System Architecture Document (SAD)

Sovereign AI Infrastructure: Bicameral Validator Ladder

Document Version: 1.0
Date: February 5, 2026
Status: Draft for Review
Owner: Solutions Architect / Technical Lead

Document Control

Version	Date	Author	Changes
1.0	2026-02-05	Architecture Team	Initial draft

Reviewers:
- [ ] Technical Lead
- [ ] Engineering Manager
- [ ] ML Lead
- [ ] DevOps Lead
- [ ] Security Architect

Dependencies:
- Product Requirements Document (PRD) v1.0
- Technical Analysis Report (Deep Agent Report)
- Comprehensive Architecture Report (Hybrid Logic Router)

Table of Contents

1. Executive Summary

1.1 Purpose

This System Architecture Document (SAD) provides a comprehensive technical blueprint for the **Sovereign AI Infrastructure**, a local multi-model AI orchestration system with built-in validation, transparent governance, and multimodal capabilities. This document translates the requirements from the PRD into detailed technical designs that implementation teams can directly execute.

1.2 Architectural Approach

The architecture is built on **three foundational principles**:

1. **Bicameral Heterogeneous Compute**: Separation of creative generation (GPU) from critical validation (CPU) to eliminate model thrashing
2. **Declarative Governance**: Routing and validation logic expressed as declarative Prolog rules, not imperative code
3. **Transparent Memory**: Markdown-based persistent memory serving as an auditable, human-readable ledger

1.3 Key Architectural Decisions

Decision	Rationale	Alternatives Considered
Hybrid Python/Prolog	Python for orchestration/integration, Prolog for logic/routing	Pure Python (less expressive for rules), Pure Prolog (poor ML ecosystem)
CPU-Resident Validator	Eliminates GPU model swapping for validation	GPU validator (causes thrashing), No validation (no quality governance)
Markdown Memory Bus	Human-readable, Git-trackable, transparent	SQL database (opaque), JSON (less readable), Neural memory (black box)
Warm Pool in RAM	Fast model swapping (<3s vs. >10s from disk)	Always load from disk (slow), Keep all in VRAM (impossible with 16GB)
Sequential Task Execution	Simpler, avoids concurrency complexity in v1.0	Concurrent execution (more complex, deferred to v2.0)

1.4 Architecture at a Glance

```
┌─────────────────────────────────────────────────────────────────┐
│                        USER INTERFACE                           │
│                      (CLI / REST API)                           │
└─────────────────────────────────────────────────────────────────┘
                                │
                                ▼
┌─────────────────────────────────────────────────────────────────┐
│                    ORCHESTRATION LAYER                          │
│                      (Python)                                   │
│                                                               │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐         │
│  │   Request    │  │   Memory     │  │  Telemetry   │         │
│  │   Handler    │  │   Manager    │  │   Logger     │         │
│  └──────────────┘  └──────────────┘  └──────────────┘         │
└─────────────────────────────────────────────────────────────────┘
                                │
        ┌───────────────────────┼───────────────────────┐
        │                       │                       │
        ▼                       ▼                       ▼
┌──────────────┐    ┌─────────────────────┐   ┌────────────────┐
│   ROUTER     │    │  VALIDATOR (CPU)    │   │  TOOLS (CPU)   │
│   (CPU)      │    │  Always Resident    │   │  - Embeddings  │
│  Granite-    │    │  Granite-H-Small    │   │  - OCR         │
│  Micro 3B    │    │  (MoE 9B active)    │   │  - Vision      │
└──────────────┘    └─────────────────────┘   └────────────────┘
        │                       │                       │
        └───────────────────────┼───────────────────────┘
                                │
                                ▼
                    ┌───────────────────────┐
                    │   MODEL LOADER        │
                    │   (Warm Pool Manager) │
                    └───────────────────────┘
                                │
        ┌───────────────────────┼───────────────────────┐
        │                       │                       │
        ▼                       ▼                       ▼
┌──────────────┐    ┌─────────────────────┐   ┌────────────────┐
│  WORKER 1    │    │  WORKER 2           │   │  WORKER 3      │
│  Qwen Coder  │    │  Nemotron 30B       │   │  GPT-OSS 20B   │
│  32B (GPU)   │    │  (GPU)              │   │  (GPU)         │
└──────────────┘    └─────────────────────┘   └────────────────┘
                                │
                                ▼
                    ┌───────────────────────┐
                    │  WORKER 4 (Optional)  │
                    │  MythoMax 13B (GPU)   │
                    └───────────────────────┘
                                │
        ╔═══════════════════════════════════════════════╗
        ║         MEMORY BUS (Markdown Files)           ║
        ║  project_state.md | scratchpad.md |          ║
        ║  knowledge_graph.md                           ║
        ╚═══════════════════════════════════════════════╝
```

---

2. Architectural Overview

2.1 Architectural Style

The system employs a **hybrid architectural style**:

- **Hub-and-Spoke** for model orchestration (Router as hub, Workers as spokes)

- **Pipes-and-Filters** for data flow (Request → Route → Generate → Validate → Commit)
- **Shared-Memory** for state management (Markdown files as shared ledger)
- **Layered** for separation of concerns (UI → Orchestration → Inference → Hardware)

2.2 Architectural Layers

```
┌─────────────────────────────────────────────────┐
│  Layer 1: Presentation Layer (CLI, REST API)    │
├─────────────────────────────────────────────────┤
│  Layer 2: Orchestration Layer (Python, Prolog)  │
│           - Request routing and coordination     │
│           - Memory management                    │
│           - Telemetry and logging                │
├─────────────────────────────────────────────────┤
│  Layer 3: Inference Layer (llama.cpp servers)   │
│           - Model serving (Router, Workers, Validator) │
│           - Tool execution (OCR, Vision, Embeddings)   │
├─────────────────────────────────────────────────┤
│  Layer 4: Resource Management Layer              │
│           - Model loading/unloading              │
│           - Warm pool management                 │
│           - GPU/CPU allocation                   │
├─────────────────────────────────────────────────┤
│  Layer 5: Storage Layer                          │
│           - Model vault (NVMe)                   │
│           - Memory ledger (Markdown files)       │
│           - Vector store (embeddings)            │
├─────────────────────────────────────────────────┤
│  Layer 6: Hardware Layer                         │
│           - GPU (Tesla A2, 16GB VRAM)            │
│           - CPU (Xeon 6-core, AVX-512)           │
│           - RAM (128GB ECC)                      │
│           - NVMe (1TB storage)                   │
└─────────────────────────────────────────────────┘
```

2.3 Core Architectural Patterns

Pattern 1: Bicameral Processing

- **Creative Hemisphere** (GPU): Fast, parallel, deep generation
- **Critical Hemisphere** (CPU): Sequential, logical, validation
- **Benefit**: No model thrashing; validation doesn't block generation

Pattern 2: Declarative Routing

- **Implementation**: Prolog predicates define routing logic
- **Benefit**: Rules are inspectable, modifiable without code changes
- **Example**: `route(coding, high_stakes) → qwen_coder + block_validation`

Pattern 3: Transparent Memory Bus

- **Implementation**: Markdown files as shared memory
- **Benefit**: Human-readable audit trail; Git version control
- **Example**: Every decision, output, correction logged in `scratchpad.md`

Pattern 4: Predictive Warm Pool

- **Implementation**: Pre-load likely-next models into RAM
- **Benefit**: 3-5x faster model swaps (3s vs. 10-15s)
- **Example**: If current task is coding, pre-load Qwen Coder

3. System Context & Boundaries

3.1 System Context Diagram (C4 Level 1)

```
graph TB
    User[Domain Expert<br/>Compliance Officer<br/>Technical Lead<br/>Researcher]

    SovAI[Sovereign AI<br/>Infrastructure<br/>Multi-model orchestration<br/>with validation]

    ModelRepo[Model Repository<br/>Hugging Face<br/>Model downloads]

    Git[Git Repository<br/>Version control for<br/>memory ledger]

    Monitor[Monitoring System<br/>Prometheus/Grafana<br/>Metrics & alerts]

    User -->|Tasks, Queries| SovAI
    SovAI -->|Validated Outputs<br/>Audit Trails| User

    SovAI -.->|One-time model download| ModelRepo
    SovAI -->|Commit memory state| Git
    SovAI -->|Export metrics| Monitor
    Monitor -->|View dashboards<br/>Receive alerts| User

    style SovAI fill:#4A90E2,stroke:#333,stroke-width:3px,color:#fff
    style User fill:#50E3C2,stroke:#333,stroke-width:2px
    style ModelRepo fill:#F5A623,stroke:#333,stroke-width:2px
    style Git fill:#7ED321,stroke:#333,stroke-width:2px
    style Monitor fill:#BD10E0,stroke:#333,stroke-width:2px
```

3.2 External Interfaces

Interface	Direction	Protocol	Purpose	Notes
CLI	Bidirectional	stdin/stdout	User interaction	Primary interface for v1.0
REST API	Bidirectional	HTTP/JSON	Programmatic access	Secondary interface
Model Repository	Outbound	HTTPS	Model downloads	One-time; offline operation afterward
Git	Outbound	Git protocol	Memory versioning	Optional; local Git sufficient
Monitoring	Outbound	Prometheus protocol	Metrics export	For observability

3.3 System Boundaries

In Scope (System Responsibilities):
- Request routing and model selection
- Model orchestration and lifecycle management
- Generation with validation
- Multimodal input processing (OCR, vision, embeddings)
- Memory ledger management
- Audit trail generation
- Performance monitoring

Out of Scope (External Responsibilities):
- Model training or fine-tuning
- User authentication (single-user system in v1.0)
- Distributed multi-node deployment
- Real-time streaming (<1s responses)
- GUI (deferred to v1.1)

4. Component Architecture

4.1 Container Diagram (C4 Level 2)

```
graph TB
    subgraph "Presentation Layer"
        CLI[CLI Interface<br/>Python Click]
        API[REST API<br/>FastAPI]
    end

    subgraph "Orchestration Layer"
        Orchestrator[Orchestrator<br/>Python<br/>Request coordination]
        RouterSvc[Router Service<br/>Prolog + Python<br/>Intent classification]
        MemMgr[Memory Manager<br/>Python<br/>Markdown I/O]
        Telemetry[Telemetry Service<br/>Python<br/>Prometheus client]
    end

    subgraph "Inference Services (CPU)"
        RouterModel[Router Model<br/>llama.cpp<br/>Granite-Micro 3B]
        ValidatorModel[Validator Model<br/>llama.cpp<br/>Granite-H-Small]
        ToolSvc[Tool Services<br/>Python<br/>OCR/Vision/Embeddings]
    end

    subgraph "Inference Services (GPU)"
        WorkerLoader[Model Loader<br/>Python<br/>Warm pool mgmt]
        Worker1[Worker 1<br/>llama.cpp<br/>Qwen Coder 32B]
        Worker2[Worker 2<br/>llama.cpp<br/>Nemotron 30B]
        Worker3[Worker 3<br/>llama.cpp<br/>GPT-OSS 20B]
        Worker4[Worker 4<br/>llama.cpp<br/>MythoMax 13B]
    end

    subgraph "Storage Layer"
        ModelVault[Model Vault<br/>NVMe SSD<br/>GGUF files]
        MemoryLedger[Memory Ledger<br/>Filesystem<br/>Markdown files]
        VectorStore[Vector Store<br/>FAISS<br/>Embeddings DB]
    end

    CLI --> Orchestrator
    API --> Orchestrator

    Orchestrator --> RouterSvc
    Orchestrator --> MemMgr
    Orchestrator --> Telemetry
    Orchestrator --> ValidatorModel
    Orchestrator --> WorkerLoader

    RouterSvc --> RouterModel
    RouterSvc --> ToolSvc

    WorkerLoader --> Worker1
    WorkerLoader --> Worker2
    WorkerLoader --> Worker3
    WorkerLoader --> Worker4

    ToolSvc --> VectorStore

    MemMgr --> MemoryLedger
    WorkerLoader --> ModelVault

    style Orchestrator fill:#4A90E2,stroke:#333,stroke-width:3px,color:#fff
    style RouterSvc fill:#50E3C2,stroke:#333,stroke-width:2px
    style WorkerLoader fill:#F5A623,stroke:#333,stroke-width:2px
    style MemMgr fill:#7ED321,stroke:#333,stroke-width:2px
```

4.2 Component Specifications

4.2.1 Orchestrator Component

Responsibility: Central coordinator for all request processing

Interfaces:
- **Input**: User request (text, document, image) via CLI/API
- **Output**: Final validated output + audit trail

Key Operations:
1. `process_request(user_input) → output`

- Calls Router to classify request
- Invokes appropriate pipelines (OCR, vision, embeddings if needed)
- Loads appropriate Worker model
- Executes generation (with or without validation based on stakes)
- Commits results to memory ledger
- Returns output to user

1. `execute_validation_loop(worker_output, context) → validated_output`
   - Implements Generate → Validate → Commit cycle
   - Handles retry logic (max 3 attempts per block)
   - Logs all validation decisions

Technology: Python 3.10+
Key Dependencies: `RouterService`, `MemoryManager`, `ModelLoader`, `ValidatorModel`
Configuration: `config/orchestrator.yaml`

4.2.2 Router Service Component

Responsibility: Intent classification and routing decision-making

Interfaces:
- **Input**: User request (text), current context
- **Output**: JSON routing decision

```
json
  {
    "domain": "coding_architecture | coding_implementation | reasoning | creative | documentation",
    "stakes": "low | medium | high",
    "recommended_model": "qwen_coder | nemotron | gpt_oss | mythomax",
    "validation_policy": "none | end_stage | block_by_block",
    "tools_required": ["ocr", "vision", "embeddings"],
    "confidence": 0.92,
    "reasoning": "High complexity coding task with architectural focus; keywords: refactor, multi-file, design patterns"
  }
```

Key Operations:
1. `classify_request(user_input, history) → routing_decision`

- Invokes Router Model (Granite-Micro 3B) for initial classification
- Queries Prolog knowledge base for routing rules
- Applies confidence thresholding

- Falls back to embedding similarity for ambiguous cases
- Logs decision with reasoning

1. `predict_next_model(current_model, task_history) → model_to_preload`
   - Predictive logic for warm pool management
   - Based on: current domain, user patterns, task context

Technology: Python (orchestration) + SWI-Prolog (rules)
Key Dependencies: `RouterModel` (llama.cpp), `Prolog KB`, `EmbeddingService` (fallback)
Configuration: `config/routing_rules.pl`, `config/router.yaml`

4.2.3 Memory Manager Component

Responsibility: All interactions with the Markdown memory ledger

Interfaces:
- **Input**: Read/write requests for memory files
- **Output**: Memory content, write confirmations

Key Operations:
1. `read_project_state() → dict`
- Parses `project_state.md`
- Returns: objectives, proven facts, constraints

1. `append_to_scratchpad(entry) → success`
   - Atomic append to `scratchpad.md`
   - Includes: timestamp, actor (Router/Worker/Validator), content

2. `commit_to_project_state(fact) → success`
   - Moves validated fact from scratchpad to project_state
   - Immutable once committed

3. `update_knowledge_graph(learning) → success`
   - Adds learned pattern to `knowledge_graph.md`
   - Examples: “Nemotron hallucinates on Rust code”, “Qwen requires explicit type hints”

4. `get_relevant_context(query, top_k=5) → list[str]`
   - Semantic search over memory using embeddings
   - Returns top-k relevant snippets for Worker context

Technology: Python
Key Dependencies: Filesystem, `EmbeddingService` (for retrieval)
Configuration: `config/memory.yaml` (file paths, backup settings)
Concurrency: File locking (fcntl) to prevent race conditions

4.2.4 Model Loader (Warm Pool Manager)

Responsibility: GPU model lifecycle and warm pool optimization

Interfaces:
- **Input**: Model load/unload requests
- **Output**: Confirmation, VRAM utilization metrics

Key Operations:
1. `load_model_to_gpu(model_name, quantization) → success`
- Checks VRAM availability
- If VRAM full: unload current model
- Load from warm pool (RAM) or cold start (NVMe)
- Register model with inference server
- Update telemetry

1. `preload_to_warm_pool(model_name) → success`
   - Load model from NVMe → RAM
   - Keep in RAM (not VRAM) for fast future access
   - Apply LRU eviction if RAM pressure

2. `get_current_gpu_model() → model_name`
   - Returns which model is currently in VRAM

3. `predict_and_preload_next() → success`
   - Predictive logic: based on Router hints, preload likely-next model

Technology: Python
Key Dependencies: llama.cpp API, GPU monitoring (pynvml)
Configuration: `config/model_loader.yaml` (warm pool size, eviction policy)
Critical Path: Model swap time <5 seconds (target <3s from RAM)

4.2.5 Router Model Service

Responsibility: Run Granite-Micro 3B for intent classification

Deployment: CPU-resident (always loaded)
Technology: llama.cpp server mode (CPU inference)
Interface: HTTP REST (OpenAI-compatible API)
Endpoint: `http://localhost:8001/v1/completions`

System Prompt:

```
You are a routing classifier for a multi-model AI system.
Analyze the user request and output ONLY a JSON object with:
{
  "domain": "coding_architecture|coding_implementation|reasoning|creative|documentation",
  "complexity": "low|medium|high",
  "keywords": ["list", "of", "key", "terms"],
  "confidence": 0.0-1.0
}
Output NOTHING else. No explanations.
```

Performance Requirements:
- Inference speed: ≥10 tokens/second
- Latency: ≤1 second per classification
- RAM footprint: ≤6GB

4.2.6 Validator Model Service

Responsibility: Run Granite-H-Small for validation

Deployment: CPU-resident (always loaded)
Technology: llama.cpp server mode (CPU inference)
Interface: HTTP REST (OpenAI-compatible API)
Endpoint: `http://localhost:8002/v1/completions`

System Prompt (varies by validation type):

```
You are a strict validator for AI-generated {content_type}.
You will receive:
1. Current project state and constraints
2. A block of {content_type} to validate
3. Validation criteria

Output format:
[PASS] - if all checks pass (optional brief summary)
[FAIL: specific reason] - if any check fails (one-sentence correction directive)

Checks:
- Logical consistency with project state
- No hallucinations (fabricated facts/APIs)
- {content_specific_checks}
- Syntax correctness (if code)
- Safety and policy compliance
```

Performance Requirements:
- Inference speed: ≥3 tokens/second (acceptable for validation)
- Latency: ≤5 seconds per block validation
- RAM footprint: ≤20GB

4.2.7 Worker Model Services (GPU)

Responsibility: Run specialist Worker models for generation

Deployment: GPU-resident (one at a time)
Technology: llama.cpp server mode (GPU inference, CUDA)
Interface: HTTP REST (OpenAI-compatible API)
Endpoint: `http://localhost:8000/v1/completions`

Worker Specifications:

Worker	Model	Role	VRAM	Speed Target	Context
Worker 1	Qwen Coder 32B (Q4_K_M)	Coding (Architecture)	18GB	≥20 tok/s	4K
Worker 2	Nemotron 30B (Q4_K_M)	Coding (Implementation)	16GB	≥25 tok/s	4K
Worker 3	GPT-OSS 20B (Q4_K_M)	Reasoning, Planning	12GB	≥30 tok/s	4K
Worker 4	MythoMax 13B (Q5_K_M)	Creative Writing	9GB	≥30 tok/s	4K

System Prompts: See Document 1.4 (Model Serving & Orchestration Design) for detailed prompts per Worker.

4.2.8 Tool Services (CPU)

Responsibility: Execute multimodal tools (OCR, Vision, Embeddings)

Sub-Components:

1. **OCR Service**
   - **Technology**: Tesseract 5.0+ or PaddleOCR
   - **Interface**: Python function: `extract_text(image_path) → {text, bounding_boxes, confidence}`
   - **Performance**: ≤5 seconds per page

2. **Vision Encoder Service**
   - **Technology**: CLIP or BLIP (Hugging Face Transformers)
   - **Interface**: Python function: `encode_image(image_path) → {caption, objects, features}`
   - **Performance**: ≤3 seconds per image

3. **Embedding Service**
   - **Technology**: Sentence-Transformers (e.g., `all-MiniLM-L6-v2`)
   - **Interface**: Python function: `embed_text(text) → vector[384]`
   - **Performance**: ≤1 second for embedding + retrieval

Deployment: CPU (can run concurrently with Router/Validator)
Configuration: `config/tools.yaml`

4.3 Component Interaction Sequence

Sequence Diagram: High-Stakes Validated Code Generation

```
sequenceDiagram
    actor User
    participant CLI
    participant Orchestrator
    participant Router
    participant MemMgr as Memory Manager
    participant Loader as Model Loader
    participant Worker as Worker (GPU)
    participant Validator as Validator (CPU)

    User->>CLI: Submit coding task
    CLI->>Orchestrator: process_request(task)

    Orchestrator->>Router: classify_request(task)
    Router->>Router: Query Prolog rules
    Router-->>Orchestrator: {domain: coding_arch, stakes: high, model: qwen_coder}

    Orchestrator->>MemMgr: read_project_state()
    MemMgr-->>Orchestrator: current_state

    Orchestrator->>Loader: load_model_to_gpu("qwen_coder")
    Loader->>Loader: Check warm pool
    Loader->>Worker: Load Qwen Coder 32B
    Loader-->>Orchestrator: Model ready

    loop Block-by-block validation
        Orchestrator->>Worker: Generate next block (5-10 lines)
        Worker-->>Orchestrator: Generated block

        Orchestrator->>MemMgr: append_to_scratchpad(block)

        Orchestrator->>Validator: validate_block(block, project_state)
        Validator->>Validator: Check logic, syntax, hallucinations

        alt Block passes
            Validator-->>Orchestrator: [PASS]
            Orchestrator->>MemMgr: commit_to_project_state(block)
        else Block fails
            Validator-->>Orchestrator: [FAIL: reason]
            Orchestrator->>MemMgr: append_correction(reason)
            Note over Orchestrator,Worker: Retry (max 3 attempts)
        end
    end

    Orchestrator->>MemMgr: finalize_output()
    Orchestrator->>CLI: validated_output + audit_trail
    CLI->>User: Display result
```

5. Data Flow Architecture

5.1 High-Level Data Flow

```
┌─────────────┐
│ User Input  │ (text, document, image)
└──────┬──────┘
       │
       ▼
┌──────────────────────────────────────────┐
│  1. INTAKE & ROUTING                     │
│  - Router classifies intent              │
│  - Prolog rules determine model          │
│  - Assess stakes, tools needed           │
└──────┬───────────────────────────────────┘
       │
       ▼
┌──────────────────────────────────────────┐
│  2. CONTEXT PREPARATION                  │
│  - Retrieve from memory ledger           │
│  - Run tools (OCR/vision if needed)      │
│  - Assemble context for Worker           │
└──────┬───────────────────────────────────┘
       │
       ▼
┌──────────────────────────────────────────┐
│  3. GENERATION (GPU Worker)              │
│  - Load appropriate specialist model     │
│  - Generate in blocks (if high-stakes)   │
│  - Write to scratchpad                   │
└──────┬───────────────────────────────────┘
       │
       ▼
┌──────────────────────────────────────────┐
│  4. VALIDATION (CPU Validator)           │
│  - Check block against constraints       │
│  - Verify grounding (if multimodal)      │
│  - Output PASS or FAIL + reasoning       │
└──────┬───────────────────────────────────┘
       │
       ├─[PASS]──────────────────────┐
       │                             │
       │                             ▼
       │                   ┌──────────────────────┐
       │                   │  5. COMMIT           │
       │                   │  Move to project_    │
       │                   │  state (immutable)   │
       │                   └─────────┬────────────┘
       │                             │
       │◄────────────────────────────┘
       │
       ├─[FAIL]───────────────────┐
       │                          │
       │                          ▼
       │              ┌─────────────────────────┐
       │              │  6. RETRY               │
       │              │  Write correction to    │
       │              │  scratchpad, retry      │
       │              │  generation (max 3x)    │
       │              └───────────┬─────────────┘
       │                          │
       │◄─────────────────────────┘
       │
       ▼
┌──────────────────────────────────────────┐
│  7. OUTPUT & AUDIT                       │
│  - Final validated output to user       │
│  - Complete audit trail logged          │
│  - Update knowledge graph (learnings)   │
└─────────────────────────────────────────┘
```

5.2 Data Structures

5.2.1 Request Object

```python
@dataclass
class Request:
    """User request with metadata"""
    id: str                      # UUID
    timestamp: datetime          # UTC
    user_id: str                 # User identifier (future multi-user)
    content: str                 # Text input
    attachments: List[Attachment]  # Documents, images
    metadata: Dict[str, Any]     # Additional context

@dataclass
class Attachment:
    """Attached file (document, image)"""
    filename: str
    filepath: str
    mime_type: str
    size_bytes: int
```

5.2.2 Routing Decision Object

```python
@dataclass
class RoutingDecision:
    """Router output"""
    domain: DomainType           # Enum: CODING_ARCH, CODING_IMPL, REASONING, CREATIVE, DOC
    stakes: StakesLevel          # Enum: LOW, MEDIUM, HIGH
    recommended_model: ModelType # Enum: QWEN, NEMOTRON, GPT_OSS, MYTHOMAX
    validation_policy: ValidationPolicy  # Enum: NONE, END_STAGE, BLOCK_BY_BLOCK
    tools_required: List[ToolType]  # Enum: OCR, VISION, EMBEDDINGS
    confidence: float            # 0.0-1.0
    reasoning: str               # Human-readable explanation
    timestamp: datetime
```

5.2.3 Validation Result Object

```python
@dataclass
class ValidationResult:
    """Validator output"""
    verdict: Verdict             # Enum: PASS, FAIL
    reasoning: str               # Explanation
    corrections: Optional[str]   # If FAIL, what to fix
    confidence: float            # Validator confidence
    checks_performed: List[str]  # E.g., ["logic", "syntax", "grounding"]
    timestamp: datetime
```

5.2.4 Memory Entry Object

```python
@dataclass
class MemoryEntry:
    """Entry in Markdown memory ledger"""
    timestamp: datetime
    actor: ActorType              # Enum: ROUTER, WORKER, VALIDATOR, USER
    entry_type: EntryType         # Enum: FACT, REASONING, CORRECTION, OUTPUT
    content: str                  # Markdown-formatted content
    metadata: Dict[str, Any]      # Additional context (model version, etc.)
```

---

6. Deployment Architecture

6.1 Physical Deployment

```
┌────────────────────────────────────────────────────────────┐
│                     SINGLE WORKSTATION                     │
│  ┌──────────────────────────────────────────────────────┐  │
│  │                  Operating System                    │  │
│  │              Ubuntu 22.04 LTS (Linux)                │  │
│  └──────────────────────────────────────────────────────┘  │
│                                                            │
│  ┌──────────────────────┐  ┌────────────────────────────┐  │
│  │   GPU (VRAM 16GB)    │  │  CPU (6 cores + AVX-512)   │  │
│  │ ┌──────────────────┐ │  │ ┌────────────────────────┐ │  │
│  │ │  Worker Model    │ │  │ │ Router (Granite-Micro) │ │  │
│  │ │  (One at time)   │ │  │ │ Always loaded          │ │  │
│  │ │                  │ │  │ └────────────────────────┘ │  │
│  │ │  - Qwen 32B      │ │  │ ┌────────────────────────┐ │  │
│  │ │  - Nemotron 30B  │ │  │ │ Validator (Granite-H)  │ │  │
│  │ │  - GPT-OSS 20B   │ │  │ │ Always loaded          │ │  │
│  │ │  - MythoMax 13B  │ │  │ └────────────────────────┘ │  │
│  │ └──────────────────┘ │  │ ┌────────────────────────┐ │  │
│  └──────────────────────┘  │ │ Tools (OCR/Vision/Emb) │ │  │
│                            │ └────────────────────────┘ │  │
│  ┌──────────────────────────────────────────────────────┐  │
│  │                System RAM (128GB ECC)                │  │
│  │  ┌──────────────┐ ┌──────────────┐ ┌──────────────┐  │  │
│  │  │ Warm Pool    │ │ Router       │ │ Validator    │  │  │
│  │  │ (60-80GB)    │ │ (6GB)        │ │ (20GB)       │  │  │
│  │  │ 2-3 models   │ │              │ │              │  │  │
│  │  │ pre-loaded   │ │              │ │              │  │  │
│  │  └──────────────┘ └──────────────┘ └──────────────┘  │  │
│  └──────────────────────────────────────────────────────┘  │
│                                                            │
│  ┌──────────────────────────────────────────────────────┐  │
│  │                    NVMe SSD (1TB)                    │  │
│  │  ┌──────────────┐ ┌──────────────┐ ┌──────────────┐  │  │
│  │  │ Model Vault  │ │ Memory       │ │ Vector DB    │  │  │
│  │  │ All models   │ │ Ledger       │ │ Embeddings   │  │  │
│  │  │ (500GB)      │ │ (Markdown)   │ │ (FAISS)      │  │  │
│  │  └──────────────┘ └──────────────┘ └──────────────┘  │  │
│  └──────────────────────────────────────────────────────┘  │
└────────────────────────────────────────────────────────────┘
```

6.2 Software Deployment

```
┌─────────────────────────────────────────────────────────┐
│  Application Layer                                      │
├─────────────────────────────────────────────────────────┤
│  sovereign-ai/                                          │
│  ├── src/                                               │
│  │   ├── orchestrator/        (Python)                  │
│  │   ├── router/              (Python + Prolog)         │
│  │   ├── memory/              (Python)                  │
│  │   ├── models/              (Python wrappers)         │
│  │   ├── tools/               (Python)                  │
│  │   └── cli/                 (Python Click)            │
│  ├── config/                                            │
│  │   ├── models.yaml                                    │
│  │   ├── routing_rules.pl                               │
│  │   ├── validation_policies.yaml                       │
│  │   └── system.yaml                                    │
│  ├── data/                                              │
│  │   ├── memory/              (Markdown files)          │
│  │   └── vector_store/        (FAISS index)             │
│  └── logs/                    (Application logs)        │
├─────────────────────────────────────────────────────────┤
│  Inference Layer                                        │
├─────────────────────────────────────────────────────────┤
│  /opt/llama.cpp/                                        │
│  ├── server_gpu     (Port 8000, GPU Worker)             │
│  ├── server_router  (Port 8001, CPU Router)             │
│  └── server_validator (Port 8002, CPU Validator)        │
├─────────────────────────────────────────────────────────┤
│  Model Storage                                          │
├─────────────────────────────────────────────────────────┤
│  /mnt/models/                                           │
│  ├── router/                                            │
│  │   └── granite-micro-3b-q8.gguf                       │
│  ├── validator/                                         │
│  │   └── granite-h-small-q4_k_m.gguf                    │
│  ├── workers/                                           │
│  │   ├── qwen-coder-32b-q4_k_m.gguf                     │
│  │   ├── nemotron-30b-q4_k_m.gguf                       │
│  │   ├── gpt-oss-20b-q4_k_m.gguf                        │
│  │   └── mythomax-13b-q5_k_m.gguf                       │
│  └── tools/                                             │
│      ├── ocr_model/                                     │
│      ├── vision_model/                                  │
│      └── embedding_model/                               │
└─────────────────────────────────────────────────────────┘
```

6.3 Process Architecture

Runtime Processes:

Process	Type	CPU/GPU	Memory	Port	Auto-Restart
`sovereign-orchestrator`	Python daemon	CPU	2GB	N/A	Yes
`llama-server-router`	llama.cpp	CPU	6GB	8001	Yes
`llama-server-validator`	llama.cpp	CPU	20GB	8002	Yes
`llama-server-worker`	llama.cpp	GPU+CPU	16-18GB	8000	Yes
`prometheus-exporter`	Python	CPU	512MB	9090	Yes

Process Management: systemd (Linux) or supervisord

Example systemd unit (`/etc/systemd/system/sovereign-orchestrator.service`):

```
[Unit]
Description=Sovereign AI Orchestrator
After=network.target

[Service]
Type=simple
User=sovereign
WorkingDirectory=/opt/sovereign-ai
ExecStart=/opt/sovereign-ai/venv/bin/python -m src.orchestrator.main
Restart=always
RestartSec=10
Environment="CUDA_VISIBLE_DEVICES=0"
Environment="OMP_NUM_THREADS=6"

[Install]
WantedBy=multi-user.target
```

---

7. Technology Stack

7.1 Core Technologies

Layer	Technology	Version	Purpose	License
Programming	Python	3.10+	Orchestration, ML tooling	PSF
Logic Programming	SWI-Prolog	8.4+	Routing rules	BSD-2
Inference Engine	llama.cpp	Latest	Model serving (GPU+CPU)	MIT
Web Framework	FastAPI	0.100+	REST API	MIT
CLI Framework	Click	8.0+	Command-line interface	BSD-3
Vector Store	FAISS	1.7+	Embedding search	MIT
Monitoring	Prometheus	2.40+	Metrics collection	Apache 2.0
Dashboards	Grafana	9.0+	Visualization	AGPLv3
Version Control	Git	2.30+	Memory ledger versioning	GPL-2.0

7.2 Python Dependencies

Core Libraries (`requirements.txt`):

```
# Orchestration & Web
fastapi>=0.100.0
uvicorn>=0.23.0
click>=8.1.0
pydantic>=2.0.0
python-multipart>=0.0.6

# LLM Integration
llama-cpp-python>=0.2.0  # With CUDA support
openai>=1.0.0  # For OpenAI-compatible API client

# Prolog Integration
pyswip>=0.2.10  # SWI-Prolog bindings

# ML & Embeddings
sentence-transformers>=2.2.0
transformers>=4.30.0
torch>=2.0.0
faiss-cpu>=1.7.4  # or faiss-gpu if available

# OCR & Vision
pytesseract>=0.3.10
Pillow>=10.0.0
opencv-python>=4.8.0

# Monitoring & Logging
prometheus-client>=0.17.0
structlog>=23.1.0

# Utilities
pyyaml>=6.0
python-dotenv>=1.0.0
psutil>=5.9.0
pynvml>=11.5.0  # GPU monitoring
filelock>=3.12.0  # File locking for memory ledger
```

7.3 System Dependencies

Ubuntu Packages (apt):

```
# NVIDIA Drivers & CUDA
nvidia-driver-535
cuda-toolkit-12-1
libcudnn8

# Build Tools
build-essential
cmake
git

# OCR
tesseract-ocr
tesseract-ocr-eng  # English language data

# Prolog
swi-prolog

# System Utilities
htop
nvtop
tmux
```

7.4 Model Formats & Quantization

Model Format: GGUF (GPT-Generated Unified Format)
Quantization Library: llama.cpp built-in quantization

Quantization Strategies:

Model Size	Target VRAM	Quantization	Perplexity Impact
32B (Qwen)	18GB	Q4_K_M	~5% increase
30B (Nemotron)	16GB	Q4_K_M	~5% increase
20B (GPT-OSS)	12GB	Q4_K_M	~4% (MoE advantage)
13B (MythoMax)	9GB	Q5_K_M	~2% (higher quality for creative)
32B (Granite-H)	20GB (CPU)	Q4_K_M	~5% increase
3B (Granite-Micro)	6GB (CPU)	Q8_0 or FP16	Minimal

Quantization Command (example):

```
./llama.cpp/quantize \
  /mnt/models/raw/qwen-coder-32b-f16.gguf \
  /mnt/models/workers/qwen-coder-32b-q4_k_m.gguf \
  Q4_K_M
```

8. Interface Specifications

8.1 REST API Specification

Base URL: `http://localhost:5000/api/v1`

8.1.1 Submit Task

Endpoint: `POST /infer`

Request:

```
{
  "task": "Generate a Python function to calculate Fibonacci numbers recursively",
  "attachments": [],
  "options": {
    "stakes_override": "high",  // Optional: force validation policy
    "model_override": "qwen_coder",  // Optional: force model selection
    "temperature": 0.7,
    "max_tokens": 2048
  }
}
```

Response (200 OK):

```
{
  "request_id": "req_1234567890",
  "status": "completed",
  "output": "def fibonacci(n):\n    if n <= 1:\n        return n\n    return
fibonacci(n-1) + fibonacci(n-2)",
  "routing_decision": {
    "domain": "coding_implementation",
    "stakes": "medium",
    "model_used": "nemotron",
    "confidence": 0.89
  },
  "validation_summary": {
    "policy": "end_stage",
    "verdict": "PASS",
    "checks_performed": ["syntax", "logic", "hallucination"]
  },
  "audit_trail_url": "/api/v1/audit/req_1234567890",
  "timestamp": "2026-02-05T10:30:00Z",
  "latency_ms": 15234
}
```

Error Response (400 Bad Request):

```
{
  "error": "ValidationError",
  "message": "Task field is required",
  "timestamp": "2026-02-05T10:30:00Z"
}
```

8.1.2 Get System Status

Endpoint: `GET /status`

Response (200 OK):

```json
{
  "system": "operational",
  "components": {
    "orchestrator": "running",
    "router": "running",
    "validator": "running",
    "worker_gpu": "running",
    "current_gpu_model": "qwen_coder_32b",
    "warm_pool": ["nemotron_30b", "gpt_oss_20b"]
  },
  "resources": {
    "vram_used_gb": 17.2,
    "vram_total_gb": 16.0,
    "ram_used_gb": 95.3,
    "ram_total_gb": 128.0,
    "disk_free_gb": 487.5
  },
  "metrics": {
    "tasks_completed_today": 42,
    "router_accuracy_7d": 0.91,
    "validator_fp_rate_7d": 0.04,
    "avg_latency_p95_7d_sec": 28.3
  },
  "timestamp": "2026-02-05T10:35:00Z"
}
```

8.1.3 Get Audit Trail

Endpoint: `GET /audit/{request_id}`

Response (200 OK):

```json
{
  "request_id": "req_1234567890",
  "timeline": [
    {
      "timestamp": "2026-02-05T10:30:00.123Z",
      "actor": "router",
      "action": "classify_request",
      "details": "Domain: coding_implementation, Confidence: 0.89"
    },
    {
      "timestamp": "2026-02-05T10:30:01.456Z",
      "actor": "orchestrator",
      "action": "load_model",
      "details": "Loading Nemotron 30B from warm pool"
    },
    {
      "timestamp": "2026-02-05T10:30:03.789Z",
      "actor": "worker",
      "action": "generate_output",
      "details": "Generated 45 tokens in 2.1 seconds"
    },
    {
      "timestamp": "2026-02-05T10:30:06.012Z",
      "actor": "validator",
      "action": "validate_output",
      "details": "[PASS] All checks passed: syntax ✓, logic ✓, no hallucinations ✓"
    },
    {
      "timestamp": "2026-02-05T10:30:07.234Z",
      "actor": "orchestrator",
      "action": "commit_output",
      "details": "Committed to project_state.md"
    }
  ],
  "memory_snapshots": {
    "project_state": "# Project State\n## Task: Fibonacci function\n## Status: Completed\n...",
    "scratchpad": "# Scratchpad\n## Generated Code:\n```python\ndef fibonacci(n):..."
  }
}
```

8.2 CLI Interface Specification

Command Structure: `sovereign <command> [options] [arguments]`

8.2.1 Main Commands

Command	Description	Example
`infer`	Submit a task	`sovereign infer "Write a function to sort a list"`
`status`	Check system health	`sovereign status`
`history`	View past tasks	`sovereign history --last 10`
`audit`	View audit trail	`sovereign audit req_1234567890`
`config`	View/edit configuration	`sovereign config show`
`logs`	View system logs	`sovereign logs --tail 100`

8.2.2 Interactive Mode

```
$ sovereign interactive
Sovereign AI > [Interactive prompt]
> analyze this code: <paste code>
[System processes request]
> show audit trail
[Display audit trail]
> exit
```

8.3 Internal Component Interfaces

8.3.1 Router ↔ Orchestrator

Interface: Function call (Python)

```python
class RouterService:
    def classify_request(
        self,
        user_input: str,
        context: Optional[Dict] = None
    ) -> RoutingDecision:
        """
        Classify user request and determine routing.

        Args:
            user_input: User's task description
            context: Optional historical context

        Returns:
            RoutingDecision with model, stakes, tools, confidence
        """
        pass
```

8.3.2 Orchestrator ↔ Memory Manager

Interface: Function calls (Python)

```python
class MemoryManager:
    def read_project_state(self) -> Dict[str, Any]:
        """Read current project state"""
        pass

    def append_to_scratchpad(self, entry: MemoryEntry) -> bool:
        """Atomic append to scratchpad"""
        pass

    def commit_to_project_state(self, fact: str) -> bool:
        """Move validated fact to immutable project state"""
        pass

    def get_relevant_context(self, query: str, top_k: int = 5) -> List[str]:
        """Semantic retrieval from memory"""
        pass
```

8.3.3 Model Loader ↔ llama.cpp Servers

Interface: HTTP REST (OpenAI-compatible)

Example Request (Generation):

````
POST http://localhost:8000/v1/completions
Content-Type: application/json

{
  "model": "qwen-coder-32b-q4_k_m",
  "prompt": "def fibonacci(n):",
  "temperature": 0.7,
  "max_tokens": 512,
  "stop": ["\n\n", "```"]
}
````

Example Response:

```
{
  "id": "cmpl-1234567890",
  "object": "text_completion",
  "created": 1707134400,
  "model": "qwen-coder-32b-q4_k_m",
  "choices": [
    {
      "text": "\n    if n <= 1:\n        return n\n    return fibonacci(n-1) + fibonacci(n-2)",
      "index": 0,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 5,
    "completion_tokens": 28,
    "total_tokens": 33
  }
}
```

9. Scalability & Performance

9.1 Performance Targets (from PRD)

Metric	Target	Measurement
Router latency	≤1 second	Time from request to routing decision
Model swap time	≤3 seconds	RAM → VRAM transfer
Worker inference	≥20 tok/s	GPU models (varies by size)
Validator inference	≥3 tok/s	CPU model (acceptable for validation)
End-to-end (simple)	≤20 seconds	Low-stakes, no validation
End-to-end (complex)	≤60 seconds	High-stakes with block validation

9.2 Scalability Constraints (v1.0)

Hard Constraints:
- **Single GPU**: One Worker model at a time in VRAM
- **Sequential Execution**: One task at a time (no concurrent tasks)
- **Single User**: No multi-user concurrency support

Bottleneck Analysis:

Bottleneck	Impact	Mitigation (v1.0)	Future (v2.0+)
GPU model swaps	Adds 3-5s per swap	Warm pool reduces swaps	Multi-GPU: parallel Workers
CPU validation speed	3-5s per block	Optimize prompts, acceptable latency	GPU-resident validator option
Memory I/O	Minimal (<100ms)	In-memory buffering	Distributed memory store
Single-threaded orchestrator	Sequential tasks only	Acceptable for v1.0	Async orchestration, task queue

9.3 Resource Utilization Targets

VRAM (16GB total):
- Worker model: 12-18GB (depending on model)
- KV cache: 1-2GB
- Overhead: 1-2GB
- **Target**: 95% utilization when Worker loaded

RAM (128GB total):
- Router: 6GB
- Validator: 20GB
- Warm pool: 60-80GB (2-3 models)
- OS + overhead: 10-15GB
- Orchestrator + tools: 5-8GB
- **Target**: 90% utilization (avoid swap to disk)

CPU:
- Router: 1-2 cores
- Validator: 2-3 cores
- Tools (OCR/Vision/Embeddings): 2-3 cores (concurrent)
- Orchestrator + system: 1-2 cores
- **Target**: 70-80% utilization under load

NVMe:
- Model vault: 500GB (multiple quantizations per model)
- Memory ledger: <1GB (grows over time)
- Vector store: <5GB
- Logs: <10GB
- **Target**: >400GB free

---

10. Security Architecture

10.1 Security Principles

1. **Data Sovereignty**: All computation local; zero external dependencies
2. **Least Privilege**: Components have minimal necessary permissions

3. **Audit Everything**: All decisions, actions, errors logged immutably
4. **Fail Secure**: Failures default to safe state (reject, not accept)
5. **Defense in Depth**: Multiple validation layers

10.2 Threat Model

Assets to Protect:
- User data (prompts, documents, images)
- AI outputs (potentially sensitive)
- Memory ledger (audit trail)
- Model files (integrity)

Threats (v1.0 single-user):

Threat	Likelihood	Impact	Mitigation
Prompt injection	High	Medium	Validator checks outputs; system prompts hardened
Model tampering	Low	High	Checksums on model files; verify before load
Memory corruption	Low	High	Atomic writes; file locking; Git versioning
Resource exhaustion	Medium	Medium	Resource monitoring; OOM detection; rate limiting (future)
Log tampering	Low	High	Append-only logs; checksums; immutable once written

Out of Scope for v1.0 (single-user on localhost):
- Network-based attacks (no external network exposure)
- Multi-user privilege escalation (single user)
- Side-channel attacks (not a concern for local deployment)

10.3 Security Controls

10.3.1 Input Validation

- **User prompts**: Sanitize, length limits (max 10K characters)
- **File uploads**: MIME type checking, size limits (max 50MB per file)
- **API requests**: JSON schema validation

10.3.2 Output Validation

- **Validator role**: Check for harmful content, PII leakage, policy violations
- **Grounding checks**: Ensure multimodal outputs cite sources correctly

- **Confidence thresholds**: Flag low-confidence outputs for review

10.3.3 Audit & Logging

- **Immutable logs**: Append-only, no deletion
- **Tamper detection**: Checksums or HMAC on log entries
- **Comprehensive coverage**: Log all routing, generation, validation decisions

10.3.4 Resource Protection

- **VRAM monitoring**: Prevent OOM crashes
- **File locking**: Prevent concurrent writes to memory ledger
- **Backup strategy**: Daily backups of memory ledger, weekly model backups

10.4 Security Configuration

File Permissions:

```
/opt/sovereign-ai/
├── src/              (755, root:sovereign)
├── config/           (750, root:sovereign)
│   ├── *.yaml        (640, sensitive configs)
│   └── *.pl          (640, routing rules)
├── data/
│   ├── memory/       (770, sovereign:sovereign, group-writable for Git)
│   └── vector_store/ (770, sovereign:sovereign)
└── logs/             (750, sovereign:sovereign, append-only)

/mnt/models/          (755, root:sovereign, read-only for models)
```

Network Exposure (v1.0):
- **All services on localhost**: No external network access
- **Firewall**: Block all incoming except SSH (if remote management)

---

11. Monitoring & Observability

11.1 Metrics to Collect

11.1.1 System Metrics (via Prometheus)

Metric	Type	Labels	Purpose
`sovereign_request_total`	Counter	`domain` , `stakes` , `status`	Track task volume
`sovereign_request_duration_seconds`	Histogram	`domain` , `stakes`	Track latency distribution
`sovereign_router_accuracy`	Gauge	-	Track routing performance
`sovereign_validator_verdicts_total`	Counter	`verdict` (pass/fail)	Track validation outcomes
`sovereign_model_swap_duration_seconds`	Histogram	`model`	Track swap performance
`sovereign_vram_used_bytes`	Gauge	-	Monitor VRAM
`sovereign_ram_used_bytes`	Gauge	`pool` (warm/system)	Monitor RAM
`sovereign_gpu_temperature_celsius`	Gauge	-	Thermal monitoring
`sovereign_model_inference_tokens_per_second`	Gauge	`model`	Track generation speed

Example Prometheus Query (p95 latency):

```
histogram_quantile(0.95,
  rate(sovereign_request_duration_seconds_bucket[5m])
)
```

11.1.2 Business Metrics

Metric	Calculation	Purpose
Router Accuracy	(Correct classifications / Total) over rolling 7-day window	Track routing quality
Validator FP Rate	(False Passes / Total Passes)	Track validation strictness
Validator FN Rate	(False Fails / Total Fails)	Track validation leniency
Task Success Rate	(Completed / Total Submitted)	Track overall reliability
Avg. Latency by Stakes	Mean latency grouped by low/ medium/high stakes	Track performance vs. rigor

11.2 Logging Strategy

Structured Logging (JSON format):

```
{
  "timestamp": "2026-02-05T10:30:00.123Z",
  "level": "INFO",
  "component": "orchestrator",
  "request_id": "req_1234567890",
  "action": "load_model",
  "details": {
    "model": "qwen_coder_32b",
    "source": "warm_pool",
    "duration_ms": 2341
  }
}
```

Log Levels:
- **DEBUG**: Detailed trace (disabled in production)
- **INFO**: Normal operations (routing decisions, model loads, validations)
- **WARNING**: Non-critical issues (low router confidence, validation retries)
- **ERROR**: Failures requiring attention (model load failures, OOM)
- **CRITICAL**: System-level failures (orchestrator crash, data corruption)

Log Rotation:
- **Daily rotation**: New log file each day
- **Retention**: 30 days
- **Compression**: gzip logs older than 7 days

11.3 Alerting Rules

Critical Alerts (PagerDuty):
- System crash (orchestrator down)
- VRAM OOM imminent (>95% utilization)
- Model load failure (after 3 retries)
- Memory ledger corruption detected

Warning Alerts (Email/Slack):
- High latency (p95 >120 seconds for 10 minutes)
- Router accuracy drop (<80% over 24 hours)
- Validator FN rate spike (>10% over 24 hours)
- Disk space low (<100GB free)

11.4 Dashboards (Grafana)

Dashboard 1: System Overview
- System status (components up/down)
- Task volume (requests/hour)
- Latency (p50, p95, p99)
- Resource utilization (VRAM, RAM, CPU, disk)

Dashboard 2: Model Performance
- Inference speed per model (tokens/second)
- Model swap times
- Current GPU model + warm pool contents
- Model load/unload frequency

Dashboard 3: Routing & Validation
- Router accuracy (7-day rolling)
- Routing confidence distribution
- Validator pass/fail rates
- Validation latency

Dashboard 4: Tasks & Errors
- Task success/failure rates
- Error breakdown by type
- Retry frequency
- Top error messages

---

12. Development & Build Architecture

12.1 Repository Structure

```
sovereign-ai/
├── .git/                           # Git version control
├── .github/                        # GitHub Actions CI (optional)
│   └── workflows/
│       └── tests.yml
├── src/                            # Source code
│   ├── __init__.py
│   ├── orchestrator/               # Orchestration layer
│   │   ├── __init__.py
│   │   ├── main.py                 # Entry point
│   │   ├── request_handler.py
│   │   └── validation_loop.py
│   ├── router/                     # Router service
│   │   ├── __init__.py
│   │   ├── classifier.py           # Python wrapper
│   │   ├── prolog_interface.py     # Prolog integration
│   │   └── rules.pl                # Prolog routing rules
│   ├── memory/                     # Memory manager
│   │   ├── __init__.py
│   │   ├── manager.py
│   │   ├── parser.py               # Markdown parser
│   │   └── retrieval.py            # Semantic retrieval
│   ├── models/                     # Model serving
│   │   ├── __init__.py
│   │   ├── loader.py               # Warm pool manager
│   │   ├── router_model.py
│   │   ├── validator_model.py
│   │   └── worker_models.py
│   ├── tools/                      # Multimodal tools
│   │   ├── __init__.py
│   │   ├── ocr.py
│   │   ├── vision.py
│   │   └── embeddings.py
│   ├── cli/                        # CLI interface
│   │   ├── __init__.py
│   │   └── commands.py
│   ├── api/                        # REST API
│   │   ├── __init__.py
│   │   ├── server.py               # FastAPI app
│   │   └── routes.py
│   └── utils/                      # Utilities
│       ├── __init__.py
│       ├── telemetry.py            # Prometheus metrics
│       ├── logging_config.py
│       └── config_loader.py
├── config/                         # Configuration files
│   ├── system.yaml                 # System-wide config
│   ├── models.yaml                 # Model paths, quantizations
│   ├── routing_rules.pl            # Prolog routing logic
│   ├── validation_policies.yaml    # Stakes-based policies
│   └── tools.yaml                  # Tool configurations
├── data/                           # Runtime data
│   ├── memory/                     # Markdown memory ledger
│   │   ├── project_state.md
│   │   ├── scratchpad.md
│   │   └── knowledge_graph.md
│   └── vector_store/               # FAISS index
│       └── embeddings.index
├── tests/                          # Test suite
│   ├── unit/
│   │   ├── test_router.py
│   │   ├── test_memory.py
│   │   └── test_validation.py
│   ├── integration/
│   │   ├── test_orchestrator.py
│   │   └── test_end_to_end.py
│   └── fixtures/
│       ├── sample_requests.json
│       └── test_models/
├── scripts/                       # Utility scripts
│   ├── setup_environment.sh       # Initial setup
│   ├── download_models.py         # Model download automation
│   ├── quantize_models.sh         # Quantization pipeline
│   └── start_services.sh          # Start all services
├── docs/                          # Documentation
│   ├── architecture/              # This document, others
│   ├── api/                       # API docs
│   └── operations/                # Runbooks
├── logs/                          # Application logs
│   └── .gitkeep
├── requirements.txt               # Python dependencies
├── requirements-dev.txt           # Dev dependencies (pytest, etc.)
├── setup.py                       # Package setup
├── pyproject.toml                 # Modern Python config
├── .env.example                   # Example environment variables
├── .gitignore
├── README.md
└── LICENSE
```

12.2 Build & Deployment Process

12.2.1 Initial Setup

```
# 1. Clone repository
git clone https://github.com/org/sovereign-ai.git
cd sovereign-ai

# 2. Run setup script
./scripts/setup_environment.sh
# This installs: system packages, CUDA, llama.cpp, Python venv, Prolog

# 3. Download and quantize models
python scripts/download_models.py --all
./scripts/quantize_models.sh

# 4. Configure system
cp .env.example .env
# Edit .env with paths, settings

# 5. Initialize memory ledger
mkdir -p data/memory
git init data/memory  # Track memory in separate Git repo

# 6. Start services
./scripts/start_services.sh
# Starts: llama.cpp servers, orchestrator, Prometheus exporter

# 7. Verify
sovereign status
```

12.2.2 Development Workflow

```
# Create feature branch
git checkout -b feature/improve-routing

# Make changes to src/router/rules.pl

# Run tests
pytest tests/ -v

# Run linter
pylint src/
black src/ --check

# Commit changes
git add src/router/rules.pl
git commit -m "Improve routing logic for ambiguous coding queries"

# Push and create PR
git push origin feature/improve-routing
```

12.2.3 CI/CD Pipeline (GitHub Actions example)

```yaml
# .github/workflows/tests.yml
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install -r requirements-dev.txt
      - name: Run linter
        run: |
          pylint src/
          black src/ --check
      - name: Run unit tests
        run: pytest tests/unit/ -v
      # Integration tests require GPU, run on self-hosted runner
```

13. Appendices

13.1 Acronyms & Abbreviations

Acronym	Meaning
SAD	System Architecture Document
PRD	Product Requirements Document
CLI	Command-Line Interface
REST	Representational State Transfer
API	Application Programming Interface
VRAM	Video Random Access Memory (GPU memory)
RAM	Random Access Memory (system memory)
NVMe	Non-Volatile Memory Express (fast SSD)
GGUF	GPT-Generated Unified Format (model format)
MoE	Mixture of Experts (model architecture)
OCR	Optical Character Recognition
LLM	Large Language Model
FP	False Positive (validation error)
FN	False Negative (validation error)
RTO	Recovery Time Objective
RPO	Recovery Point Objective
OOM	Out Of Memory
SLO	Service Level Objective

13.2 Glossary of Terms

Term	Definition
Bicameral Architecture	Design separating creative generation (GPU) from critical validation (CPU)
Warm Pool	Pre-loaded models in RAM for fast GPU loading
Stakes	Risk level of a task (low/medium/high) determining validation rigor
Grounding	Ensuring outputs are attributable to source material (e.g., OCR text)
Provenance	Tracking the origin and lineage of data
Memory Ledger	Markdown files serving as persistent, auditable state
Quantization	Reducing numerical precision (FP32 → INT4) to reduce model size
Router	Component classifying requests and selecting appropriate model
Validator	Component checking outputs for correctness, safety, grounding
Worker	Specialist AI model performing generation (coding, reasoning, creative)
Orchestrator	Central coordinator managing request flow

13.3 C4 Model Diagrams Summary

Level 1 - System Context: User ↔ Sovereign AI ↔ External Systems (ModelRepo, Git, Monitoring)

Level 2 - Container: Presentation (CLI/API) → Orchestration (Python/Prolog) → Inference (llama.cpp) → Storage (Models, Memory, VectorDB)

Level 3 - Component: See Section 4 (Component Architecture) for detailed component diagram

Level 4 - Code: Implementation-specific; deferred to code documentation

---

13.4 Design Decisions Log

ID	Date	Decision	Rationale	Alternatives	Status
DD-001	2026-02-05	Use Hybrid Python/Prolog	Python for ML ecosystem, Prolog for declarative rules	Pure Python (less expressive), Pure Prolog (poor ML support)	**Approved**
DD-002	2026-02-05	CPU-resident Validator	Eliminates GPU thrashing	GPU validator (causes thrashing), No validation (no governance)	**Approved**
DD-003	2026-02-05	Markdown Memory Ledger	Human-readable, Git-trackable, transparent	SQL (opaque), JSON (less readable), Neural memory (black box)	**Approved**
DD-004	2026-02-05	Warm Pool in RAM	3-5x faster model swaps	Always from disk (slow), All in VRAM (impossible)	**Approved**
DD-005	2026-02-05	Sequential Execution (v1.0)	Simpler, avoids concurrency complexity	Concurrent execution (more complex, v2.0)	**Approved**
DD-006	2026-02-05	FAISS for Vector Store	Lightweight, CPU-efficient	ChromaDB (heavier), Pinecone (cloud, violates sovereignty)	**Approved**
DD-007	2026-02-05	Tesseract for OCR	Free, good accuracy	PaddleOCR (better but heavier), Commercial APIs (violates sovereignty)	**Under Review**

13.5 References

1. **Product Requirements Document (PRD)** v1.0, 2026-02-05
2. **Technical Analysis Report** (Deep Agent Report), 2026-01-03
3. **Comprehensive Architecture Report** (Hybrid Logic Router), 2026-01-16
4. **Project Roadmap**, 2026-02-05
5. **Document Roadmap**, 2026-02-05
6. **llama.cpp Documentation**: https://github.com/ggerganov/llama.cpp
7. **SWI-Prolog Documentation**: https://www.swi-prolog.org/
8. **GGUF Specification**: https://github.com/ggerganov/ggml/blob/master/docs/gguf.md
9. **C4 Model**: https://c4model.com/

13.6 Document Revision History

Version	Date	Author	Changes
0.1	2026-02-05	Architecture Team	Initial draft
1.0	2026-02-05	Architecture Team	Complete first version for review

Document Status: Draft for Review
Next Steps: Review by Technical Lead, Engineering Manager, ML Lead, DevOps Lead, Security Architect
Target Approval Date: 2026-02-12
Owner: Solutions Architect / Technical Lead

End of System Architecture Document