Comprehensive Report (Final Iteration): Sovereign AI Infrastructure, Bicameral Validator Ladder, and Hybrid Logic Router

Source basis: "Deep Agent Report.pdf" (final evolution: bicameral, zero-swap validation) + prior language assessment (Python default, Prolog fit) + explicit inclusion of embedding models, OCR, and vision encoders.

Executive Summary

You are building a **sovereign-grade local multi-model system** optimized for constrained VRAM (Tesla A2 16GB) but abundant system RAM (128GB ECC). The final iteration converges on a **Bicameral Architecture**:

- **GPU (Worker hemisphere):** high-throughput generation (coding/reasoning/creative specialists)
- **CPU (Validator + Router hemisphere):** always-on governance—routing, validation, and policy enforcement—without GPU model thrashing
- **Markdown memory bus:** transparent, audit-friendly persistent state (project_state.md, scratchpad.md, knowledge_graph.md)

This report formalizes that final iteration and extends it to include the capabilities you explicitly want:

- **Embedding models** (for retrieval, routing support, semantic memory)
- **OCR** (for document ingestion and grounded extraction)
- **Vision encoders** (for image understanding, multimodal routing and validation inputs)

A key design conclusion (from the prior response) is that while **Python remains essential** for orchestration and integration, the router you're building is fundamentally a **logic/constraint engine**; therefore **Prolog (or a comparable logic programming layer)** is the best structural fit for the governance/routing core. The strongest architecture is hybrid: **Python = muscle**, **Prolog = prefrontal cortex**.

---

1. System Goal and Design Principles (Final Iteration)

1.1 Sovereign objectives (as defined in the document)

- **Local control:** no external API dependencies
- **Quality governance:** multi-layer validation; line-by-line proof-checking where needed
- **Resource optimization:** maximize capability under VRAM constraints via warm pools and heterogeneous compute
- **Transparency:** fully observable, auditable decisions via a persistent memory ledger

1.2 Constraint-driven architecture

The final iteration is shaped by one core reality: **VRAM is scarce; RAM is abundant**. The system must avoid repeated GPU swaps and instead use CPU persistence for control functions.

2. Final Iteration Architecture: Bicameral "Zero-Swap" Validator Ladder

2.1 Core components and placement

- **Worker (GPU resident):** a single specialist model loaded in VRAM at a time (e.g., Qwen Coder / Nemotron / GPT-OSS / MythoMax)
- **Validator (CPU resident, always loaded):** Granite-H-Small (MoE; ~9B active) used for strict review and governance
- **Router (CPU resident, always loaded):** Granite-Micro for intent classification and orchestration decisions
- **Memory bus:** Markdown files updated continuously and treated as the shared "brain ledger"

2.2 The validated workflow (Generate → Validate → Commit)

1. **Router classifies** request (domain, stakes, depth, tools needed)
2. **Scout/Context stage** gathers relevant context (embedding retrieval; OCR/vision if needed)
3. **Worker generates** in small blocks (code chunk, paragraph, step)
4. **Validator checks** block against constraints + project state
5. If **PASS** → commit to memory + continue
6. If **FAIL** → write correction directive to scratchpad.md; worker retries

This is the report's "proof checker" loop—made practical by avoiding GPU swaps.

---

3. Language Choice: Python Default, Prolog Fit (Governance Router)

3.1 Why Python remains necessary

Python should own:

- model serving orchestration (llama.cpp endpoints, GPU/CPU dispatch)
- file I/O (Markdown memory ledger)
- embeddings pipeline, OCR pipeline, vision pipeline
- monitoring/telemetry and operational glue

3.2 Why Prolog is structurally aligned with your router

Your router is not merely dispatch logic; it is a **constraint-enforcing symbolic controller**. Prolog naturally supports:

- declarative routing rules ("what must hold")
- pattern matching over structured symbolic forms (facts, metadata, tool results)

- backtracking over candidate plans/models
- constraint solving for invariants and “epistemic hygiene”
- inspectable decision traces (important for sovereignty posture)

3.3 Recommended hybrid boundary

- **Python:** runs pipelines, calls models, manages memory files, performs OCR/vision/embedding compute
- **Prolog:** decides *which* pipeline/model/validator policy applies and *why*, and emits an inspectable decision trace (that Python logs into the ledger)

---

4. Expanded Capability Set You Requested

The final iteration in the document already includes a “Scout” concept via embeddings. You want this explicitly broadened to OCR + vision encoders. Below is the integrated design.

4.1 Embedding models (Semantic retrieval + routing support)

Roles:

1. **Retrieval Augmented Context (RAG):** fetch relevant snippets before generation/validation
2. **Semantic memory indexing:** index Markdown memory entries so the system can recall “what we already decided”
3. **Router assistance:** when Granite-Micro is uncertain, embeddings provide similarity-based fallback signals (e.g., nearest known prompt clusters)

Where embeddings live: CPU (fast enough), with a vector store on disk/RAM.

Key outputs to ledger:

- retrieved sources/snippets
- similarity scores/top-k metadata
- a “why these docs” explanation for auditability

4.2 OCR (Document ingestion and grounded extraction)

Why OCR matters here: the architecture is governance-first. OCR lets you ingest scanned PDFs/images while preserving a provenance trail.

Workflow integration:

- When input is image/PDF scan → route to **OCR pipeline** first
- OCR output becomes a **grounded text artifact** committed to memory (with page references if available)
- Worker generates only from OCR-extracted text + citations; Validator checks that claims map back to OCR text where required

Ledger artifacts:

- OCR text blocks with coordinates/page refs if possible
- confidence / unreadable regions
- "extraction caveats" surfaced to validator

4.3 Vision encoders (Image understanding + multimodal routing)

Roles:

1. **Image captioning / dense description** to create a text representation for the rest of the stack
2. **Visual feature embeddings** for image retrieval ("find similar prior images / diagrams")
3. **Multimodal validation triggers:** if output depends on image content, validator can require explicit grounding (e.g., "claim must be supported by detected objects/text in image")

Workflow integration:

- Router detects image modality → call vision encoder
- Vision output becomes structured intermediate state (objects, layout, captions, detected text overlay if OCR combined)
- Worker/Validator operate over that structured representation

---

5. Model Stack (As per document, with modality additions)

The document's stack (roles preserved):

- **Router:** Granite-Micro 3B (CPU)
- **Validator:** Granite-H-Small (CPU)
- **Workers (GPU one at a time):**
    - GPT-OSS 20B (reasoning/planning)
    - Qwen Coder 32B (architecture/refactoring)
    - Nemotron-3 Nano 30B (practical implementation/perf)
    - MythoMax-L2-13B (creative/narrative)

Additions you want (tooling models, typically CPU-bound):

- **Embedding model(s):** for retrieval and semantic memory
- **OCR engine:** for scanned docs/images
- **Vision encoder(s):** for image understanding and vision embeddings

These should be treated as **tools** feeding the router/worker/validator, not as "workers" themselves.

---

6. Governance, Validation, and Epistemic Hygiene (Operational Policy)

To align with the “sovereign-grade” claim in the document, define explicit policies the router must enforce:

6.1 Stakes-based validation policy

- **Low stakes:** single pass generation, optional light validation
- **Medium stakes:** mandatory validator review before commit
- **High stakes:** block-by-block proof-checking + at least one secondary specialist review (e.g., Nemotron practicality check after Qwen architecture)

6.2 Grounding policy (new, driven by OCR/vision)

- If OCR/vision was used:
  - outputs that cite image/doc content must be traceable to extracted text/features
  - validator can require “source-backed assertions only” mode
  - any uncertainty must be surfaced in the final answer (“OCR unclear on line X”)

6.3 Router uncertainty policy

If router confidence is low:

- run embedding similarity fallback
- default to safer path: more validation, more conservative model selection
- write an explicit “uncertainty note” into the ledger (auditable)

---

7. Implementation Blueprint (Practical Build Plan)

This aligns with the document’s phased recommendations but updates them for Prolog + OCR/vision/embeddings.

Phase 1 — Foundation (instrumentation-first)

- Stand up CPU router + CPU validator endpoints
- Create Markdown memory schema and commit rules
- Add structured logging for: routing decision, model chosen, validation result, and timing metrics

Phase 2 — Hybrid router core

- Implement routing rules as declarative predicates (Prolog)
- Python calls Prolog to obtain:
  - selected worker model

    - whether OCR/vision/embeddings are required
    - validation strictness level
    - retry policy
- Store Prolog "decision trace" in ledger for auditability

Phase 3 — Embeddings + retrieval

- Build ingestion pipeline that indexes:
    - project memory markdown
    - local docs
    - OCR outputs
- Use retrieval before generation and before validation (validator gets the same evidence)

Phase 4 — OCR + vision encoders

- Add OCR pipeline for scanned documents/images
- Add vision encoder pipeline for image understanding and image embeddings
- Update routing rules to detect multimodal inputs and enforce grounding constraints

Phase 5 — Full Validator Ladder (sequential specialist review)

- Enable multi-validator steps for high stakes:
    - Granite governance
    - Nemotron practicality checks (code)
    - optional "structure review" passes for docs/policy text

---

8. Risks, Metrics, and Acceptance Criteria (from the document, extended)

8.1 Key risks (document + new modality risks)

- Router misclassification → wrong model/tool path
- Validator false rejects → endless retries / latency blowups
- OCR errors → false grounding
- Vision misinterpretation → incorrect claims about images
- Complexity overhead → maintenance burden

8.2 Metrics to track (minimum viable governance)

- Router accuracy + router "uncertainty rate"
- Validation pass/fail rate; false positive/negative (spot-checked)
- OCR confidence and error rate (manual sampling)

- Vision pipeline latency and failure rate
- End-to-end latency breakdown (router / retrieval / generation / validation / OCR / vision)
- "Grounded claim ratio" for OCR/vision-based answers (how many claims are source-backed)

8.3 Acceptance criteria (practical)

- No GPU thrashing under proof-checking mode
- Typical tasks remain usable under your latency tolerance
- High-stakes mode produces auditable traces: decision → evidence → output
- OCR/vision pipelines produce explicit uncertainty notes rather than silent failure

---

9. Final Recommendation

Proceed with the **final bicameral architecture** from the uploaded document, but implement the router/governance logic as a **declarative logic layer (Prolog)** invoked by a Python orchestrator. Add **embeddings, OCR, and vision encoders** as first-class tool pipelines controlled by the same governance logic, with strong grounding and audit requirements baked into the validator policy.

If you want, paste (or describe) what you consider "final iteration" details that differ from what's written in the PDF (e.g., any updated model choices or policies), and I'll adapt this report to match your exact final spec.