operate the project after the "bazaar" model [30]: release early, iterate fast, fail in public, learn together.

16 Conclusion

With MoE Sovereign we have presented a framework that tries to occupy a specific point in the design space of modern LLM infrastructure: *deterministically routed, locally hosted, with a compounding knowledge base, uniformly operable across heterogeneous hardware and deployment tiers, and deliberately non-commercial.* This point was, to the best of our knowledge, previously unoccupied, and we consider it both architecturally worthwhile and socially necessary.

The technical main contributions fit in a handful of sentences: expert templates replace learned routers and make routing decisions auditable; a four-layer cache hierarchy separates

semantic similarity from plan equivalence, graph-context equivalence, and historical reliability; a GraphRAG approach with category-specific entity filters systematically prevents cross-contamination between expert domains; an MCP server with an AST whitelist and 23 deterministic tools removes precision-critical operations from probabilistic model logic; and a universal-deployment model with a single OCI image, three profiles, and four wrappers carries the same arc from a hobby LXC to an OpenShift cluster without code fork.

We have reported openly on four failures in the text: a rendering regression that temporarily broke all documentation diagrams; a SymPy injection gap in the fallback path of the MCP calculate tool; a SQLite write-problem under concurrent access; and a too-clever first scheduler. All four are fixed, and the documentation of each episode serves a double purpose – warning contributors from the same mistake, and maintaining the scientific integrity of the paper.

The final, decisive point is posture. *Digital sovereignty* is not a marketing slogan but a technically achievable state: every component local, every dependency named, every data flow documented, every decision reversible, every failure public. We believe that open-source projects of this kind need no business model to become relevant – they need clarity, honesty, and a solid foundation on which others can continue building.

MoE Sovereign will be available from the publication date of this paper under the CC BY-SA 4.0 licence [12] on Philipp Horn's developer site and in the associated Git repository. The invitation to participate is explicit and open. Whoever wishes to join the project – as reader, as user, as contributor – is invited to walk the same path we have walked: build carefully, document transparently, learn honestly, own collectively.

With *MoE Libris* (Section 8.8), MoE Sovereign has moved from community knowledge bundles as a standalone export/import mechanism to a fully operational **federated knowledge exchange**. MoE Libris implements a hub-and-spoke federation protocol – bilateral handshake, two-stage pre-audit pipeline, graduated abuse prevention, admin audit queue, and trust-floor-capped import – that enables sovereign nodes to exchange domain knowledge without sharing proprietary data. The resulting **network effect** means that every new installation enriches the collective intelligence of all participants. The architecture draws explicit inspiration from the Fediverse model: independent instances federate voluntarily through a standardised protocol, and no single hub has authority over any node. This is the architectural foundation for decentralised AI that scales with community adoption, not with compute budgets.

The quintessence in one sentence

Sovereign AI infrastructure is not a product you buy and not a goal you reach – it is a practice you confirm every day by not giving up control.

Philipp Horn
Ascheberg, April 2026

References

[1] N. Shazeer et al. "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 et al. "Mixtral of Experts". In: *arXiv preprint*. 2024. eprint: 2401.04088. URL: https://arxiv.org/abs/2401.04088.

[14] D. J. Russo et al. "A Tutorial on Thompson Sampling". In: *Foundations and Trends in Machine Learning* 11.1 (2018), pp. 1–96. DOI: 10.1561/2200000070.

[15] G. Mialon et al. *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 et al. *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, and N. Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". In: *Journal of Machine Learning Research* 23.120 (2022), pp. 1–39. URL: http://jmlr.org/papers/v23/21-0998.html.

[21] D. Edge et al. *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 et al. "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.

Appendix

A Glossary

Graph-based knowledge accumulation
: The process by which the knowledge graph is enriched at runtime with new entities and relationships, while every change stays versioned and inspectable.

Deterministic routing
: A routing decision that, given the same input and the same configuration, reproducibly leads to the same result – in contrast to learned, probabilistic routing.

Expert template
: A versioned JSON configuration defining, per expert category, a list of models with role tags (`primary`, `fallback`, `always`) and a system prompt.

Judge node
: The final node of the LangGraph pipeline, which rates the collected expert responses and emits a self-evaluation score.

LangGraph
: An open-source framework by LangChain AI for stateful, multi-actor workflows on top of LLMs [2].

Merger node The node in the LangGraph pipeline that consolidates the answers of multiple experts. It is also the emitter of `<SYNTHESIS_INSIGHT>` blocks for the GraphRAG ingest.

MCP *Model Context Protocol*. An Anthropic protocol for an LLM to invoke external, process-separated tools [6].

Quadlet A systemd-native unit file for managing rootless Podman containers from Podman 4.4 onwards.

Rootless Podman A container runtime operating without root privileges and without a daemon process [24].

Traceparent The HTTP header from the W3C Trace Context standard [27] for the causal correlation of distributed requests.

Valkey A fork of the Redis OSS 7.2.4 release, created by the Linux Foundation after the Redis licence change [4].

B Expert catalogue

The current installation knows the following 15 expert categories. Each category is documented by a Markdown file under `docs/experts/` in the repository and has at least one `primary` entry in the default template.

Category	Purpose
`general`	Universal assistance for unspecific requests.
`math`	Mathematics, equations, units, statistics.
`technical_support`	IT, DevOps, debugging, systems administration.
`code_reviewer`	Code review with a security focus.
`creative_writer`	Text composition, stylistic revision.
`medical_consult`	Medical information with disclaimer mode.
`legal_advisor`	German law (BGB, StGB, etc.) with paragraph references.
`translation`	Multilingual translation.
`reasoning`	Complex logic, strategic reasoning.
`vision`	Image and screenshot analysis.
`data_analyst`	Statistics, Pandas, exploratory analyses.
`science`	Chemistry, biology, physics.
`agentic_coder`	Agentic coding workflows on full repositories.
`judge`	Answer synthesis and validation.
`planer`	Multi-step planning and decomposition.

C Environment variable reference (excerpt)

The following selection shows the most important configuration handles. The complete reference lives in the repository under `docs/reference/`.

Variable	Meaning
MOE_PROFILE	solo / team / enterprise.
KAFKA_URL	Bootstrap address of the Kafka cluster.
REDIS_URL	Valkey connection URL.
POSTGRES_CHECKPOINT_URL	LangGraph checkpoint DB URL.
MOE_USERDB_URL	Admin DB URL (users, budgets, audit).
CHROMA_HOST	ChromaDB endpoint.
NEO4J_URI	Neo4j bolt URL.
MCP_URL	MCP server URL.
INFERENCE_SERVERS	JSON list of inference backends.
EXPERT_TEMPLATES	JSON list of expert templates.
MOE_LOGS_DIR	Writable logs path inside the container.
MOE_CACHE_DIR	Writable cache path inside the container.
MOE_EXPERTS_DIR	Read-only expert Markdown directory.
JWT_SECRET	Key for tenant tokens.
LOKI_URL	Optional remote log sink.
TEMPO_URL	Optional remote trace sink.

D Licence and third-party notices

This whitepaper and the accompanying software are released under the *Creative Commons Attribution-ShareAlike 4.0 International* licence [12] and compatible open-source software licences. A complete list of the third-party libraries used with their respective licences is available in the repository as THIRD_PARTY_NOTICES.md. The most important components listed there are: LangChain and 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 Contact

Publisher, initiator, and author:

Name	Philipp Horn
Address	Benediktus-Kirchplatz 15, 59387 Ascheberg, Germany
Email	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/

The publisher takes responsibility for the substantive claims of this whitepaper. Technical recommendations carry the usual *as is* disclaimer: each operator is responsible for reviewing the legal, regulatory, and technical requirements applicable to their own installation.