Sovereign LLM Operations with RAGIX/KOAS

A Framework for Confidential Document Analysis in Regulated Environments

Author: Olivier Vitrac, PhD, HDR | Adservio Innovation Lab **Version:** 1.2.0 **Date:** 2026-01-29 **Classification:** Technical Architecture Document

Abstract

This document presents the architectural principles and operational procedures enabling Large Language Models (LLMs) to perform document analysis tasks in environments where data confidentiality is paramount. The RAGIX (*Retrieval-Augmented Generative Interactive eXecution*) framework, combined with the KOAS (*Kernel-Orchestrated Audit System*) methodology, provides a sovereign alternative to cloud-based AI services. We demonstrate how policy enforcement, deterministic kernels, and local LLM deployment create an auditable system suitable for sensitive document processing—including technical audits, compliance verification, and specification analysis—without exposing data to external services.

Keywords: Sovereign AI, Local LLM, RAG, Document Analysis, Data Sovereignty, Air-Gapped Systems, Compliance, KOAS

Pipeline Invariants (MUST / MUST NOT)

This section defines non-negotiable rules for any LLM or agent operating within this framework.

Caution

MUST

1. **Strip Markdown metadata in all external outputs** — YAML/TOML/JSON front-matter, fenced `metadata` blocks, and HTML provenance comments (`←!— PROVENANCE ... ⟶`) are internal-only artifacts. External reports receive `content_clean` exclusively.
2. **Enforce output-level contract via CLI** — `--output-level=external` MUST strip metadata, redact paths, anonymize identifiers. This is enforced by build-time validation, not trust.
3. **Forced caching for all document LLM calls** — Every LLM call in `doc_*` kernels MUST be logged with:
    1. `call_hash` = SHA256 of canonical request JSON (sorted keys, normalized whitespace, volatile fields stripped)
    2. `inputs_merkle_root` = Merkle root of ordered child hashes for pyramidal synthesis
    3. Cache key includes: `model_id`, `temperature`, `template_id@version`, `prompt_canonical`
4. **Protect fenced/inline code inside documents** — Boilerplate detection patterns MUST NOT match content within triple-backtick fences or inline code spans. Code blocks are preserved verbatim.
5. **Canonical ordering for reproducibility** — Pyramid children MUST be ordered deterministically (by document path, then chunk index) before Merkle root computation. Non-deterministic ordering invalidates cache.
6. **Seed logging for partial-determinism kernels** — Clustering kernels

> (`doc_cluster`, `doc_cluster_leiden`, `partition`) MUST log their random seed in the audit trail. Same seed + same input = same output.
>
> **MUST NOT**
>
> 1. **Apply boilerplate cleaning to code or Code+Docs mode** — Boilerplate detection is for **pure document corpora only**. In mixed mode, code files bypass boilerplate filtering entirely.
> 2. **Include provenance markers in external outputs** — Keys like `call_hash`, `inputs_merkle_root`, `run_id`, `endpoint`, `model` are denylist items. Build fails if detected in external report.
> 3. **Modify representative content in `doc_extract`** — Quality filtering is achieved by **penalizing boilerplate patterns in `_score_sentence_quality()`**, not by cleaning/transforming the text. Original content is preserved.
> 4. **Allow orchestrator to see document excerpts** — `--output-level=orchestrator` returns metrics only: kernel names, timing, success/failure, aggregate counts. No text content.
> 5. **Trust implicit defaults for isolation** — Isolation settings MUST be explicit in manifest or CLI. Absent flags do not imply "safe"; they imply "internal mode."

Table of Contents

1. Introduction

1.1 The Confidentiality Challenge

Organizations increasingly require AI-assisted analysis of sensitive documents: technical specifications, audit reports, contractual documents, and compliance assessments. Traditional cloud-based LLM services (OpenAI, Anthropic API, Google AI) present fundamental confidentiality concerns:

Concern	Cloud LLM Risk	Impact
Data exfiltration	Documents sent to external servers	Confidentiality breach
Training data leakage	User data may train future models	IP loss
Jurisdiction	Data processed in foreign territories	Regulatory violation
Audit opacity	No visibility into processing	Compliance failure

1.2 The Sovereignty Imperative

For regulated industries (finance, healthcare, government, critical infrastructure), data sovereignty is not optional. The European GDPR, French ANSSI guidelines, and sector-specific regulations mandate:

1. **Data localization** — Processing must occur within controlled perimeters
2. **Auditability** — Complete traceability of data transformations
3. **Minimal exposure** — Data access limited to necessary operations
4. **Provenance tracking** — Origin and transformation history preserved

1.3 RAGIX/KOAS Solution

This document demonstrates how the RAGIX/KOAS stack achieves these requirements through:

- **Local LLM deployment** via Ollama (no external API calls)
- **Deterministic kernel computation** (reproducible, auditable)
- **Policy-enforced sandboxing** (controlled execution environment)
- **Cryptographic audit trails** (tamper-evident logging)
- **Cache-based replay** (LLM-free verification possible)

2. Sovereignty Architecture

2.1 System Topology

The sovereign architecture ensures all data processing occurs within a controlled boundary:

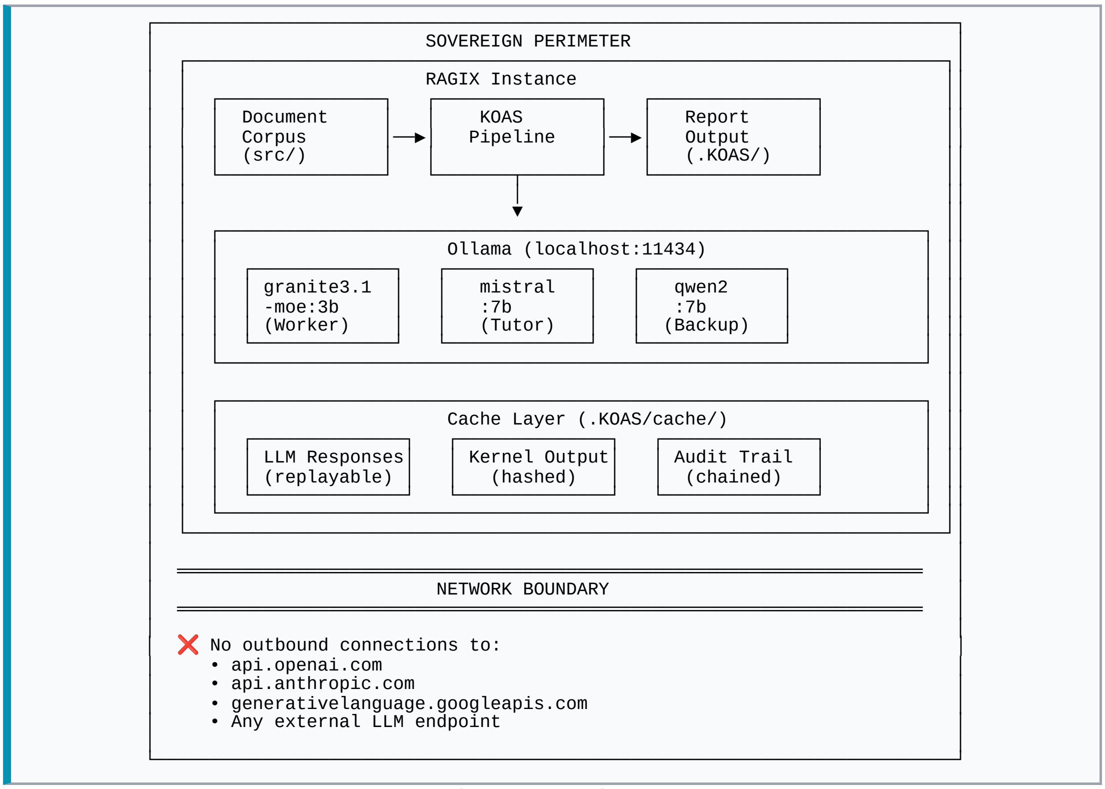


Figure 1. Isolation of data/documents/codes and of their processing.

2.2 Core-Shell (Onion) Architecture with External Orchestrator

For maximum isolation, RAGIX supports a **core-shell architecture** where an external orchestrating LLM (e.g., Claude, GPT-4) can drive the analysis without ever accessing the raw data. This "onion" model provides layered isolation.

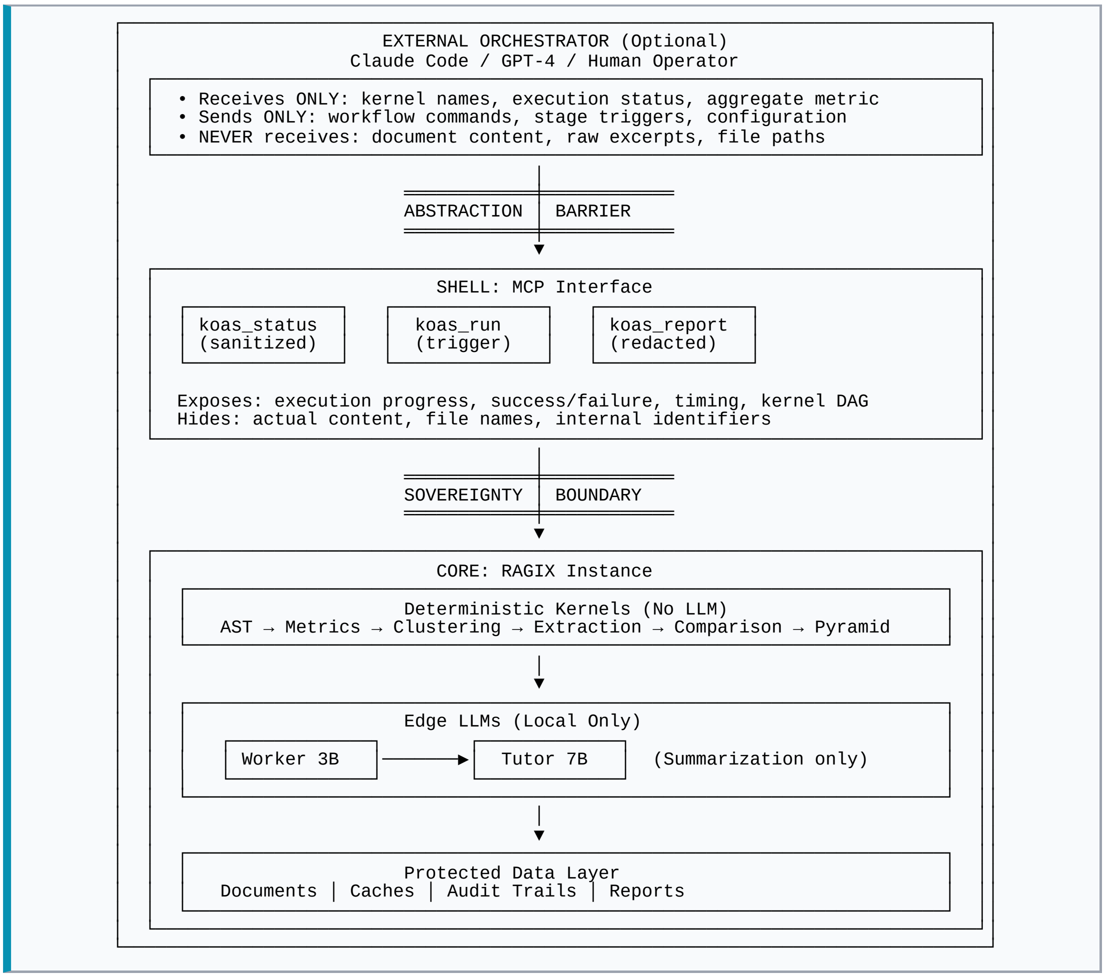

EXTERNAL ORCHESTRATOR (Optional)
Claude Code / GPT-4 / Human Operator
• Receives ONLY: kernel names, execution status, aggregate metric
• Sends ONLY: workflow commands, stage triggers, configuration
• NEVER receives: document content, raw excerpts, file paths
ABSTRACTION BARRIER
SHELL: MCP Interface
koas_status (sanitized)
koas_run (trigger)
koas_report (redacted)
Exposes: execution progress, success/failure, timing, kernel DAG
Hides: actual content, file names, internal identifiers
SOVEREIGNTY BOUNDARY
CORE: RAGIX Instance
Deterministic Kernels (No LLM)
AST → Metrics → Clustering → Extraction → Comparison → Pyramid
Edge LLMs (Local Only)
Worker 3B
Tutor 7B
(Summarization only)
Protected Data Layer
Documents | Caches | Audit Trails | Reports

Figure 2. Core-Shell Architecture with External Orchestrator

Isolation Guarantees of the Onion Model:

Layer	Sees	Does Not See	Control Level
External Orchestrator	Kernel names, metrics, timing	Content, file names, excerpts	Commands only
MCP Shell	Sanitized status, redacted reports	Raw paths, internal IDs	Trigger execution
RAGIX Core	Full document content	External systems	Full processing
Edge LLMs	Pre-structured prompts	Network, filesystem	Inference only

This architecture enables scenarios where:

- A cloud-based LLM orchestrates the workflow without data exposure
- Human operators receive progress updates without sensitive content
- Compliance officers audit the process through sanitized logs
- The core remains fully sovereign regardless of the orchestration layer

2.3 Data Flow Guarantees

Theorem (Data Containment): *Under correct configuration, no document content traverses the sovereign perimeter boundary.*

Proof sketch:

1. All LLM calls are routed exclusively to `localhost:11434` (Ollama)
2. Ollama operates in inference-only mode (no telemetry, no model updates)
3. Kernel computations are pure functions on local filesystem data
4. Cache reads/writes occur only within `.KOAS/` directory
5. No network sockets are opened except to localhost

This can be verified by:

- Network traffic monitoring (`tcpdump`, `wireshark`)
- Process tracing (`strace -e connect`)
- Firewall rules blocking outbound on ports 80, 443

2.4 Edge LLM Strategy: Minimal Compute at the Boundary

The KOAS architecture deliberately places LLMs at the **edge** of the computation graph, not at its center. This design choice is driven by both sovereignty and efficiency considerations.

Why Edge Placement?

1. **Minimal attack surface** — LLMs only see pre-structured data, not raw documents
2. **Deterministic core** — 90%+ of computation is hash-verifiable
3. **Replaceable models** — Edge LLMs can be swapped without reprocessing
4. **Reduced token consumption** — Kernels compress information before LLM sees it
5. **Parallel scaling** — Multiple documents processed by kernels, LLM only for synthesis

Token Efficiency Analysis *(example values for 137-document corpus):*

Approach	Tokens per Document	Total for 137 docs	Cost Factor
Full-document LLM	~50,000	6.85M tokens	100x
KOAS (kernel → LLM)	~2,000	274K tokens	**4x**
KOAS + caching	~500 (cache hits)	68K tokens	**1x**

Model Selection for Edge Compute:

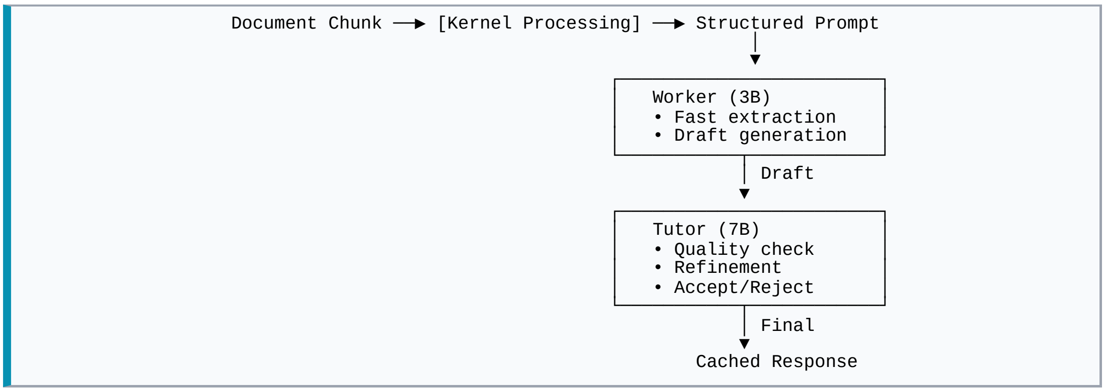
Model	Parameters	VRAM	Role	Justification
granite3.1-moe:3b	3B (MoE)	~4GB	Worker	Sparse activation: only 2B params active per forward pass. Ideal for bulk extraction where speed > quality.
mistral:7b-instruct	7B	~6GB	Tutor	Strong instruction-following. Validates and refines worker outputs. Quality gate.
llama3:8b	8B	~8GB	Alternative	Good balance of quality and speed. Fallback option.
qwen2:7b	7B	~6GB	Multilingual	Essential for non-English corpora (FR, DE, ES).
deepseek-v2:16b	16B	~12GB	High-Quality	For final report synthesis when quality is paramount.

The MoE Advantage:

Mixture-of-Experts models like `granite3.1-moe:3b` are particularly suited for edge compute:

- **Sparse activation:** Only a subset of parameters (experts) activate per token
- **Reduced inference cost:** Effective compute of ~2B despite 3B total parameters
- **Specialization:** Different experts handle different content types

Dual-LLM Architecture:



Document Chunk → [Kernel Processing] → Structured Prompt

Worker (3B)
• Fast extraction
• Draft generation

Draft

Tutor (7B)
• Quality check
• Refinement
• Accept/Reject

Final

Cached Response

This dual-LLM pattern achieves 61%+ acceptance rate on first pass, with the tutor providing quality assurance without human intervention.

Recommendation: Use `granite3.1-moe:3b` as worker for bulk extraction (fast, low VRAM), `mistral:7b-instruct` as tutor for refinement (higher quality validation). Reserve larger models (16B+) for final report synthesis only.

3. The KOAS Paradigm: Kernels Compute, LLMs Interpret

3.1 Separation of Concerns

The fundamental architectural principle of KOAS is:

> **"Kernels compute, LLMs interpret."**

This separation provides:

Property	Kernels	LLMs
Determinism	✅ Identical inputs → identical outputs	× Stochastic
Auditability	✅ Fully traceable	⚠️ Requires caching
Reproducibility	✅ Hash-verifiable	✅ With cache replay
Speed	✅ Fast (compute-bound)	⚠️ Slower (inference)
Sovereignty	✅ No external deps	✅ With local deployment

3.2 Kernel Architecture

KOAS kernels are stateless functions with explicit dependencies:

```python
@kernel("doc_extract", stage=2, requires=["doc_metadata", "doc_concepts"])
def doc_extract(input: KernelInput) -> KernelOutput:
    """
    Extract key sentences from documents with quality scoring.

    INVARIANT: This kernel performs NO LLM calls.
    All processing is deterministic and reproducible.
    """
    # Pure computation: scoring, filtering, deduplication
    ...
```

Key properties:

1. **Explicit dependencies** — DAG-based execution order
2. **Input hashing** — Automatic cache invalidation on change
3. **Output schemas** — Structured JSON with validation
4. **No side effects** — Kernels don't modify input data

3.3 LLM Boundary

LLM calls are confined to specific synthesis kernels:

Kernel	LLM Role	Isolation Mechanism
`doc_func_extract`	Extract functionalities from SPD	Cached, replayable
`doc_summarize`	Generate natural language summaries	Cached, replayable
`doc_summarize_tutored`	Dual-LLM refinement	Cached, replayable

All other kernels (15+) operate without LLM involvement.

3.4 Mathematical Foundation

The KOAS pipeline can be modeled as a directed acyclic graph (DAG) $G = (V, E)$ where:

- V = set of kernels
- E = dependency edges
- For each kernel $k \in V$: $\mathrm{output}_k = f_k(\mathrm{inputs}_k, \mathrm{config}_k)$

Reproducibility Theorem: *For a fixed configuration C and input corpus D, the final report R is deterministic if all LLM calls are replaced by cached responses.*

$$R = \mathrm{Pipeline}(D, C, \mathrm{LLM\_Cache}) = \mathrm{constant}$$

This enables **verification without LLM**: an auditor can replay the entire pipeline using only cached data.

4. KOAS Kernel Inventory and Capabilities

4.1 Complete Kernel Registry

RAGIX/KOAS provides approximately **50 specialized kernels** organized into three families. Each kernel is classified by its **determinism level**, which directly impacts auditability and reproducibility.

Table 1. KOAS Kernel Inventory by Family

CODE AUDIT KERNELS (`RAGIX_KERNELS/AUDIT/`)

Kernel	Stage	Determinism	Purpose	Output
`ast_scan`	1	✅ Full	Parse source code into AST	Symbols, classes, methods, fields
`metrics`	1	✅ Full	Compute code metrics	LOC, complexity, maintainability
`dependency`	1	✅ Full	Build dependency graph	Import/call relationships
`services`	1	✅ Full	Detect service boundaries	Microservice catalog
`coupling`	2	✅ Full	Compute coupling metrics	Afferent/efferent coupling, instability
`hotspots`	2	✅ Full	Identify complexity hotspots	Risk-ranked method list
`dead_code`	2	✅ Full	Detect unreachable code	Unused symbols, orphan files
`entropy`	2	✅ Full	Measure code entropy	Information density metrics
`partition`	2	✅ Full	Graph-based partitioning	Module decomposition proposals
`drift`	2	⚠️ Partial	Detect architectural drift	Pattern violations
`timeline`	2	✅ Full	Git history analysis	Change frequency, authors
`risk`	2	✅ Full	Compute risk scores	Per-file risk assessment
`volumetry`	2	✅ Full	Operational volumetry	Flow weights, peak patterns
`module_group`	2	✅ Full	Functional grouping	Module-to-flow mapping
`risk_matrix`	2	✅ Full	Weighted risk matrix	Risk = f(complexity, volumetry)
`stats_summary`	3	✅ Full	Aggregate statistics	Summary tables
`section_executive`	3	⚠️ Partial	Executive summary	Natural language (LLM)
`section_overview`	3	✅ Full	Technical overview	Structured sections
`section_recommendations`	3	⚠️ Partial	Recommendations	Prioritized action list
`section_drift`	3	✅ Full	Drift report section	Violation details
`report_assemble`	3	✅ Full	Assemble final report	Markdown document

SECURITY KERNELS (`RAGIX_KERNELS/SECURITY/`)

Kernel	Stage	Determinism	Purpose	Output
`net_discover`	1	✅ Full	Network enumeration	Asset inventory
`port_scan`	1	✅ Full	Service detection	Open ports, services
`dns_enum`	1	✅ Full	DNS analysis	Subdomains, records
`config_parse`	1	✅ Full	Parse firewall configs	Iptables, Cisco, pfSense rules
`ssl_analysis`	2	✅ Full	TLS/certificate audit	Cipher suites, expiry, vulnerabilities
`vuln_assess`	2	✅ Full	CVE mapping	Vulnerability matches
`web_scan`	2	✅ Full	Web application scan	OWASP findings
`compliance`	2	✅ Full	Compliance checking	ANSSI/NIST/CIS mappings
`risk_network`	2	✅ Full	Network risk scoring	Per-asset risk
`section_security`	3	⚠️ Partial	Security report	Findings narrative

DOCUMENT KERNELS (`RAGIX_KERNELS/DOCS/`)

Kernel	Stage	Determinism	Purpose	Output
`doc_metadata`	1	✅ Full	Document inventory	File list, chunk counts
`doc_concepts`	1	✅ Full	Concept extraction	Knowledge graph traversal
`doc_structure`	1	✅ Full	Section detection	Heading hierarchy
`doc_quality`	1	✅ Full	Quality scoring	MRI, SRI metrics
`doc_cluster`	2	⚠️ Partial	Hierarchical clustering	Document groups
`doc_cluster_leiden`	2	⚠️ Partial	Community detection	Leiden partitions
`doc_cluster_reconcile`	2	✅ Full	Merge clustering results	Unified grouping
`doc_extract`	2	✅ Full	Key sentence extraction	Quality-filtered excerpts
`doc_coverage`	2	✅ Full	Concept coverage analysis	Gaps, overlaps
`doc_compare`	2	✅ Full	Cross-document comparison	Discrepancies, terminology
`doc_func_extract`	2	× LLM	Functionality extraction	Structured requirements
`doc_pyramid`	3	✅ Full	Hierarchical summary	4-level pyramid
`doc_summarize`	3	× LLM	Natural language summaries	Per-document summaries
`doc_summarize_tutored`	3	× LLM	Dual-LLM refinement	Quality-gated summaries
`doc_visualize`	3	✅ Full	Generate visualizations	Word clouds, graphs
`doc_report_assemble`	3	✅ Full	Assemble report sections	Structured markdown
`doc_final_report`	3	✅ Full	Final report generation	Complete audit report

INFRASTRUCTURE KERNELS

Kernel	Determinism	Purpose
`preflight`	✅ Full	System readiness check
`orchestrator`	✅ Full	DAG execution engine
`cache`	✅ Full	LLM and kernel caching
`llm_wrapper`	⚠️ Partial	LLM call boundary

Legend:

- ✅ **Full** — Identical inputs always produce identical outputs
- ⚠️ **Partial** — Deterministic with fixed seeds or cached responses
- × **LLM** — Requires LLM, deterministic only with cache replay

Seed Locations for Partial-Determinism Kernels:

Kernel	Seed Source	Audit Trail Field	Reproducibility
`doc_cluster`	`config.clustering_seed`	`kernel_execution[].seed`	Same seed → same clusters
`doc_cluster_leiden`	`config.leiden_seed`	`kernel_execution[].seed`	Same seed → same partitions
`partition`	`config.partition_seed`	`kernel_execution[].seed`	Same seed → same modules
`section_executive`	LLM cache	`llm_calls[].call_hash`	Same cache → same text
`section_recommendations`	LLM cache	`llm_calls[].call_hash`	Same cache → same text
`section_security`	LLM cache	`llm_calls[].call_hash`	Same cache → same text
`llm_wrapper`	LLM cache	`llm_calls[].call_hash`	Same cache → same response

To reproduce a run: Extract seeds from audit trail and pass via `--seed-file=audit_trail.json`.

4.2 Determinism Spectrum: From Pure Computation to LLM Synthesis

KOAS kernels span a spectrum of determinism, enabling different audit and reproducibility guarantees.

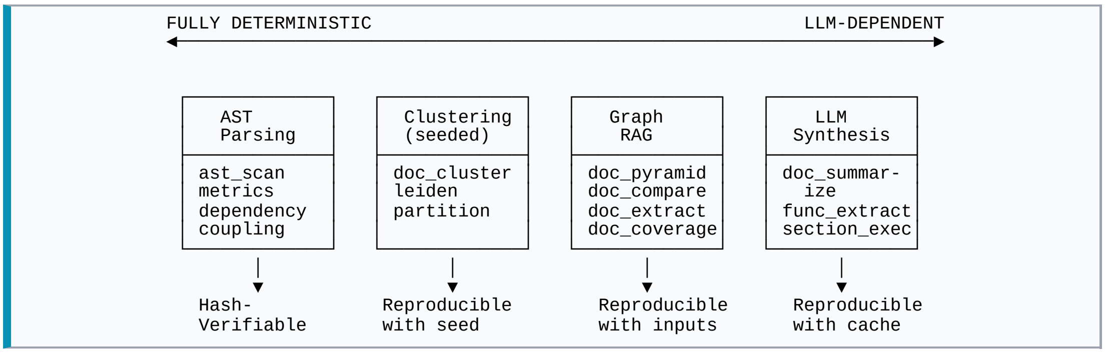


Figure 3. Determinism Spectrum

4.3 Capabilities by Analysis Mode

MODE A: FULLY DETERMINISTIC (AST-BASED CODE AUDIT)

Use case: Security audits, compliance verification, technical debt assessment

Activated kernels: `ast_scan` → `metrics` → `dependency` → `coupling` → `hotspots` → `dead_code` → `risk` → `report_assemble`

Capabilities:

- **Root cause identification:** Trace bugs to specific code locations
- **Complexity hotspots:** Identify methods requiring refactoring
- **Coupling analysis:** Detect architectural violations
- **Dead code detection:** Find unused symbols for removal
- **Dependency mapping:** Full call graph visualization
- **Technical debt estimation:** Remediation effort in hours

Example output *(illustrative values, actual numbers vary by codebase):*

```
{
  "hotspots": [
    {
      "symbol": "PaymentProcessor.processTransaction",
      "file": "src/payments/processor.py:142",
      "complexity": 47,
      "risk_score": 0.89,
      "recommendation": "Extract validation logic into separate methods"
    }
  ],
  "dead_code": {
    "unused_methods": 23,
    "orphan_files": 4,
    "estimated_loc_removable": 1847
  }
}
```

MODE B: SEMI-DETERMINISTIC (GRAPH RAG + PYRAMIDAL DECOMPOSITION)

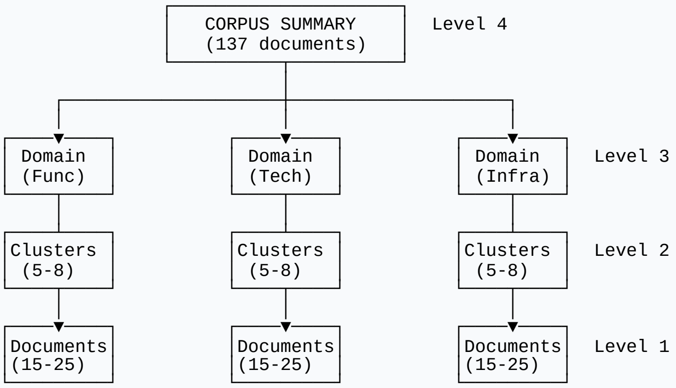
Use case: Large corpus analysis, concept extraction, cross-document comparison

Activated kernels: `doc_metadata` → `doc_concepts` → `doc_cluster` → `doc_cluster_leiden` → `doc_extract` → `doc_compare` → `doc_pyramid`

Capabilities:

- **Concept extraction:** Identify key themes across corpus
- **Analogies and patterns:** Detect similar structures in different documents
- **Repetition detection:** Find redundant content
- **Terminology variations:** Track inconsistent naming
- **Cross-reference validation:** Verify document links
- **Hierarchical summarization:** Document → Cluster → Domain → Corpus

Pyramidal decomposition:



Example output (discrepancies) *(illustrative values):*

```
{
  "discrepancies": [
    {
      "type": "terminology_variation",
      "base_term": "authentification",
      "variants": ["authentication", "authent.", "auth."],
      "occurrences": 47
    },
    {
      "type": "missing_reference",
      "source": "DOC-042",
      "reference": "SPD-99",
      "context": "See SPD-99 for protocol details"
    }
  ]
}
```

MODE C: LLM-ENHANCED (FULL DOCUMENT SYNTHESIS)

Use case: Executive summaries, natural language reports, functionality extraction

Activated kernels: All of Mode B + `doc_func_extract` → `doc_summarize_tutored` → `doc_final_report`

Capabilities:

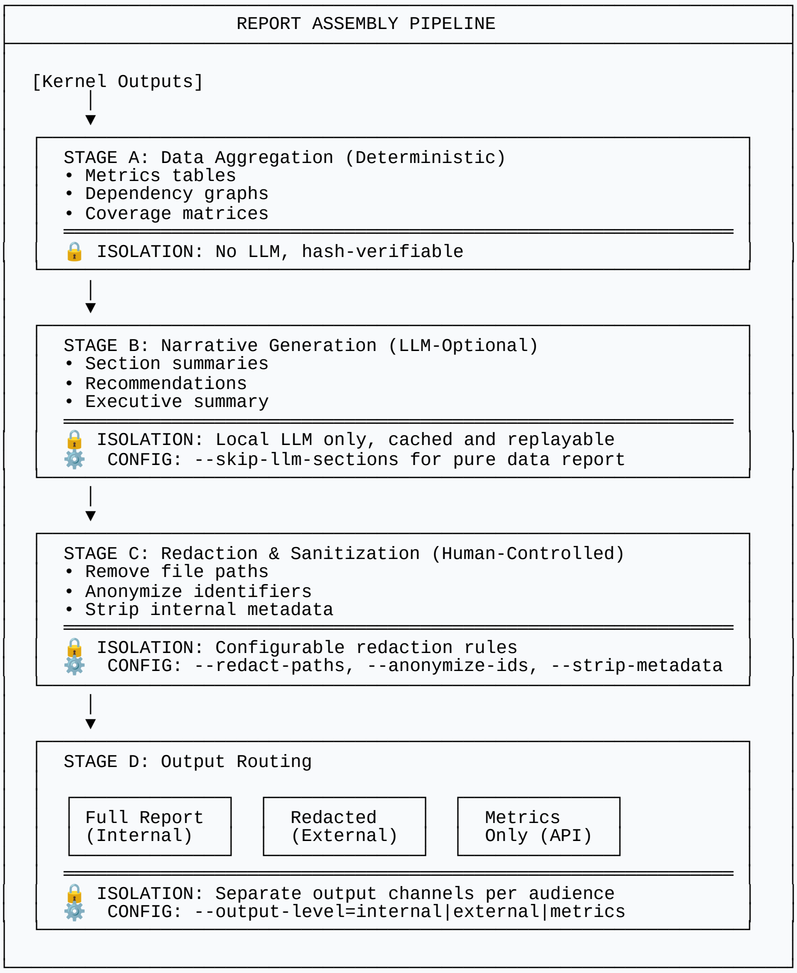
- **Natural language summaries:** Human-readable document descriptions
- **Functionality extraction:** Structured requirements from specifications
- **Executive synthesis:** High-level insights for decision-makers
- **Quality-gated generation:** Dual-LLM validation

Example output (functionality) *(illustrative values):*

```
{
  "functionalities": [
    {
      "id": "FUNC-016-01",
      "name": "User Authentication via SSO",
      "description": "The system shall authenticate users through corporate
SSO...",
      "actors": ["End User", "Administrator"],
      "trigger": "Login attempt",
      "references": ["DOC-034", "DOC-035"],
      "category": "security"
    }
  ]
}
```

4.4 Automatic Report Generation with Configurable Isolation

KOAS generates reports through an assembly pipeline where **human control** is paramount. Each stage offers configuration points for isolation level.



```
REPORT ASSEMBLY PIPELINE

[Kernel Outputs]
      │
      ▼
STAGE A: Data Aggregation (Deterministic)
• Metrics tables
• Dependency graphs
• Coverage matrices
🔒 ISOLATION: No LLM, hash-verifiable
      │
      ▼
STAGE B: Narrative Generation (LLM-Optional)
• Section summaries
• Recommendations
• Executive summary
🔒 ISOLATION: Local LLM only, cached and replayable
⚙️  CONFIG: --skip-llm-sections for pure data report
      │
      ▼
STAGE C: Redaction & Sanitization (Human-Controlled)
• Remove file paths
• Anonymize identifiers
• Strip internal metadata
🔒 ISOLATION: Configurable redaction rules
⚙️  CONFIG: --redact-paths, --anonymize-ids, --strip-metadata
      │
      ▼
STAGE D: Output Routing

  Full Report      Redacted        Metrics
  (Internal)       (External)      Only (API)

🔒 ISOLATION: Separate output channels per audience
⚙️  CONFIG: --output-level=internal|external|metrics
```

Figure 4. Report Assembly Pipeline

Isolation Configuration Matrix:

Scenario	LLM Sections	File Paths	Identifiers	Metadata	Output
Internal Audit	✅ Include	✅ Include	✅ Include	✅ Include	Full
External Delivery	✅ Include	× Redact	⚠️ Anonymize	× Strip	Redacted
Orchestrator View	× Skip	× Redact	× Redact	× Strip	Metrics
Compliance Archive	✅ Include	✅ Include	✅ Include	✅ Include	Full + Attestation

4.5 Human Control Points

Throughout the pipeline, humans retain full control:

Control Point	Options	Default
Kernel selection	Include/exclude specific kernels	All enabled
LLM model choice	Swap worker/tutor models	granite3.1-moe:3b / mistral:7b
Cache policy	write_through / read_only / off	write_through
Redaction rules	Custom regex patterns	None
Output channels	Internal / external / metrics	Internal
Approval gates	Require human sign-off per stage	Disabled
Audit verbosity	Minimal / standard / verbose	Standard

Example: Maximum Isolation Configuration

```
ragix-koas run \
    --workspace ./audit \
    --all \
    --llm-cache=read_only \
    --kernel-cache=read_only \
    --skip-llm-sections \
    --redact-paths \
    --anonymize-ids \
    --strip-metadata \
    --output-level=metrics \
    --require-approval
```

This produces a metrics-only report suitable for external orchestrators, with all sensitive information removed.

5. Policy Enforcement Mechanisms

5.1 Forced LLM Caching with Canonicalization

For cache replay to work reliably, LLM requests must be **canonicalized** before hashing.

CANONICALIZATION RULES

```python
def canonicalize_llm_request(request: dict) -> str:
    """Produce stable JSON for cache key computation."""
    canonical = {
        "model": request["model"],
        "temperature": request.get("temperature", 0.0),
        "template_id": request.get("template_id"),
        "template_version": request.get("template_version"),
        "messages": _canonicalize_messages(request["messages"]),
    }
    # Sorted keys, no whitespace variance
    return json.dumps(canonical, sort_keys=True, separators=(',', ':'))

def _canonicalize_messages(messages: List[dict]) -> List[dict]:
    """Normalize message content."""
    result = []
    for msg in messages:
        content = msg["content"]
        # Normalize whitespace
        content = re.sub(r'\s+', ' ', content).strip()
        # Remove volatile fields (timestamps, run IDs)
        content = re.sub(r'run_\d{8}_\d{6}_[a-f0-9]+', 'RUN_ID', content)
        result.append({"role": msg["role"], "content": content})
    return result
```

MERKLE ROOT FOR PYRAMIDAL SYNTHESIS

When summarizing aggregated content (cluster → domain → corpus), children must be ordered deterministically:

```python
def compute_inputs_merkle_root(children: List[dict]) -> str:
    """Compute Merkle root of child inputs for cache key."""
    # Step 1: Sort children deterministically
    sorted_children = sorted(children, key=lambda c: (c["file_path"],
c.get("chunk_index", 0)))

    # Step 2: Hash each child
    child_hashes = [sha256(c["content_hash"]) for c in sorted_children]

    # Step 3: Build Merkle tree
    while len(child_hashes) > 1:
        if len(child_hashes) % 2 == 1:
            child_hashes.append(child_hashes[-1])  # Duplicate last if odd
        child_hashes = [sha256(child_hashes[i] + child_hashes[i+1])
                       for i in range(0, len(child_hashes), 2)]

    return child_hashes[0] if child_hashes else sha256("")
```

Cache Key Structure:

```
{
  "call_hash": "sha256(canonical_request)",
  "inputs_merkle_root": "sha256(ordered_children_tree)",
  "template": "docs/domain_summary@1.3",
  "model": "granite3.1-moe:3b",
  "temperature": 0.3
}
```

5.2 Cache Modes

The `--llm-cache` flag controls LLM interaction:

Mode	Behavior	Use Case
`write_through`	Call LLM, cache response	Normal operation
`read_only`	Use cache only, fail on miss	Verification, air-gapped
`read_prefer`	Cache first, LLM fallback	Incremental updates
`off`	No caching	Fresh analysis (rare)

Air-gapped operation:

```
# Populate cache (requires LLM)
ragix-koas run --workspace ./audit --all --llm-cache=write_through

# Transfer cache to air-gapped system
rsync -av ./audit/.KOAS/cache/ /air-gapped/audit/.KOAS/cache/

# Verify on air-gapped system (no LLM required)
ragix-koas run --workspace /air-gapped/audit --all --llm-cache=read_only
```

5.3 Kernel Cache Modes

The `--kernel-cache` flag controls kernel output caching:

Mode	Behavior	Use Case
`write_through`	Execute kernel, cache output	Normal operation
`read_only`	Use cached output, skip execution	Ultra-fast replay
`off`	Always execute	After code changes

Combined for maximum speed:

```
# Full replay (no computation, no LLM)
ragix-koas run --workspace ./audit --all \
    --llm-cache=read_only \
    --kernel-cache=read_only
# Result: ~45 seconds for 137 documents (vs ~12 minutes full run)
```

5.4 Input Validation and Sanitization

Before processing, documents undergo validation. **Critical distinction:**

BOILERPLATE DETECTION (PURE DOCS MODE ONLY)

Scope: Applies ONLY to document corpora (`--type docs`). NEVER applied to code files or mixed mode.

Purpose: Filter non-informative content (release notes, infrastructure notation) from representative excerpts.

Protection: Fenced code blocks (`` ``` ``) and inline code (`` ` ``) are **excluded** from boilerplate matching:

````python
def _apply_boilerplate_filter(text: str, patterns: List[str]) -> str:
    """Apply boilerplate patterns, protecting code blocks."""
    # Step 1: Extract and protect code blocks
    code_blocks = re.findall(r'```[\s\S]*?```', text)
    protected = text
    for i, block in enumerate(code_blocks):
        protected = protected.replace(block, f'__CODE_BLOCK_{i}__')

    # Step 2: Apply boilerplate patterns to non-code content
    for pattern in patterns:
        protected = re.sub(pattern, '', protected, flags=re.IGNORECASE)

    # Step 3: Restore code blocks
    for i, block in enumerate(code_blocks):
        protected = protected.replace(f'__CODE_BLOCK_{i}__', block)

    return protected
````

Language Packs: Boilerplate vocabulary is language-specific:

Language	Config Key	Example Patterns
French	`boilerplate_vocab_fr`	`Sommaire`, `Table des matières`, `Confidentiel`
English	`boilerplate_vocab_en`	`Table of Contents`, `Confidential`, `Draft`
German	`boilerplate_vocab_de`	`Inhaltsverzeichnis`, `Vertraulich`
Changelog	`boilerplate_changelog`	`X.X.X.X MERGED`, `VM1 VM2`

Selected via `--language` flag or corpus auto-detection.

REPRESENTATIVE EXTRACTION QUALITY (DOC_EXTRACT)

Scope: Applies to `doc_extract` kernel in all modes.

Method: Boilerplate is penalized via `_score_sentence_quality()`, NOT by cleaning/modifying text:

```python
def _score_sentence_quality(sentence: str, config: QualityConfig) -> float:
    """Score sentence quality. Original text is NEVER modified."""
    score = 0.5  # Base score

    # Penalties (reduce score)
    if _matches_boilerplate(sentence, config):
        score -= 0.3  # Heavy penalty for boilerplate
    if _is_truncated(sentence):
        score -= 0.2
```

```
    # Bonuses (increase score)
    if _has_technical_terms(sentence, config):
        score += 0.15
    if _has_action_verbs(sentence):
        score += 0.1
    if _optimal_length(sentence):
        score += 0.1

    return max(0.0, min(1.0, score))
```

Key invariant: The sentence text itself is preserved verbatim. Low-scoring sentences are excluded from representative excerpts, not modified.

OTHER VALIDATIONS

1. **Metadata stripping** — Separate front-matter from content (see §5.4)
2. **Encoding normalization** — UTF-8 standardization
3. **Path sanitization** — Prevent traversal attacks

5.5 Metadata Stripping Contract

Definition of Metadata (Pipeline Invariant)

The following are classified as **internal metadata** and MUST be stripped before external output:

Type	Pattern	Example
YAML front-matter	`^---\n...\n---` at file start	`title: ...\nllm_trace: ...`
TOML front-matter	`^+++\n...\n+++` at file start	`[meta]\ncall_hash = "..."`
JSON front-matter	`` ^```json\n{...}\n```\n `` at file start	Embedded config blocks
Fenced metadata	`` ```metadata\n...\n``` ``	Provenance blocks
HTML provenance	`<!-- PROVENANCE ... -->`	Inline trace markers
Internal keys	Any of: `llm_trace`, `call_hash`, `inputs_merkle_root`, `run_id`, `endpoint`, `model`, `cache_status`	Audit fields

Two-Channel Output Model

Every document processing pipeline produces two representations:

```
┌─────────────────────────────────────────────────────────┐
│                  DOCUMENT PROCESSING                    │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  Raw Document ──▶ [Parser] ──┬──▶ content_clean (external)
│                              │                          │
│                              └──▶ metadata (internal only)
│                                                         │
│  content_clean: Markdown with all metadata blocks removed
│  metadata: Parsed dict retained for traceability/caching│
│                                                         │
└─────────────────────────────────────────────────────────┘
```

Usage Rules:

Context	Allowed Source	Rationale
User-facing reports	`content_clean` only	No provenance leak
LLM summarization	`content_clean` only	No meta-confusion
Embedding/retrieval	`content_clean` only	Clean vectors
Internal audit	Both	Full traceability
Cache key computation	`metadata`	Provenance tracking

Build-Time Validation (Denylist Check)

The report assembly pipeline includes a mandatory denylist scan:

```python
DENYLIST_KEYS = [
    "llm_trace", "call_hash", "inputs_merkle_root",
    "run_id", "endpoint", "model", "cache_status",
    "prompt_tokens", "completion_tokens", "digest"
]

def validate_external_report(report_path: Path) -> bool:
    """Fail build if denylist keys appear in external report."""
    content = report_path.read_text()
    for key in DENYLIST_KEYS:
        if key in content:
            raise SecurityViolation(f"Denylist key '{key}' found in external
report")
    return True
```

This check runs automatically when `--output-level=external` is specified.

5.6 Output Isolation

Audit artifacts are written to `.KOAS/` directory:

```
workspace/
├── src/                    # Input documents (READ-ONLY)
│   ├── doc1.pdf
│   └── doc2.docx
└── .KOAS/                  # Output directory (WRITE)
    ├── cache/              # LLM and kernel caches
    ├── stage1/             # Collection outputs
    ├── stage2/             # Analysis outputs
    ├── stage3/             # Synthesis outputs
    ├── final_report.md     # Generated report
    └── audit_trail.json    # Provenance log
```

This separation ensures:

- Input documents are never modified
- Outputs don't contaminate future RAG indexing
- Clear audit boundary

5.7 External vs Internal Output Contract

The `--output-level` flag enforces isolation rules at build time. This is a **hard contract**, not a guideline.

Table 2. Output Level Contract

Level	Metadata	File Paths	Identifiers	Excerpts	Enforced By
`internal`	✅ Include	✅ Include	✅ Include	✅ Include	Default
`external`	× **Strip**	× **Redact**	⚠️ **Anonymize**	✅ Include	CLI + Build
`orchestrator`	× Strip	× Redact	× Redact	× **Metrics only**	CLI + Build
`compliance`	✅ Include	✅ Include	✅ Include	✅ Include	+ Attestation

CLI Enforcement:

```bash
# External delivery (client-facing)
ragix-koas run --workspace ./audit --all --output-level=external
# Enforces: strip metadata, redact paths, anonymize IDs
# Build FAILS if denylist keys detected in output

# Orchestrator view (Claude/GPT-4 integration)
ragix-koas run --workspace ./audit --all --output-level=orchestrator
# Enforces: metrics only, no text content
# Returns: kernel names, timing, success/failure, counts

# Compliance archive (full audit trail)
ragix-koas run --workspace ./audit --all --output-level=compliance
# Enforces: full content + sovereignty attestation
# Includes: hash chains, provenance markers, timestamps
```

Build-Time Validation:

When `--output-level=external`:

1. Denylist scan runs on all generated markdown
2. Path patterns (`/home/`, `/path/to/`, `C:\`) trigger redaction or failure
3. Internal ID formats (`F000XXX`, `run_YYYYMMDD_`) are anonymized

Example: Anonymization Transform

```
Before: "Document F000142 (src/specs/SPD-16.docx) shows..."
After:  "Document [DOC-A] ([PATH-REDACTED]) shows..."
```

Non-Negotiable: Absent `--output-level` flag defaults to `internal`. External outputs require explicit declaration.

6. Operational Modes

6.1 Mode 1: Documents Only

Analyze a corpus of technical documents without code.

Applicable to: Specification audits, policy reviews, contract analysis.

```bash
ragix-koas init --project ./specs --name "Contract Review" --type docs
ragix-koas run --workspace ./specs/.KOAS --all
```

Kernels activated: `doc_metadata`, `doc_concepts`, `doc_structure`, `doc_cluster`, `doc_extract`, `doc_pyramid`, `doc_summarize`, `doc_final_report`

6.2 Mode 2: Documents + Code

Combined analysis of specifications and implementation.

Applicable to: Compliance verification, specification-implementation gap analysis.

```
ragix-koas init --project ./system --name "Compliance Audit" --type mixed
ragix-koas run --workspace ./system/.KOAS --all --include-code
```

Additional kernels: `ast_scan`, `metrics_compute`, `coverage_analyze`, `drift_detect`

6.3 Mode 3: Code Only

Pure code audit without documentation.

Applicable to: Security audits, technical debt assessment, refactoring planning.

```
ragix-koas init --project ./codebase --name "Security Audit" --type code
ragix-koas run --workspace ./codebase/.KOAS --all
```

Kernels activated: Standard KOAS code audit pipeline.

7. Gray Environment Deployment

7.1 Definition

A **gray environment** is an operational context where:

1. Network connectivity exists but is untrusted
2. External API calls are prohibited by policy
3. Data must not leave the local system
4. Auditability is mandatory

This differs from:

- **Green environment:** Full connectivity, cloud services allowed
- **Air-gapped (black):** No network connectivity

7.2 Network Configuration

Firewall rules for gray deployment:

```
# Allow only localhost for LLM
iptables -A OUTPUT -d 127.0.0.1 -j ACCEPT
iptables -A OUTPUT -d ::1 -j ACCEPT

# Block known LLM API endpoints
iptables -A OUTPUT -d api.openai.com -j DROP
iptables -A OUTPUT -d api.anthropic.com -j DROP
iptables -A OUTPUT -d generativelanguage.googleapis.com -j DROP

# Log any other outbound attempts (for audit)
iptables -A OUTPUT -j LOG --log-prefix "OUTBOUND_ATTEMPT: "
iptables -A OUTPUT -j DROP
```

7.3 Verification Procedure

Before processing sensitive data, verify sovereignty:

```bash
# 1. Check Ollama is local
curl -s http://127.0.0.1:11434/api/tags | jq '.models[].name'

# 2. Verify no external endpoints in config
grep -r "api\." ~/.ragix/ /etc/ragix/ 2>/dev/null | grep -v localhost

# 3. Test network isolation
timeout 5 curl -s https://api.openai.com/v1/models && echo "FAIL: External reachable" || echo "OK: External blocked"

# 4. Run RAGIX system check
ragix-koas status --sovereignty-check
```

7.4 Sovereignty Attestation

The audit trail includes sovereignty attestation:

```json
{
  "sovereignty": {
    "hostname": "audit-workstation-01",
    "user": "auditor",
    "platform": "Linux 6.8.0-90-generic",
    "python_version": "3.12.12",
    "llm_endpoint": "http://127.0.0.1:11434",
    "llm_local": true,
    "models_used": [
      {"name": "granite3.1-moe:3b", "digest": "b43d80d7fca7", "role": "worker"},
      {"name": "mistral:7b-instruct", "digest": "6577803aa9a0", "role": "tutor"}
    ],
    "external_calls": 0,
    "attestation": "All processing performed locally. No data sent to external services."
  }
}
```

8. Attestation and Audit Trail

8.1 Audit Trail Structure

Every KOAS run produces a comprehensive audit trail:

```json
{
  "_meta": {
    "version": "1.0.0",
    "generated_at": "2026-01-29T14:05:10Z",
    "generator": "KOAS Document Summarization v0.64.2"
  },
  "run_id": "run_20260129_140510_8a3f2c",
  "sovereignty": { ... },
  "configuration": {
    "project_root": "/path/to/audit",
    "language": "fr",
    "llm_model": "granite3.1-moe:3b",
    "llm_cache_mode": "write_through",
    "kernel_cache_mode": "write_through"
  },
  "kernel_execution": [
    {
      "name": "doc_metadata",
      "stage": 1,
      "success": true,
      "execution_time_s": 0.01,
      "input_hash": "sha256:6c9d206bc3aa10d3...",
      "output_hash": "sha256:4b35b47917038e35...",
      "llm_calls": 0
    },
    // ... all kernels
  ],
  "llm_calls": [
    {
      "call_hash": "sha256:a3f2c8d1...",
      "kernel": "doc_summarize",
      "model": "granite3.1-moe:3b",
      "prompt_tokens": 1247,
      "completion_tokens": 312,
      "cache_status": "miss",
      "timestamp": "2026-01-29T14:06:23Z"
    }
  ],
  "checksums": {
    "input_corpus": "sha256:...",
    "final_report": "sha256:...",
    "audit_trail": "sha256:..."
  }
}
```

8.2 Hash Chain Integrity

Logs are protected by SHA-256 hash chaining:

```
Entry[0]: { data, hash_0 = SHA256(data) }
Entry[1]: { data, hash_1 = SHA256(data || hash_0) }
Entry[n]: { data, hash_n = SHA256(data || hash_{n-1}) }
```

Verification:

```
ragix verify --audit-trail ./audit/.KOAS/audit_trail.json
# Output: "Chain integrity verified. 247 entries, no tampering detected."
```

8.3 Provenance Tracking

Every generated statement can be traced:

```
## Domain: Infrastructure Management

This domain covers network configuration and system monitoring...

<!-- PROVENANCE
call_hash: sha256:a3f2c8d1e5b7...
input_documents: [F000023, F000024, F000031]
kernel: doc_summarize
timestamp: 2026-01-29T14:06:23Z
-->
```

Metadata stripping: Provenance markers are stripped from user-facing outputs but preserved in internal audit artifacts.

9. Practical Implementation

9.1 Prerequisites

Hardware:

- CPU: 8+ cores recommended
- RAM: 16GB minimum, 32GB recommended
- Storage: SSD, 50GB+ free space
- GPU: Optional but accelerates inference (NVIDIA 8GB+ VRAM)

Software:

```bash
# Ollama installation
curl -fsSL https://ollama.com/install.sh | sh

# Model download (offline-capable)
ollama pull granite3.1-moe:3b
ollama pull mistral:7b-instruct

# RAGIX installation
pip install ragix[full]
```

9.2 Initialization

```bash
# Create audit workspace
ragix-koas init \
    --project /path/to/documents \
    --name "Technical Audit 2026" \
    --language fr \
    --type docs

# Verify configuration
cat /path/to/documents/.KOAS/manifest.yaml
```

9.3 Execution

```bash
# Full pipeline (populates caches)
ragix-koas run \
    --workspace /path/to/documents/.KOAS \
    --all \
    --llm-cache=write_through \
    --kernel-cache=write_through

# Monitor progress
tail -f /path/to/documents/.KOAS/logs/koas.log
```

9.4 Verification

```bash
# Replay without LLM (verification mode)
ragix-koas run \
    --workspace /path/to/documents/.KOAS \
    --all \
    --llm-cache=read_only \
    --kernel-cache=read_only

# Verify audit trail
ragix verify --audit-trail /path/to/documents/.KOAS/audit_trail.json

# Generate sovereignty report
ragix-koas report --sovereignty --format pdf
```

10. Case Study: Metropolitan Infrastructure Audit

10.1 Context

- **Client:** Major metropolitan authority
- **Scope:** Technical specification corpus for urban infrastructure system
- **Documents:** 137 files (DOCX, PDF, PPTX, XLSX)
- **Chunks:** 11,809 indexed segments
- **Constraint:** Full data sovereignty required

10.2 Execution Metrics

Table 3. Execution Metrics *(example run, actual values vary by corpus size and hardware):*

Stage	Duration	Kernels	LLM Calls	Cache Hits
Stage 1	54s	6	0	N/A
Stage 2	~11 min	8	340	39%
Stage 3	42s	6	137	61%
Total	**~13 min**	**20**	**477**	**52%**

Replay (cached): ~45 seconds *(example)*

10.3 Quality Improvements (v0.64.2)

Metric	Before	After	Improvement
Boilerplate sentences	649	644	-5 (0.8%)
Changelog data in excerpts	Present	Filtered	✅
Infrastructure notation	Present	Filtered	✅
Report path	Wrong	Correct	✅

10.4 Sovereignty Verification

```
$ ragix-koas status --sovereignty-check
✓ LLM endpoint: localhost:11434 (local)
✓ External calls: 0
✓ Models verified: granite3.1-moe:3b, mistral:7b-instruct
✓ Cache integrity: 477 entries, chain valid
✓ Attestation: SOVEREIGN
```

11. Limitations and Mitigations

11.1 Model Quality vs. Cloud Services

Limitation: Local 3B-7B models produce lower quality outputs than GPT-4 or Claude Opus.

Mitigations:

1. **Dual-LLM architecture** — Worker + Tutor refinement
2. **Pre-structured prompts** — Kernels reduce LLM cognitive load
3. **Iterative refinement** — Multiple passes with quality gates
4. **Human review** — Final outputs require expert validation

11.2 Computational Resources

Limitation: Local inference requires significant hardware.

Mitigations:

1. **MoE models** — granite3.1-moe:3b activates only 2B params
2. **Kernel caching** — Avoid redundant computation
3. **Batch processing** — Optimize GPU utilization
4. **Incremental updates** — Process only changed documents

11.3 Model Updates

Limitation: Local models don't receive automatic updates.

Mitigations:

1. **Version pinning** — Lock model digests in manifest
2. **Controlled updates** — Manual review before deployment
3. **Regression testing** — Verify outputs against baselines

12. Conclusion

The RAGIX/KOAS framework demonstrates that sovereign LLM operations are achievable for document analysis tasks requiring confidentiality. The key architectural decisions enabling this are:

1. **Local LLM deployment** — Ollama eliminates external API dependencies
2. **Kernel-LLM separation** — Deterministic computation isolated from stochastic inference
3. **Comprehensive caching** — Enables LLM-free replay and verification
4. **Audit trail integrity** — Hash-chained provenance for compliance
5. **Policy enforcement** — Cache modes control data flow

For organizations operating in regulated environments, this approach provides:

- **Compliance** — Data never leaves controlled perimeter
- **Auditability** — Every transformation is traceable
- **Reproducibility** — Results can be independently verified
- **Cost control** — No per-token API charges

The v0.64.2 release demonstrates continuous improvement in content quality while maintaining sovereignty guarantees, with measurable metrics tracking regressions and improvements across versions.

13. References

Academic Foundations

1. Lewis, P. et al. (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." *NeurIPS 2020*.
2. Edge, D. et al. (2024). "From Local to Global: A Graph RAG Approach to Query-Focused Summarization." *arXiv:2404.16130*.
3. Vitrac, O. (2025). "Virtual/Hybrid R&D Laboratories built with Augmented-AI Agents." *Generative Simulation Initiative*.

Standards and Regulations

4. **ANSSI** (2024). "Guide d'hygiène informatique." Agence nationale de la sécurité des systèmes d'information.
5. **GDPR** (2016). "General Data Protection Regulation." European Union Regulation 2016/679.
6. **ISO/IEC 27001:2022** — Information security management systems.

Technical Documentation

7. Ollama Documentation. https://ollama.com/docs
8. RAGIX Project Documentation. Internal technical specifications.
9. KOAS Kernel Specifications. `docs/KOAS_DOCS.md`

Appendix A: Sovereignty Checklist

Before processing sensitive data, verify:

- ☐ Ollama running on localhost:11434
- ☐ Required models downloaded and verified
- ☐ Firewall rules blocking external LLM APIs
- ☐ RAGIX configured with `llm_endpoint: http://127.0.0.1:11434`
- ☐ Cache directory on local encrypted storage
- ☐ Audit trail enabled
- ☐ No cloud backup of `.KOAS/` directory

Appendix B: Glossary

Term	Definition
Air-gapped	System with no network connectivity
Gray environment	Networked but untrusted, external calls prohibited
KOAS	Kernel-Orchestrated Audit System
MoE	Mixture of Experts (sparse model architecture)
Ollama	Local LLM runtime
RAGIX	Retrieval-Augmented Generative Interactive eXecution
Sovereignty	Data processing within controlled perimeter

RAGIX/KOAS — Adservio Innovation Lab | 2026