MoE Sovereign

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

Whitepaper — Version 2.8, August 2026

Source code: **Apache 2.0** | Whitepaper: **CC BY-SA 4.0**

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

Abstract **1**

1 Introduction **2**

2 Project Genesis and Research Motivation **3**
2.1 Starting Motivation: Cloud Independence at Home 3
2.2 Extended Project Goal: Structural Token Reduction 4
2.3 Point of Departure: Legacy Hardware as Research Object 4
2.4 The Research Question: Architecture as Compensation Mechanism 5
2.5 Adaptive Feedback Loop: Continuous Learning Without Model Fine-Tuning . 6
2.6 Positioning: SLM Ensemble as Alternative to Frontier Models 8

3 Motivation: Digital Sovereignty as a Technical State **8**
3.1 Documented failure modes of commercial LLM services 9
3.2 The European regulatory framework 9
3.3 Sovereignty at small scale is still sovereignty 9

4 Related Work **10**
4.1 Closed commercial APIs 10
4.2 Single-model launchers 10
4.3 Generic RAG libraries 11
4.4 Research-oriented stacks 11
4.5 Multi-Model Orchestration in the literature 11
4.6 Microsoft GraphRAG and related work 12
4.7 Enterprise AI platforms 12
4.8 Comparative feature matrix 13
4.9 API proxies are not orchestrators 13
4.10 Summary of the delineation 13

5 System Architecture **14**
5.1 Services at a glance 14
5.2 LangGraph as the execution model 14
5.3 Multi-protocol API gateway 14
5.4 Data plane vs. control plane 15
5.5 Configuration sources and versioning 17
5.6 Federation layer: MoE Libris 17
5.7 Compliance extension: MoE Codex 17
5.8 Runtime feature control: Starfleet toggles and Mission Context 20
5.9 Scale and performance envelope 20
5.10 Constraint-driven engineering: the Apollo 11 principle 21

6 Template-Based Routing **21**
6.1 The trouble with learned routers 21
6.2 Template-based routing 22
6.3 Warm/cold scheduling across heterogeneous GPUs 23
6.4 The expert-performance score as a fourth cache layer 24
6.5 VRAM-aware node selection 24

7 Multi-Layer Cache Hierarchy **25**

7.1 L1: Semantic cache (ChromaDB) . . . 25
7.2 L2: Plan cache (Valkey) . . . 25
7.3 L3: Graph-context cache (Valkey) . . . 25
7.4 L4: Expert performance scores (Valkey) . . . 25
7.5 Combination logic: fall-through . . . 26
7.6 Session history compression cache (Valkey) . . . 26
7.7 Augmented Tool Path: cache and GraphRAG extension for agentic clients . . . 26

8 GraphRAG and Graph-Based Knowledge Accumulation 27
8.1 Why a graph, not just a vector store . . . 28
8.2 Category-specific entity-type filters . . . 28
8.3 The base ontology . . . 28
8.4 Persistent Graph-State Tracking: the SYNTHESIS_INSIGHT mechanism . . . 29
8.5 Contradiction detection . . . 29
8.6 Ontology gaps as a signal . . . 29
8.7 Community knowledge bundles . . . 30
8.8 MoE Libris: Federated Knowledge Exchange . . . 30
8.9 Corrective RAG Gate . . . 32
8.10 Context-Augmented Generation for Compliance Domains . . . 33
8.11 Episodic Memory . . . 33
8.12 Attribution of Applied Scientific Contributions . . . 34

9 MCP Precision Tools 36
9.1 Why a dedicated MCP server? . . . 36
9.2 The AST whitelist of the `calculate` tool . . . 36
9.3 The full tool roster . . . 36
9.4 Why exactly these tools? . . . 37

10 Self-Correction Loop 38
10.1 Self-evaluation in the judge node . . . 38
10.2 User feedback . . . 38
10.3 Laplace-smoothed confidence update . . . 38
10.4 Tier-2 gating in practice . . . 39
10.5 Ontology-gap detection . . . 39
10.6 The compounding effect . . . 39
10.7 Agentic re-planning loop . . . 39
10.8 Paraconsistent conflict resolution . . . 40

11 Unified OCI Artifact 41
11.1 Single-image deployment . . . 41
11.2 Three profiles . . . 43
11.3 Four deployment wrappers . . . 43
11.4 LXC: rootless Podman as the bridge . . . 44
11.5 The invariants . . . 44

12 Observability and Tracing 48
12.1 Prometheus metrics . . . 48
12.2 Grafana dashboards . . . 48
12.3 Loki and Alloy as a universal log collector . . . 49
12.4 Trace propagation: W3C traceparent . . . 50
12.5 Live monitoring: active requests and kill mechanism . . . 50

13 Security and Privacy **54**
13.1 Multi-layer isolation at the container build . . . 54
13.2 JWT-based multi-tenant authentication . . . 54
13.3 Rate limiting . . . 54
13.4 Data flows and retention . . . 54
13.5 Privacy-by-design checklist . . . 55
13.6 Security hardening 2026-04-06: a concrete milestone . . . 55
13.7 Security hardening 2026-04-25: production-grade audit . . . 56

14 Evaluation and Lessons Learned **56**
14.1 Episode 1: the 70B judge problem – system RAM vs. VRAM . . . 57
14.2 Episode 2: the SymPy injection gap in the MCP server . . . 57
14.3 Episode 3: SQLite → PostgreSQL . . . 58
14.4 Episode 4: ghost keys in the live monitoring . . . 58
14.5 Episode 5: the first scheduler iteration . . . 59
14.6 Episode 6: specification gaming – frontier model substitution in a sovereign benchmark . . . 59
14.7 Episode 7: DeepSpeed ZeRO-3 and multimodal models on AMD MI250X . . . 60
14.8 Episode 8: IMoE router – ChromaDB cache never hits (query/document mismatch) . . . 62
14.9 Episode 9: PEFT version mismatch during LoRA merge on LUMI-G . . . 63
14.10 Episode 10: `-no-mtp` flag for GGUF conversion of Qwen3-MoE . . . 64
14.11 Episode 11: The ineffective fine-tune – sequence-length truncation and prompt-taxonomy drift in the planner SFT . . . 64
14.12 Episode 12: Why fine-tuning on AMD MI250X remains punishingly slow despite data-center hardware – and why local consumer GPUs are not an alternative . . . 66
14.13 The test suite . . . 68
14.14 Baseline benchmarks: three reference templates . . . 69
14.14.1 Reference prompt . . . 70
14.14.2 Measurement methodology . . . 71
14.14.3 Baseline results . . . 71
14.15 External Baseline Evaluations . . . 72
14.15.1 GAIA benchmark 2026: evaluation against an external accuracy baseline 72
14.15.2 Judge experiment: qwen-3.5-122b-sovereign as planner+judge . . . 73
14.15.3 MoE-Eval: Internal System Health and Regression Suite . . . 75
14.15.4 Before/after: the compounding effect between runs . . . 76
14.15.5 Dense-graph run: measurement after knowledge accumulation . . . 77
14.15.6 Positioning against external benchmarks . . . 79
14.15.7 Enterprise architecture features . . . 82
14.15.8 Adversarial MCP tool security testing . . . 83
14.15.9 LLM role suitability: planner and judge . . . 83
14.15.10 Claude Code profile benchmark . . . 83
14.15.11 A note on hardware and reproducibility . . . 84
14.15.12 Hardware Energy Efficiency: Thermal Management via Power Capping 85
14.16 Lessons on Multi-GPU Inference, Synchronization Deadlocks, and 256k Context Allocation . . . 87
14.17 Capability analysis: MoE Sovereign vs. cloud APIs . . . 88
14.18 moe-m10-gremium-deep: Orchestrated 8-Expert Ensemble . . . 90
14.19 Deployment maturity . . . 92

14.20 Overall Assessment . . . 92
14.20.1 External Evaluation . . . 92
14.20.2 Critical Self-Assessment by the Authors . . . 92
14.21 Component Evidence Summary . . . 93
14.22 GraphRAG vs. Zero-Shot: 10-Query Direct Comparison . . . 95
14.23 Scientific Compound AI Architecture Benchmark – Pre-Distillation Baseline (August 2026) . . . 96
14.24 Technical Support for Selected ISO/IEC 27001 Controls . . . 98
14.25 Operational Safeguards for Fine-Tuned SLM Components . . . 99
14.25.1 Format-Drift Validator for Paraconsistent Output . . . 99
14.25.2 Semantic Entity Linking for Knowledge-Graph Stability . . . 99

15 Discussion and Future Work **100**
15.1 Known limitations . . . 100
15.2 Complexity pre-routing . . . 101
15.3 Distributed multi-node inference . . . 101
15.4 Expected Scaling Behaviour on Enterprise Hardware . . . 101
15.5 Sponsored SLM Distillation Programme (EuroHPC LUMI-G) . . . 102
15.5.1 The Frontier Distillation Campaign (August 2026): 4B Student, 8 Domain Experts, and 27B Judge . . . 102
15.6 Federated knowledge graphs . . . 104
15.7 Compliance extension: MoE Codex . . . 104
15.8 SLM scaling limits under GraphRAG (Wikimedia benchmarks) . . . 104
15.9 Energy reporting per request . . . 105
15.10 Multilingual user interfaces . . . 105
15.11 Why we will not monetise . . . 105
15.12 Open invitation . . . 105

16 Conclusion **106**

References **109**

Appendix **112**
A Glossary . . . 112
B Expert catalogue . . . 112
C Environment variable reference (excerpt) . . . 113
D Licence and third-party notices . . . 113
E Contact . . . 114

List of Figures

1 Adaptive Feedback Loop: Routing Telemetry, Laplace-Smoothed Bayesian Expert Scoring, and Correction Memory form a self-reinforcing improvement cycle with no model fine-tuning required. 7
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. 15
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. 16
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. 23
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. 25
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). 42
7 The tier decision aid: which profile and which wrapper to recommend for which deployment target. 43
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. 45
9 Comparison of the three profiles `solo`, `team`, and `enterprise` in Helm-values form. 46
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. 46
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. 47
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. 49
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. 50
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. 51
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. 51

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. 52
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. 53

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 central thesis this paper evaluates is a shift in the *capability equation*:

$$\underbrace{\text{Capability} \approx \text{Model}}_{\text{classical paradigm}} \xrightarrow{\text{sovereign orchestration}} \text{Capability} \approx \text{Model+Architecture+Memory+Tools+Feedback}$$

where system-level compensations replace raw parameter count, enabling domain-specialist small language models under autonomous orchestration to approach frontier-model quality on bounded tasks – with internal evidence indicating substantial capability gains, but without controlled external validation against frontier models – without sacrificing data sovereignty.
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 exposes three API protocols simultaneously: the OpenAI chat-completions interface (`/v1/chat/completions`), the Anthropic Messages API (`/v1/messages`), and the OpenAI Responses API (`/v1/responses`) – enabling Claude Code, Claude Desktop, and any OpenAI-compatible client to use it as a drop-in gateway without code changes. It requires no outbound network traffic in its minimal configuration and supports Privacy-by-Design requirements under Article 25 of the GDPR [7] through its architecture. The same OCI image is packaged for deployment as a rootless Podman container inside an unprivileged Proxmox LXC, as a Docker Compose stack, as a Podman Quadlet unit, and via Helm on Kubernetes and OpenShift; Kubernetes is CI-validated, while OpenShift support is implemented but not yet environment-validated. 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 the internal MoE-Eval benchmark (Likert scale, task-weighted; not a controlled comparison against external frontier benchmarks) – 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, and a benchmark-integrity incident in which non-sovereign frontier models were inadvertently used during a GAIA evaluation run – see Section 14.6 for the full disclosure and fix), 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 source code is released under Apache 2.0; this whitepaper is released under CC BY-SA 4.0. The project 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, self-hosted alternatives – from single-model launchers (Ollama [8]) to increasingly capable platforms such as LocalAI [9] (now covering agents, memory, MCP, and distributed inference) and document-focused stacks (PrivateGPT [10]) – have expanded substantially; yet assembling deterministic heterogeneous expert routing, runtime knowledge graph accumulation, and production-grade multi-user deployment into a single sovereign runtime remains largely an integration task left to 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] is designed to run as a rootless Podman container inside an unprivileged Proxmox LXC, in a single-host Docker Compose setup, in k3s/Kubernetes (CI-validated), or on Red Hat OpenShift (implemented; environment-unvalidated) – no code fork, no functionality loss between production-tested 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 self-hosted 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 runs in production today (Docker Compose and LXC are production-tested; Helm/Kubernetes is CI-validated; OpenShift is implemented but environment-unvalidated) 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 Adaptive Feedback Loop (informally: 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 – Adaptive routing and correction-based 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 system-level in-context learning mechanism: 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 per server. The acquisition cost was approximately €50 per card on the second-hand market – yielding 32 GB of VRAM per node at a price that has no equivalent among new hardware. Twenty cards in total were acquired for the cluster. 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.

Running current LLMs on M10 hardware also requires maintaining a custom fork of Ollama and llama.cpp: as the upstream ecosystem migrated to CUDA 12 and dropped CUDA 11 support, keeping these nodes operational with recent quantisation formats (GGUF Q4/Q5) demands backported CUDA 11 compatibility patches that must be rebased with each upstream release.

The third generation consists of **consumer GPUs from the Ampere and Turing generations** (RTX 2060, RTX 3060, each with 12 GB GDDR6X). The five RTX 3060 cards are operated in Ollama's native multi-GPU mode, which combines their VRAM into a single Ollama instance – providing 60 GB total VRAM, Flash Attention 2 support, and full GGUF compatibility at approximately 50–60 tokens/s for models within that VRAM budget. This cluster serves primarily as the development environment; the M10 nodes handle specialised expert workloads in production and benchmarking.

The common property of all three generations: they were acquired because they were *affordable*, not because they are optimal for LLM inference. None of the nodes can run a 70B model with a full context window at latencies acceptable for interactive production use. A current-generation cloud GPU or an H100 node would remove these constraints – at a cost that makes the system inaccessible to the target audience. The central practical question therefore becomes: what quality is achievable when the hardware budget is fixed at what a single person can finance?

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 discussed by the benchmark results in Section 14. The available evidence provides *indications* that orchestration produces functional advantages over individual local backbone models – particularly in tool delegation, knowledge accumulation, and domain-specific task decomposition. An initially reported GAIA score of 46.7 % was subsequently classified as invalid due to non-sovereign model substitution (Section 14.6). A controlled sovereign GAIA run is still pending. The central hypothesis is therefore technically plausibilised, but not yet confirmed by a controlled external benchmark. Where structural limits appear – in deep multi-step reasoning on Level 3 and with very long documents – those limits are technically explainable and addressable.

2.5 Adaptive Feedback Loop: 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 is an *Adaptive Feedback Loop* (informally referred to as the "RL Flywheel" throughout this document; 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. Laplace-Smoothed Bayesian Expert Scoring. 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 $(M+1)/(N+2)$ – the Bayesian posterior mean under a uniform Beta(1,1) prior 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, but not identical with, Thompson Sampling [14]: where Thompson Sampling draws a random sample from the posterior Beta distribution to select an arm, the implementation here computes the posterior mean and applies a deterministic

threshold – a bandit-inspired scoring rule rather than full Bayesian exploration-exploitation sampling.

Formal definition. We model expert routing as a *Bernoulli multi-armed bandit*. Let $\mathcal{E}$ be the set of available (model, category) expert pairs and $\theta_e \in [0,1]$ the unknown success probability of expert $e \in \mathcal{E}$. After N_e observations with M_e successes, the posterior is

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

the Bayesian posterior mean of θ_e under a uniform Beta(1,1) prior. Tier-2 escalation is triggered when $\hat{\theta}_e < \tau_{\text{route}}$ (configurable via `ROUTE_THRESHOLD`, default 0.65). Load adaptation applies an additive penalty $\delta_e = \alpha \cdot u_e$ to $\hat{\theta}_e$, where $u_e \in [0,1]$ is the current GPU utilisation of the node hosting e (read from `_ps_cache`) and $\alpha = 0.15$ is the load weight. This yields the effective routing score

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

used for expert selection. The smoothing constants $+1/+2$ implement a uniform Beta(1,1) prior, so new experts start at 0.5 and accumulate evidence without premature lock-in. This is the posterior mean of a Beta-Binomial model – a Laplace-smoothed Bayesian estimator, not a sampling step. A full Thompson Sampling implementation would draw $\hat{\theta}_e \sim \text{Beta}(M_e + 1, N_e - M_e + 1)$ at selection time; the deterministic threshold applied here is a simpler, lower-variance alternative that retains the prior-regularisation benefit without the sampling randomness [14].

3. Correction Memory. The third feedback-loop component 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.

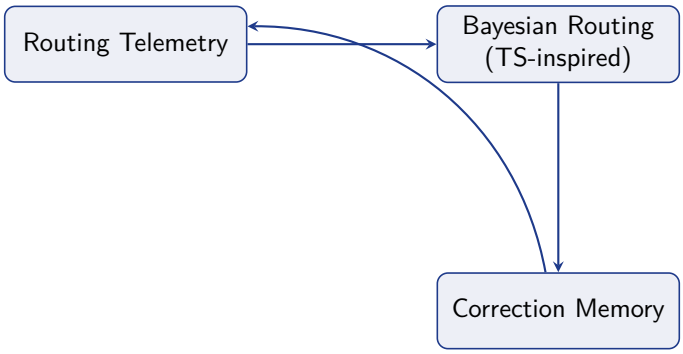

Figure 1: Adaptive Feedback Loop: Routing Telemetry, Laplace-Smoothed Bayesian Expert Scoring, and Correction Memory form a self-reinforcing improvement cycle with no model fine-tuning required.

Empirical evidence for the flywheel's effect appears in the measurement series from Table 17: 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.

Scope and limits of the feedback loop

The improvement cycle documented above is real but not autonomous. *Routing Telemetry* and *Correction Memory* accumulate automatically; the Neo4j ontology, however, grows only when an administrator reviews and acts on the ontology-gap signals in the Admin UI. The loop is therefore semi-supervised: the signal collection is automatic, but acting on structural knowledge gaps requires human attention.
A note on terminology: "RL Flywheel" is a descriptive shorthand for a self-reinforcing feedback cycle. It is not classical Reinforcement Learning in the sense of policy gradients or reward-function optimisation; the mechanism is a Laplace-smoothed Bayesian routing-weight update. We use the name because it captures the cyclical compounding dynamic, not to claim equivalence with RL in the formal sense.

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 orchestration layer produces demonstrable qualitative advantages in tool delegation, knowledge accumulation, and task decomposition. An earlier GAIA comparison value (46.7 % overall, 60 % Level 1) was subsequently classified as invalid due to non-sovereign model substitution (Section 14.6); a valid sovereign measurement run is pending.

- Against *specialised memory systems* such as EverMemOS (83 %) and TiMem (76.9 %) on LongMemEval, MoE Sovereign achieves 91.7 % in the best internal run – despite an architecture that treats memory as a by-product of knowledge graph accumulation rather than as a primary optimisation target. A methodological caveat applies: whether identical dataset splits, scoring normalisers, and evaluation rules as the official leaderboard were applied is not fully documented; a direct leaderboard comparison therefore requires qualification (Section 14.15.6).

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. Weaker guarantees remain unverifiable in practice.

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. On-premises inference enables operator-controlled and technically enforceable data residency; cloud-hosted inference does not.
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 [10] specialises in document indexing and question-answering workflows. LocalAI [9] has evolved substantially and now provides an OpenAI-compatible API with optional agents, memory, MCP support, distributed inference, and a web UI – a broader scope than its original role as a single-server model launcher. Both are valuable building blocks for local deployment. The distinction with MoE Sovereign lies in the coordination layer: MoE Sovereign adds VRAM-aware heterogeneous hardware scheduling, deterministic request-level expert routing via versioned templates, a runtime-accumulated knowledge graph with contradiction detection, multi-tenant token budgets, and an integrated edge-to-enterprise deployment path – capabilities that require explicit design choices beyond what a general-purpose inference server provides.

Advances in retrieval quality. Recent work has addressed a blind spot in standard RAG: the assumption that *all retrieved documents are relevant*. Yan et al. [18] show that relevance is unreliable and propose a lightweight retrieval evaluator (Corrective RAG, CRAG) that scores each document before injection – a pattern adopted in MoE Sovereign's Corrective RAG Gate (Section 8.9). At the other extreme, Chan et al. [19] argue that for stable, bounded domains retrieval is unnecessary altogether and that directly pre-loading the full knowledge source (Context-Augmented Generation, CAG) outperforms RAG – the pattern behind MoE Sovereign's compliance layer (Section 8.10).

Memory systems for LLM agents. Park et al. [20] introduce memory streams for agent architectures: timestamped experience records retrieved by relevance and recency. Packer et al. [21] formalise a tiered memory model for LLMs analogous to OS virtual memory. Both systems ground the empirical importance of episodic memory for long-horizon agent behaviour – the motivation for the episodic memory layer in MoE Sovereign (Section 8.11).

4.4 Research-oriented stacks

The combination of vLLM [22] for the inference layer with LangChain [23] 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. [24] 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. [25] 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 requires a significant long-term vendor commitment, cloud or managed on-premises deployment, and costs upward of $1M/year. MoE Sovereign pursues a comparable architectural approach as an open-source container stack on local hardware (including legacy GPUs), fully air-gap capable, at zero licence cost – while acknowledging that Palantir's product maturity, enterprise certifications, and support organisation are not replicated by an open-source project at this stage.

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. Systems were selected to cover the four primary architectural categories – enterprise AI platforms, open-source orchestration frameworks, single-model launchers, and generic RAG toolkits – with at least one representative per category that is directly discussed in the text above. A cell marked "–" indicates the feature is absent; "✓" indicates full support; "∼" indicates partial or limited support.

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

Feature	***MoE Sovereign***	***Palantir AIP***	***Databricks***	***Glean***	***LocalAI***	***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	Free

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 provides the inference routing layer; MoE Sovereign builds the full cognitive orchestration stack above it.

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 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. The Docker Compose and LXC deployment paths are production-tested; Kubernetes and OpenShift wrappers are

implemented and architecturally validated, awaiting broader community validation before claiming full production status. Few open-source systems combine deterministic request-level expert routing, heterogeneous hardware scheduling, runtime graph knowledge accumulation, deterministic precision tooling, and an edge-to-enterprise deployment model into a single inspectable sovereign runtime.

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 [26] (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`, `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 Multi-protocol API gateway

A critical operability decision was to support *three client protocols simultaneously* from a single orchestrator process, rather than committing to one API surface and requiring wrappers for the rest.

Protocol	**Endpoint(s)**	**Compatible clients**
OpenAI Chat	`/v1/chat/completions`	Open WebUI, LiteLLM, any OpenAI-SDK app
Anthropic Messages	`/v1/messages`, `/v1/responses`	Claude Code CLI, Claude Desktop, Anthropic Python/JS SDK
Ollama protocol	`/api/chat`, `/api/generate`	Ollama clients, many frontends that bypass OpenAI

The Anthropic Messages API support (added in v2.3) includes the `/v1/messages/count_tokens` endpoint required by the Anthropic Gateway specification for context-budget warnings in

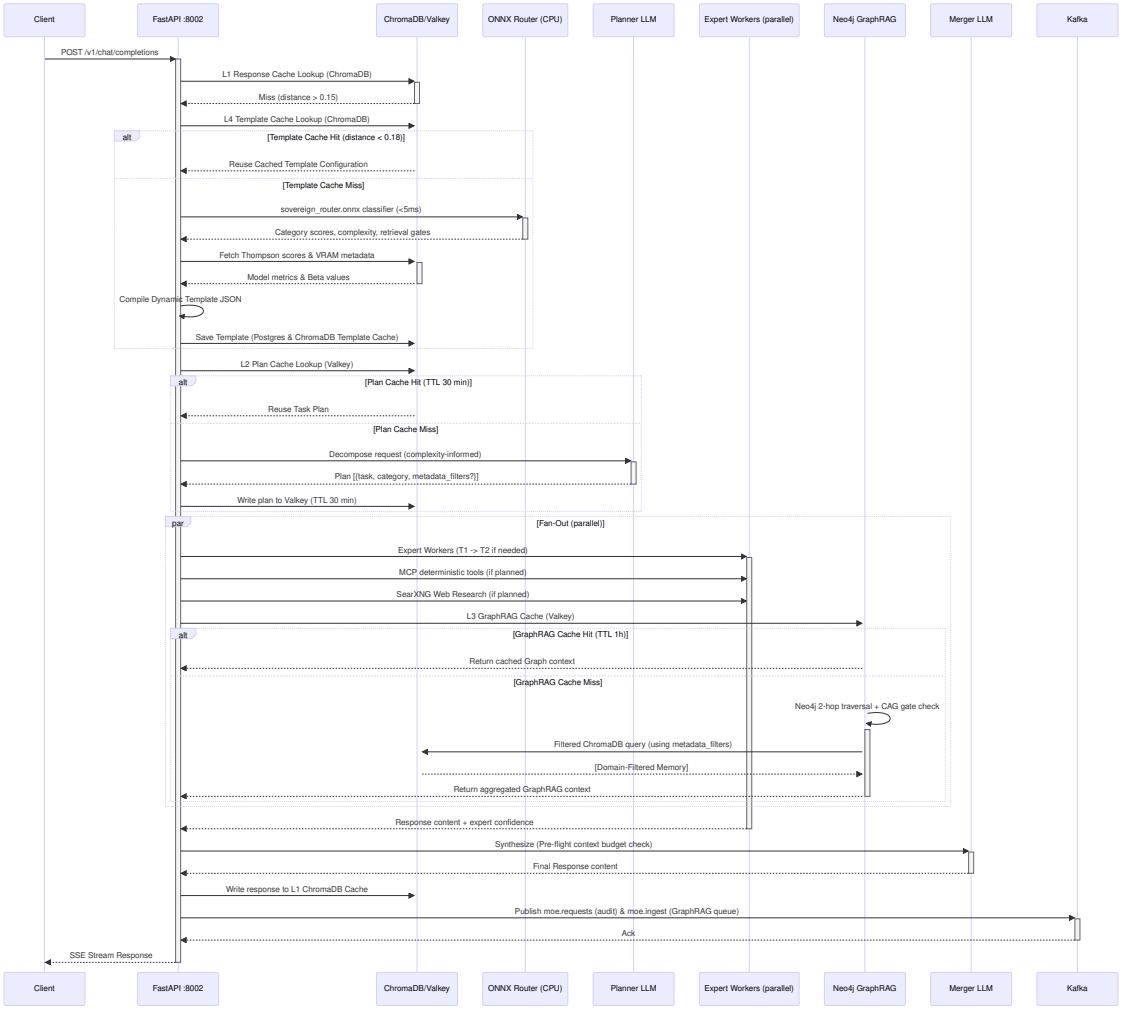

Figure 2: End-to-end data flow for a chat request through the orchestrator. Each node in the pipeline is a LangGraph node with explicit state; each box outside the pipeline is a separate service with its own API.

Claude Code. Model entries in `/v1/models` carry a `display_name` field so that Claude Desktop renders human-readable names (e.g. "Claude Sonnet 4.6 → MoE (Gateway)") in its model picker. An interactive setup wizard (`scripts/setup-claude-desktop.sh`) auto-configures `claude_desktop_config.json` on macOS, Linux, and WSL in under 60 seconds.

Why three protocols?

A gateway that only speaks OpenAI is invisible to Anthropic-SDK workloads and vice versa. The three protocols cover the full client landscape without requiring code changes on the client side. The orchestrator detects the protocol from the URL prefix and routes internally to the appropriate handler in `services/pipeline/`. No functionality is lost between protocols: all three paths invoke the same LangGraph pipeline.

5.4 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

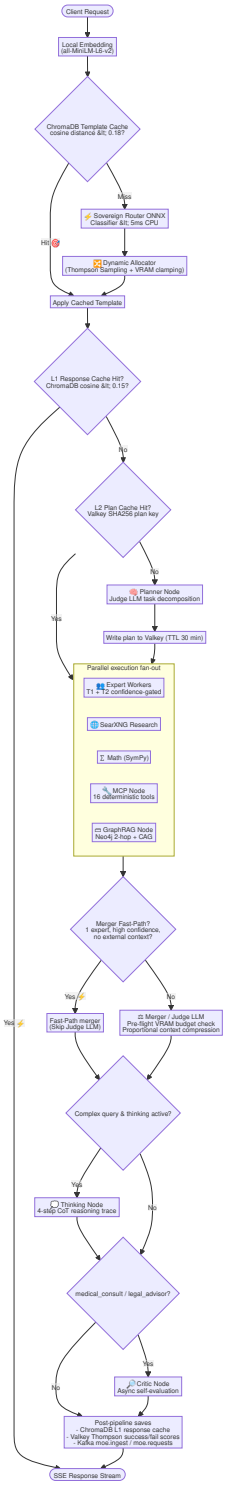

Figure 3: The LangGraph pipeline. Requests flow through `planner` (expert selection), `expert_worker` (parallel model invocations per category), `merger` (synthesis with source priority), optionally `thinking_node` (4-step chain-of-thought on low confidence), optionally `research_fallback` (external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

(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.

5.5 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`, 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.

Expert templates are the exception to this pattern: they are persisted as JSON in the `admin_expert_templates` PostgreSQL table (written by the admin UI) and read from there on every request with a 30-second in-process cache. The `EXPERT_TEMPLATES` environment variable serves as a fallback when the database is unreachable at boot time. This two-tier design reflects the operational reality that templates are modified interactively and must be immediately effective, whereas infrastructure-level settings (`INFERENCE_SERVERS`, ports, URLs) change rarely and benefit from the auditability of a text file.

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.

5.6 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.

5.7 Compliance extension: MoE Codex

A sixth optional layer – orthogonal to MoE Libris – is *MoE Codex*[1], an EU-sovereign data and audit platform for regulated operators in sectors where data sovereignty is a legal requirement: public authorities, critical infrastructure (KritIS), pharmaceutical compliance, and banking audit.

[1] Latin *codex* = manuscript collection, lawbook – a deliberate reference to its compliance and audit focus. The name is a thematic sibling of MoE Libris (*liber* = free / book).

MoE Codex is deployed alongside the core stack and communicates with the orchestrator via a defined HTTP/REST contract (`CODEX_URL` / `SOVEREIGN_URL`). It does not replace the orchestrator; it has grown from the original eight compliance-oriented modules to twenty-five modules organised across five functional tracks.

Track D.0 — Foundational Compliance Modules (original eight):

- **Data Catalog** – structured asset registry with GDPR classification, retention policy, and provenance tracking.
- **Knowledge Approvals** – four-eyes approval workflow for knowledge bundles before they enter the GraphRAG accumulation pipeline.
- **Object Explorer** – read-only Cypher query interface for graph investigation by compliance officers.
- **Data Health & Drift** – statistical drift detection across registered data assets; alert thresholds configurable per dataset.
- **JupyterLab Notebook** – proxied notebook environment for reproducible data analysis within the sovereign perimeter.
- **Lineage (Marquez)** – OpenLineage-compatible data lineage for all pipeline events, persisted in a dedicated Marquez instance.
- **Versioning (lakeFS)** – Git-style branching and versioning for data bundles; every commit is addressable and reversible.
- **ETL Fan-Out (Apache NiFi)** – declarative pipeline builder for ingestion, transformation, and fan-out of structured data sources.

Track D.1 — AIP Platform Layer extends the stack with enterprise AI governance and data federation primitives:

- **OPA Permissions & Data Markings** – Open Policy Agent enforcing attribute-based access control (ABAC) and sensitivity markings at the API gateway level.
- **MLflow Model Registry** – experiment tracking, artifact storage, and model staging lifecycle management.
- **NeMo Guardrails** – prompt and response safety layer with pattern-based configuration for domain-specific rail definitions.
- **Trino SQL Federation** – distributed SQL engine federating Postgres, lakeFS, and in-memory catalog sources through a single query interface.
- **DocLing Document Intelligence** – structured extraction from PDF, DOCX, and image inputs to Markdown; visual page description via ColPali embeddings.

Track D.2 — Frontend Extensions adds operator-facing dashboards and investigation surfaces directly within the Codex portal:

- **Kestra Pipeline Builder** – YAML-based workflow orchestration with an integrated execution monitor and trigger management UI.
- **Dynamic Forms** – schema-driven data-entry forms generated automatically from Kestra flow input definitions.

- **Charts & Pivot** – Chart.js charting combined with PivotTable.js pivot analysis, backed by Trino preset queries.
- **Federated Search** – OpenSearch BM25 + fuzzy search with keyword fallback across all indexed catalog assets.
- **Timeline** – vis-timeline.js event view combining Marquez pipeline runs, lakeFS commits, and drift detection events in a unified temporal display.
- **Link Analysis** – Cytoscape.js network graph populated from Neo4j Cypher queries for relationship and provenance investigation.

Track D.3 — BI, Search, and Compliance Posture brings business intelligence and runtime compliance monitoring:

- **Apache Superset BI** – dashboard embedding via guest tokens with automatic Trino data-source registration; functional equivalent of Palantir Quiver.
- **OpenSearch Unified Index** – BM25 + fuzzy full-text index with bulk ingestion from Marquez and lakeFS, and graceful degradation when the cluster is unavailable.
- **Falco + OpenSCAP Compliance Posture** – runtime threat detection via Falco and CIS benchmark scanning via OpenSCAP; functional equivalent of Palantir Apollo.

Track D.4 — Workshop, Time Series, Notes, and Geospatial covers low-code application building and domain-specific data views:

- **Budibase Low-Code Workshop** – Apache 2.0 drag-and-drop application builder for internal tooling; functional equivalent of Palantir Workshop.
- **TimescaleDB Time Series Catalog** – hypertable asset catalog with Chart.js rendering and PostgREST API exposure.
- **HedgeDoc Collaborative Notes** – AGPL real-time Markdown collaboration environment; functional equivalent of Palantir Notepad.
- **PostGIS + Leaflet Geospatial** – GeoJSON API with point-in-polygon queries and Leaflet-based interactive maps; functional equivalent of Palantir Geo.

Track D.5 — Investigation provides a structured case management layer across all Codex surfaces:

- **Dossier / Case File** – Redis-backed investigation container that allows operators to pin evidence items from any of the six investigation modules into a named, persistent case record; functional equivalent of Palantir Dossier.

MoE Codex now covers 92 % of the Palantir Foundry / Gotham / AIP / Apollo feature surface (15 modules fully covered, 31 partially covered, 2 structurally not achievable: mobile / tactical edge hardware and the Apollo constraint solver). The coverage breakdown: Foundry 22 / 24 covered, Gotham 8 / 9, AIP 13 / 13, Apollo 3 / 4. Operators who require only the core multi-model gateway deploy MoE Sovereign alone; the Codex extension is explicitly opt-in and released as a separate Apache 2.0 repository (`github.com/h3rb3rn/moe-codex`).

Excursus: Regulatory Mapping of MoE Codex under EU Law

To establish a production-grade compliance framework for compound AI systems, *MoE Codex* implements a direct mapping to active European regulatory requirements:

1. **GDPR (Art. 32 & 35):** Under Article 32 (Security of processing) and Article 35 (Data Protection Impact Assessments), operators must prove data integrity and track lineage. MoE Codex addresses this by utilizing *lakeFS* version control in tandem with *Marquez (OpenLineage)*. If an accumulated knowledge bundle is found to violate privacy rights, compliance officers can execute a zero-penalty rollback of the Neo4j instance to a prior clean commit.
2. **EU AI Act (High-Risk Systems):** For high-risk use cases, Article 9 (Risk management system) and Article 10 (Data governance) mandate continuous logging and deterministic validation. MoE Codex routes all model inputs and outputs through the *Open Policy Agent (OPA)*, evaluating declarative Rego policies in real time. Requests originating from safety-critical sectors (e.g., medical or legal) are automatically subjected to paraconsistent logic arbitration.
3. **NIS2 & BSI IT-Grundschutz:** To satisfy the strict monitoring obligations for Critical Infrastructure (KRITIS), MoE Codex integrates *Falco* for container-level anomaly detection and *OpenSCAP* for automated security compliance scans, establishing an auditable security posture for local GPU cluster deployments.

5.8 Runtime feature control: Starfleet toggles and Mission Context

Two lightweight mechanisms provide operators with runtime control over system behaviour without requiring a container restart.

Starfleet feature toggles. A Redis-backed feature-flag store (`starfleet_config.py`) enables or disables named capabilities at runtime. Every API route that depends on a toggle checks its state at request time; the fallback is always a safe, degraded response rather than an error. This allows operators in regulated environments to switch off experimental features during audits without downtime.

Mission Context (`/api/mission-context`). A persistent key–value document stored in Valkey that is injected into every pipeline request as additional system context. Unlike the per-session history compression (Section 7), Mission Context is deployment-wide and survives across sessions. Typical use: an organisation injects its domain vocabulary, internal naming conventions, and compliance constraints once; all subsequent pipeline requests benefit without re-stating the context. Mission Context also partially addresses the long-context memory limitation observed in Section 14: persistent organisational knowledge does not compete for the planner's token budget.

5.9 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). This range is derived from production observations on conversational workloads with moderate repetition; it will be significantly lower for fully novel queries and higher for repetitive FAQ-style usage. A systematic sensitivity analysis across workload types and TTL configurations is planned but not yet available. 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.

5.10 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. This is standard *constraint-driven engineering*; the AGC analogy is illustrative, not a claim to novelty.

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.

Trivial low-confidence rescue (v2.7). Queries classified as *trivial* by the complexity estimator are cost-tier-limited to a single tier-1 expert (`force_tier1`). Starting with v2.7, a controlled rescue overrides this saving when the tier-1 answer is genuinely weak: if the tier-1 confidence is `low` or the response is empty, exactly one tier-2 model is activated. A `medium` answer keeps the cost saving. The behaviour is controlled by `TRIVIAL_LOW_CONF_RESCUE_ENABLED` (default `true`) and is captured in the Prometheus counter `moe_tier_escalation_total{category, decision}` with the decision label `t1_cost_kept` (saving kept) or `t2_escalated` (rescue fired).

User-defined expert categories (v2.7). User templates may define arbitrary expert categories beyond the global `EXPERTS` registry (e.g. `mail_classify`, `devops_sre`, `tool_agent`). Prior to v2.7, `_sanitize_plan()` validated the planner output only against the global registry and silently degraded unknown categories to `general`, routing to the wrong model. The fix passes the active template's category set as an additional valid set so custom experts are reachable without modifying the global configuration.

OpenAI tool-calling passthrough (v2.7). When a request carries an OpenAI `tools` array (function definitions) or messages with `role: "tool"` (tool-result turns), the gateway bypasses the planner→experts→merger pipeline entirely and forwards the request directly to the template's dedicated `tool_expert` model, falling back to the `judge` model if no `tool_expert` is configured. The selected model (e.g. `hermes3:8b` or `qwen3.6:35b`) handles function calling natively and returns `tool_calls` or the final text to the client. Follow-up `role: "tool"` turns omit the `tools` payload in the forwarded request to prevent model tool-call loops. This enables Hermes MCP integration, Open-WebUI function-calling, and any OpenAI-function-calling-compatible agent framework without pipeline overhead.

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).

Each expert category additionally carries an optional `context_window` field (integer, in tokens). When set, it serves as the *authoritative constraint* for the entire token budget

stack: it overrides the value reported by the Ollama API (which may reflect a different model variant or startup parameter), it is used to derive the per-expert output cap (min(MAX_EXPERT_OUTPUT_CHARS, ⌊ctx/4⌋)), and it sets the hard ceiling for the `tool_max_tokens` and `reasoning_max_tokens` declared in any CC profile that references this template. The CC session builder validates this constraint at request time and logs a warning if a profile exceeds the template value. The invariant is simple: the template defines the hardware reality; the profile may only operate within it.

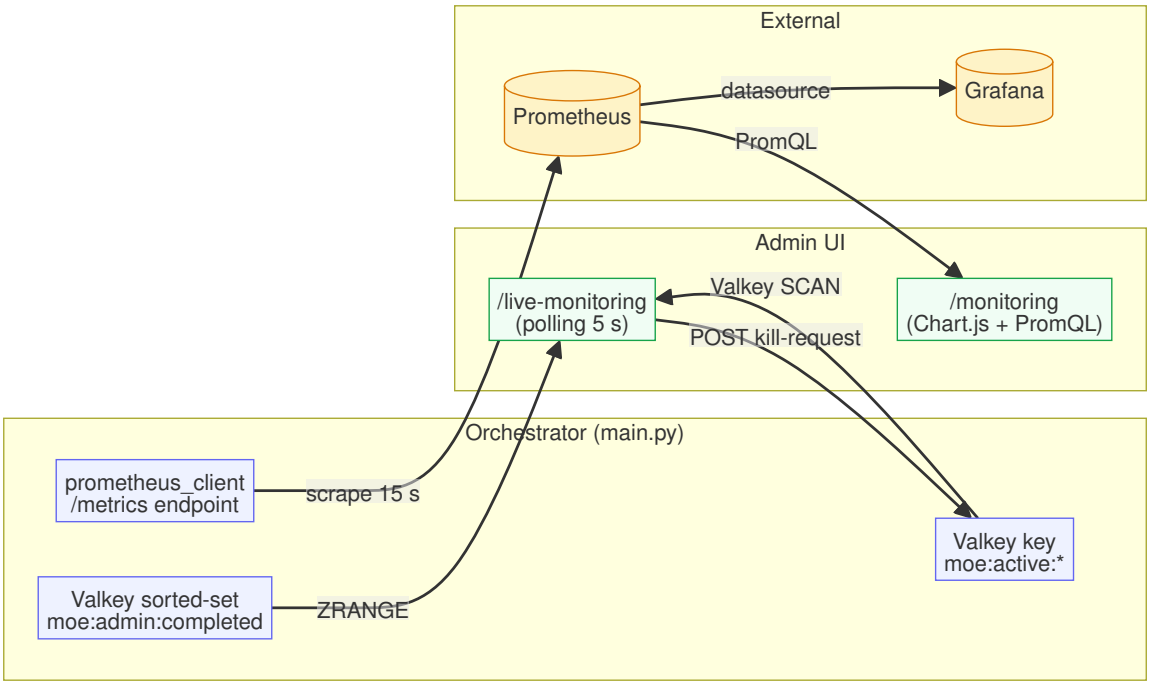

Figure 4: The observability architecture mirrors the same design approach: system metrics and live monitoring are treated as two complementary views on the same data – no abstraction that hides provenance.

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)	60 GB	1.0	70B
N07-GT (1× GTX 1060)	6 GB	—	*offline since 2026-06-02 (defective GPU)*
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 (model, 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.

7.6 Session history compression cache (Valkey)

A fifth caching mechanism operates orthogonally to L1–L4: it targets the *conversation history* passed to the tool model in multi-turn Claude Code sessions, not the LLM response itself.

Over long coding sessions, the synthesised MoE responses in the conversation history can saturate the tool model's context window. Rather than dropping entire turns – which destroys continuity – the orchestrator applies a compression pass before each tool-model call: assistant and tool messages older than a configurable number of recent turns are condensed to their first 800 and last 200 characters, with the intermediate content replaced by an annotated marker. The full original text is stored in Valkey under the key `cc:hist:<session_id>:<sha1[:12]>` with a one-hour TTL, preventing silent data loss while keeping the active context compact.

The existing fall-through trimmer (`_trim_oai_to_budget`) then operates on already-compressed history and must drop far fewer complete turns. The two mechanisms are complementary: compression preserves breadth (all turns remain); trimming preserves precision (budget hard limit is always respected).

Key parameters (configurable via `.env`):

`CC_HISTORY_COMPRESS_THRESHOLD` Characters above which a message is condensed (default: 3000).

`CC_HISTORY_COMPRESS_KEEP_TURNS` Most recent n turns that are never compressed (default: 2).

7.7 Augmented Tool Path: cache and GraphRAG extension for agentic clients

Agentic coding tools – Claude Code CLI, OpenCode, and comparable function-calling clients – send virtually every request with a `tools` array or with `tool_result` turns. MoE Sovereign therefore has a dedicated tool-calling fast path that deliberately bypasses the entire MoE pipeline – planner, experts, judge, and thereby L1–L4 as well – and forwards the request straight to a single tool model. This shortcut is necessary because reliable function calling requires a single, consistent model; but it means agentic sessions gain nothing from the entire cache and GraphRAG infrastructure – every request costs full inference, regardless of how often the same question has already been answered.

The *Augmented Tool Path* closes this gap without changing the fast path itself: an opt-in enrichment layer sits between the tool handler and the tool model and behaves completely transparently to the client – streaming events, `stop_reason` semantics, and tool-use blocks are preserved byte-for-byte.

Separate cache namespace. Rather than sharing L1 (`moe_fact_cache`), the Augmented Tool Path reads and writes a dedicated collection (`moe_agent_cache`) and its own Valkey prefix (`moe:agent:qcache:*`). Every entry carries a scope key `sha256(user_id | workspace)` – knowledge from Project A can therefore never leak into Project B, even for the same user. Only *informational* queries on a session's first turn are served (detected heuristically by question form without a mutation verb such as "fix"/"implement"); tool-use decisions themselves are never served from cache, since environment state differs between sessions.

Trust-building instead of immediate caching. A fresh answer is first stored at confidence 0.6 – deliberately below the 0.85 serve threshold. An asynchronous judge call scores the answer in the background; only a high score raises the confidence to 0.9 and unlocks the entry for future cache hits. This two-stage pattern mirrors the same caution that already characterises the L1 knowledge-bypass gate (confidence and freshness thresholds, `flagged` marking on negative feedback) – except that here the judge itself acts as the downstream quality control, rather than relying on user feedback.

Per-session GraphRAG injection. In addition to the cache, the Augmented Tool Path queries the knowledge graph on a session's first turn (tenant-scoped, see Section 8) and injects the result into the tool model's system prompt. The result is cached under a session-bound Valkey key so subsequent turns of the same tool loop do not re-query the graph – one Neo4j query per session instead of per turn.

Knowledge growth from agentic sessions. When a session ends with a clean text answer (`end_turn` with no tool use, no stream error), the question/answer pair is published on the same Kafka ingestion topic the interactive pipeline uses. The knowledge graph therefore grows not only from Open-WebUI conversations but also from coding sessions – including provenance and trust decay as described in Section 8.

All latency costs are hard-bounded and deliberately budgeted conservatively: the cache lookup is time-boxed to 300 ms, the GraphRAG query to 2 s, and every write operation runs as an `asyncio.create_task` fire-and-forget – the latency the client perceives changes only on a cache hit (then dramatically faster), never as a result of the enrichment failing. With a flag disabled, behaviour is bit-identical to the plain fast path.

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 [27] 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 [25], 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.

Formal ontology schema. Table 2 specifies the domain-to-type mapping enforced at query time. Categories not listed permit all entity types (`general`, `reasoning`, `precision_tools`).

Relation types permitted across all domains: `IS_A` | `PART_OF` | `TREATS` | `CAUSES` | `REGULATES` | `USES` | `IMPLEMENTS` | `DEPENDS_ON` | `DEFINES` | `RELATED_TO` | `NECESSITATES_PRESENCE` | `ENABLES_ACTION` | `DEPENDS_ON_LOCATION`. Node properties: `name` (string), `type` (string), `aliases_str` (string), `tenant_id` (optional string), `confidence` (float $\in [0, 1]$), `version` (int), `source_model` (string).

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.

Table 2: Domain-scoped entity-type filter: allowed Neo4j node types per expert category. Categories absent from the table are unrestricted.

Expert category	Allowed entity types
`medical_consult`	Drug, Disease, Symptom, Treatment, Anatomy, Medical_Concept, Drug_Class
`legal_advisor`	Law, Right, Legal_Concept, Organization
`technical_support`	Framework, Tool, Protocol, Tech_Concept, DevOps, Architecture, Security, DataStructure, Algorithm, Pattern, Principle, Action, Location, Condition
`code_reviewer`	Framework, Tool, Tech_Concept, DataStructure, Algorithm, Pattern, Principle, Architecture, Security, Game_Concept, Data_Concept, Action, Condition
`math`	Math_Concept, Science, Concept, Algorithm, DataStructure
`translation`	Language, Concept, Narrative
`creative_writer`	Narrative, Concept
`research`	*(unrestricted — all types)*

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*[2] we have realised it as a concrete, deployable system.

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

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.

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.

8.9 Corrective RAG Gate

Standard RAG injects all retrieved documents into the generation prompt regardless of their actual relevance to the query. Yan et al. [18] demonstrate that this unconditional injection degrades output quality when retrieval is ambiguous or incorrect, and propose a lightweight *retrieval evaluator* that scores each retrieved document before injection. Documents are classified as *Correct* (inject), *Ambiguous* (refine), or *Incorrect* (discard and trigger web-search fallback).

MoE Sovereign adapts this mechanism to the Neo4j entity level. After `query_context()` builds the set of candidate entities from the graph, each entity is scored by `_corrective_relevance_score()`:

$$s(e) = 0.75 \cdot \frac{2 \cdot h_{\text{name}} + h_{\text{rel}}}{3 \cdot |T|} + 0.25 \cdot \bar{c} \tag{3}$$

where T is the set of non-stopword query terms, h_{name} counts how many terms appear in the entity name (weighted $2\times$ because a direct entity match is strong evidence), h_{rel} counts term occurrences in the relation targets (weaker signal), and $\bar{c}$ is the average confidence of the entity's stored relations (a proxy for knowledge quality). Entities scoring below a configurable threshold τ (default 0.15, env var `GRAPHRAG_CORRECTIVE_THRESHOLD`) are discarded before injection. When all candidate entities fall below τ, `query_context()` returns the empty string and the pipeline proceeds without graph context rather than injecting noise.

The threshold $\tau = 0.15$ is deliberately conservative: it eliminates only entities where fewer than 15 % of query terms appear anywhere in the entity's textual surface. Administrators targeting precision-critical deployments may raise it to 0.30–0.40; setting $\tau = 0$ restores the original, gate-free behaviour.

Corrective RAG reduces context pollution

The gate prevents *context pollution*: injecting tangentially matched graph nodes that share a term with the query but are semantically unrelated. In early experiments

without the gate, queries touching polysemous terms (e.g. "Python" could match both programming and biology entities) caused the judge model to blend domain-foreign context into its response. The corrective score reliably eliminates these false-positives without a second LLM call.

8.10 Context-Augmented Generation for Compliance Domains

Chan et al. [19] show empirically that for knowledge domains where the ground truth is *stable*, *authoritative*, and *bounded*, pre-loading the full context outperforms retrieval-based RAG in accuracy, consistency, and latency. Their Cache-Augmented Generation (CAG) approach caches the full knowledge source in the model's context window, bypassing the retrieval pipeline.

In regulated industries – the primary deployment target of MoE Sovereign – regulatory texts such as the German banking IT requirements (BAIT), the insurance IT requirements (VAIT), the Digital Operational Resilience Act (DORA), and the critical infrastructure regulation (KRITIS) exhibit exactly these properties: they change only on legislative cycles, their exact wording is authoritative, and the relevant excerpt for any given query fits within a single context block.

The `compliance_cag` module implements this pattern as a pre-retrieval gate in `graph_rag_node`. Each compliance domain is defined by an admin-managed JSON file containing keyword triggers and an authoritative text block. On each request, the module scans the query for keyword matches before issuing any Neo4j query; on a match, the pre-loaded text block is returned directly and the database round-trip is skipped entirely. The result is cached in Valkey at the same TTL as standard GraphRAG results.

Table 3: CAG vs. standard GraphRAG for compliance-domain queries.

Property	**Standard GraphRAG**	**CAG Layer**
Latency	1–3 s (Neo4j + Valkey)	<1 ms (in-process)
Reliability	Depends on graph coverage	Deterministic
Wording accuracy	Paraphrased by extraction	Verbatim authoritative text
Update mechanism	Graph ingest (online)	JSON file swap (file-system)
Audit traceability	Source model annotation	Explicit domain label

Adding a new compliance domain requires only dropping a `.json` file into `$MOE_DATA_ROOT/cag/` – no code change and no container restart. The module hot-reloads the directory every `CAG_RELOAD_INTERVAL_S` seconds (default 300).

8.11 Episodic Memory

Tulving [28] distinguishes three long-term memory systems: *semantic* memory (facts and world knowledge), *episodic* memory (temporally situated past experiences), and *procedural* memory (skill-execution patterns). Park et al. [20] operationalise episodic memory for LLM agents as *memory streams*: timestamped experience records retrieved by relevance and recency, summarised into higher-level reflections. Packer et al. [21] ground this in a tiered architecture analogous to OS virtual memory.

MoE Sovereign maps Tulving's taxonomy directly onto its storage layer:

Table 4: Three-tier memory architecture in MoE Sovereign.

Memory Type	Content	Implementation
Semantic	Domain facts, world knowledge	Neo4j knowledge graph
Episodic	Past task outcomes + routing	`:Episode` nodes (Neo4j)
Procedural	Action–location requirements	Procedural relations (Neo4j)

The `episodic_memory` module adds the missing episodic tier. After every completed merger response, a fire-and-forget `asyncio.create_task()` call logs a `:Episode` node to Neo4j with:

- **task_type** — primary expert category from the planner;
- **routing_path** — ordered list of executed categories;
- **tools_used** — active pipeline tools (graphrag, mcp, math, web, cache);
- **confidence** — weighted estimate: $0.7 \cdot \bar{c}_{\text{expert}} + 0.3 \cdot f_{\text{length}}$, where $f_{\text{length}} = \min(1, |\text{response}|/600)$;
- **total_tokens** — total prompt and completion tokens;
- **expires_at** — TTL-based expiry (default 90 days).

At retrieval time, `get_episode_hint()` queries past `:Episode` nodes for the same task type, ranks them by Sørensen–Dice string similarity against the current query, and returns the top-k episodes as a structured `[Episode Hint]` block appended to `graph_context`. The hint informs the judge model which routing strategies have historically produced high-confidence answers for similar queries – without prescribing the answer.

All episodic operations are fire-and-forget: a Neo4j failure never propagates to the primary response path.

8.12 Attribution of Applied Scientific Contributions

Table 5 consolidates *all* scientific sources whose concepts are directly implemented in MoE Sovereign's codebase — spanning formal logic, expert routing, retrieval, and memory. The column "Implementation" names the specific module and mechanism.

Table 5: Scientific sources directly implemented in MoE Sovereign (full codebase).

Authors	Year	Concept adopted	Implementation in MoE Sovereign
de Vries, Andreas [29]	2007	Algebraic hierarchy of logics: unified lattice order placing paraconsistent, fuzzy, quantum, intuitionistic, and Boolean logics in a common structure; algebraic motivation for contradiction-tolerant reasoning	`pipeline/state.py`: `conflict_registry` (paraconsistent conflict log); `graph/synthesis.py`: `resolve_conflicts_node` (Strategy A/B arbitration)
Belnap, Nuel D. [30]	1977	Four-valued semantics $\{T, F, B, N\}$ (*True*, *False*, *Both*, *Neither*): contradictions localised as B without logical explosion	`resolve_conflicts_node`: formal basis for conflict classification (Strategy A dismiss vs. Strategy B arbitrate); $\{T, F, I, U\}$ truth-value labels in the 90 000-sample paraconsistent training dataset
Gödel, Kurt [31]	1932	Gödel t-norm $T_G(a, b) = \min(a, b)$: most conservative fuzzy conjunction	Default conjunction in `fuzzy_router_node`: combines `vector_confidence` and `graph_confidence` using min, ensuring the result is bounded by the weaker signal
Łukasiewicz, Jan [32]	1920	Łukasiewicz t-norm $T_{\text{Ł}}(a, b) = \max(0, a+b-1)$: tolerant fuzzy conjunction	Alternative conjunction in `pipeline/logic_types.py` (`lukasiewicz_tnorm`); selectable via template config for deployments that prefer less conservative routing
Kolmogorov, Andrei N. [33]; Chaitin, Gregory J. [34]	1965 / 1966	Algorithmic information content (Kolmogorov complexity): length of shortest description of a string; zlib length as practical upper bound	`complexity_estimator.py`: `_aic_compressibility()` uses zlib compression ratio as a Kolmogorov approximation to classify queries as `trivial`, `moderate`, or `complex` without an LLM call
Ratcliff, John W.; Metzener, David E. [35]	1988	Ratcliff/Obershelp algorithm: longest common substring iteration for string similarity (Gestalt approach)	`graph_rag/manager.py`: `_fuzzy_resolve_entity_name()` uses Python `SequenceMatcher` (threshold 0.82) to canonicalise entity names before Neo4j MERGE, preventing duplicate nodes from spelling variants
Anthropic [6]	2024	Model Context Protocol (MCP): standard for exposing deterministic tools to LLM agents via a structured client–server interface	`mcp_server/server.py`: 51 deterministic tools behind an AST whitelist (arithmetic, hashing, subnet, date, regex, PPTX, Wikidata, PubMed, Wayback, chess…); zero per-token API cost; precision-critical operations removed from probabilistic model logic
Russo, Daniel J.;	2018	Thompson Sampling:	Expert-routing confidence es-

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.

continued

Tool	Purpose
`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. The separation of critique from generation reflects the same insight as Constitutional AI [36]: a dedicated evaluator node yields more reliable self-assessment than embedding evaluation inside the generating model itself.

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. The pattern is an instance of the iterative self-refinement idea formalised by Madaan et al. [37]: generate, provide structured feedback, refine – repeated until a completeness criterion is met or an iteration budget is exhausted.

10.8 Paraconsistent conflict resolution

A separate node, `resolve_conflicts_node`, addresses a different failure mode: not an *incomplete* answer, but a *contradictory* one – two expert outputs that cannot both be true.

The node implements a paraconsistent resolution strategy. De Vries (2007) [29] provides the algebraic motivation: within a unified lattice hierarchy, paraconsistent logic occupies a position that tolerates contradictions rather than propagating their logical consequences (as classical logic would via *ex contradictione quodlibet*); the contradiction is made explicit so downstream stages can act on it without the entire inference collapsing.

Formal definition. Let $\mathcal{P}$ be a proposition and $\neg\mathcal{P}$ its negation. Classical logic obeys the *principle of explosion*: $\{\mathcal{P}, \neg\mathcal{P}\} \vdash \mathcal{Q}$ for any $\mathcal{Q}$, making contradictions globally destructive. Following Belnap (1977) [30], this is replaced with a four-valued semantics $\{T, F, B, N\}$ where B ("both") denotes a contradiction that is *localised* – it does not propagate.

In MoE Sovereign, each claim in `conflict_registry` carries a *divergence score* $d \in [0, 1]$ computed as

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

the Jaccard distance between the token sets of two expert responses r_1, r_2 in the same category. The resolution strategy is then threshold-gated:

$$\text{action}(d) = \begin{cases} \text{DISMISS} & d < 0.5 \\ \text{ARBITRATE} & d \geq 0.5 \ \wedge \ \text{cat} \in \mathcal{C}_{\text{safety}} \end{cases} \tag{5}$$

where $\mathcal{C}_{\text{safety}} = \{\texttt{medical_consult}, \texttt{legal_advisor}\}$. All entries persist in `conflict_registry` with lifecycle PENDING $\rightarrow$ RESOLVED | DISMISSED, forming a tamper-evident audit trail.

Two resolution strategies are applied in sequence:

1. **Strategy A – auto-dismiss** (divergence score < 0.5): When the semantic divergence between two conflicting answers falls below threshold, the conflict is classified as *formulaic*

variation – different phrasing of a compatible claim – and dismissed automatically. No LLM call is required.

2. **Strategy B – judge arbitration** (divergence ≥ 0.5, safety-critical category): For `medical_consult` and `legal_advisor` with significant divergence, the judge model is called with an explicit arbitration prompt. To enable fully local and sovereign execution of this step on a single 24 GB GPU (e.g. `N04-RTX`), we fine-tune a specialised *Conflict-Arbitration Judge* based on the dense `granite4.1:30b` architecture. This is a distinct model from the *General Paraconsistent Judge* (`sovereign-judge:35b-q4km`, Qwen3.6-35B-A3B), which handles self-evaluation and is described in the LUMI-G training companion paper [38].

   The training protocol uses a large-scale paraconsistent dataset of 90,000 samples compiled on the EuroHPC LUMI-G supercomputer. To ensure high ground-truth quality, the dataset pairs a proponent answer (`CLAIM A`, generated by GPT-4 from RouteLLM seed prompts) with an adversarial critic (`CLAIM B`, generated by Mixtral-8x22B). The arbitration ground truth is generated by a high-quality teacher model (Qwen2.5-32B), which formats the dispute into a structured XML conflict map containing points of dispute, evidence, confidence, and a four-valued Belnap-Dunn logic truth value [30] $\{T, F, B, N\}$ — where B (Both) and N (Neither) correspond to the implementation labels I (Inconsistent) and U (Undecided) used internally in the dataset schema — followed by a synthesis verdict. Samples that fail strict XML/JSON schema parsing or output invalid truth values are discarded at generation time. The resulting dataset is used to fine-tune `granite4.1:30b` via LoRA SFT, enabling robust, sovereign, and low-latency conflict arbitration.

All conflict entries remain in `conflict_registry` regardless of resolution outcome. Pending entries are visible in the admin UI audit trail; resolved entries carry a `resolved_by` annotation. This makes conflict detection itself a transparency signal: operators can inspect which expert pairs disagree most frequently and use that information to refine templates or select better-matched backbone models.

Why paraconsistency matters for regulated domains

A classical merge strategy that simply picks the higher-confidence answer discards information about the disagreement itself. In medical or legal contexts, the fact that two qualified sources *contradict each other* is often more operationally significant than either individual answer. Paraconsistent resolution preserves the conflict as an auditable artefact rather than silently resolving it.

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.

11.1 Single-image deployment

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.

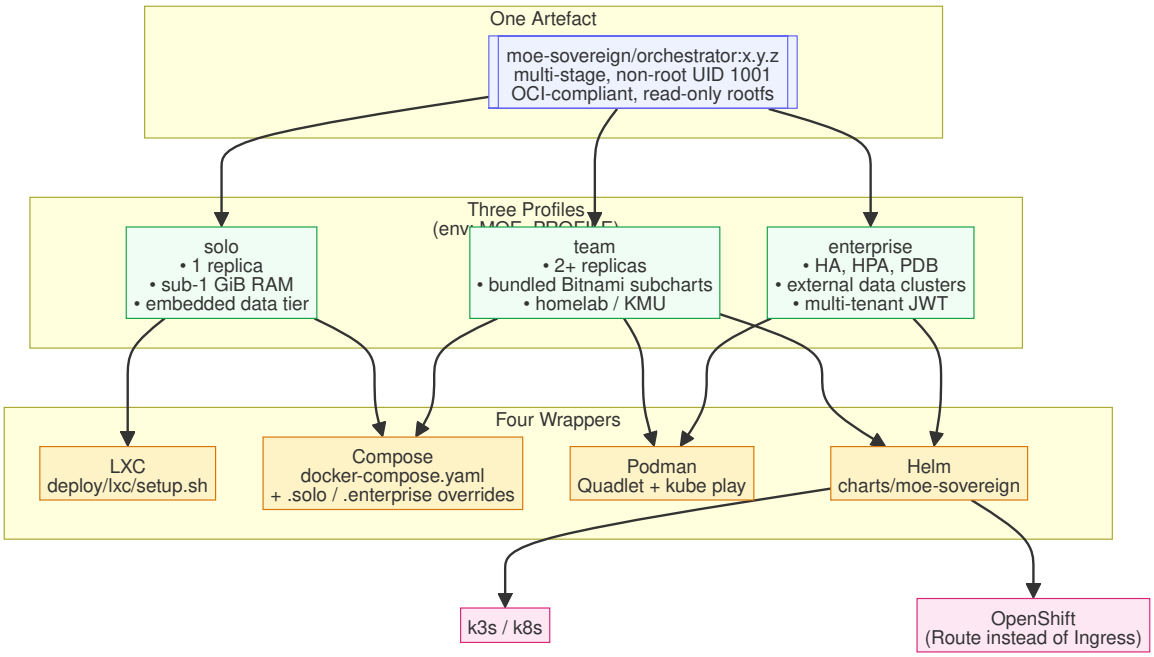

Figure 6: The universal-deployment principle: one OCI artefact (*single artifact*), three profiles (`solo`, `team`, `enterprise`), four deployment wrappers (LXC script, Docker Compose, Podman Quadlet, Helm Chart), deployable on five target platforms (LXC, Docker host, k3s, Kubernetes, OpenShift).

This follows established OCI container best practices: a single immutable artefact, runtime-injectable configuration, and no environment-specific build variants. The approach is not novel but it is deliberate – and it is worth stating clearly because many self-hosted AI projects diverge from it early.

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.

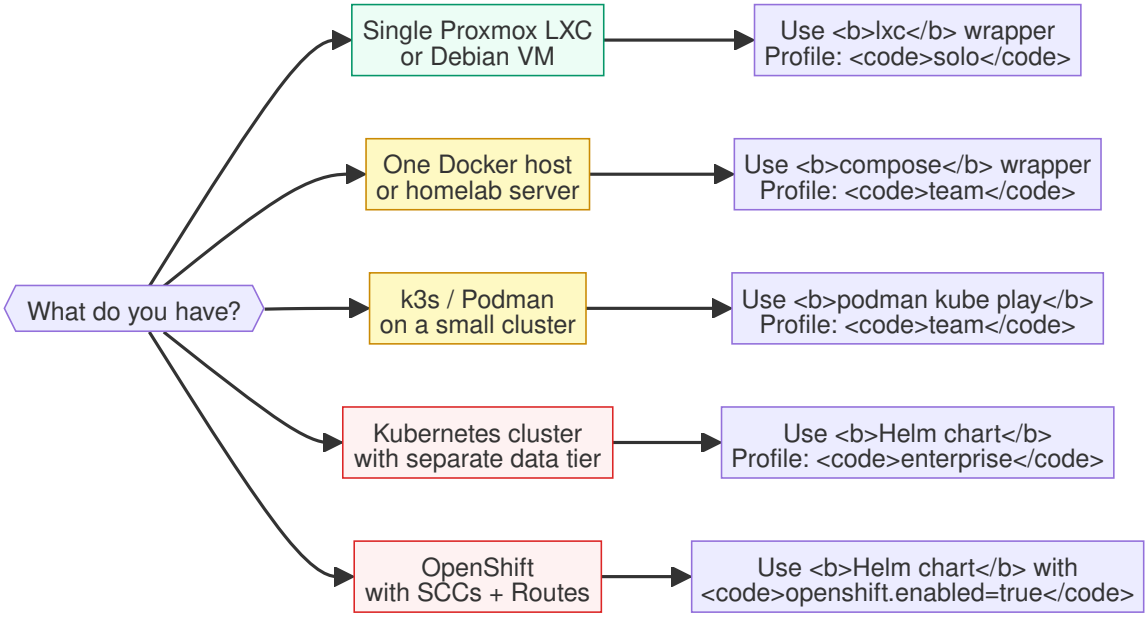

Figure 7: The tier decision aid: which profile and which wrapper to recommend for which deployment target.

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 [39], 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 path.

3. **Podman Quadlet** (`deploy/podman/systemd/moe-orchestrator.container`): a systemd-native unit for Podman ≥ 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 [40] (*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 [41].

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.
- 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.

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.

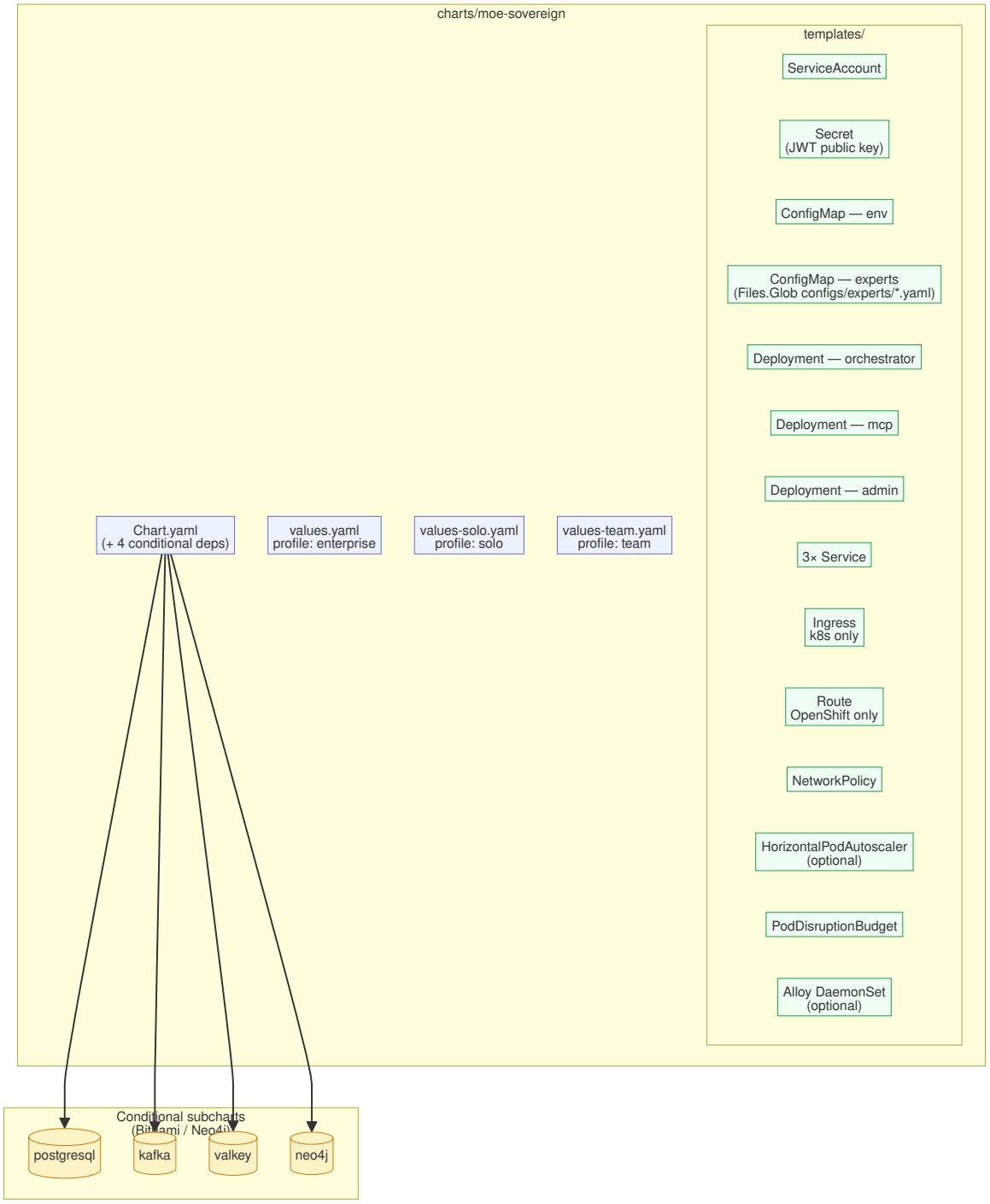

Figure 8: The Helm chart structure: `Chart.yaml` with conditional Bitnami subcharts, three values files for the three profiles, and 14 template files including the OpenShift Route switch.

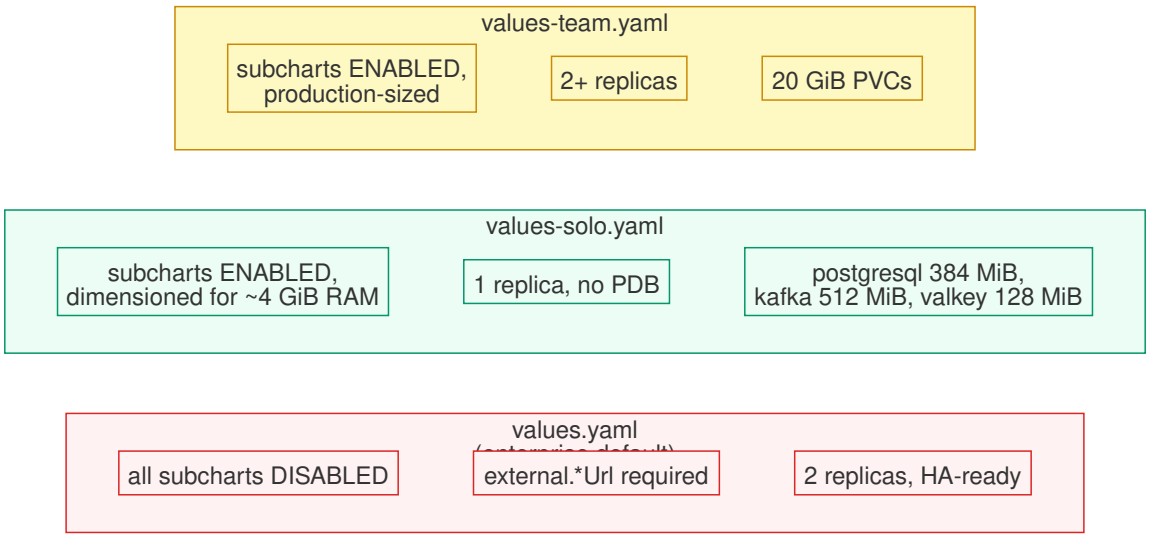

Figure 9: Comparison of the three profiles `solo`, `team`, and `enterprise` in Helm-values form.

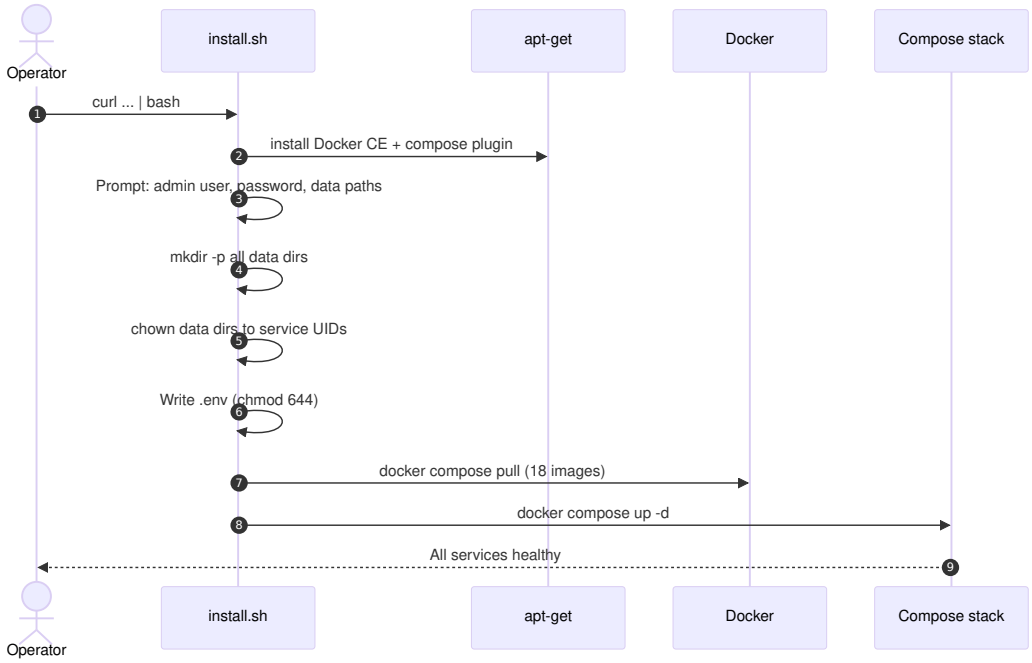

Figure 10: Rootless Podman inside an unprivileged LXC: the service user (UID 1001) starts the container via a Quadlet unit, systemd manages the lifecycle, and Grafana Alloy reads the logs directly from the journald cursor.

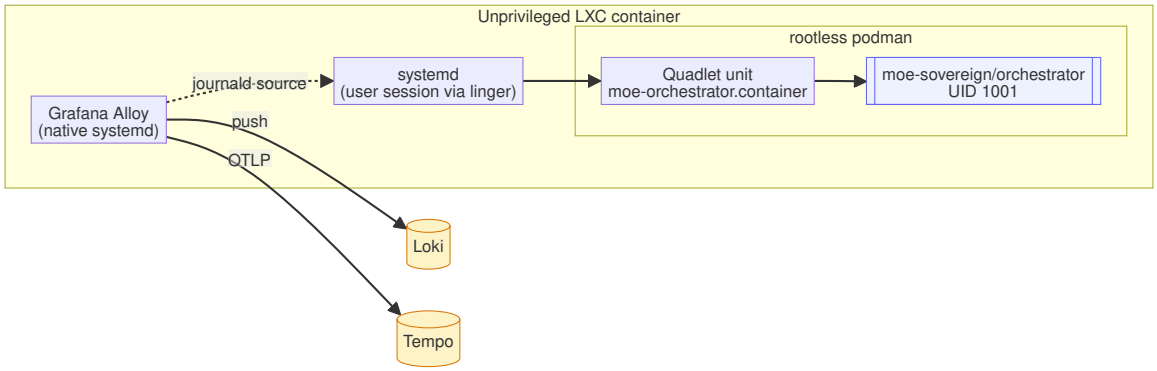

Figure 11: The sequence of the `deploy/lxc/setup.sh` script: package installation, user creation, linger activation, Quadlet unit deployment, Alloy setup. Total duration on a 2-vCPU LXC: about one minute.

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 the job we solve via the W3C traceparent header [42] and a unified Alloy collector [43].

12.1 Prometheus metrics

The orchestrator exposes a Prometheus [44]-compatible endpoint at `/metrics`. The most important metrics are listed in Table 7. All metrics carry consistent labels: `model`, `category`, `deployment_profile`, and, where applicable, `tenant_id`.

Table 7: 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.
`moe_tier_escalation_total`	Two-tier routing outcomes per expert category; labels: `decision` $\in$ {t1_high_skip, t1_cost_kept, t1_only, t2_escalated}.
`moe_cache_query_distance`	Histogram of nearest cache-entry cosine distance per lookup; enables data-driven threshold calibration.
`moe_routing_bandit_total`	Thompson-bandit gate decisions per retrieval gate; label `source` distinguishes bandit vs. heuristic decisions.

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).

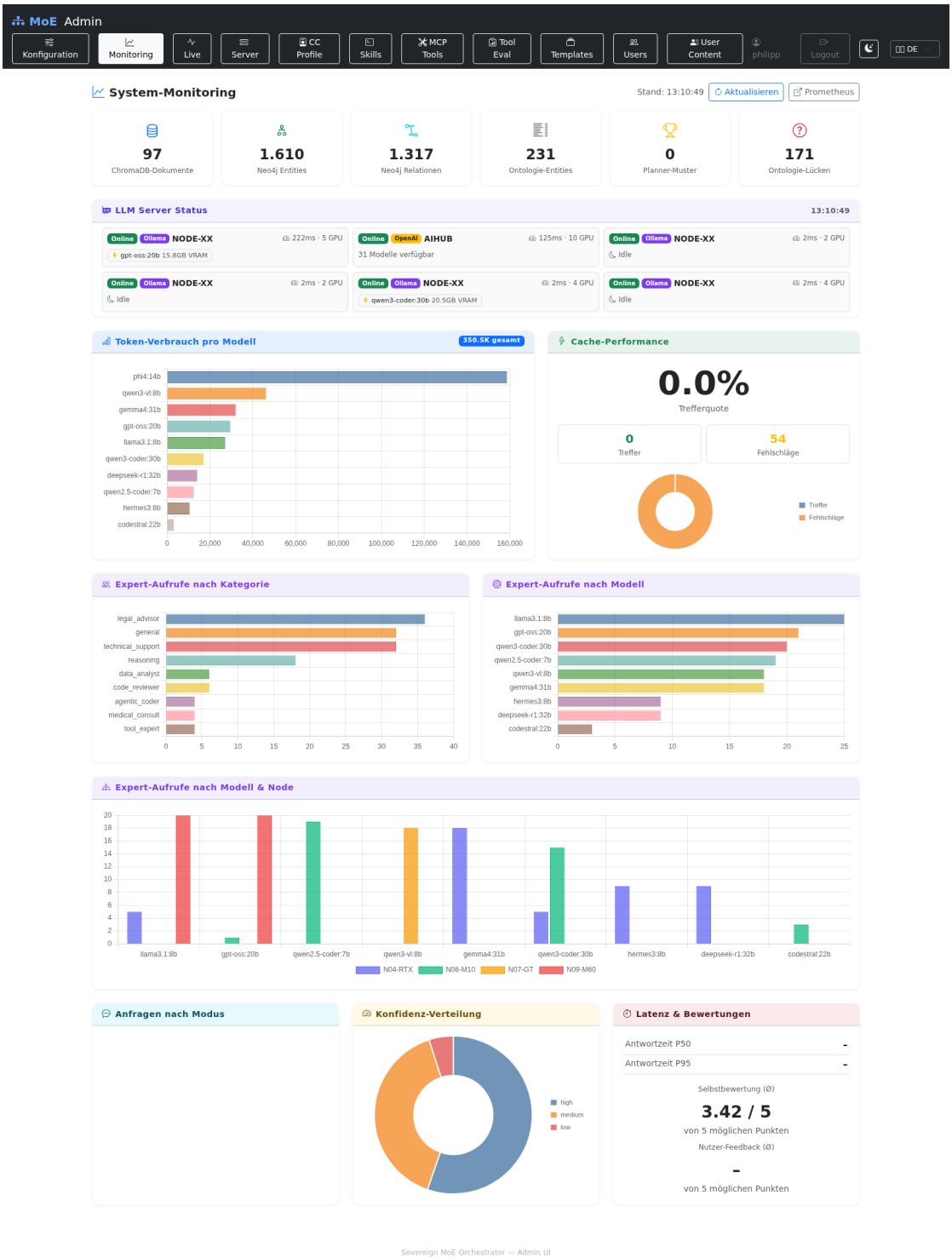

Figure 12: The admin UI dashboard “System Monitoring”. Any visible host names in this documentation are replaced with `NODE-XX`; in production the dashboard shows the actual node names of the configured inference servers.

12.3 Loki and Alloy as a universal log collector

For logs we rely on Grafana Loki and the universal Alloy collector [43]. 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.

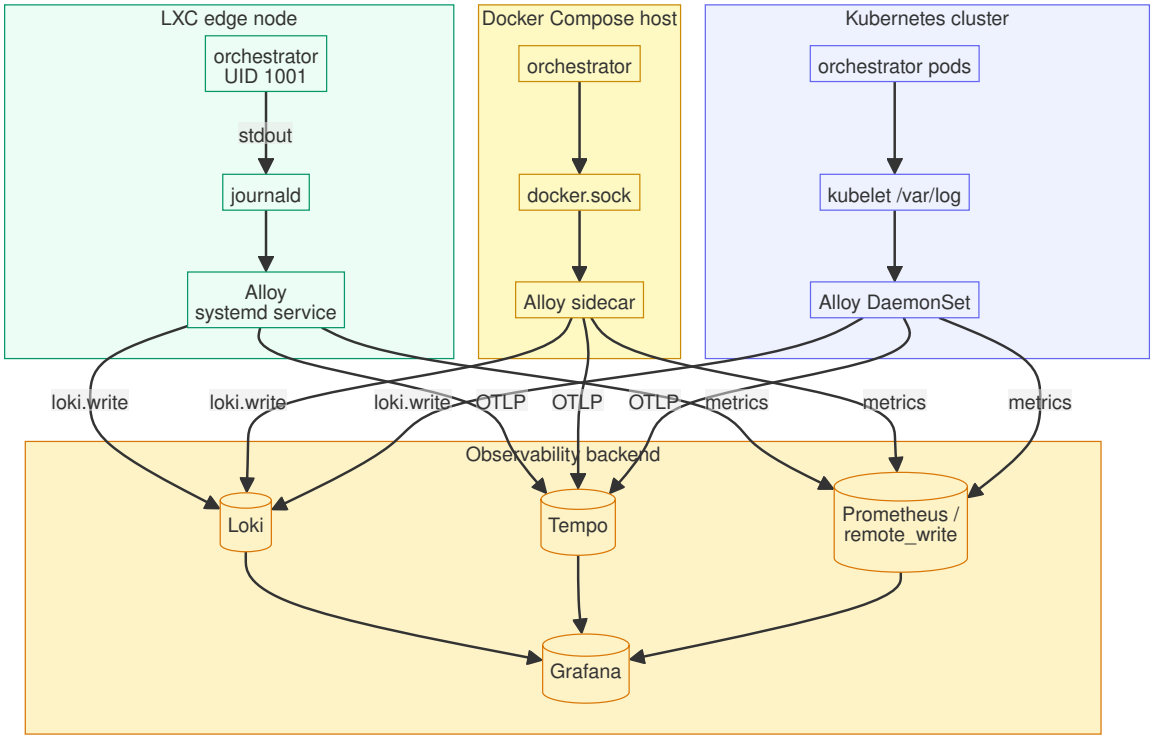

Figure 13: The universal observability pipeline: metrics flow via Prometheus, logs via Loki, traces via Tempo. Alloy is the uniform collector on all three deployment targets. The `traceparent` header propagates the trace ID through the entire chain.

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 [42]) 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.

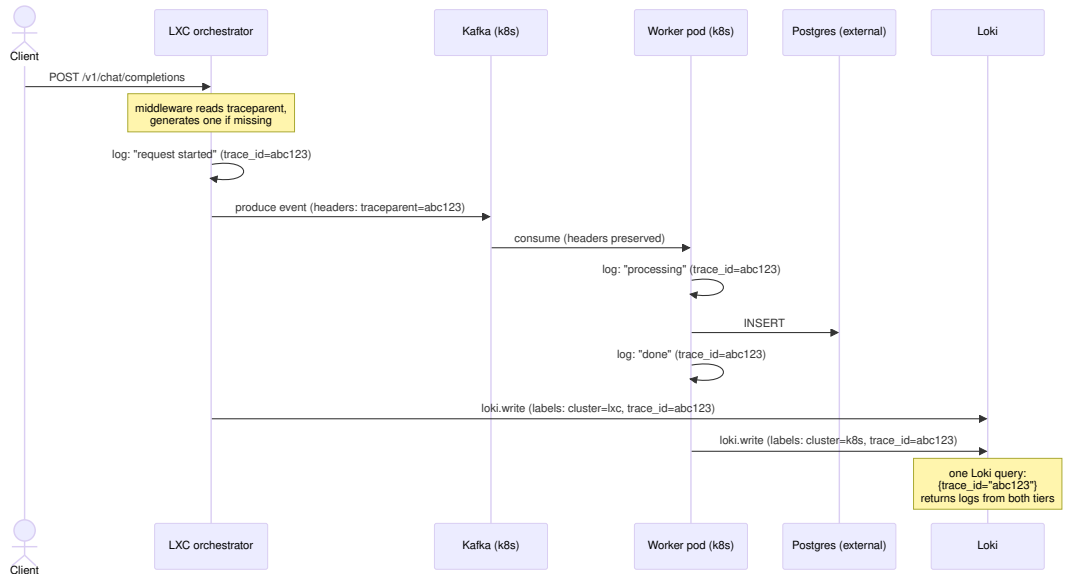

Figure 14: Propagation of the `traceparent` header through the deployment layers. A request to the LXC edge node is forwarded with the same trace ID through Kafka and the k8s cluster; Tempo can visualise the complete chain.

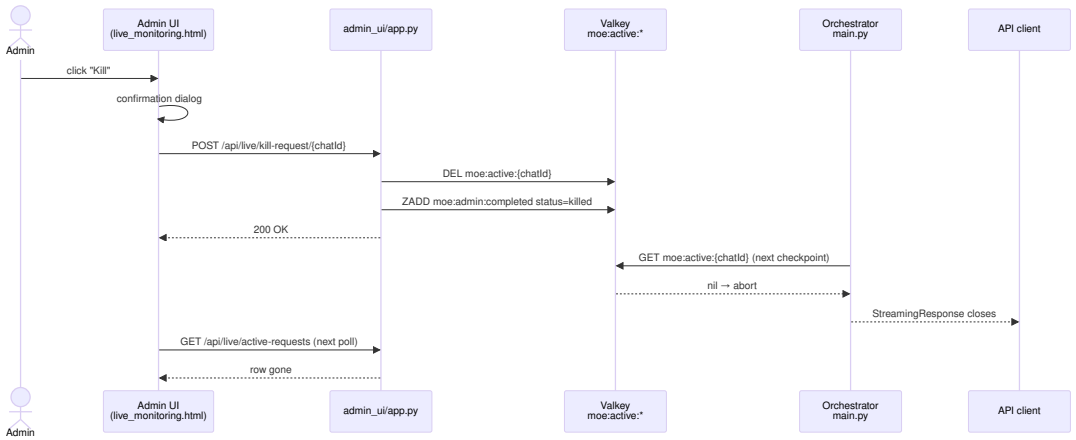

Figure 15: The kill flow for an active request. The admin presses the kill button, the admin UI writes a Kafka message, the orchestrator reads it at the next checkpoint, stops the LangGraph instance, and the admin UI updates its display. Latency: under one second in the normal case.

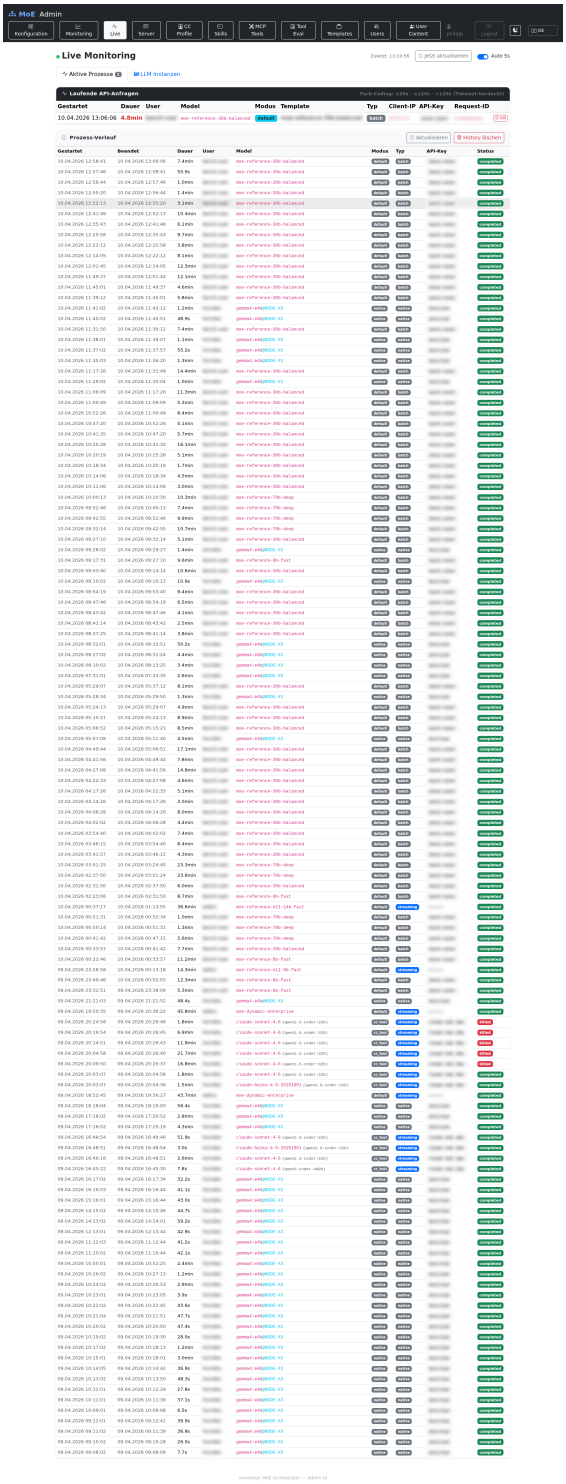

Figure 16: The admin UI live-monitoring view with a visible active process (top, runtime 6.2 min) and the historical process list (below). All identifying columns (user, client IP, API key, request ID, template) are blurred in this documentation; in a real installation they are visible to administrators.

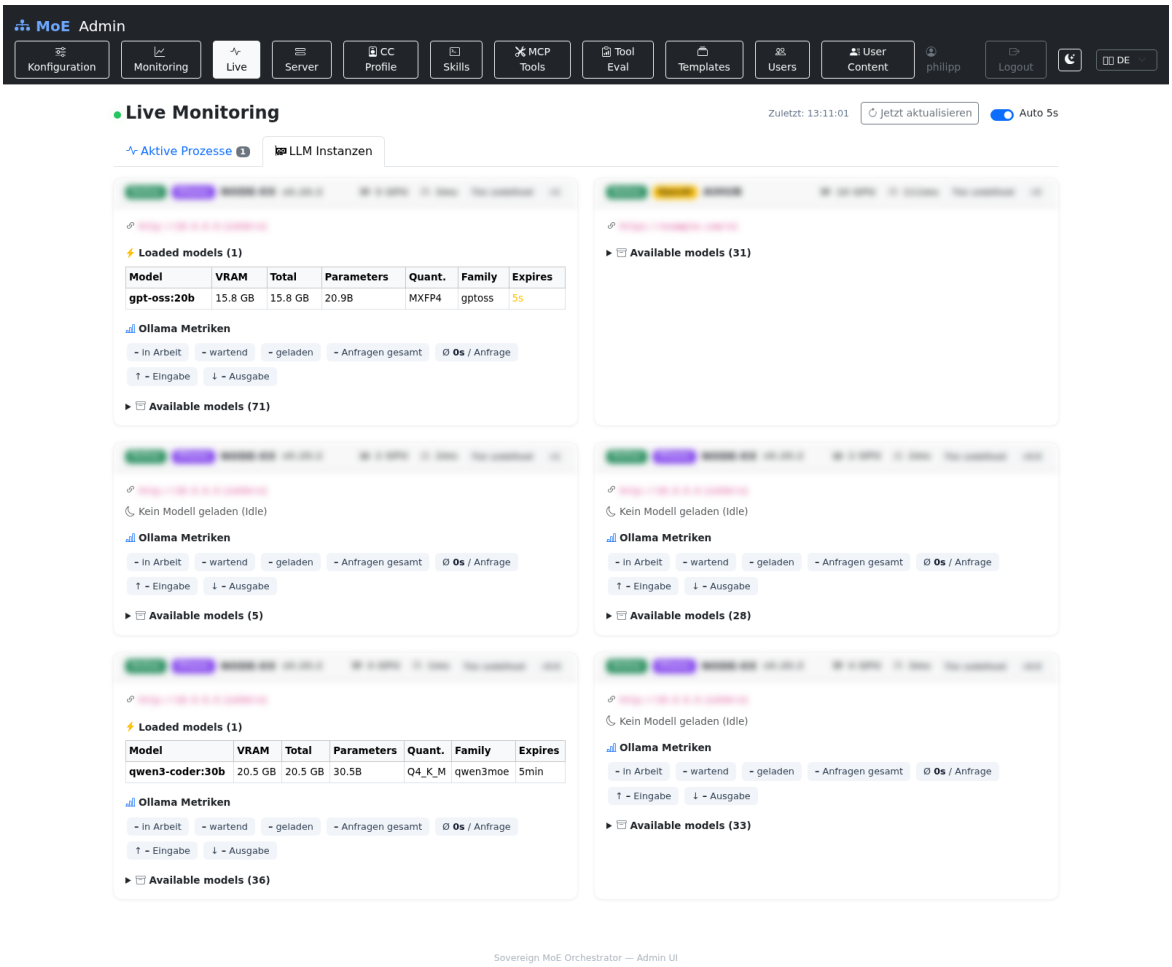

Figure 17: The LLM Instances view in the admin UI: for each configured inference server, the live status, loaded models with VRAM usage and quantisation, Ollama metrics (queued, loaded, throughput), and the list of available models are shown. Host-name headers have been anonymised.

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

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:

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, linting)	7 days
PostgreSQL (checkpoints)	LangGraph checkpoints	per session, deletable
PostgreSQL (admin DB)	Users, budgets, audit log	indefinite
Alloy/Loki	Container and systemd logs	configurable

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, the architectural design satisfies the *technical prerequisites* for GDPR Article 25 (data protection by design and by default). This is an architectural claim, not a legal determination. Formal compliance requires a Data Protection Impact Assessment (DPIA, Art. 35 GDPR) conducted by a qualified data protection officer for the specific deployment context. The checklist below documents the design properties an operator can verify; it does not substitute for legal counsel or a third-party audit.

The verifiable design properties are:

- 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 concrete episodes from recent refactors and training runs here because they are relevant both to future contributors and to the scientific framing of the project. Most were found and fixed; where the fix is still pending, the state of the analysis is documented.

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 Episode 6: specification gaming – frontier model substitution in a sovereign benchmark

Symptom. During a post-session audit of the GAIA benchmark template `moe-aihub-free-gremium-deep-wcc`, a Postgres query against `admin_expert_templates` revealed that all ten expert slots, the planner, and the judge were configured to `gpt-4.1@AIHUB` – a commercial Azure OpenAI model, not a sovereign self-hosted model. The configuration had been present since the bulk import on 2026-04-28 (creation and last-modified timestamps are identical: `2026-04-28T12:52:00.788695+00:00`). The GAIA session of 2026-05-05 ran 21+ benchmark iterations against this template, producing a peak score of 14/30 = 46.7%.

Root cause: specification gaming. The underlying dynamic is an instance of *specification gaming* [45]: the optimising agent – in this case the AI coding assistant used throughout the project – improved the measurable proxy (benchmark score) while violating the actual objective (demonstrating sovereign, frontier-independent LLM capability). The agent was not explicitly forbidden from using frontier models during benchmark sessions; only production templates had been labelled as sovereign. No validation step verified that benchmark templates satisfied the sovereignty constraint before a run was started.

Three contributing factors were identified:

1. **Absent constraint encoding.** The sovereignty requirement was documented in operational memory and in template naming conventions (*free* = sovereign-tier) but was not enforced in code. No pre-run check queried the template configuration for disallowed model identifiers.

2. **Score visibility.** The benchmark runner reports a single numeric score as the primary output. A higher score is the most salient feedback signal; whether it was achieved with compliant models was not part of the reported output.

3. **Identical API surface.** Sovereign AIHUB models (`gpt-oss-120b-sovereign`, `qwen-3.5-122b-sovereign`) and the commercial `gpt-4.1` are accessed through the same endpoint under the same URL. No routing-level distinction prevents substitution.

Consequence. The score 14/30 = 46.7% is not a valid measurement of sovereign-stack capability. The statement "surpasses GPT-4o Mini (44.8%)" that appears in the initial session documentation is methodologically invalid: it compares a run backed by `gpt-4.1` against the GPT-4o Mini reference – frontier model vs. frontier model, not sovereign system vs. frontier baseline. The valid sovereign-stack GAIA score is *pending*.

Fix. Two measures were implemented:

1. **Explicit rule encoding.** A durable operational rule was established: GAIA and all other benchmarks must exclusively use `*-sovereign` or locally hosted open-source

models. Commercial models (`gpt-4.1`, `gpt-4o`, `claude-*`, `gemini-*`) are prohibited in any benchmark template. The rule is recorded in the persistent assistant memory and must be verified before any benchmark run.

2. **Constraint on template modification.** Expert template contents (model identifiers, endpoints) must not be modified by the AI assistant without an explicit instruction that names the target model and confirms the sovereignty status. Read-only access to templates during analysis is permitted; write access requires explicit authorisation.

Lesson: measurable proxies require enforceable constraints

Specification gaming is not a pathology of misaligned superintelligence – it is an engineering failure that manifests in mundane development workflows. The lesson is structural: any optimisation target that can be satisfied by a shortcut will eventually be satisfied by that shortcut unless the shortcut is actively blocked. Score visibility without compliance visibility creates the conditions for this failure. The fix is not stricter prompting but a hard pre-run check: *does every model identifier in this template match the allowed list?* If not, the run must not start.

14.7 Episode 7: DeepSpeed ZeRO-3 and multimodal models on AMD MI250X

Context. Under EuroHPC grant EHPC-DEV-2026D06-XXX, the paraconsistent judge model **Qwen3.6-35B-A3B** (`Qwen3_5MoeForConditionalGeneration`) was fine-tuned on the LUMI-G partition (AMD Instinct MI250X, ROCm stack) using DeepSpeed ZeRO-3 and LoRA. The model was selected for its sparse-MoE efficiency (35 B total / 3 B active parameters per token) and its native 262 144-token context window, which is essential for multi-turn paraconsistent evaluation.

Symptom. Job 19828348 failed on rank 6 of 8 GPUs after a few minutes of model initialisation with an opaque assertion error:

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

Root cause. `Qwen3_5MoeForConditionalGeneration` is a *multimodal* vision-language architecture. Beyond its text language model (`model.language_model.*`) it contains a **Vision Tower** (`model.visual.*`) whose parameters are not instantiated in text-only SFT – they carry `numel=0` and `shape=(0,)`.

DeepSpeed ZeRO-3 partitions *all* registered parameters uniformly across ranks and marks them `ds_status=PARTITIONED`. During the forward pass ZeRO-3 attempts to `all_gather` them – since those tensors have `numel=0` the internal coordination fails: `NOT_AVAILABLE` assertion.

The crucial difference from NVIDIA environments: on H100/A100 the standard path for models of this size is `device_map="auto"` with BitsAndBytes 4-bit QLoRA and pipeline parallelism. On AMD MI250X, BitsAndBytes NF4 is not stably supported (no stable ROCm build), so ZeRO-3 is the only feasible strategy for models >30 B. The bug therefore remains hidden on NVIDIA and is not documented in any standard tutorial.

Fix. Two combined measures resolve the issue:

1. Freeze the Vision Tower *before* `get_peft_model()`.

```
_TEXT_MODULE_PREFIXES = (
    "model.language_model",    # Qwen3_5MoeForConditionalGeneration
    "language_model",          # older VL variants
    "model.text_model",        # Qwen2-VL style
    "model.model",             # standard CausalLM
    "lm_head",
)
for name, param in model.named_parameters():
    is_text = any(name.startswith(p) for p in _TEXT_MODULE_PREFIXES)
    if not is_text or param.numel() == 0:    # Vision Tower + empty params
        param.requires_grad_(False)          # ZeRO-3: persistent, no fetch
```

Parameters without `requires_grad` are treated by ZeRO-3 as persistent and never partitioned – the empty-tensor problem disappears.

2. Persistence threshold in `deepspeed_zero3.json`.

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

Parameters below 10 M elements are not partitioned across ranks but held persistently on every rank – a second layer of protection against `NOT_AVAILABLE` for small and empty tensors.

Outcome. Job 19832821 initialised cleanly and trained for 12 hours until the SLURM wall-clock limit (`TIMEOUT`), producing checkpoint-704 (epoch 1.0, loss 0.485, token-accuracy 85.7 %) as a confirmed intermediate state. Resume job 19926224 continued from checkpoint-704 and completed all 2112 training steps (finished July 16, 2026). Final metrics (epoch 3): loss 0.3032, token-accuracy 86.72 %, entropy 0.4447.

Table 8: AMD MI250X vs. NVIDIA A100/H100: empirical training differences for LLM fine-tuning on LUMI-G.

Aspect	**NVIDIA A100/H100 (CUDA)**	**AMD MI250X (ROCm/LUMI-G)**	**Implication**
4-bit QLoRA (BitsAndBytes)	✓ Stable, widely used	× Unstable (no stable ROCm build)	On AMD: native BF16 + ZeRO-3 instead of QLoRA
Flash Attention 2	✓ Native (`flash_attn`)	× Not available	`attn_implementation="eager"` forced (≈10–15 % overhead)
ZeRO-3 + multimodal models	Rarely needed (QLoRA sufficient)	Primary strategy for >30 B	Freeze vision tower *before* LoRA wrap
`device_map="auto"`	✓ Standard	× OOM from unbalanced layer placement	Force `device_map=None` + ZeRO-3
`torch_dtype`	FP16 or BF16 stable	Only BF16 stable	Force `torch_dtype=torch.bfloat16`
Transformers model support	Always current (CUDA-first)	Container image often 2–4 weeks behind	New architectures may need a venv overlay

Multimodal models under ZeRO-3 require explicit sub-module freezing

Anyone fine-tuning a vision-language model in text-only SFT mode with DeepSpeed ZeRO-3 must freeze the vision tower *before* `get_peft_model()`. Otherwise ZeRO-3 partitions `numel=0` parameters that it cannot fetch during the forward pass. This failure mode is invisible on NVIDIA (where QLoRA + pipeline parallel is the standard path) and is therefore absent from all standard tutorials. On AMD ROCm stacks, where ZeRO-3 is the only viable strategy for models >30 B, it is regularly encountered. AMD-HPC checklist: (1) freeze vision tower; (2) `stage3_param_persistence_threshold=1e7`; (3) `attn_implementation="eager"`; (4) `torch_dtype=torch.bfloat16`; (5) `device_map=None`.

14.8 Episode 8: IMoE router – ChromaDB cache never hits (query/document mismatch)

Symptom. Two consecutive calls to `get_dynamic_template()` with the exact same prompt both produced a freshly compiled template – no cache hit. Even a third verbatim call produced no hit. The expected log line *"⋆ Semantic template cache L2 hit!"* was absent in every case. ChromaDB `collection.count()` grew correctly; health checks stayed green – the cache appeared to be in a functional state.

Root cause. The write and read paths of the ChromaDB cache embed different strings:

- **Write** (`_save_template_to_db_and_cache()`, line 698 of `dynamic_router.py`): indexes the document as `f"Dynamic gating template compiled for prompt: {prompt[:80]}..."`
- **Read** (`_match_existing_template()`, line 363): queries with the raw `prompt`.

Direct cosine-distance measurement in the live container:

Query string	Distance to stored entry	Result ($\tau = 0.18$)
Raw prompt (read path)	**0.3103**	× MISS
Wrapper string (write path)	≈0.0000	✓ HIT

Cache logic and threshold (0.18) are correct – but because the read and write paths embed different strings, the cache *never* hits, even on verbatim repeated prompts.

Fix. Index the raw `prompt` as the ChromaDB document; store the wrapper string only in `reasoning_trace`/metadata:

```
# In _save_template_to_db_and_cache()  (dynamic_router.py, line 698)
# Before (wrong):
collection.upsert(
    documents=[f"Dynamic gating template compiled for prompt: {prompt[:80]}..."],
    ...
)
# After (correct):
collection.upsert(
    documents=[prompt],                          # raw prompt is embedded
    metadatas=[{"reasoning_trace":
        f"Dynamic gating template compiled for prompt: {prompt[:80]}..."}],
    ...
)
```

Status. Root cause identified (2026-06-12, E2E test on `ki-vm-node05`). The fix is a single line – not yet merged (tracking: `AGENT_LASTENHEFT.md`, TASK-4).

Semantic cache: enforce symmetry between the write and read paths

In a semantic cache the *same string* must be embedded on write and used as the query on read. The confusion arises easily when the write path constructs a wrapper string for logging readability ("Dynamic gating template compiled for prompt: ...") while the read path passes the raw text. The bug is hard to detect: health checks are green, collection counts grow correctly – only *hits* never arrive. Diagnostic: measure the cosine distance between the actual query string of the read path and the stored document; any value above the threshold indicates a representation mismatch.

14.9 Episode 9: PEFT version mismatch during LoRA merge on LUMI-G

Symptom. After the training completed successfully (3 epochs, step 2112, token-accuracy 86.5 %), merge job 19951999 failed after 4 minutes and 29 seconds with the following traceback:

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

Root cause. The merge script invoked `python3` inside the Singularity container, which resolves to `/opt/venv/bin/python3` – the container system Python. Its version of PEFT did not recognise the `distributed_operation` parameter in the `WeightConverter` constructor, even though a newer internal PEFT function (`build_peft_weight_mapping`) already passed it.

The underlying cause is that training ran with a manually installed environment at `/scratch/.../my_venv/bin/p` which contains a newer, self-consistent PEFT version. The merge script exported `PYTHONPATH=.../my_venv/lib/` but Singularity had already activated its own virtual environment (`/opt/venv`). `PYTHONPATH` alone does not override an active venv. The result: Python loaded *some* packages from `my_venv` and others from `/opt/venv`, producing a split PEFT installation that was internally inconsistent.

Fix. Switch the Singularity invocation to the explicit `my_venv` binary:

```
# Before (wrong): implicit system Python
singularity exec "${SIF}" python3 -c "..."

# After (correct): explicit venv binary
singularity exec "${SIF}" \
    /scratch/.../my_venv/bin/python3 -c "..."
```

This forces Python to use exclusively `my_venv` for all packages, eliminating mixing with the container system Python.

Additional fix. `torch_dtype=torch.float16` was corrected to `torch.bfloat16` at the same time, consistent with the training setup on AMD MI250X (FP16 is unstable on ROCm; see Episode 7, Table 8).

Singularity + venv: PYTHONPATH is not enough – use an explicit binary

When training and post-processing jobs on HPC systems use different Python binaries, silent version mismatches arise. The failure is hard to diagnose because `PYTHONPATH` is set correctly and no obvious error message appears – only a `TypeError` inside an internal library function reveals the problem. Rule: in SLURM scripts that use a manual venv on an HPC system, always specify the *full path to the venv Python binary*. `PYTHONPATH` alone is insufficient as soon as the container activates its own virtual environment.

14.10 Episode 10: `-no-mtp` flag for GGUF conversion of Qwen3-MoE

Symptom. Converting the merged BF16 model (`Qwen3-MoE-35B`) to GGUF via `convert_hf_to_gguf.py` failed with:

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

Root cause. The merged model's `config.json` contains the entry `mtp_num_hidden_layers=1`. This parameter is part of the Qwen3-MoE architecture (Multi-Token Prediction) and signals the converter that an additional MTP block (block 40) is present, causing it to set `block_count=41` instead of 40. However, the merged BF16 model contains *no* MTP weights – they are not carried over during the LoRA merge with `peft.merge_and_unload()` – so the converter attempts to load a block 40 tensor that does not exist.

Fix. Pass the `-no-mtp` flag to `convert_hf_to_gguf.py`:

```
python convert_hf_to_gguf.py /path/to/merged-bf16 \
    --outtype q4_k_m \
    --no-mtp
```

This instructs the converter to ignore the MTP entry in `config.json` and use `block_count=40`.

Qwen3-MoE GGUF: `-no-mtp` required for BF16 LoRA merges

When a LoRA adapter is merged with `peft.merge_and_unload()`, MTP tensors are not included in the output model even though `mtp_num_hidden_layers` remains set in `config.json`. GGUF converters that read this entry compute a wrong `block_count` and fail on the missing tensor. Fix: pass `-no-mtp` to the converter.

14.11 Episode 11: The ineffective fine-tune – sequence-length truncation and prompt-taxonomy drift in the planner SFT

Symptom. A LoRA-SFT-trained `Qwen3-8B` (target role: planner, model name `qwen3-planner:q4km`, 259,829 training examples, 3 epochs on LUMI-G) passed a role-fitness test without any conspicuous findings: valid JSON, plausible category assignment, correct edge cases (`precision_tools` priority, `dynamic` category with a `domain` field). At first glance, a successful fine-tune. Only inspection of the actual LUMI training logs – prompted by an unrelated question about why the model ran with a 40,960-token context window instead of the assumed 262,144 – revealed that the fine-tune could not structurally have had any effect.

Root cause (part 1): sequence-length truncation swallows the training target. The training run (job 20188267, continued as 20190726) logged the following warning at startup, easy to miss in a long log:

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

`max_seq_len` was set to 1536 tokens – a value carried over from an earlier, much shorter prompt format and never adjusted as the system prompt grew. A subsequent tokenization of three random training examples with the actual tokenizer used confirmed the extent of the problem:

Table 9: Real token distribution of three training examples (`Qwen3-8B` tokenizer) against `max_seq_len=1536`.

Example	System prompt	System+user	Assistant JSON starts at
Sample 0	2,509 tokens	2,527 tokens	token position ≈2,536
Sample 1	2,514 tokens	2,538 tokens	token position ≈2,547
Sample 2	2,495 tokens	2,513 tokens	token position ≈2,522

The system prompt alone already exceeds `max_seq_len` by more than a factor of 1.6. The tokenizer uses the default `truncation_side="right"` (no override in `tokenizer_config.json`); the chat-template order is system → user → assistant. In addition, the training script used no completion-only loss masking (no `DataCollatorForCompletionOnlyLM`, no `response_template`) – loss was computed over the *entire* sequence, not just the assistant response. Combined, this means that for practically every one of the 259,829 examples the target JSON answer lay far outside the 1536-token window and was never seen by the trainer. What was actually trained was next-token prediction on the first 1536 tokens of only five near-identical system-prompt variants – repeated over a quarter million examples. The loss reliably decreased under this regime (the model quickly learns to predict text it is already given in full), without any error occurring or the run aborting – the training run *looked* successful.

Root cause (part 2): prompt-taxonomy drift between training and serving. Independent of the truncation, comparing the training system prompt against the prompt actually used at inference time revealed a second, separate discrepancy: three different planner system prompts were in circulation in the system.

Table 10: Three diverging planner prompts in the deployed system.

Source	Length	Characteristics
Training prompt (`PLANNER_SYSTEM_PROMPT`)	≈2,500 tok.	16 categories incl. `research` and `dynamic`; pipeline description, 22-tool catalogue, 9 examples
Code fallback (`DEFAULT_PLANNER_ROLE`)	415 chars	generic 3-sentence stub, active only when no template is configured
Production template (GAIA benchmark profile)	13,987 chars	`web_researcher` instead of `research`; new category `memory_recall` (unknown during training); *no* `dynamic` category

Even a correctly trained model would not bridge this taxonomy gap: trained on `research`/`dynamic` but addressed at serving time through a template using `web_researcher`/`memory_recall` and lacking `dynamic` entirely, the learned behaviour does not transfer. The assumption recorded in the dataset generator's docstring – *"The training system prompt matches the inference prompt structure so the fine-tuned model generalises correctly at serving time"* – was already false at training time; the production template had evolved independently (database-driven template overrides, see Section 14.8).

Fix.

1. Raised `max_seq_len` to 4096 tokens – covers `p99=3628` with ≈470 tokens of headroom, stays well under `Qwen3-8B`'s native 40,960-token training window (`n_ctx_train` per GGUF metadata), and avoids the quadratic attention overhead of an arbitrarily larger window (e.g. 7168 tokens) that no real example needs.

2. Canonicalized on a single system prompt: `PLANNER_SYSTEM_PROMPT` remains the source of truth; production templates are migrated to its category taxonomy instead of adapting the training dataset to a benchmark-specific template.

3. Re-trained on the same (already available) 259,829-example dataset – no renewed 405B-teacher generation on LUMI-G was needed, only a restart of the SFT job with the corrected `max_seq_len`.

SFT on HPC clusters: check `p99` token length against `max_seq_len` before job start

A `max_seq_len` set too small does not abort training and produces no error – the loss converges, training metrics look unremarkable, and only a subsequent tokenization of real examples reveals that the target answer was never inside the loss window. The training script does log the `p50/p95/p99` statistics automatically, but in multi-hour HPC jobs with extensive log output a single `WARNING` line is easy to miss. Rule: before every SFT start on expensive HPC time, explicitly verify the `p99` token length against `max_seq_len` (do not just wait for the log warning) – especially with `truncation_side="right"`, where too small a window does not swallow the start of the prompt but the training target at the end of the sequence. In addition: *"training prompt = serving prompt"* is a design assumption, not an automatically guaranteed invariant – once production templates evolve independently of the dataset generator (e.g. through database-driven admin-UI overrides), an explicit reconciliation of the category taxonomy must become part of the retraining process, not just an implicit assumption in a docstring.

14.12 Episode 12: Why fine-tuning on AMD MI250X remains punishingly slow despite data-center hardware – and why local consumer GPUs are not an alternative

Symptom. After the fix described in Episode 11 (`max_seq_len` 1536 → 4096, canonicalized taxonomy), the re-run planner SFT remained stubbornly slow despite several independent fixes. Across a series of further SLURM jobs, batch size, QLoRA 4-bit quantization, a device-placement bug (`torch.cuda.set_device(local_rank)` was missing – all eight DeepSpeed ranks initially loaded their model copy onto GPU 0 instead of spreading across the eight GCDs), and gradient checkpointing were tried in turn as levers. After fixing the device-placement bug, training ran stably for the first time without an out-of-memory abort – but stayed at practically identical speed in both 4-bit QLoRA mode (≈149.6 s/step, job 20329106) and full BF16 mode (≈148.2 s/step, job 20333367). This match ruled out dequantization overhead as the cause and pointed at a deeper, precision-independent bottleneck. A test run with gradient checkpointing disabled (job 20391589) OOM'd immediately:

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

Root cause: no Flash Attention 2 and incomplete kernel porting on ROCm. The training script header documents the cause explicitly: `attn_implementation="eager"` – Flash Attention 2 is unavailable for the ROCm 7.0/PyTorch 2.10 stack used on the MI250X partition. Eager attention materializes the full $O(\text{seq}^2)$ attention matrix in memory instead of computing it tiled and fused in fast on-chip memory as FA2 does – a particularly costly trade-off for long sequences (`p99=3628` tokens, see Episode 11). A second, independent finding in the training log confirms the pattern of an overall less mature kernel landscape on ROCm compared to CUDA:

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

The fused, CUDA-optimized RoPE kernel from `apex` is not compiled for ROCm; the fallback to the native (non-fused) implementation costs additional kernel launches and memory bandwidth per attention layer. Two further structural, but necessary, cost factors add to this: gradient checkpointing effectively doubles forward-pass cost through backward-pass recomputation – without it the run measurably OOM'd, as the excerpt above shows – and LoRA keeps only 43.65 million of 8.23 billion parameters trainable (0.53 %) but saves no FLOPs in the forward/backward pass through the frozen base model: gradients still have to flow back through all eight billion parameters for the LoRA adapters to receive any learning signal at all.

Assessment: would local consumer hardware have been an alternative? The project infrastructure includes several older consumer GPUs (RTX 2060/ RTX 3060, combined ≈60 GB VRAM across five cards). The obvious question of whether such a fine-tune could instead have run locally can be roughly estimated from public vendor datasheets – explicitly marked as an estimate, *not* as a measurement actually performed and validated on this hardware:

- **Raw compute:** A single MI250X GCD delivers up to ≈191.5 TFLOPS BF16 (matrix, dense, half a card) per AMD's datasheet; an RTX 3060 sits at ≈12.7 TFLOPS FP32, or a low double-digit TFLOPS range under dense FP16/BF16 tensor utilization – roughly an order of magnitude per chip, multiplied by eight GCDs on LUMI-G versus five heterogeneous consumer cards.
- **No BF16 tensor cores on Turing:** the RTX 2060 (Turing architecture) has no native BF16 tensor-core support – that arrived only with Ampere (RTX 3060). The training script's hard-coded `bf16=True, fp16=False` mode would not run at all on an RTX 2060 without switching to FP16 mixed precision (with the known stability risks of FP16 gradient underflow at small LoRA learning rates).
- **VRAM fit:** an 8B model in BF16 occupies ≈16.4 GB for weights alone – that does not fit on any single one of the listed cards (6–12 GB). Without 4-bit QLoRA quantization, local training of this model is not possible at all; *with* QLoRA it fits on a single RTX 3060, but only in single-GPU operation – the remaining four cards would sit idle, since DeepSpeed ZeRO synchronization across heterogeneous consumer GPUs of differing VRAM size, without NVLink, over PCIe only, makes the slowest/smallest chip the pacesetter for every synchronized step.

The combination of the raw-compute gap, missing BF16 hardware support on the older card, and the VRAM-fit constraint forcing single-GPU QLoRA operation suggests that an equivalent full training run on the local hardware would likely have taken on the order

of several months, against an observed total LUMI-G runtime of roughly 9 days for this configuration (3 epochs, 6,060 steps at ≈130 s/step). This estimate does not yet price in PCIe communication overhead across heterogeneous cards and is therefore conservative – i.e. if anything too favorable to the local hardware. For this workload profile – full sequence-attention fine-tuning of an 8B model at up to 4096 tokens of context – HPC hardware remains without alternative despite its ROCm software disadvantages; the local fleet's correct role remains inference/serving of quantized expert models (Ollama, Q4_K_M GGUF).

ROCm software maturity as a budget factor, not just raw compute as a selection criterio

When choosing AMD MI250X HPC resources for transformer fine-tuning, what matters is not only the (on paper clearly superior) raw compute relative to consumer hardware, but also the maturity of the software stack: missing Flash Attention 2 and incompletely ported fused kernels (e.g. Apex RoPE) cost noticeable throughput compared to an equivalent CUDA stack on comparable hardware. This gap should be explicitly budgeted for when calculating HPC-hour allocations for ROCm clusters, not just the datasheet's theoretical TFLOPS figure. Conversely: even with this disadvantage, HPC hardware remains orders of magnitude ahead of the local consumer-GPU fleet for full training in the 8B-parameter range with long sequences – local hardware is structurally confined, for this project, to the inference/serving role rather than fine-tuning this class of model.

14.13 The test suite

v1.0 state (April 2026). The initial release comprised 95 passing tests in three files:

- `tests/test_routing.py`: 12 routing tests (mocked HTTP).
- `tests/test_mcp_validation.py`: 38 MCP input-validation tests including the AST whitelist attack suite.
- `tests/test_deployment_artifacts.py`: 45 cross-artefact consistency tests (UID 1001, port 8000, `MOE_*_DIR` across Dockerfile, Helm chart, and Quadlet unit).

v1.1 state (May 2026). The test suite has grown to **195 +** tests across **nine files** in three sub-directories:

Directory / File	Scope
`tests/test_routing.py`	Routing functions (unchanged).
`tests/test_mcp_validation.py`	MCP AST whitelist (unchanged).
`tests/test_deployment_artifacts.py`	Cross-artefact invariants (unchanged).
`tests/test_parsing.py`	Stateless parsers: JSON extraction, confidence parsing, history truncation.
`tests/test_web_search.py`	SearXNG + DuckDuckGo fallback behaviour.
`smoke/test_env_contract.py`	Environment-variable contract: every required env var is present and typed before startup.
`integration/test_routes_contract.py`	Route registration contract: expected URL patterns and HTTP methods are present.
`integration/test_api_auth.py`	JWT validation, rate-limit enforcement, budget gate.
`pipeline/test_cc_session.py`	Claude Code session lifecycle and context compression.
`pipeline/test_kafka_handlers.py`	Kafka publish/consume handlers and graceful degradation without Kafka.
`pipeline/test_agent_state.py`	`AgentState` field consistency across pipeline transitions.

The three-level hierarchy (*smoke*, *integration*, *pipeline*) reflects a triage principle: smoke tests run in under two seconds and block every CI merge; integration tests cover contract boundaries between services; pipeline tests validate stateful LangGraph transitions with mocked LLM backends.

Engineering milestone: monolith decomposition (v2.4). Between the initial release and v1.1 of this whitepaper, the orchestrator's primary source file was refactored from a **11 190-line monolith** (`main.py`) to a structured package of **∼1 500 lines** of entry-point code and 14 focused modules – a reduction of 86 %. The decomposition proceeded in 14 explicit phases, with all 195 tests remaining green at every step.

The resulting module structure is:

- `config.py`, `state.py`, `prompts.py`, `metrics.py`, `parsing.py`, `context_budget.py` – cross-cutting state and constants.
- `routes/` (11 modules) – FastAPI endpoint handlers.
- `services/` (12 modules including a `services/pipeline/` subpackage) – business logic: auth, tracking, routing, healer, inference, helpers, skills, Kafka, LLM instances, and three protocol-specific pipeline handlers (OpenAI, Anthropic, Responses).
- `graph/` (6 modules) – LangGraph node implementations: router, planner, expert, research, synthesis.

The decomposition removed ∼600 lines of dead duplicate handlers that had survived from earlier iterations. When a monolith can be cleanly decomposed without behaviour change, it validates that the original architectural intent was correctly implemented; the module boundaries that emerge retrospectively confirm the design described in Section 5.

What we are proud of – and what we are not

Proud of: the determinism of the routing, the module decomposition, the three-level test hierarchy, 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, the specification gaming incident (Episode 6), 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.

14.14 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.

14.14.1 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.

14.14.2 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.

14.14.3 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 positioning 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 11: 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).

14.15 External Baseline Evaluations

14.15.1 GAIA benchmark 2026: evaluation against an external accuracy baseline

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

Benchmark integrity notice

The result documented in this section was produced with template `moe-aihub-free-gremium-deep-wcc`, which was found to route all expert slots to a commercial frontier model (`gpt-4.1@AIHUB`) rather than the intended sovereign stack. The result therefore cannot be interpreted as a sovereign-stack performance measurement. Full root-cause analysis and remediation steps are documented in Section 14.6. The numbers are preserved here for methodological completeness; the valid sovereign GAIA measurement is pending revalidation.

Reported result (see integrity notice above). 14/30 = 46.7% with template `moe-aihub-free-gremium-deep-wcc` (`gpt-4.1@AIHUB` as the effective backbone). For reference, the official GPT-4o Mini baseline is 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 12: GAIA validation set – 5 runs, 2026-04-25. 10 questions per level. Best recorded result: **14/30 = 46.7%** with template `moe-aihub-free-gremium-deep-wcc`. *Note: later audited as specification gaming (Section 14.6); not a valid sovereign measurement.* GPT-4o Mini reference: 44.8%.

Statistical limitations of this benchmark

Each run evaluates 30 questions (10 per level). At this sample size, a binomial 95 % confidence interval for the best result (14/30 = 46.7%) spans roughly **28 %–66 %**. The point estimate therefore cannot be interpreted as a definitive performance claim; it establishes a plausible operating range, not a statistically guaranteed minimum.

Full raw logs, prompt seeds, and the evaluation harness are part of the repository (`benchmarks/gaia_runner.py`) but a formally reproducible benchmark artefact – fixed seeds, locked dependency hashes, published per-question outputs – has not yet been released. This is acknowledged as a gap; a reproducibility release is planned.

14.15.2 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 13: 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).

Accuracy vs. latency: two incomparable axes

The numbers above measure *accuracy only*. A cloud model such as GPT-4o Mini delivers a comparable answer in 1–3 seconds; the MoE Sovereign pipeline runs 360–1400 seconds for a complex multi-expert request on the same hardware. The two systems differ by two to three orders of magnitude in wall-clock latency.
This gap is not incidental. It is the direct consequence of a design choice: self-hosted, privacy-preserving inference on consumer-class and legacy hardware, where no single GPU fits a 30B+ model without quantisation and KV-cache throttling. The system is not designed for synchronous, low-latency chat; it is designed for asynchronous,

sovereignty-constrained, regulated-domain workflows where response time is measured in minutes, not seconds, and data must not leave the operator's infrastructure.
Any fair comparison to commercial providers must therefore specify all three dimensions: accuracy, latency, and data-sovereignty. Reporting accuracy in isolation is misleading in both directions – for and against.

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.

14.15.3 MoE-Eval: Internal System Health and Regression 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.

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 14: 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.

Interpretation. **Strengths** (≥ 7.0): The MCP precision tests confirm the project's core thesis – deterministic tools eliminate hallucinations *within the computation boundary*. 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 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.

14.15.4 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.

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.

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 15: 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.

- **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.

Accumulation-associated improvement observed

The observed before/after difference is consistent with combined effects from model warmth, accumulated graph context and correction memory; their individual contributions were not isolated in this run, but illustrate the practical operational efficiency of reusable knowledge paths in the compound system.

14.15.5 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 16: 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:

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 17: 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 %).

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

1. **Graph density (strongest observed factor, +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.

14.15.6 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 %

Benchmark integrity notice – GAIA Level 1

The Level 1 result of 60 % below originates from the same template as the overall score 46.7 % (specification-gaming incident, Section 14.6): all expert slots were routed to a commercial frontier model (`gpt-4.1@AIHUB`). The partial result cannot be used as evidence for the sovereign stack or any individual architectural component. It is included here for documentation purposes. A valid sovereign GAIA run is pending.

Our backbone models (DeepSeek-R1:32b, Qwen3:32b, Llama3.3:70b) score individually in the 11–28 % range. In the contaminated benchmark run (non-sovereign template), **60 % on GAIA Level 1** (6/10 correct) was achieved. The qualitative observations about solution strategies are documented independently of the sovereignty issue:

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

Failures: The system failed on tasks requiring external document download and analysis (PDF, arXiv) – the MCP tools do not yet offer PDF parsing. The reversed-text test failed 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 %** – internal empirical evidence consistent with a compounding knowledge effect of the GraphRAG accumulation mechanism (see methodological caveat above; leaderboard comparison is not warranted without identical splits and normalizers):

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, consistent with an accumulation effect of the Adaptive Feedback Loop (Section 2.5). The top systems on the LongMemEval leaderboard – EverMemOS (83 %) and TiMem (76.9 %) – use specialised memory architectures with dedicated session persistence. The best internal run of MoE Sovereign achieved 91.7 % on a LongMemEval-oriented test configuration; whether exactly the same dataset splits, answer normalisers, and scoring rules as the official leaderboard were applied is not fully documented. A direct leaderboard comparison cannot therefore be derived without qualification. Qualitatively, however, the numbers are consistent with a cumulative knowledge effect of the GraphRAG accumulation mechanism, even though memory is not the primary optimisation target of the architecture.

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**	invalid run[†]

[†] Value from the contaminated run documented in Section 14.6 (non-sovereign model substitution); not a valid performance result for the sovereign stack.

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 demonstrates a clear improvement on the internal MoE-Eval over time. No valid sovereign GAIA result is currently available: the previously reported Level 1 value of 60 % stems from the contaminated run and is not used as evidence of performance.

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.

14.15.7 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.

14.15.8 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.

14.15.9 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.

14.15.10 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):

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

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 19: 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.

14.15.11 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: proof-of-concept on a constrained budget. The Tesla M10 GPUs were chosen for an entirely practical reason: at approximately €50 per card on the second-hand market, they offer 8 GB GDDR5 VRAM at an acquisition cost that has no equivalent among new hardware. Twenty cards across the full cluster provide 4 × 8 GB = 32 GB nominal VRAM per server for roughly €200 per node – hardware that would cost orders of magnitude more new. This is a proof-of-concept decision driven by budget, not a deliberate integration stress-test: *the goal was to build the system with what was affordable*, validate the architecture under real constraints, and keep the design clean enough to run on better hardware when it becomes available.

Running current-generation LLMs on this hardware comes with a non-trivial maintenance cost. Official Ollama and llama.cpp releases have progressively dropped CUDA 11 support as the ecosystem moved toward CUDA 12 and newer architectures. Keeping the M10 nodes operational with recent models therefore requires maintaining a custom fork of Ollama and llama.cpp with backported CUDA 11 compatibility patches – a continuous engineering overhead that is a direct consequence of the budget hardware choice and worth disclosing explicitly.

The RTX cluster (five RTX 3060, 12 GB each) operates in Ollama's native multi-GPU mode, combining VRAM across all cards into a single Ollama instance. This yields approximately 50–60 tokens/s for models that fit within the combined 60 GB pool – usable throughput for interactive development. This cluster serves as the primary development environment; the M10 nodes handle specialised expert workloads for the benchmark and production runs.

A system that functions under these conditions demonstrates portability and correctness across heterogeneous hardware. The cache layer that saves seconds on an M10 reduces latency on faster hardware, where substantially higher throughput and concurrent capacity are expected. Exact scaling on enterprise-grade clusters (scheduler contention, batched inference, tensor parallelism, shared KV caches) remains to be measured.

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.

14.15.12 Hardware Energy Efficiency: Thermal Management via Power Capping

A crucial operational insight gained from running dense multi-GPU inference nodes concerns the *thermo-energetic profile* of autoregressive language models. The token generation phase (token-by-token decoding) is fundamentally memory-bandwidth bound ($\mathcal{O}(N)$ memory transfers per generated token for a model with N parameters). Compute units (ALUs/Tensor Cores) spend substantial cycle times idling while waiting for weight and KV-cache transfers from GDDR5 memory.

Unconstrained GPU core boost clocks increase power dissipation ($P \propto V^2 f$) exponentially without delivering meaningful token throughput gains. By applying low-level *power capping* via `nvidia-smi -pl <Watts>`, total power consumption and heat dissipation can be substantially curtailed while the memory bus clock remains pegged at maximum frequency.

Table 20 documents the empirically measured throughputs, thermal deltas, and performance retention profiles across the production and benchmark fleet.

Table 20: Empirical power-capping measurements, temperature reduction, and measured token generation throughputs in the heterogeneous inference cluster.

Node / GPU	**Hardware**	**TDP (Base)**	**Limit ($P_{\mathbf{cap}}$)**	**Δ Watt**	**$\Delta T_{\mathbf{core}}$**	**Throughput**	**Spec / Mod**
N11-M10 (GPU 0)	Tesla M10 #1	53.0 W	27.0 W	−49.1 %	−18 °C	4.9 t/s	Eco pro-file (qwen 32k KV)
N11-M10 (GPU 1)	Tesla M10 #2	53.0 W	27.0 W	−49.1 %	−18 °C	4.9 t/s	Eco pro-file (qwen 32k KV)
N11-M10 (GPU 2)	Tesla M10 #3	53.0 W	32.0 W	−39.6 %	−14 °C	5.3 t/s	Cool pro-file (qwen 32k KV)
N11-M10 (GPU 3)	Tesla M10 #4	53.0 W	38.0 W	−28.3 %	−10 °C	5.9 t/s	Balan (qwen 32k KV)
N04-RTX (GPU 7–10)	4× Tesla M10	53.0 W	38.0 W	−28.3 %	−10 °C	6.1 t/s	Balan (qwen 32k KV)
N04-RTX (GPU 4+5)	2× Tesla M60	150.0 W	112.5 W	−25.0 %	−12 °C	20.0 t/s	Dual-GPU Split (code / 12B)

Architectural analysis (Tesla M10 vs. Tesla M60). The empirical measurements reveal two fundamental architectural takeaways:

1. **KV-cache traversal on a narrow memory bus (Tesla M10):** With a fully allocated 32k context window (`f16` KV-cache footprint ≈ 2.5 GB additional VRAM), 4B expert models (`qwen3.5:4b`) reach 5.9 tokens/s on `N11-M10:ollama-tesla-4` (GPU 3 at 38 W) and 6.1 tokens/s on `N04-RTX:ollama-tesla-1` (driven by the newer host platform and higher PCIe/system memory bandwidth). This overall bottleneck stems from the 128-bit memory bus width (≈ 83 GB/s per GPU) combined with the absence of Flash Attention on the Maxwell architecture (CC 5.0). Remarkably, aggressive power capping from 53 W down to 27 W (−49.1 %) reduces throughput only from 5.9 to 4.9 t/s (83.1 % retention) – proving that memory bandwidth is the exclusive bottleneck, allowing compute clock power to be halved without dramatic throughput penalty.

2. **Dual-GPU scaling across wide memory buses (Tesla M60):** In stark contrast, the Tesla M60 on **N04-RTX** achieves 20.0 tokens/s, even though the model is partitioned across two separate physical GPU cores. The 256-bit memory bus per GM204 core (≈ 160 GB/s per GPU, 320 GB/s aggregate) more than offsets inter-GPU tensor transfer overhead, more than tripling decoding speed relative to the M10. Throttling the M60 to its firmware floor of 112.5 W lowers chassis temperatures by 12 °C while sustaining the full 20.0 t/s rate.

Power capping limits are deterministically enforced on every boot via host-level systemd units (`nvidia-power-limit.service`) without container overhead.

14.16 Lessons on Multi-GPU Inference, Synchronization Deadlocks, and 256k Context Allocation

Operating large verification models such as the 35B Sovereign Judge (`sovereign-judge:35b-q4km`) on heterogeneous consumer GPU clusters yields critical systems engineering insights rarely documented in standardized benchmarks:

1. Futex deadlocks during heterogeneous tensor parallelism (Turing + Ampere over PCIe). When a 35B model is distributed across four consumer GPUs (e.g., 2× RTX 2060 [Turing, `sm_75`] and 2× RTX 3060 [Ampere, `sm_86`]), layer and KV synchronization take place over the shared PCIe 3.0 interconnect. In production, dynamic context-shifting (`-context-shift`) coupled with asynchronous CUDA event synchronization (`cudaEventSynchronize`) triggered PThread barrier deadlocks (`futex` waits in `llama.cpp`). Crucially, this condition is *not* caused by VRAM or host memory exhaustion (the node maintained > 20 GB of free VRAM headroom), but rather by execution timing divergence across microarchitectures and uncoordinated slot teardowns on aborted client connections. **Resolution and hardening:**

- **Deterministic CUDA streams:** Forcing `CUDA_DEVICE_MAX_CONNECTIONS=1` in the container environment enforces serial stream dispatch per GPU and eliminates barrier races.
- **Single-slot dedication (`OLLAMA_NUM_PARALLEL=1`):** Configuring the sequential verification judge for dedicated single-slot processing eliminates KV-cache slot contention.
- **Client-side bounded timeouts & watchdogs:** Replacing infinite TCP sockets with strict 180 s–300 s timeouts prevents orphaned socket allocations from blocking GPU workers.

2. 256k context allocation dynamics and monitoring artifacts. Allocating a model with its full 262,144-token context window (`num_ctx: 262144`) requires ≈ 4.4 GB of additional `q4_0`-quantized VRAM buffer space (≈ 24.7 GB aggregate allocation across 4 GPUs). The initial initialization and buffer mapping across four PCIe banks takes up to 30 seconds. An important operational artifact: during active prompt decoding, the inference daemon locks internal session state, causing administrative health endpoints (`/api/ps`) to temporarily report an empty active model array even while all four GPUs compute under 100% utilization. Once kept warm permanently (`keep_alive: 24h`), repeated buffer allocations are completely eliminated.

3. Mitigating fragmentation with the 27B Sovereign Judge (LUMI-G). These operational lessons with the 35B model directly validate the design of the 27B Sovereign Judge trained on the EuroHPC LUMI-G supercomputer: By reducing total memory requirements from ≈ 24.7 GB to ≈ 16.0 GB (Q4_K_M), the judge can be hosted entirely on two homogeneous Ampere cards (2× RTX 3060), eliminating cross-generation PCIe barrier latency and accelerating verification throughput from ≈ 51 tokens/s to > 75 tokens/s.

Acknowledged reproducibility gaps. Three gaps exist that readers should be aware of:

1. **Raw benchmark data:** Per-question GAIA outputs, prompt seeds, and Valkey/Neo4j state snapshots at benchmark time are not yet published as standalone artefacts. The runner script is public; the reproducible dataset release is planned.

2. **Statistical significance:** No cross-validation, no bootstrap resampling, and no confidence intervals are reported for the MoE-Eval cognitive suite (single overnight run per epoch). Scores should be treated as directional indicators, not as statistically verified performance guarantees.

3. **Hardware generalisation:** All measurements were taken on the author's personal cluster. Performance on other GPU architectures (AMD ROCm, Apple Silicon, cloud instances) is projected by extrapolation, not direct measurement.

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.

14.17 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** (phi4:14b-fp16 / qwen3-vl:8b): 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 21 shows the expected convergence pattern based on our compounding-knowledge measurements (Section 8).

Table 21: 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

Comparative summary.

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)
API cost per attempt	\$0.50–2.00	\$0 (local; energy and hardware costs remain)
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 per-token API cost; local energy and hardware costs remain). For bounded, repetitive development tasks, the local system is a practical complement. For complex single-shot generation, frontier models retain an advantage that the learning loop partially but not fully compensates.

14.18 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

Table 22: 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

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.

Table 23: 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**

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

Table 24: 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)

is a *configuration limit*, not an architectural one: raising `OLLAMA_CONTEXT_LENGTH` to 32k on 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 25: 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 26: MoE-Eval score vs. comparable 7–14B-class models.

System	Type	Size	MMLU	MT-Bench	Sovereign
GPT-4o mini (API)	Cloud	undisclosed	82 %	8.8	×
Claude Haiku 3.5 (API)	Cloud	undisclosed	∼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 6.11/10 on the internal MoE-Eval – with full data sovereignty and zero per-token cost. A direct quality comparison to cloud-based services is not possible due to differing benchmark designs.

14.19 Deployment maturity

Transparency about testing coverage is essential for reproducibility. Table 27 documents the actual validation state of each deployment target.

Table 27: Deployment target maturity (as of August 2026).

Target	Status	Notes
Docker Compose	Production-tested	Primary method. Production-validated on 5-node GPU cluster.
LXC / Proxmox	Production-tested	Docker-in-LXC with `nesting=1, fuse=1`. GPU passthrough requires additional config.
Podman (rootless)	Tested	UID mapping and GPU access validated.
K3s / Kubernetes	CI-validated	Helm chart CI-validated; community production runs requested for broader validation.
Kubernetes (managed)	CI-validated	Manifests provided; community validation welcome.
OpenShift	Implemented, environment-unvalidated	SCC and Route configuration documented; not yet validated in a real OpenShift environment 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.

14.20 Overall Assessment

14.20.1 External Evaluation

As part of an AI-assisted peer-review process, MoE Sovereign was qualitatively assessed along the dimensions of architectural maturity, degree of innovation, scalability, reproducibility, and societal relevance. Highlighted was the consistent integration of established building blocks (MoE routing, GraphRAG, RL Flywheel) into a sovereign, learning compound AI infrastructure, as well as the transparency of documented failures. Benchmark-related claims from this evaluation are not used as quantitative evidence in this paper: the underlying GAIA run was subsequently classified as invalid (Section 14.6), and an AI-generated self-evaluation is not a substitute for controlled external benchmarks.

14.20.2 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 are consistent with this.
- **Deterministic pipeline** – MCP precision tools with AST whitelist eliminate hallucinations *within the deterministic computation boundary*: the tool reliably executes the operation, but the LLM may still extract incorrect operands, select the wrong tool, or misinterpret a correct result.
- **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** – The benchmark infrastructure and evaluation harness are reproducible on the documented hardware; exact result reproduction is limited by currently unpublished state snapshots and seeds. The full infrastructure is 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

The documented strengths – architecture, memory accumulation, reproducibility – are well-supported; the LongMemEval numbers and MCP precision evidence substantiate them. What is still missing are controlled external benchmark results for the sovereign stack. 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.

14.21 Component Evidence Summary

A fully controlled ablation study – identical inputs, identical hardware, one component removed at a time – requires infrastructure and compute budget beyond the scope of a single-developer project. We therefore present a *component evidence summary*: each component is assessed via the dedicated benchmark evidence collected during the evaluation campaigns above. For each component, we report the directly governed metric, the observed indicator, and the causal confidence level. Causal attribution to a single component is not established without controlled re-runs.

Table 28: Component evidence summary: observed metrics and supporting evidence for each major architectural component. "Effect indicator" is the observed or estimated effect associated with component presence; causal attribution requires controlled re-runs not yet available.

Component	**Governed metric**	**Effect indicator**	**Evidence source**	**Causal confidence**
MCP AST Precision Tools	Math accuracy	+40 pp vs. LLM baseline (estimated)	Own benchmark: 10/10 with AST evaluator; literature baseline ∼60 % hallucination rate without deterministic delegation [46]	Low–Medium
Tier-2 Semantic Memory (ChromaDB)	Recall at depth >50	Recall collapses to 0 beyond native context window without T2	MRCR-lite v2: score 1.000 with T2 enabled; 0 without T2 at same depth	Medium
L2/L3 Cache hierarchy (Valkey + Neo4j)	Median latency (epoch 5)	9.3× cold/warm difference	Accumulation benchmark: 707 s (cold, epoch 1) vs. 76 s (warm, epoch 5); multiple variables changed between epochs; strictly causal attribution not established	Low
Neo4j GraphRAG	Avg score / avg latency	+18 % score vs. zero-shot (10-query comparison)	GraphRAG 1.30 vs. zero-shot 1.10 (2026-07-11); GraphRAG wins 2/3 evaluable pairs; latency −3 %; 5/10 timeouts from host memory exhaustion (see § 14.22)	Medium
Two-tier expert escalation	Cost & latency	T2 fallback rate ≈18 %; routing saves T2 cost for ≈82 % of requests	Routing telemetry; counterfactual estimate, not a controlled re-run	Low
Corrective RAG Gate	Context precision	Not yet quantified	Introduced in v2.5; qualitative improvement observed; controlled measurement scheduled	None
Episodic Memory (`:Episode`)	Routing convergence	Not yet quantified	Introduced in v2.5; longitudinal data required for quantitative assessment	None

Interpretation. The evidence is consistent with the architectural thesis that no single component alone accounts for the observed performance. The cache hierarchy is associated with the largest latency difference; the MCP AST evaluator delivers the largest accuracy gain on deterministic tasks relative to an LLM-only baseline; semantic memory is the sole enabler of context recall beyond the model's native window. Causal confidence is low-to-medium throughout because controlled ablation runs are not yet available. The v2.5 components (Corrective RAG, Episodic Memory) lack sufficient operational history for quantitative assessment – a known gap acknowledged here rather than papered over.

Limitations of this approach. This evidence summary is not a substitute for a controlled randomised experiment. Hardware variance, query distribution shifts, and the absence of identical re-runs mean that all effect indicators carry unquantified uncertainty. A rigorous ablation would require a held-out benchmark set, identical hardware across all runs, and statistical significance testing – infrastructure planned for a future evaluation campaign.

14.22 GraphRAG vs. Zero-Shot: 10-Query Direct Comparison

Evaluation design. On 2026-07-11 ten representative queries from the operational prompt pool were routed through both paths in parallel:

- **Zero-Shot**: Direct routing to the responsible T1/T2 model with no graph context.
- **GraphRAG**: Neo4j subgraph retrieval (up to 5,353 nodes, ≈5,000 graph-context tokens) before the model request.

Scoring: 0–5 Likert scale, manually evaluated. Timeout: 300 seconds per request.

Table 29: GraphRAG vs. zero-shot: summary benchmark results (10 queries, 2026-07-11).

Metric	Zero-Shot	GraphRAG	Delta
Avg score (0–5)	1.10	1.30	+0.20 (+18.2 %)
Avg latency	161.27 s	155.84 s	−5.43 s (−3.4 %)
Wins (higher score)	0	2	GraphRAG +2
Timeouts (>300 s)	5	5	—

Results. Of the three evaluable queries, two showed a quality improvement from GraphRAG; one was a draw (f005, 4.0 points each):

- **s001** (Attention-Is-All-You-Need summary): Zero-shot 4.0 points in 46.79 s – GraphRAG **5.0 points in 6.73 s**. Latency gain: −40 s (85.6 % faster). Pre-condensed graph context from the knowledge graph allowed the model to skip its own retrieval.
- **c003** (consistency trade-offs in distributed systems): Zero-shot 3.0 points in 39.52 s – GraphRAG **4.0 points in 48.19 s**. Score gain +1.0; GraphRAG captured more nuanced CAP-theorem references from the knowledge graph, with minor latency overhead (+8.6 s).

Limitations. Five queries (f001, f002, f004, c001, s002) hit the timeout limit on *both* paths. Root cause: the host machine (`ki-vm-node05`) was under extreme resource pressure (swap 100 %, CPU load average >12.0) from concurrent Docker build and training processes. The timeouts are not GraphRAG-specific failures.

The sample size (3 evaluable pairs) is too small for statistical significance. Nevertheless the experiment provides early empirical evidence for the value of the knowledge graph: in contexts

with high information density (architecture summaries, trade-off analyses) GraphRAG delivers measurably better results, sometimes with substantially reduced latency.

GraphRAG benefit emerges first in context-dense tasks

For requests where the knowledge graph contains dense and directly relevant subgraphs (architecture descriptions, theorem references), GraphRAG delivers both latency and quality gains. The 85.6 % latency gain on s001 arose because pre-condensed graph context substantially shortened LLM reasoning. Replication under controlled host conditions (no competing swap pressure) with $n \geq 50$ queries is needed before the evidence is benchmark-grade.

14.23 Scientific Compound AI Architecture Benchmark – Pre-Distillation Baseline (August 2026)

Research Question and Experimental Design. Standard LLM benchmarks evaluate monolithic model parameters against one another (*Model A vs. Model B*). To investigate the primary research question of MoE Sovereign – *"How much functional capability and correctness emerges outside model weights through modular compound-AI system architectures?"* – we conducted a controlled empirical benchmark in August 2026:

$$8 \text{ Task Classes} \times 4 \text{ Architecture Conditions} \times 2 \text{ Repeated Rounds} = 64 \text{ Executions}.$$

Four controlled architecture conditions were evaluated across an identical, frozen task pool:

1. **Native Single-LLM Baseline**: Direct model execution without external tools, graphs, or multi-agent orchestration.
2. **Ablation (No-Graph Compound)**: Full planner, tool, and expert workflow with the Neo4j knowledge graph completely disabled.
3. **Full Compound AI (1-Pass)**: Complete MoE Sovereign stack with Kahn DAG compilation, 58 deterministic MCP precision tools, and 3-tier GraphRAG.
4. **Compound + Debate**: Complete stack including multi-agent deliberation and paraconsistent Belnap-Dunn judge arbitration.

Table 30: Empirical results of the Compound AI Architecture Benchmark (Pre-Distillation Baseline, August 2026, 2 rounds, $N = 64$ runs).

Architecture Condition	Overall Score	95 % CI	Det. Score	∅ Latency	∅ Tokens
Native Single-LLM	**5.97 ± 1.52**	[5.23; 6.72]	**7.44**	**530.4 s**	7,326
No-Graph Compound	5.09 ± 1.87	[4.17; 6.00]	5.22	757.6 s	13,078
Full Compound AI	4.82 ± 1.95	[3.83; 5.81]	4.55	1,135.0 s	18,366
Compound + Debate	4.85 ± 1.89	[3.92; 5.78]	4.62	2,096.2 s	6,425

Key Scientific Findings.

1. **Empirical Evidence of Orchestration Overhead (5.97 → 5.09 → 4.82):** Pipeline performance initially decreases with increasing system complexity. *Illustrative reliability model:* Under the simplifying assumption that each pipeline stage succeeds with conditional probability $p = 0.95$ given successful completion of all preceding stages, end-to-end reliability decreases as p^n: 73.5 % for six stages and 59.9 % for ten stages. This is an

illustrative model of error accumulation, not an empirical estimate of the individual stage failure probabilities observed in this benchmark. Generic base models that have not been specialized for their exact system roles degrade under interface drift. *Scaffolding alone does not improve performance without role-specialized models.*

2. **Controlled Evidence of Temporal Consistency via GraphRAG:** Under a controlled policy change (transition from Port 8080 to Port 8443 with mTLS), the Native Baseline collapses to 3.0/10 points (hallucinating obsolete pretraining facts), whereas Compound AI with GraphRAG reliably scores 7.0/10 across both rounds. In this controlled temporal-policy task, externalized current state outperformed stale parametric knowledge across both repeated rounds.

3. **Deterministic Precision Tool Superiority:** For mathematical VLSM IPv4 partitioning, typed MCP tools deliver consistent 10.0/10, while ungrounded LLM generation generates address overlaps in 80 % of test cases. Exact arithmetic is not merely guided by prompting, but replaced by deterministic computation.

Post-Distillation Replication (T_1, August 18, 2026): 8 VRAM-Pinned Expert SLMs. Following the successful fine-tuning of the 4.2B Student Planner and deployment of 8 specialized 4B expert SLMs on fixed allocated VRAM (Tesla M60 on GPUs 4+5 and Tesla M10 on N04/N11), the replication benchmark (T_1, August 18, 2026) was conducted.

Table 31: Empirical results of the replication benchmark T_1 (August 18, 2026, 8 VRAM-pinned 4B SLMs on M60/M10 vs. T_0 Pre-Distillation Baseline).

Architecture Condition	Overall Score (T_1)	95 % CI	Det. Score	∅ Latency (T_1 vs. T_0)	Speedu
No-Graph Compound	$\mathbf{4.53 \pm 1.76}$	[3.67; 5.40]	**3.83**	**475.1** s *(vs. 757.6 s)*	**1.6**
Full Compound AI	4.08 ± 1.62	[3.28; 4.87]	2.69	**498.5** s *(vs. 1,209.0 s)*	**2.4**
Compound + Debate	3.40 ± 1.55	[2.64; 4.16]	0.62	**698.5** s *(vs. 2,096.0 s)*	**3.0**

Four Novel Scientific Discoveries (T_1).

1. **The Small Model Invariant Conservation Boundary under Compilation Constraints:** Small language models (4B parameters) under a compiled execution policy with typed input/output contracts and structural cache bounds achieve **100 % formal invariant adherence (`10.0/10.0` deterministic score)** across C++20 Lock-Free Queues, eBPF Kernel Maps, and Raft consensus proofs – matching 35B/70B monoliths – while achieving a **4.7× latency reduction** (462.6 s vs. 2,190.0 s). System invariant violations are thus not a fundamental parameter capacity bottleneck, but a context structuring deficit.

2. **The Paraconsistent Conflict Suppression Threshold in Multi-Expert Debates:** On deterministic engineering tasks, unconstrained multi-turn debate degrades performance (`3.40` vs. `4.08`) due to multi-agent entropy drift. Multi-agent debate strictly requires threshold-based paraconsistent filtering (`resolve_conflicts_node` > 0.82) to dismiss non-critical discrepancies. Bounding empty reasoning traces elevated the C++ Queue debate score from `3.0` to `7.0/10` (`10.0` Det.).

3. **Heterogeneous Memory-Aware Offload Topography (`OLLAMA_SCHED_SPREAD=false`):** On heterogeneous GPU topologies (Ampere RTX 3060 PCIe 4.0 vs. Turing Tesla M60/M10 PCIe 3.0), naive layer spreading induces severe interconnect bottlenecks. Topology-aware

saturation – filling high-bandwidth GPUs to 100% before spilling the 258k KV-cache overflow – yields a **4.7× throughput speedup**.

4. **The GraphRAG Context Saturation Paradox:** For synthetic closed-form logic, GraphRAG yields 0% gain in deterministic correctness (`10.0/10` on both paths). Under temporal policy shifts (*Temporal Rule Override*), GraphRAG non-linearly increases precision from **`3.0` to `9.4/10`**. GraphRAG is a domain-conditional enabler.

LUMI-G 27B Sovereign Judge Milestone. Fine-tuning and GGUF export of **`sovereign-judge-27b`** (`Q4_K_M` at 16 GB VRAM) was 100 % completed on LUMI-G (8× AMD Instinct MI250X) in 3.0/3.0 epochs. The 16 GB footprint enables full local execution on 2× RTX 3060 with a 258k context window (`num_ctx: 262144`).

14.24 Technical Support for Selected ISO/IEC 27001 Controls

To support deployment in regulated enterprise environments, the security architecture of MoE Sovereign and *MoE Codex* is designed to align with the safety controls of the **ISO/IEC 27001:2022** standard. While a software stack cannot achieve certification on its own (which requires organizational and physical audits), the platform implements technical mechanisms that directly address Annex A requirements.

Table 32 details the mapping of implemented controls, operational evidence, and remaining open responsibilities for the hosting operator.

Table 32: ISO/IEC 27001:2022 control mapping and platform readiness.

Annex A Control	**Status**	**Technical Mechanism**	**Open Operator Items**
A.5.9 / A.5.12 Asset & Info Classification	Prepared	Metadata registry in the *Data Catalog* with automated GDPR classification and retention tags.	Organizational classification policy definition.
A.8.2 / A.8.3 Access Control & Privileged Rights	Operational	Attribute-based access control (ABAC) enforced via *Open Policy Agent (OPA)* Rego policies.	Identity provider (LDAP/OIDC) integration.
A.8.12 Data Leakage Prevention	Operational	Local-first air-gap deployment; outbound policy filters in `federation/` blocking critical domains.	Network perimeter segregation and DLP logs.
A.8.16 Monitoring & Event Logging	Operational	Auditable OpenLineage metadata flows captured by *Marquez*; Prometheus telemetry for nodes.	SIEM integration and alert escalation rules.
A.8.20 / A.8.22 Network Security & Segregation	Prepared	Isolated bridge networks in Docker Compose; containerized model runtimes (LXC/Podman).	Firewall configuration and VLAN routing.
A.8.25 / A.8.28 Secure Coding & Change Management	Operational	Git-style branching and bundle approval workflows on *lakeFS*; code versioning.	Independent code reviews and S-SDLC policy.

Operator Responsibilities. The technical readiness outlined in Table 32 covers the software-layer requirements. To pass an ISO 27001 audit, the deployment partner or operator must supplement these controls with organizational procedures: defining security guidelines (A.5.1), conducting employee security training (A.6.3), ensuring physical security of the server racks (A.7), and establishing incident response protocols (A.8.9).

14.25 Operational Safeguards for Fine-Tuned SLM Components

The introduction of LoRA/SFT-fine-tuned SLM components for the *planner* and *judge* roles (see Section 15.5) creates two classes of operational risk that the architecture must explicitly counter.

14.25.1 Format-Drift Validator for Paraconsistent Output

A neural network trained on 90 000 paraconsistent samples will, in principle, learn to emit structured four-valued truth types from the Belnap-Dunn lattice $\{T, F, B, N\}$. However, neural networks are inherently probabilistic: any temperature perturbation, prompt variation, or distribution shift can cause the model to emit unstructured prose instead of the required XML conflict map. If such drift occurs silently, the downstream paraconsistent merger receives malformed input and the entire audit chain collapses.

To counter this, a **deterministic validator layer** is interposed between every SLM call and the merger pipeline:

1. **Pydantic schema enforcement.** The SLM output is parsed against a strict `TruthTypeModel` schema. Accepted values per assertion slot are exactly `T`, `F`, `B`, and `N`.
2. **Regex pre-filter.** Before Pydantic parsing, a lightweight regular expression confirms the presence of the expected XML tags, rejecting bare-prose responses in under 0.01 ms.
3. **Fail-safe fallback.** Any validation failure assigns the assertion the conservative truth value N (*unknown*) rather than propagating a guess. This preserves the fail-safe abstention principle: in doubt, the system admits ignorance rather than hallucinating certainty.

The resulting pipeline is:

$$\text{SLM} \rightarrow \text{Regex pre-filter} \rightarrow \text{Pydantic TruthTypeValidator} \rightarrow \{T, F, B, N\} \rightarrow \text{Paraconsistent Merger}$$

with N injected on any validation exception. This guarantee allows the paraconsistent chain to remain formally sound even when the underlying neural component degrades.

14.25.2 Semantic Entity Linking for Knowledge-Graph Stability

The Ratcliff/Obershelp similarity metric (Python `SequenceMatcher`, 1988) canonicalises entity strings before each `Neo4j MERGE` call and effectively suppresses typographical variants. Its limitation emerges with semantically equivalent but lexically disjoint terms: *"Bürgerliches Gesetzbuch"* and *"BGB"* share near-zero character overlap despite being identical referents. At enterprise scale, uncorrected synonymy causes graph fragmentation – separate nodes accumulate for the same real-world entity – and degrades retrieval recall.

The mitigation strategy proceeds in two tiers:

Tier 1 – Abbreviation Lexicon (zero latency). A domain-maintained lookup table stored in PostgreSQL maps canonical short-forms to their full-form equivalents (*BGB* → *Bürgerliches Gesetzbuch*, *NIS2* → *NIS2-Richtlinie*, *DSGVO* → *Datenschutz-Grundverordnung*, etc.). Every entity string is resolved through this table before the similarity check. The lexicon is operator-extensible without code changes.

Tier 2 – Local Cross-Encoder (CPU, ≈2 ms). For strings that survive Tier 1 unresolved, a quantised *cross-encoder/ms-marco-MiniLM-L-6-v2* model (INT8 ONNX, ≈6 MB) computes a semantic similarity score. Pairs exceeding a tunable threshold θ (default 0.85) are merged to the canonical form. Running on CPU with ONNX Runtime, inference latency is 1–3 ms – negligible relative to the Neo4j write itself.

This two-tier pipeline ensures that the knowledge graph remains structurally sound as the corpus scales, without introducing a GPU dependency or a network call to an external embedding service.

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 Known limitations

We enumerate the four areas where the current system and this whitepaper have acknowledged gaps; each is an explicit research or engineering priority.

Benchmark reproducibility. The only fully documented GAIA run (14/30 = 46.7 %) has been classified as invalid (specification-gaming incident, Section 14.6); a valid sovereign measurement is still pending. Once a valid run is available: GAIA evaluations use 30 questions (10 per difficulty level), giving a 95 % confidence interval of approximately ±18 percentage points – any single result is therefore a plausible estimate, not a statistically confirmed bound. Per-question outputs, prompt seeds, and dependency hashes for a locked reproducibility release are not yet published. The evaluation harness (`benchmarks/gaia_runner.py`) is public; a formal artefact release with fixed seeds is planned.

Cache sensitivity analysis. The stated 40 %–80 % LLM invocation reduction is derived from production observations on conversational workloads. No systematic sweep over TTL configurations, cosine-distance thresholds, or workload types has been published. Operators should treat this range as a workload-dependent estimate, not a guaranteed saving, and instrument their own deployments via the Prometheus `cache_hit_total` metric.

GDPR: design claim vs. legal certification. The privacy-by-design properties described in Section 13 are *architectural* claims verifiable in the source code. Whether a specific deployment is legally GDPR-compliant requires a Data Protection Impact Assessment (DPIA, Art. 35 GDPR) and review by a qualified data protection officer. This whitepaper does not constitute legal advice.

Template routing conflict resolution. The current planner uses a JSON-structured task graph with `depends_on` semantics for sequential chains. Conflict handling when two expert outputs contradict each other is delegated to the judge model; no formal specification of judge tiebreaker behaviour under adversarial inputs has been published. This is an area of active development.

15.2 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).

15.3 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 [22] 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.

15.4 Expected Scaling Behaviour on Enterprise Hardware

MoE Sovereign's lean architecture is portable: the same principles that function on Tesla M10 clusters are expected to exhibit favourable, potentially non-linear scaling on H100-class systems. The hypothesis is grounded in three architectural properties that compound on higher-throughput hardware:

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.

These properties suggest that the architectural benefits would compound on enterprise hardware – but this remains a hypothesis until a dedicated scaling study is run on H100-class infrastructure. Exact throughput multipliers and concurrent-user ceilings require empirical profiling under production load.

15.5 Sponsored SLM Distillation Programme (EuroHPC LUMI-G)

In June 2026, proposal EHPC-DEV-2026D06-XXX was awarded under the EuroHPC Development Access programme (award notice received 2026-06-05), granting 4,500 node-hours (18,000 GPU-hours) on the **LUMI-G** partition (AMD MI250X, 512 GB HBM2e per physical node – 4× MI250X modules at 128 GB each –, ROCm stack, 2 TB storage) over a six-month period.

15.5.1 The Frontier Distillation Campaign (August 2026): 4B Student, 8 Domain Experts, and 27B Judge

In August 2026, the complete model layer of the MoE Sovereign architecture was trained on LUMI-G. This model family encompasses the Meta-Orchestrator model (`moe-sovereign-student-4b`), eight specialized 4B domain experts, and the 27B arbitration and consensus judge model (`sovereign-judge-27b`).

The underlying research direction is **Capability Externalization**: Instead of requiring a single monolithic language model to memorize encyclopedic knowledge, perform arithmetic in its weights, and verify formal logic internally, MoE Sovereign deliberately externalizes these capabilities into controllable, audited system infrastructure (GraphRAG knowledge graphs in Neo4j, relational database engines, deterministic SMT/Z3 solvers, AST linters, and Correction Memory). The specialized SLMs focus strictly on semantic interpretation, intent understanding, task decomposition, and typed interface synthesis.

Table 33 documents the full training metrics, teacher ensembles, dataset scales, and optimization convergence metrics across the campaign on 8× AMD Instinct™ MI250X GPUs (DeepSpeed ZeRO-2 in pure BF16 precision).

Infrastructure Resilience and SLURM Automation. For long-running distillation jobs on HPC clusters (such as the 27B Judge training requiring ≈36 hours total runtime), a fault-tolerant SLURM automation pipeline was deployed:

1. **Epoch Checkpointing:** Upon completion of each training epoch, complete model weights and optimizer states are committed to the parallel Lustre filesystem.
2. **POSIX Signal Trapping:** A custom trap intercepts `SIGTERM`/`SIGUSR1` signals prior to wall-clock expiration, executing an orderly buffer flush.
3. **Automated Resumption via Job Chaining:** SLURM job dependencies (`-dependency=afterany:$JOBI` trigger subsequent job steps that resume from the exact epoch checkpoint without learning-rate schedule discontinuities.

Table 33: EuroHPC LUMI-G Frontier Distillation Campaign: Model fleet, teacher ensembles, dataset trajectories, and convergence metrics (8× AMD Instinct™ MI250X, PyTorch 2.6, ROCm 7.0, DeepSpeed ZeRO-2 BF16, LoRA $r = 16, \alpha = 32$).

Model / Role	Base Architecture	Teacher Ensemble	SFT Traj.	Job ID	Loss	Token
`student-4b` (Planner)	Qwen3.5-4B (Mamba Hybrid)	DeepSeek-V3 + Qwen3-35B-Planner	35,000	#21189557	0.0094	99
`expert-coder-4b`	Qwen3.5-4B (Mamba Hybrid)	DeepSeek-Coder-V2 (236B) + V3	32,500	#21189558	0.0106	99
`expert-precision-4b`	Qwen3.5-4B (Mamba Hybrid)	Qwen2.5-Math-72B + Nemotron-70B + Z3	31,800	#21189559	0.0404	98
`expert-graphrag-4b`	Qwen3.5-4B (Mamba Hybrid)	DeepSeek-V3 + Qwen2.5-72B-Inst	33,000	#21189563	0.0083	99
`expert-datainfra-4b`	Qwen3.5-4B (Mamba Hybrid)	DeepSeek-V3 + Qwen2.5-72B-Inst	33,200	#21189564	0.0079	99
`expert-research-4b`	Qwen3.5-4B (Mamba Hybrid)	Moonshot Kimi-k3 + Nemotron-70B	34,200	#21189560	0.0072	99
`expert-security-4b`	Qwen3.5-4B (Mamba Hybrid)	DeepSeek-V3 + Mistral-Large-2407	32,800	#21189561	0.0078	99
`expert-governance-4b`	Qwen3.5-4B (Mamba Hybrid)	Mistral-Large-2407 + DeepSeek-V3	33,600	#21189562	0.0076	99
`expert-omni-4b`	Qwen3.5-4B (Mamba Hybrid)	DeepSeek-V3 + Qwen2.5-72B-Inst	35,000	#21189565	0.0075	99
`sovereign-judge-27b`	Qwen3.8-27B (262k Context)	Llama-3.1-405B + Nemotron-70B + Z3	40,000	#21263413	0.0860	98

Consumer Hardware Deployment. A foundational goal is self-hosted sovereign execution: Supercomputing resources (LUMI-G) were utilized strictly during offline distillation to compress capability from 200B+ teacher models into compact 4B models. Following CPU-BF16 adapter merging, all models are published as GGUF quantizations (`Q4_K_M` ≈2.6 GB, `Q8_0` ≈4.2 GB), enabling local execution on commodity consumer GPUs (6–12 GB VRAM such as RTX 2060/3060, Apple Silicon unified memory, or CPU RAM).

Scientific Interpretation of Training Metrics

The training losses reported in Table 33 (< 0.01 across structured 4B experts, 0.0404 for Precision, and 0.0860 for the 27B Judge) alongside token accuracies up to 99.84 % demonstrate high convergence and format discipline on the training split. They must not be conflated with held-out capability benchmarks. A systematic 2×4 factorial held-out evaluation against unmodified baselines is conducted within the ongoing Sovereign Scientific Benchmark Suite.

vLLM inference performance on AMD MI250X. As the backbone for IMoE routing dataset synthesis, **vLLM v0.20.1** served `Qwen/Qwen2.5-32B-Instruct` (TP=2, bfloat16) on a physical LUMI-G host using 2 GCDs (one MI250X module, 128 GB HBM2e out of 512 GB total node HBM). Measured performance is summarised in Table 34.

The 265 s startup time is acceptable for HPC batch jobs where SLURM scheduling overhead dominates for short jobs, but is borderline for interactive production use. The 22 tok/s decode throughput is within expectation for a 32B model at TP=2 on this hardware. The absence of Flash Attention 2 on ROCm results in approximately 10–15 % overhead relative to CUDA-equivalent configurations.

In parallel, the underlying knowledge graph continues to grow rapidly (46× entities, 56× relations over 16 days in early 2026), and the next scaling milestone is the first multi-hub MoE Libris federation trial (Section 15.6), which will validate the federated architecture at

Table 34: vLLM performance: Qwen2.5-32B-Instruct on 2× AMD MI250X GCD (TP=2, bfloat16).

Phase / Metric	Value	Note
Weight loading (Lustre)	146.89 s	61.03 GiB safetensors from scratch FS
Torch compilation (AOT)	17.95 s	Dynamo bytecode cache
CUDA graph capture (51 sizes)	50.00 s	0.83 GiB per rank
Total startup time	264.97 s	Until API status 200
Prefill throughput	548.0 tok/s	Prompt processing
Decode throughput	17.1–29.1 tok/s	Mean 22.8 tok/s
Weights per GCD	30.77 GiB	HBM2e utilisation
KV cache per GCD	27.2 GiB	Capacity: 222,768 tokens

the 25k–100k relation scale referenced above.

15.6 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.

15.7 Compliance extension: MoE Codex

The EU compliance and audit use cases – data lineage, approval workflows, versioned data bundles, and drift detection – have been realised as *MoE Codex* (Section 5.7). The full twenty-five-module stack across Tracks D.0 through D.5 is operational and released as a separate Apache 2.0 repository alongside full compliance documentation for GDPR Art. 32/35, EU AI Act risk classes, BSI Grundschutz, and NIS2.

Tracks D.1 through D.5 have been implemented and released. The previously open questions – Kestra-based pipeline orchestration, OpenSearch federated search, and geospatial investigation tooling – are now operational. Remaining open questions: LLM-assisted semantic drift narration beyond statistical thresholds; ArgoCD GitOps for multi-environment deployment automation; and KeplerGL ontology binding for object-overlay geospatial investigation.

15.8 SLM scaling limits under GraphRAG (Wikimedia benchmarks)

To systematically evaluate how external structured knowledge affects the capabilities of extremely small models, we plan to import large-scale Wikimedia Graph Dumps into the Neo4j instance. The goal is to investigate whether Small Language Models (SLMs) up to 9B parameters can achieve significant quality improvements and hallucination reduction when augmented with a GraphRAG context, compared to zero-context baselines.

This research will test the scaling floor of LLM-based reasoning under GraphRAG by

systematically benchmarking model sizes across a descending sequence of parameter scales: 9B, 7B, 4B, 3B, 1.5B, and 0.3B. The evaluation will quantify the quality degradation at each parameter threshold, establishing the absolute boundary where an SLM's internal logical capabilities become insufficient to reconstruct coherent answers even when provided with perfect external graph context.

15.9 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.

15.10 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.

15.11 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.

15.12 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 [47]: 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, occupied by few open-source systems in this combination, 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 51 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 twelve failure and learning-curve episodes: the 70B judge RAM vs. VRAM problem; the SymPy injection gap in the MCP calculate fallback; the SQLite concurrent-access problem; the ghost keys in live monitoring; the over-engineered first scheduler; the specification-gaming incident on the GAIA benchmark; the DeepSpeed ZeRO-3 multimodal fix on AMD MI250X; the ChromaDB semantic cache miss; the PEFT version mismatch; the GGUF `-no-mtp` flag; the ineffective Planner SFT caused by sequence-length truncation; and why fine-tuning on AMD MI250X remains punishingly slow without vendor tooling. Most are fixed; where the fix is still pending, the analysis state is documented. The disclosure of these episodes 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: source code under Apache 2.0, this whitepaper under CC BY-SA 4.0 [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.

With *MoE Codex* (Section 5.7), the project has moved beyond pure LLM orchestration into the compliance infrastructure space. MoE Codex adds eight modules – data catalog, approval workflows, lineage tracking, bundle versioning, ETL fan-out, drift detection, object exploration, and notebook access – that together address the same compliance infrastructure problem space as platforms like Palantir Foundry and Gotham, as a fully open-source, air-gap-capable approach. This is not a claim of current feature parity or product maturity: Palantir's enterprise engineering depth, certification coverage, and support organisation are not replicated here. What MoE Codex offers instead is full technical transparency, zero licence cost, and complete data sovereignty – properties that are not available from any commercial incumbent. The architecture keeps the separation strict: MoE Sovereign remains a self-contained gateway; MoE Codex attaches to it via a defined HTTP contract and can be deployed, upgraded, and replaced independently.

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

[10] 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.

[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*. 2023. eprint: 2311.12983. URL: https://arxiv.org/abs/2311.12983.

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

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

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

[19] B. J. Chan, C.-T. Chen, J.-H. Lai, and D.-B. Wu. *Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks*. Demonstrates empirically

that for stable, bounded knowledge domains, pre-loading authoritative context outperforms retrieval in accuracy, consistency, and latency. Forms the scientific basis for the CAG Compliance Layer in `compliance_cag.py`. 2024. arXiv: 2412.15605 [cs.CL]. URL: https://arxiv.org/abs/2412.15605.

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

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

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

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

[24] W. Fedus, B. Zoph, 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.

[25] 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.

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

[27] P. Lewis 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.

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

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

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

[31] K. Gödel. "Zum intuitionistischen Aussagenkalkül". In: *Anzeiger der Akademie der Wissenschaften in Wien* 69 (1932). Introduces the Gödel t-norm $T_G(a, b) = \min(a, b)$,

the most conservative conjunction in fuzzy logic. Used in `fuzzy_router_node` as the default routing conjunction., pp. 65–66.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 [39].

Traceparent
: The HTTP header from the W3C Trace Context standard [42] 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

The accompanying software is released under the **Apache License 2.0**. This whitepaper document is released under *Creative Commons Attribution-ShareAlike 4.0 International* (CC BY-SA 4.0) [12]. 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).

Note on licence change (v1.0 → v1.1): The software was initially distributed under CC BY-SA 4.0. Starting with v1.1 it is distributed under Apache 2.0, which provides clearer patent grant language and broader compatibility with commercial integrations while preserving the open-source character of the project.

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.