Project Roadmap: Sovereign AI Infrastructure

Bicameral Validator Ladder - Implementation Timeline

Project: Local Multi-Model AI System with GPU/CPU Heterogeneous Compute
Version: 1.0
Date: February 5, 2026
Project Duration: 6-8 months (Proof of Concept to Production-Ready)

Executive Summary

This roadmap outlines the **phased implementation** of the Sovereign AI Infrastructure, from initial proof-of-concept through production deployment. The project is structured in **8 phases** over 6-8 months, with clear milestones, deliverables, and success criteria.

Key Principles:
- **Incremental validation**: Start small (2-3 models), add complexity only when justified
- **Instrumentation-first**: Measure everything from day one
- **Risk mitigation**: Validate critical assumptions early (router accuracy, validation effectiveness)
- **Documentation-driven**: Design docs before code; operational docs before deployment

---

Timeline Overview

```
Month 1    Month 2    Month 3    Month 4    Month 5    Month 6    Month 7-8
|----------|----------|----------|----------|----------|----------|----------|
[ Phase 0: Foundation & Setup                        ]
           [ Phase 1: Core Router + Basic Orchestration           ]
                      [ Phase 2: Validation Loop + Memory System        ]
                                 [ Phase 3: Full Model Stack + Warm Pools     ]
                                            [ Phase 4: Multimodal Pipelines        ]
                                                       [ Phase 5: Optimization & Tun-
ing   ]
                                                                  [ Phase 6: Ops & De-
ployment ]
                                                                             [ Phase
7: Production Hardening ]

Milestone:    M0         M1          M2            M3              M4              M
5           M6
              Setup     Router Live    Validation    Full Stack    Multimodal      Optim-
ized    Production
```

Phase 0: Foundation & Infrastructure Setup

Duration: 3-4 weeks
Team: 1-2 engineers (DevOps + 1 developer)
Objective: Establish hardware, tooling, and baseline measurements

Goals

1. Procure and configure hardware
2. Install and verify all dependencies
3. Download and quantize initial models (2-3 only)
4. Establish baseline performance metrics
5. Set up development environment and repository

Key Activities

Week 1: Hardware & OS Setup

- [ ] **Procure hardware** (if not available)
- NVIDIA Tesla A2 (16GB VRAM) or equivalent
- System with 128GB ECC RAM (minimum 64GB acceptable for PoC)
- 1TB NVMe SSD
- Intel Xeon or AMD Ryzen with 6+ cores
- [ ] **Install OS**: Ubuntu 22.04 LTS (recommended)
- [ ] **Install drivers**:
- NVIDIA Driver 535+
- CUDA 12.1+
- cuDNN 8.9+
- [ ] **Verify GPU**: Run `nvidia-smi`, check VRAM, run basic CUDA test

Week 2: Software Stack Installation

- [ ] **Install core dependencies**:
- Python 3.10+ (with pip, venv)
- llama.cpp (with CUDA support)
- SWI-Prolog 8.4+ or GNU Prolog
- Git, Docker (optional for containerization)
- [ ] **Install Python packages**:
- `llama-cpp-python` (with CUDA)
- `requests`, `pydantic`, `fastapi`, `uvicorn`
- `sentence-transformers` (for embeddings)
- `pytesseract` or equivalent (for OCR)
- `torch`, `transformers` (for vision encoders)
- `prometheus-client` (for monitoring)
- [ ] **Test installations**: Run sample inference with llama.cpp, verify Prolog installation

Week 3: Initial Model Deployment (Minimal Stack)

- [ ] **Download and quantize 3 models**:
  1. **Granite-Micro 3B** (Router): FP16 or Q8 (CPU-resident)
  2. **Qwen Coder 32B** (Worker): Q4_K_M quantization
  3. **Granite-H-Small** (Validator): Q4_K_M or Q5_K_M (CPU-resident)

- [ ] **Model vault structure**: Create `/mnt/models/` with organized subdirectories
- [ ] **Benchmark each model**:
- Inference speed (tokens/second)
- Memory footprint (VRAM for GPU, RAM for CPU)
- Load time (cold start, warm start)

Week 4: Baseline Measurements & Repository Setup

- [ ] **Performance baselines**:
- Single-model inference latency (Qwen Coder alone)
- Model swap time (NVMe → RAM → VRAM)
- CPU inference speed (Granite-Micro, Granite-H-Small)
- [ ] **Git repository setup**:
- Initialize repo with `.gitignore` (exclude models)
- Create directory structure: `src/`, `tests/`, `docs/`, `configs/`
- Set up branching strategy (main, develop, feature branches)
- [ ] **Documentation**:
- Hardware specifications document
- Installation guide (capture exact steps)
- Baseline performance report

Deliverables

- ✅ **Operational hardware** with all dependencies installed
- ✅ **3 quantized models** (Router, Worker, Validator) successfully running
- ✅ **Baseline performance report** with measurements
- ✅ **Git repository** with project structure
- ✅ **Infrastructure documentation**

Success Criteria

- [ ] GPU successfully runs Qwen Coder 32B (Q4_K_M) with context window of 4K+ tokens
- [ ] CPU successfully runs Granite-Micro (Router) at 10+ tokens/second
- [ ] CPU successfully runs Granite-H-Small (Validator) at 3+ tokens/second
- [ ] Model swap (VRAM unload + RAM → VRAM load) completes in <10 seconds
- [ ] All baseline metrics documented

Risks & Mitigations

Risk	Likelihood	Impact	Mitigation
GPU VRAM insufficient for Qwen 32B	Medium	High	Use more aggressive quantization (Q3_K_M) or smaller model (Qwen 20B)
CPU too slow for validation	Medium	High	Accept slower validation (5-7 tok/s) or reduce validator model size
RAM pressure (128GB not enough)	Low	Medium	Start with 2 models in RAM; add warm pool later
Dependency conflicts	Medium	Low	Use Docker containers for isolation

---

Phase 1: Core Router + Basic Orchestration

Duration: 3-4 weeks
Team: 2-3 engineers (1 Backend, 1 Logic/Prolog, 1 ML)
Objective: Build working router that classifies requests and routes to correct model

Goals

1. Implement Granite-Micro router (Python wrapper)
2. Define routing logic in Prolog (declarative rules)
3. Build orchestration layer (Python)
4. Test router accuracy on 100+ prompts
5. Implement single-worker execution (no validation yet)

Key Activities

Week 1: Router Implementation (Python)

- [ ] **Design router API**:
- Input: User prompt (text)
- Output: JSON with `{domain, stakes, model, tools_required, confidence}`
- [ ] **Implement Granite-Micro wrapper**:
- Python class `RouterEngine` that calls llama.cpp endpoint
- System prompt for classification task
- JSON parsing and validation
- [ ] **Create test dataset**:
- 100 prompts covering: coding (architecture, implementation), reasoning, creative, documentation
- Ground truth labels (manually assigned)

Week 2: Routing Logic (Prolog)

- [ ] **Define Prolog predicates**:
- `route(Input, Domain, Stakes, Model)` - main routing rule
- `classify_domain(Input, Domain)` - domain classification
- `assess_stakes(Input, Domain, Stakes)` - risk assessment
- `select_model(Domain, Stakes, Model)` - model selection
- [ ] **Implement decision rules**:
- Keyword-based classification (initial simple rules)
- Domain mapping (coding → Qwen, reasoning → GPT-OSS, etc.)
- Stakes mapping (complexity indicators → high/medium/low)
- [ ] **Python-Prolog integration**:
- Use `pyswip` or subprocess calls to query Prolog
- Pass router output to Prolog for validation/refinement

Week 3: Orchestration Layer

- [ ] **Build orchestration engine** (`orchestrator.py`):
- Receives user request
- Calls Router to get routing decision
- Loads appropriate Worker model (GPU)
- Executes generation
- Returns output to user
- [ ] **Implement model loading/unloading**:
- GPU model management (load to VRAM, unload)
- Simple synchronous execution (no parallel validation yet)
- [ ] **Logging and telemetry**:
- Log every routing decision with reasoning
- Log model load/unload times
- Log inference times and token counts

Week 4: Testing & Evaluation

- [ ] **Router accuracy testing**:
- Run 100-prompt test dataset through router
- Calculate classification accuracy (correct domain, correct stakes)
- Identify misclassification patterns
- [ ] **End-to-end testing**:
- 20 realistic tasks (user prompt → routed output)
- Measure total latency breakdown
- Validate correct model was used
- [ ] **Iterate on routing logic**:
- Refine Prolog rules based on errors
- Add edge case handling
- Improve confidence scoring

Deliverables

- ✅ **Working router** (Python + Prolog) with >80% accuracy
- ✅ **Orchestration engine** that routes requests to appropriate Worker

- ✅ **Test dataset** with 100+ labeled prompts
- ✅ **Router accuracy report** with error analysis
- ✅ **API specification** for router and orchestrator

Success Criteria

- [ ] Router achieves ≥80% domain classification accuracy on test set
- [ ] Router correctly assesses stakes (high/medium/low) ≥75% of the time
- [ ] End-to-end latency (prompt → output) is acceptable (≤30 seconds for typical task)
- [ ] All routing decisions are logged with human-readable reasoning
- [ ] Zero GPU OOM (Out-of-Memory) errors during testing

Risks & Mitigations

Risk	Likelihood	Impact	Mitigation
Router accuracy below 80%	Medium	High	Enhance Prolog rules; add embedding-based fallback
Python-Prolog integration brittle	Medium	Medium	Isolate Prolog calls; add error handling
Model loading too slow	Low	Medium	Implement warm pool (Phase 3)
Ambiguous prompts confuse router	High	Medium	Add confidence thresholding; default to safe model

---

Phase 2: Validation Loop + Memory System

Duration: 3-4 weeks
Team: 2-3 engineers (1 Backend, 1 ML, 1 Data/Storage)
Objective: Add CPU-resident validator and Markdown memory ledger

Goals

1. Implement Granite-H-Small validator (CPU-resident)
2. Build Markdown memory system (project_state.md, scratchpad.md, knowledge_graph.md)
3. Implement Generate → Validate → Commit loop
4. Test validation accuracy (false positive/negative rates)
5. Enable block-by-block generation with validation

Key Activities

Week 1: Validator Implementation

- [ ] **Deploy Granite-H-Small on CPU**:
- Configure llama.cpp for CPU inference (no GPU)

- System prompt for validation role
- Output format: `[PASS]` or `[FAIL: reason]`
- [ ] **Design validator API**:
- Input: Worker output block + project state + scratchpad
- Output: Verdict (pass/fail) + reasoning
- [ ] **Create validation test cases**:
- 50 code blocks (25 correct, 25 with errors: logic, syntax, hallucination)
- Ground truth labels

Week 2: Markdown Memory System

- [ ] **Define memory schemas**:
- `project_state.md`: Immutable facts, global objective, proven constraints
- `scratchpad.md`: Active reasoning, pending steps, validator feedback
- `knowledge_graph.md`: Learned patterns, model-specific gotchas
- [ ] **Implement memory manager** (`memory.py`):
- Read/write functions for each memory file
- Append operations (for scratchpad)
- Update operations (for project state)
- Query operations (retrieve relevant sections)
- [ ] **Version control integration**:
- Memory files tracked in Git
- Each task commits memory state

Week 3: Validation Loop Integration

- [ ] **Implement Generate → Validate → Commit workflow**:
- Worker generates block (5-10 lines of code or 1 logical step)
- System pauses Worker
- Validator loads, reads context + new block from scratchpad
- Validator returns verdict
- If PASS: commit block to project_state.md
- If FAIL: write correction to scratchpad.md, Worker retries
- [ ] **Orchestration updates**:
- Modify orchestrator to support block-by-block execution
- Add retry logic (max 3 retries per block)
- Add escape hatch (user override if validation loops)

Week 4: Validation Testing & Tuning

- [ ] **Validation accuracy testing**:
- Run 50 test cases through validator
- Calculate: True Positive, False Positive, True Negative, False Negative rates
- Acceptable: FP <10%, FN <5%
- [ ] **Prompt engineering**:
- Refine validator prompts to reduce false rejections
- Add few-shot examples to prompts
- Test different validation strictness levels
- [ ] **Performance testing**:

- Measure validation latency per block
- Measure impact on end-to-end task completion time
- Identify if CPU validation is bottleneck

Deliverables

- ✅ **CPU-resident validator** (Granite-H-Small) operational
- ✅ **Markdown memory system** with defined schemas
- ✅ **Working validation loop** (Generate → Validate → Commit)
- ✅ **Validation accuracy report** (FP/FN rates)
- ✅ **Prompt library** for Worker and Validator roles

Success Criteria

- [ ] Validator False Positive rate <10% (doesn't reject good outputs)
- [ ] Validator False Negative rate <5% (catches real errors)
- [ ] Validation adds <5 seconds per block (at 3-5 tok/s on CPU)
- [ ] Memory system successfully persists state across sessions
- [ ] Validation loop completes 20 realistic tasks without infinite retry loops

Risks & Mitigations

Risk	Likelihood	Impact	Mitigation
Validator too strict (high FP)	High	High	Tune prompts; add confidence thresholds
Validator too lenient (high FN)	Medium	High	Enhance validation criteria; add test suite
Validation latency unacceptable	Medium	Medium	Accept slower execution; optimize prompts
Memory file corruption	Low	Medium	Implement file locking; add integrity checks
Infinite retry loops	Medium	Medium	Max retry limit; user override mechanism

---

Phase 3: Full Model Stack + Warm Pool Strategy

Duration: 3-4 weeks
Team: 2-3 engineers (1 ML, 1 Backend, 1 DevOps)
Objective: Add remaining specialist models and implement RAM warm pool

Goals

1. Add GPT-OSS 20B (reasoning), Nemotron-3 Nano 30B (coding), MythoMax-L2-13B (creative)
2. Expand router logic to handle all specialists
3. Implement warm pool strategy (pre-load models into RAM)
4. Optimize model swap times
5. Test full stack under load

Key Activities

Week 1: Additional Model Deployment

- [ ] **Download and quantize remaining models**:
- GPT-OSS 20B (Q4_K_M)
- Nemotron-3 Nano 30B (Q4_K_M)
- MythoMax-L2-13B (Q5_K_M for higher quality)
- [ ] **Update model vault structure**:
- Organize by role: `/mnt/models/reasoning/`, `/mnt/models/coding/`, `/mnt/models/creative/`
- Multiple quantization levels per model (Q4, Q5, Q6)
- [ ] **Benchmark new models**:
- Inference speed, memory footprint, load time
- Quality assessment (subjective evaluation on sample tasks)

Week 2: Router Logic Expansion

- [ ] **Update Prolog routing rules**:
- Add reasoning domain → GPT-OSS
- Add creative domain → MythoMax
- Add specialist selection logic (Qwen for architecture, Nemotron for implementation)
- [ ] **Test routing with expanded model set**:
- 200-prompt test dataset (cover all domains)
- Validate correct model selection
- Measure router confidence across domains

Week 3: Warm Pool Implementation

- [ ] **Design warm pool manager** (`warm_pool.py`):
- Track which models are loaded in RAM
- Predictive pre-loading (based on current domain)
- Eviction policy (LRU or domain-based)
- [ ] **Implement RAM → VRAM transfer**:
- Fast model loading from RAM
- GPU memory management (unload current, load next)
- Minimize VRAM fragmentation
- [ ] **Optimize model loading**:
- Benchmark: NVMe → VRAM vs. RAM → VRAM
- Implement parallel loading where possible
- Test PCIe bandwidth saturation

Week 4: Full Stack Testing

- [ ] **Integration testing**:

- 50 end-to-end tasks covering all domains (coding, reasoning, creative, mixed)
- Measure: latency, model swap frequency, RAM utilization
- [ ] **Load testing**:
- Sustained load (sequential tasks over 1 hour)
- Monitor: GPU temp, CPU usage, RAM pressure, swap to disk
- [ ] **Failure mode testing**:
- OOM scenarios
- Model swap failures
- Recovery procedures

Deliverables

- ✅ **6-model stack** (1 Router, 1 Validator, 4 Workers) operational
- ✅ **Warm pool manager** with predictive loading
- ✅ **Expanded routing logic** covering all specialist models
- ✅ **Performance report**: Model swap times, end-to-end latencies, resource utilization
- ✅ **Failure recovery procedures**

Success Criteria

- [ ] All 4 Worker models successfully generate outputs in their domains
- [ ] Warm pool reduces model swap time to <3 seconds (RAM → VRAM)
- [ ] Router correctly selects specialist models ≥85% of the time
- [ ] System operates for 1 hour under sustained load without crashes
- [ ] RAM utilization stays below 90% (avoid swap to disk)

Risks & Mitigations

Risk	Likelihood	Impact	Mitigation
RAM pressure causes swapping	High	High	Reduce number of models in warm pool; prioritize most-used
Model thrashing (frequent swaps)	Medium	Medium	Improve routing prediction; batch similar tasks
OOM crashes	Medium	High	Implement aggressive VRAM monitoring; fallback to smaller models
Performance worse than single model	Low	High	Measure vs. baseline; optimize swap overhead

Phase 4: Multimodal Pipelines (Embeddings, OCR, Vision)

Duration: 3-4 weeks
Team: 2-3 engineers (1 ML, 1 Backend, 1 Integration)
Objective: Add embeddings, OCR, and vision encoder pipelines

Goals

1. Implement embedding model for retrieval (RAG)
2. Implement OCR pipeline for document ingestion
3. Implement vision encoder for image understanding
4. Integrate multimodal tools into routing and validation logic
5. Test grounding and provenance tracking

Key Activities

Week 1: Embedding Model + RAG

- [ ] **Deploy embedding model**:
- Sentence-Transformers (e.g., `all-MiniLM-L6-v2` or `all-mpnet-base-v2`)
- CPU-resident (fast enough for <1s encoding)
- [ ] **Implement vector database**:
- Options: FAISS (lightweight), ChromaDB, or simple numpy
- Index: Markdown memory files, project documentation
- [ ] **Build retrieval pipeline** (`retrieval.py`):
- Embed user query
- Retrieve top-k relevant snippets (k=5-10)
- Inject into Worker context
- [ ] **Scout phase integration**:
- Before generation, router triggers retrieval if needed
- Relevant context added to project_state.md

Week 2: OCR Pipeline

- [ ] **Select OCR engine**:
- Options: Tesseract (open-source), PaddleOCR, or commercial API
- Test on sample scanned documents
- [ ] **Implement OCR pipeline** (`ocr.py`):
- Input: Image or PDF
- Output: Extracted text + bounding boxes (if available)
- Provenance: Store source image reference with text
- [ ] **Integrate into routing**:
- Router detects image/PDF input → route to OCR first
- OCR output becomes structured artifact in memory
- [ ] **Validation integration**:
- Validator can check if claims are grounded in OCR text
- Flag ungrounded assertions

Week 3: Vision Encoder Pipeline

- [ ] **Deploy vision encoder**:
- Options: CLIP (OpenAI), BLIP (Salesforce), or lightweight alternative
- CPU or GPU-resident (depending on performance needs)
- [ ] **Implement vision pipeline** (`vision.py`):
- Image → dense caption
- Image → object detection (if needed)
- Image → embeddings (for visual retrieval)
- [ ] **Integrate into routing**:
- Router detects visual input → trigger vision encoder
- Caption/features become context for Worker
- [ ] **Multimodal validation**:
- Validator checks if outputs align with detected image content

Week 4: Grounding & Provenance Testing

- [ ] **Create multimodal test dataset**:
- 30 tasks: 10 with OCR (scanned docs), 10 with images, 10 with both
- Ground truth: Expected grounding citations
- [ ] **Test grounding accuracy**:
- Do outputs cite OCR sources correctly?
- Do outputs reference image content accurately?
- [ ] **Test provenance tracking**:
- Can we trace every claim back to source (OCR text, image features)?
- Audit trail in memory system
- [ ] **Performance testing**:
- Latency of OCR, vision, embeddings
- Impact on end-to-end task completion

Deliverables

- ✅ **Embedding model + RAG** operational
- ✅ **OCR pipeline** for document ingestion
- ✅ **Vision encoder pipeline** for image understanding
- ✅ **Multimodal routing logic** (router handles text, image, document inputs)
- ✅ **Grounding & provenance report** (accuracy of source citations)

Success Criteria

- [ ] Embeddings retrieval adds <2 seconds to task latency
- [ ] OCR successfully extracts text from scanned documents with >90% accuracy
- [ ] Vision encoder generates accurate captions/descriptions
- [ ] Multimodal tasks produce outputs with correct source citations (>80% grounding rate)
- [ ] Provenance trail is auditable in memory system

Risks & Mitigations

Risk	Likelihood	Impact	Mitigation
OCR errors create false grounding	High	Medium	Surface OCR confidence; validator flags uncertain citations
Vision encoder misinterprets images	Medium	Medium	Use conservative claims; validator checks alignment
Multimodal latency too high	Medium	Medium	Optimize pipelines; consider GPU for vision if needed
Grounding logic too complex	Medium	Low	Start with simple keyword matching; iterate

---

Phase 5: Optimization & Performance Tuning

Duration: 3-4 weeks
Team: 2-3 engineers (1 Performance Eng, 1 ML, 1 Backend)
Objective: Optimize latency, accuracy, and resource utilization

Goals

1. Profile system bottlenecks (CPU, GPU, I/O, network)
2. Optimize router accuracy and confidence scoring
3. Optimize validator prompts (reduce false positives/negatives)
4. Optimize warm pool strategy (predictive loading accuracy)
5. Implement caching and memoization where applicable

Key Activities

Week 1: Profiling & Bottleneck Identification

- [ ] **End-to-end profiling**:
- Instrument all major components (router, worker, validator, tools)
- Measure: latency distribution, resource utilization, waiting times
- [ ] **Identify top 3 bottlenecks**:
- E.g., validator CPU inference too slow, model swaps frequent, retrieval slow
- [ ] **Create optimization plan**:
- Prioritize by impact × feasibility
- Set optimization targets (e.g., reduce latency by 20%)

Week 2: Router & Routing Logic Optimization

- [ ] **Improve router accuracy**:

- Analyze misclassifications from Phase 1-4 testing
- Refine Prolog rules, add edge cases
- Consider embedding-based fallback for ambiguous prompts
- [ ] **Confidence scoring**:
- Implement confidence thresholds (low confidence → safer model choice)
- Add “uncertainty notes” to audit trail
- [ ] **Routing prediction accuracy**:
- Improve warm pool prediction (reduce unnecessary model swaps)
- Add domain persistence (if user asks follow-up, stay in same domain)

Week 3: Validator & Prompt Optimization

- [ ] **Validator prompt tuning**:
- Reduce false positives (unnecessary rejections)
- Maintain low false negatives (don’t miss real errors)
- Add few-shot examples to prompts
- [ ] **Validation granularity**:
- Test different block sizes (line-by-line vs. function-by-function)
- Find optimal trade-off between rigor and latency
- [ ] **Parallel validation** (if feasible):
- Explore running validator on separate CPU thread
- Overlap Worker generation with Validator checking

Week 4: System-Wide Optimizations

- [ ] **Caching**:
- Cache router decisions for similar prompts
- Cache embeddings for repeated queries
- Cache validator decisions for identical blocks
- [ ] **Memory optimizations**:
- Optimize Markdown file I/O (in-memory buffers)
- Reduce memory fragmentation
- [ ] **GPU optimizations**:
- Enable KV cache quantization (INT8) for longer contexts
- Optimize batch size for GPU utilization
- [ ] **Benchmark improvements**:
- Re-run all performance tests
- Document improvements (before/after)

Deliverables

- ✅ **Performance optimization report** (before/after metrics)
- ✅ **Optimized router** (accuracy ≥90%, confidence scoring)
- ✅ **Optimized validator** (FP <5%, FN <3%)
- ✅ **Optimized warm pool** (swap frequency reduced by ≥30%)
- ✅ **Caching layer** for repeated operations

Success Criteria

- [ ] Router accuracy ≥90% on test set

- [ ] Validator false positive rate <5%, false negative rate <3%
- [ ] End-to-end latency reduced by ≥20% vs. Phase 4
- [ ] Model swap frequency reduced by ≥30%
- [ ] No performance regressions (all Phase 4 tests still pass)

Risks & Mitigations

Risk	**Likelihood**	**Impact**	**Mitigation**
Optimizations break functionality	Medium	High	Comprehensive regression testing after each change
Optimization targets not met	Medium	Medium	Accept incremental improvements; prioritize most impactful
Over-optimization (diminishing returns)	Medium	Low	Set time box; stop when targets met

---

Phase 6: Operations, Monitoring & Deployment

Duration: 3-4 weeks
Team: 2-3 engineers (1 DevOps, 1 SRE, 1 Backend)
Objective: Prepare for production deployment with full observability

Goals

1. Implement comprehensive monitoring and alerting
2. Create deployment automation (scripts, containers)
3. Write operations runbook and troubleshooting guide
4. Set up backup and disaster recovery
5. Conduct deployment dry-run

Key Activities

Week 1: Monitoring & Observability

- [ ] **Metrics collection**:
- Prometheus instrumentation for all components
- Metrics: router accuracy, validation rates, latency (p50/p95/p99), resource utilization
- Custom metrics: model swap frequency, warm pool hit rate, grounding accuracy
- [ ] **Logging**:
- Structured logging (JSON format)
- Centralized log aggregation (if multi-node)
- Audit trail logging (all routing/validation decisions)
- [ ] **Dashboards**:
- Grafana dashboards: system health, performance, error rates
- Real-time monitoring of GPU/CPU/RAM

- [ ] **Alerting**:
- Alert rules: OOM warnings, high latency, validation bottlenecks, router errors
- Alert channels: email, Slack, PagerDuty

Week 2: Deployment Automation

- [ ] **Containerization** (optional but recommended):
- Docker containers for: orchestrator, router, validator, tools
- Docker Compose or Kubernetes manifests
- [ ] **Deployment scripts**:
- Automated model downloads and quantization
- Configuration management (environment variables, config files)
- Health check scripts
- [ ] **CI/CD pipeline** (if applicable):
- Automated testing on code changes
- Automated deployment to staging environment
- Rollback procedures

Week 3: Operations Documentation

- [ ] **Write operations runbook**:
- Startup procedures
- Shutdown procedures
- Routine maintenance (log rotation, model updates)
- Troubleshooting guide (common errors and fixes)
- Escalation procedures
- [ ] **Disaster recovery plan**:
- Backup strategy (models, memory files, configuration)
- Recovery procedures (hardware failure, corruption, etc.)
- RTO/RPO targets
- [ ] **Security documentation**:
- Access controls
- Audit logging
- Incident response plan

Week 4: Deployment Dry-Run & Validation

- [ ] **Staging deployment**:
- Deploy full stack to staging environment
- Run full test suite (functionality, performance, load)
- [ ] **Failure testing**:
- Simulate failures (GPU crash, OOM, model corruption)
- Validate recovery procedures
- [ ] **Production readiness review**:
- Checklist: all docs complete, monitoring working, tests passing
- Sign-off from: Tech Lead, DevOps, Security

Deliverables

- ✅ **Monitoring infrastructure** (Prometheus + Grafana + Alerting)
- ✅ **Deployment automation** (scripts, containers, CI/CD)

- ✅ **Operations runbook** and troubleshooting guide
- ✅ **Disaster recovery plan** and backup procedures
- ✅ **Production readiness report**

Success Criteria

- [ ] All critical metrics are monitored and visualized
- [ ] Alerting triggers correctly on test failures
- [ ] Deployment can be executed by any team member following runbook
- [ ] Disaster recovery tested and validated (restore from backup successful)
- [ ] Staging environment passes all tests
- [ ] Security review completed and signed off

Risks & Mitigations

Risk	Likelihood	Impact	Mitigation
Monitoring overhead impacts performance	Low	Medium	Optimize metric collection; sample if needed
Deployment automation brittle	Medium	Medium	Test thoroughly in staging; manual fallback procedures
Inadequate documentation	Medium	High	Peer review all docs; test with new team member

---

Phase 7: Production Hardening & Iteration

Duration: 4-6 weeks (ongoing)
Team: Full team
Objective: Deploy to production, monitor, iterate based on real-world usage

Goals

1. Deploy to production environment
2. Monitor real-world performance and errors
3. Collect user feedback
4. Iterate on routing, validation, and prompt quality
5. Establish continuous improvement process

Key Activities

Week 1-2: Production Deployment

- [ ] **Production deployment**:
- Execute deployment runbook
- Gradual rollout (canary or blue-green if applicable)

- Monitor closely during initial hours/days
- [ ] **User onboarding**:
- Train initial users on system capabilities and limitations
- Provide user guide and FAQ
- Set up feedback channels (Slack, email, issue tracker)

Week 3-4: Real-World Monitoring & Issue Resolution

- [ ] **Performance monitoring**:
- Track all metrics in production
- Compare to baseline and optimization targets
- Identify anomalies or degradation
- [ ] **Error analysis**:
- Review logs for errors, failures, edge cases
- Prioritize and fix critical issues
- Update troubleshooting guide with new patterns
- [ ] **User feedback collection**:
- Survey users on satisfaction, pain points, feature requests
- Categorize feedback (bugs, performance, features, usability)

Week 5-6: Iteration & Continuous Improvement

- [ ] **Router refinement**:
- Analyze real-world routing decisions
- Identify and fix misclassifications
- Update Prolog rules based on actual usage patterns
- [ ] **Validator tuning**:
- Review false positive/negative rates in production
- Adjust prompts and thresholds
- Consider domain-specific validation rules
- [ ] **Prompt engineering**:
- Refine Worker prompts based on output quality
- Add domain-specific guidance
- Implement A/B testing for prompt variations
- [ ] **Performance improvements**:
- Optimize identified bottlenecks
- Adjust warm pool strategy based on actual usage
- [ ] **Feature additions** (if needed):
- Implement high-priority user requests
- Extend multimodal capabilities
- Add new specialist models if justified

Ongoing: Maintenance & Evolution

- [ ] **Regular reviews** (monthly):
- Performance review (metrics vs. targets)
- Router/validator accuracy review
- User satisfaction review
- Incident review (post-mortems)

- [ ] **Model updates** (quarterly):
- Evaluate new model releases
- Test alternatives for underperforming specialists
- Update quantizations or model sizes
- [ ] **System evolution**:
- Plan next-generation features
- Architectural improvements
- Scalability enhancements

Deliverables

- ✅ **Production system** operational and stable
- ✅ **User feedback report** with prioritized improvements
- ✅ **Post-deployment review** (lessons learned)
- ✅ **Continuous improvement plan** (roadmap for next 6 months)

Success Criteria

- [ ] System runs in production for 30 days with <99% uptime
- [ ] No critical incidents (P0 outages)
- [ ] User satisfaction ≥7/10 (survey)
- [ ] Router accuracy in production ≥85%
- [ ] Validator effectiveness validated (users trust outputs)
- [ ] All P0/P1 bugs resolved

Risks & Mitigations

Risk	Likelihood	Impact	Mitigation
Production issues not seen in testing	High	High	Comprehensive monitoring; rapid response team
User dissatisfaction with latency	Medium	Medium	Set clear expectations; optimize critical paths
Model quality issues in production	Medium	High	Continuous evaluation; A/B testing; rollback capability
Unexpected usage patterns	High	Medium	Flexible architecture; iterative improvements

---

Resource Requirements

Team Composition (Full Project)

- **Technical Lead**: 1 (across all phases)
- **ML Engineers**: 1-2 (Phases 0-5)
- **Backend Engineers**: 1-2 (Phases 1-6)
- **DevOps/SRE**: 1 (Phases 0, 3, 6, 7)
- **Logic/Prolog Engineer**: 0.5-1 (Phases 1, 2, 5)
- **QA/Test Engineer**: 0.5-1 (Phases 2-6)
- **Technical Writer**: 0.5 (Phase 6-7)

Total team size: 3-5 people (not all full-time, roles may overlap)

Hardware Requirements

- **Development**: 1 workstation (Tesla A2 spec or similar)
- **Staging**: Same spec as production (can be same machine with careful isolation)
- **Production**: 1 workstation (Tesla A2, 128GB RAM, 1TB NVMe) - can scale to multiple if needed

Budget Estimate (Rough)

- **Hardware**: $3,000-5,000 (one-time, if purchasing new)
- **Personnel**: 3-5 engineers × 6-8 months × $100-150/hr = $192k-480k (highly variable by location/seniority)
- **Tools & Services**: $1,000-2,000 (monitoring, CI/CD, misc)
- **Total**: ~$200k-$500k (mostly personnel costs)

Note: If this is an internal/solo project, hardware is already available, and labor is not externally costed, the out-of-pocket cost is nearly zero.

Milestones & Go/No-Go Decision Points

M0: Foundation Complete (Week 4)

Criteria: Hardware operational, baseline models running, baselines documented
Go/No-Go: If GPU cannot run Qwen 32B or CPU validator is too slow (<1 tok/s), STOP and re-evaluate hardware or model choices

M1: Router Live (Week 8)

Criteria: Router achieves ≥80% accuracy, end-to-end execution works
Go/No-Go: If router accuracy <70%, STOP and invest in better routing logic (embeddings, more sophisticated Prolog rules)

M2: Validation Loop Working (Week 12)

Criteria: Validation loop completes tasks, FP <10%, FN <5%
Go/No-Go: If validation is too slow (>10s per block) or inaccurate (FN >10%), STOP and simplify (fewer validation steps, lighter validator model)

M3: Full Stack Operational (Week 16)

Criteria: All 6 models working, warm pool functional, system stable
Go/No-Go: If RAM pressure causes swapping or OOM, STOP and reduce model count or upgrade RAM

M4: Multimodal Pipelines Ready (Week 20)

Criteria: OCR, vision, embeddings working, grounding tested
Go/No-Go: If multimodal adds >10s latency or grounding is inaccurate (<70%), STOP and simplify (remove least critical pipeline)

M5: Optimization Complete (Week 24)

Criteria: Performance targets met (latency, accuracy), system optimized
Go/No-Go: If targets not met, assess if “good enough” or if fundamental redesign needed

M6: Production Deployed (Week 28)

Criteria: System running in production, monitored, stable for 7 days
Go/No-Go: If critical incidents occur repeatedly, STOP production use and return to hardening phase

---

Risk Management Summary

Critical Risks (Project-Level)

Risk	Impact	Mitigation Strategy
Hardware insufficient	Project failure	Validate in Phase 0; have contingency models (smaller/ more aggressive quantization)
Router accuracy too low	Poor user experience	Invest heavily in Phase 1; consider alternative routing approaches (embeddings, hybrid)
Validation adds unacceptable latency	System unusable	Test early (Phase 2); have fallback (optional validation, validation only for high-stakes)
Complexity overwhelming	Slow progress, bugs	Start small (2-3 models), add complexity only when justified; rigorous testing
Real-world performance ≠ lab performance	Production issues	Comprehensive monitoring, gradual rollout, rapid iteration capability

---

Success Metrics (Project-Level)

Technical Success

- [ ] System runs on specified hardware without OOM or crashes
- [ ] Router classification accuracy ≥85% in production
- [ ] Validator false positive <5%, false negative <3%
- [ ] End-to-end latency acceptable (≤30s for typical tasks, ≤60s for high-stakes)
- [ ] System uptime ≥99% in production (first 3 months)
- [ ] Multimodal grounding accuracy ≥80%

Operational Success

- [ ] All documentation complete and accurate
- [ ] Deployment can be executed by any team member
- [ ] Incidents resolved within SLA (to be defined)
- [ ] Disaster recovery tested and validated

Business/User Success

- [ ] User satisfaction ≥7/10
- [ ] Users trust validation outputs (subjective, measured via survey/interviews)
- [ ] System provides value vs. simpler alternatives (measured via user feedback)
- [ ] "Sovereign-grade" claims validated (local control, quality governance, transparency, resource efficiency)

Next Steps

1. ✅ **You are here**: Reviewing Project Roadmap
2. 📄 **Next**: Review PRD (Product Requirements Document) below
3. 🛠️ **Then**: Begin Phase 0 (Foundation & Infrastructure Setup)
4. 📅 **Schedule**: Kickoff meeting, assign roles, set up project tracking (Jira, GitHub Projects, etc.)

Project Tracking Recommendations

Tools

- **Project Management**: Jira, Linear, or GitHub Projects
- **Documentation**: Confluence, Notion, or Git-based wiki
- **Communication**: Slack or Teams
- **Version Control**: Git (GitHub, GitLab, or Bitbucket)

Cadence

- **Daily standups**: 15-minute sync (Phases 1-7)
- **Weekly planning**: Sprint planning, task breakdown
- **Bi-weekly demos**: Show progress to stakeholders
- **Monthly reviews**: Metrics review, roadmap adjustments

- **Retrospectives**: After each phase, learn and adjust

Roadmap maintained by: Technical Lead / Project Manager
Last updated: February 5, 2026
Next review: End of Phase 0 (Week 4)

Appendix: Phase Dependency Chart

```
Phase 0 (Foundation)
    ↓
Phase 1 (Router) ← Must complete Phase 0
    ↓
Phase 2 (Validation) ← Must complete Phase 1
    ↓
Phase 3 (Full Stack) ← Must complete Phase 2
    ↓
Phase 4 (Multimodal) ← Must complete Phase 3
    ↓
Phase 5 (Optimization) ← Must complete Phase 4
    ↓
Phase 6 (Ops & Deployment) ← Must complete Phase 5
    ↓
Phase 7 (Production) ← Must complete Phase 6

Note: Some phases can have parallel work streams (e.g., documentation in parallel
with implementation), but core technical dependencies must be respected.
```

Appendix: Estimated Effort by Phase

Phase	Duration	FTE	Total Person-Weeks
Phase 0: Foundation	3-4 weeks	1.5	4.5-6
Phase 1: Router	3-4 weeks	2.5	7.5-10
Phase 2: Validation	3-4 weeks	2.5	7.5-10
Phase 3: Full Stack	3-4 weeks	2.5	7.5-10
Phase 4: Multimodal	3-4 weeks	2.5	7.5-10
Phase 5: Optimization	3-4 weeks	2.5	7.5-10
Phase 6: Ops & Deployment	3-4 weeks	2.5	7.5-10
Phase 7: Production	4-6 weeks	3	12-18
Total	**25-34 weeks**		**62-84 person-weeks**

Interpretation: With a team of 3 full-time engineers, expect 6-8 months to production. With 5 engineers, 4-6 months.

End of Project Roadmap