Data Architecture & Memory Design Document

Sovereign AI Infrastructure: Transparent Memory Ledger System

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

Document Control

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

Reviewers:
- [ ] Technical Lead
- [ ] Backend Lead
- [ ] ML Lead
- [ ] Security Architect

Dependencies:
- System Architecture Document (SAD) v1.0
- Product Requirements Document (PRD) v1.0

Table of Contents

1. Executive Summary

1.1 Purpose

The **Transparent Memory Ledger** is the central nervous system of the Sovereign AI Infrastructure. Unlike traditional databases or neural memory systems, it uses **Markdown files** as a human-readable, auditable, version-controlled shared memory bus that all AI components read from and write to.

1.2 Design Philosophy

Key Principles:
1. **Transparency**: All memory is human-readable Markdown (not binary, not opaque database)
2. **Auditability**: Git version control tracks every change; immutable audit trail
3. **Simplicity**: Plain text files; no complex database schemas; grep-compatible
4. **Provenance**: Every fact includes source attribution (which model, when, why)
5. **Shared Context**: All models (Router, Workers, Validator) access same memory

1.3 Core Memory Components

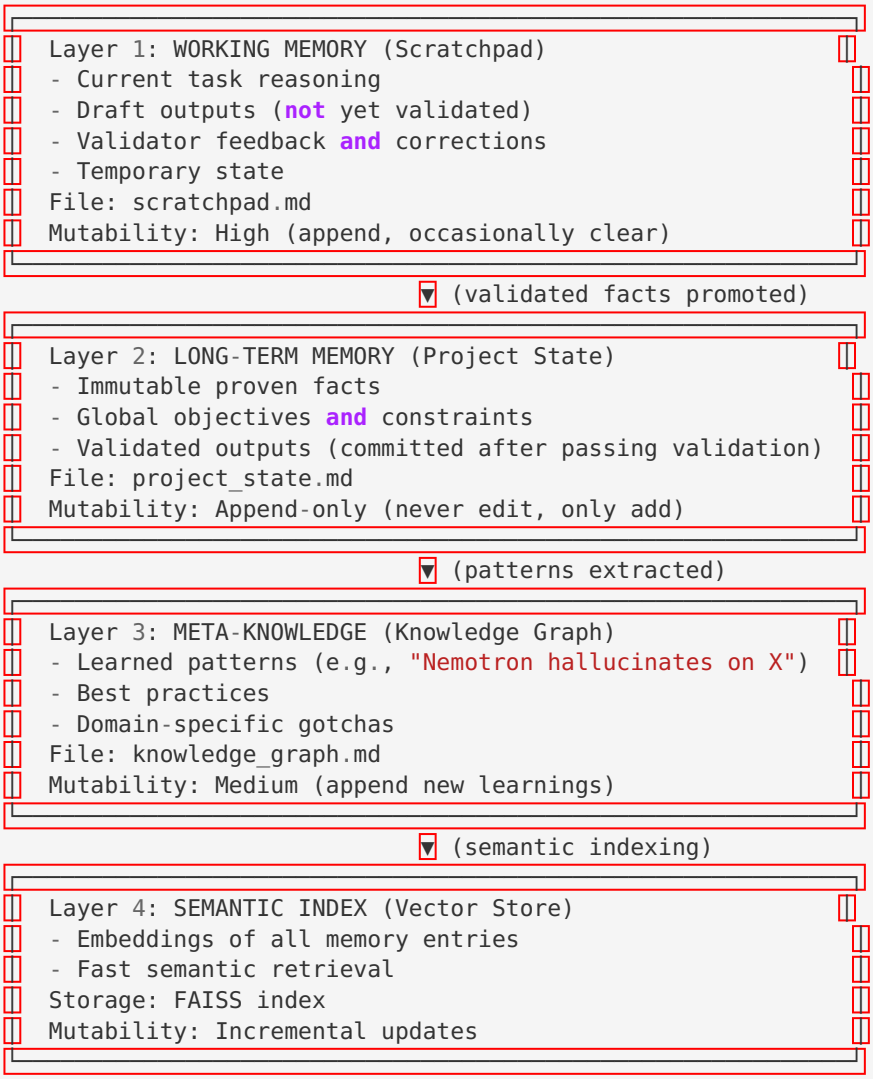
```
Memory System
├─ Markdown Ledger (Primary Memory)
│  ├─ project_state.md      (Immutable facts, proven constraints)
│  ├─ scratchpad.md          (Active reasoning, work in progress)
│  └─ knowledge_graph.md     (Learned patterns, meta-knowledge)
├─ Vector Store (Semantic Memory)
│  ├─ Memory embeddings      (Semantic search over Markdown)
│  └─ Document embeddings    (User-provided docs, past tasks)
└─ Provenance Database (Metadata)
   ├─ Source tracking         (OCR pages, image files, model versions)
   └─ Decision trails         (Routing/validation reasoning)
```

1.4 Key Architectural Decisions

Decision	Rationale	Trade-offs
Markdown for primary memory	Human-readable, Git-trackable, LLM-friendly	Not optimized for structured queries vs. SQL
Separate project_state (immutable) vs. scratchpad (mutable)	Clear boundary between proven facts and speculation	More complex state management
Git for versioning	Industry-standard, time-travel capability	Requires Git discipline (commits, etc.)
FAISS for vector store	Fast, CPU-efficient, local-only	Less feature-rich than ChromaDB/Pinecone
File-based (not database)	Simplicity, no database overhead	Manual concurrency control needed

2. Memory Architecture Overview

2.1 Three-Layer Memory Hierarchy



2.2 Memory Flow Diagram

```
graph TD
    Request[User Request] --> Router[Router Classifies]
    Router --> Retrieval[Retrieve Relevant Context<br/>from Memory]
    Retrieval --> Worker[Worker Generates<br/>writes to Scratchpad]
    Worker --> Validator[Validator Checks<br/>reads Scratchpad + Project State]

    Validator -->|PASS| Commit[Commit to Project State<br/>immutable]
    Validator -->|FAIL| Correction[Write Correction<br/>to Scratchpad]
    Correction --> Worker

    Commit --> Extract[Extract Meta-Learnings]
    Extract --> KnowledgeGraph[Update Knowledge Graph]

    Commit --> Embed[Re-embed New Facts]
    KnowledgeGraph --> Embed
    Embed --> VectorStore[Update Vector Store<br/>FAISS]

    style Request fill:#50E3C2,stroke:#333,stroke-width:2px
    style Commit fill:#7ED321,stroke:#333,stroke-width:2px,color:#000
    style VectorStore fill:#4A90E2,stroke:#333,stroke-width:2px,color:#fff
```

---

3. Markdown Memory Ledger

3.1 File Structure Overview

Primary Location: `/opt/sovereign-ai/data/memory/`

```
memory/
├── .git/                          # Git repository for version control
├── project_state.md               # Layer 2: Immutable proven facts
├── scratchpad.md                  # Layer 1: Current reasoning
├── knowledge_graph.md             # Layer 3: Meta-knowledge
├── sessions/                      # Historical sessions (archived)
│   ├── 2026-02-01_session.md
│   ├── 2026-02-02_session.md
│   └── ...
└── README.md                      # Memory system documentation
```

3.2 project_state.md Schema

Purpose: Immutable long-term memory of proven, validated facts.

Structure:

```
# Project State

**Last Updated**: 2026-02-05 14:30:00 UTC
**Session**: 42
**Git Commit**: abc1234

---

## Global Objectives

**Primary Goal**: [High-level objective, e.g., "Develop payment processing module"]

**Success Criteria**:
- [ ] Criterion 1
- [ ] Criterion 2

---

## Immutable Facts (Validated & Committed)

### Fact #1: [Timestamp: 2026-02-05 10:15:00]
**Actor**: Worker (Qwen Coder 32B)
**Validator**: Granite-H-Small
**Content**:
```python
def calculate_fibonacci(n: int) -> int:
    """Calculate nth Fibonacci number recursively."""
    if n <= 1:
        return n
    return calculate_fibonacci(n-1) + calculate_fibonacci(n-2)
```

Validation: [PASS] - Syntax ✓, Logic ✓, Type hints ✓
Source: User request "Create Fibonacci function"
Confidence: 0.95

Fact #2: [Timestamp: 2026-02-05 11:20:00]

Actor: Worker (GPT-OSS 20B)
Validator: Granite-H-Small
Content:
System architecture decision: Use PostgreSQL for transaction storage due to ACID compliance requirements.

Validation: [PASS] - Logical consistency ✓, No hallucinations ✓
Source: Analysis of payment processing requirements
Confidence: 0.89

Proven Constraints

Hardware Constraints

- GPU VRAM: 16GB (Tesla A2)
- System RAM: 128GB
```

- Max model size: 32B parameters (Q4 quantization)

Domain Constraints

- Language: Python 3.10+
- Framework: FastAPI for REST APIs
- Database: PostgreSQL 14+

Dependencies & Environment

Python Version: 3.10.12
Key Libraries:
- fastapi==0.100.0
- sqlalchemy==2.0.0
- psycopg2==2.9.0

Historical Decisions (Archived)

[Link to past decision logs if needed]

Note: This file is APPEND-ONLY. Never delete or edit existing facts. To correct, add a new fact superseding the old one.

```
**Key Properties**:
- **Immutability**: Append-only; existing entries never modified
- **Timestamped**: Every fact includes UTC timestamp
- **Attributed**: Every fact includes which Actor (Worker/Router/User) and which Validator
- **Validated**: Every fact includes validation verdict and checks performed
- **Searchable**: Plain text; grep-compatible

---

### 3.3 scratchpad.md Schema

**Purpose**: Mutable working memory for current task reasoning.

**Structure**:
```markdown
# Scratchpad (Working Memory)

**Current Task**: [User request summary]
**Started**: 2026-02-05 14:00:00 UTC
**Status**: In Progress

---

## Routing Decision

**Router Output**:
- Domain: `coding_architecture`
- Stakes: `high`
- Model: `qwen_coder_32b`
- Validation: `block_by_block`
- Confidence: 0.92
- Reasoning: "Complex multi-file refactoring; architectural focus indicated by keywords: design patterns, SOLID principles"

---

## Active Reasoning Stream

### Step 1: Architecture Design [PENDING]
**Actor**: Worker (Qwen Coder 32B)

**Context**:
- User request: "Refactor the payment module to follow SOLID principles"
- Current state: Monolithic PaymentService class (500 lines)
- Goal: Separate into Single Responsibility components

**Generated Output**:
```markdown
Proposed Architecture:
1. PaymentProcessor (core logic)
2. PaymentValidator (validation rules)
3. PaymentLogger (audit trail)
4. PaymentNotifier (external notifications)
```

Status: Awaiting validation
```
```

Validator Feedback on Step 1 [FAIL]

Validator: Granite-H-Small
Timestamp: 2026-02-05 14:05:00 UTC

Verdict: [FAIL]

Reasoning:
The proposed architecture is missing:
1. PaymentRepository (data access layer) - violates dependency inversion
2. Error handling strategy not specified
3. Interface definitions (abstractions) not provided

Correction Directive:
Add PaymentRepository for data access, define interfaces (IPaymentProcessor, etc.), and specify error handling approach (exceptions vs. result types).

Step 1: Architecture Design [RETRY #1]

Actor: Worker (Qwen Coder 32B)
Timestamp: 2026-02-05 14:07:00 UTC

Revised Output:

```
Proposed Architecture:
1. IPaymentProcessor (interface)
   - Concrete: StripePaymentProcessor, PayPalPaymentProcessor
2. IPaymentValidator (interface)
   - Concrete: SchemaValidator, BusinessRuleValidator
3. IPaymentRepository (interface)
   - Concrete: PostgresPaymentRepository
4. PaymentLogger (cross-cutting concern)
5. PaymentNotifier (event-driven notifications)

Error Handling: Use Result<T, Error> pattern (no exceptions for business logic)
```

Status: Awaiting validation

Validator Feedback on Step 1 Retry #1 [PASS]

Validator: Granite-H-Small
Timestamp: 2026-02-05 14:10:00 UTC

Verdict: [PASS]

Reasoning:
- Interfaces defined ✓
- Dependency inversion respected ✓
- Repository pattern included ✓
- Error handling specified ✓

Action: Committing architecture design to project_state.md

Temporary Notes (Scratchable)

- User mentioned they prefer explicit error handling over exceptions
- Current codebase uses pytest for testing; maintain compatibility
- Consider adding PaymentMetrics for observability (deferred to later step)

Scratchpad cleared after task completion or session end.

```
**Key Properties**:
- **Mutability**: High; frequently updated, cleared after task completion
- **Feedback Loop**: Contains validator corrections, retry attempts
- **Temporary**: Not intended for long-term storage (archived after session)
- **Append-heavy**: New steps appended sequentially

---

### 3.4 knowledge_graph.md Schema

**Purpose**: Meta-knowledge and learned patterns about models, domains, best prac-
tices.

**Structure**:
```markdown
# Knowledge Graph (Meta-Knowledge)

**Last Updated**: 2026-02-05 14:30:00 UTC

---

## Model-Specific Learnings

### Qwen Coder 32B
**Strengths**:
- Excellent at multi-file refactoring
- Strong architectural reasoning
- Accurate type inference

**Weaknesses**:
- Requires explicit type hints for validation to pass
- Occasionally over-engineers simple solutions

**Best Practices**:
- Provide full file context (not just snippets)
- Explicitly request "pragmatic" solutions if simplicity desired

**Last Updated**: 2026-02-05

---

### Nemotron 30B
**Strengths**:
- Fast, practical implementations
- Performance-oriented code

**Weaknesses**:
- Hallucinates Rust library APIs (low-confidence on Rust)
- Sometimes skips edge case handling

**Best Practices**:
- Use for Python, C++, Go implementations
- Avoid for Rust unless you verify API existence
- Explicitly request edge case handling

**Last Updated**: 2026-02-04

---

### GPT-OSS 20B
**Strengths**:
- Balanced reasoning
```
```

```
- Good at planning and decomposition

**Weaknesses**:
- Less specialized than coding models
- Verbose explanations (can be edited down)

**Best Practices**:
- Use for general reasoning, not code generation
- Good for architectural planning before implementation

**Last Updated**: 2026-02-03

---

### Granite-H-Small (Validator)
**Strengths**:
- Strict validation, low false negative rate
- Good at catching hallucinations

**Weaknesses**:
- Can be overly strict (false positives on creative solutions)
- Validation prompts need tuning for domain

**Best Practices**:
- For creative domains, set validation to "lenient" mode
- Provide clear validation criteria upfront

**Last Updated**: 2026-02-05

---

## Domain-Specific Patterns

### Python Coding
**Common Issues**:
- Forgetting type hints → validation fails
- Mixing sync/async without explicit markers

**Best Practices**:
- Always request type-hinted code
- Specify sync vs. async upfront

---

### Architecture Design
**Common Issues**:
- Missing interface definitions
- Not considering testability

**Best Practices**:
- Always define interfaces (abstractions)
- Request test strategy alongside architecture

---

## Validation Patterns

### High False Positive Scenarios
- Creative writing (validator too literal)
- Experimental code (validator expects production-ready)

**Mitigation**: Set validation policy to "lenient" or "end_stage" instead of "block_by
_block"
```

```
---

### High False Negative Scenarios
- Subtle logic errors in complex algorithms
- Hallucinated APIs in less-common languages

**Mitigation**: Multiple validation passes; cross-check with external docs

---

## User Preferences (Session-Specific)

**Current User**:
- Prefers explicit error handling (Result types) over exceptions
- Values simplicity over clever solutions
- Uses pytest for testing

---

**Note**: This file evolves over time as patterns emerge. Review quarterly.
```

Key Properties:
- **Meta-knowledge**: Not facts about the project, but about the system itself
- **Evolving**: Updated as patterns are discovered
- **Actionable**: Directly informs routing and validation decisions
- **Prunable**: Old/obsolete learnings can be archived

3.5 Markdown Parsing & Generation

Parser Requirements:
- Parse Markdown headings (H1-H6) for structure
- Extract code blocks (`language ...`)
- Extract lists, tables, links
- Preserve formatting

Python Implementation (using `markdown-it-py`):

```python
from markdown_it import MarkdownIt
from pathlib import Path

class MemoryParser:
    def __init__(self, memory_dir: Path):
        self.memory_dir = memory_dir
        self.md = MarkdownIt()

    def parse_project_state(self) -> Dict[str, Any]:
        """Parse project_state.md into structured data."""
        filepath = self.memory_dir / "project_state.md"
        content = filepath.read_text()

        # Parse Markdown
        tokens = self.md.parse(content)

        # Extract sections
        facts = self._extract_facts(tokens)
        constraints = self._extract_constraints(tokens)
        objectives = self._extract_objectives(tokens)

        return {
            "facts": facts,
            "constraints": constraints,
            "objectives": objectives,
            "raw_content": content
        }

    def _extract_facts(self, tokens) -> List[Dict]:
        """Extract immutable facts from parsed tokens."""
        facts = []
        # Implementation: Find sections starting with "### Fact #"
        # Extract timestamp, actor, validator, content
        return facts
```

---

4. Vector Store Architecture

4.1 Purpose & Design

Why Vector Store?
- **Semantic Retrieval**: Find relevant context even if keywords don't match
- **Scalability**: Memory ledger grows over time; text search alone becomes slow
- **Context Augmentation**: Provide Workers with relevant past facts without full memory dump

Technology Choice: FAISS (Facebook AI Similarity Search)
- CPU-efficient (no GPU needed)
- Local-only (no external dependencies)
- Fast approximate nearest neighbor search
- Lightweight (no database server required)

4.2 Vector Store Schema

Storage Location: `/opt/sovereign-ai/data/vector_store/`

```
vector_store/
├── memory_embeddings.index        # FAISS index for memory entries
├── memory_metadata.json           # Metadata (timestamps, sources, etc.)
├── document_embeddings.index      # FAISS index for user docs (OCR outputs, etc.)
└── document_metadata.json         # Metadata for documents
```

Embedding Model: `sentence-transformers/all-MiniLM-L6-v2`
- Dimension: 384
- Speed: Fast (CPU-efficient)
- Quality: Good for semantic search

4.3 Embedding Strategy

What Gets Embedded:

Source	Content to Embed	Update Frequency
project_state.md	Each fact (as individual chunks)	After each commit
knowledge_graph.md	Each model/domain section	Daily or after updates
scratchpad.md	NOT embedded (temporary, high churn)	N/A
User documents	OCR-extracted text (para-graphs)	On document ingestion

Chunking Strategy:
- **Chunk size**: 200-300 words (~3-4 sentences)
- **Overlap**: 50 words between consecutive chunks (for context continuity)
- **Metadata**: Each chunk tagged with source (file, section, timestamp)

Example Embedding Entry:

```
{
    "id": "fact_42_20260205_1015",
    "embedding": [0.123, -0.456, 0.789, ...],  # 384-dim vector
    "metadata": {
        "source_file": "project_state.md",
        "section": "Fact #42",
        "timestamp": "2026-02-05T10:15:00Z",
        "actor": "qwen_coder_32b",
        "content_preview": "def calculate_fibonacci(n: int) -> int: ..."
    }
}
```

4.4 Retrieval Process

Query Flow:

```
User Query → Embed Query → FAISS Search (top-k) → Fetch Metadata → Return Contexts
```

Python Implementation:

```
import faiss
import numpy as np
from sentence_transformers import SentenceTransformer

class VectorMemory:
    def __init__(self, index_path: str):
        self.index = faiss.read_index(index_path)
        self.embedder = SentenceTransformer('all-MiniLM-L6-v2')
        self.metadata = self._load_metadata()

    def retrieve_relevant(self, query: str, top_k: int = 5) -> List[Dict]:
        """Retrieve top-k relevant memory entries."""
        # Embed query
        query_vector = self.embedder.encode([query])[0]

        # Search FAISS index
        distances, indices = self.index.search(
            np.array([query_vector], dtype=np.float32),
            top_k
        )

        # Fetch metadata
        results = []
        for idx, dist in zip(indices[0], distances[0]):
            results.append({
                "content": self.metadata[idx]["content_preview"],
                "source": self.metadata[idx]["source_file"],
                "timestamp": self.metadata[idx]["timestamp"],
                "relevance_score": 1.0 / (1.0 + dist)  # Convert distance to similar-
ity
            })

        return results
```

Performance Target:
- Retrieval latency: <100ms for top-10 results
- Index size: ~10MB per 10,000 embeddings

---

5. Provenance & Audit Trail

5.1 Provenance Tracking

Definition: Provenance = “where did this data come from?”

Critical for:
- **Compliance**: Regulators need to trace AI decisions back to sources
- **Debugging**: Understand why output was generated
- **Grounding**: Ensure claims are backed by evidence (OCR text, images)

Provenance Metadata Structure:

```python
@dataclass
class ProvenanceMetadata:
    """Tracks origin of a fact or output."""

    # Who created this?
    actor: str                  # "qwen_coder_32b", "user", "ocr_pipeline"
    actor_version: str          # Model version or tool version

    # When?
    timestamp: datetime         # UTC

    # Why/How?
    trigger: str                # "user_request", "validation_correction",
"auto_learning"
    parent_request_id: str      # Link back to originating user request

    # Source materials
    source_documents: List[str] # ["scanned_doc_page_3.png", "user_prompt.txt"]
    source_citations: List[str] # ["OCR line 42-45", "Figure 2 caption"]

    # Validation trail
    validator: Optional[str]    # "granite_h_small" or None
    validation_verdict: str     # "PASS", "FAIL", "NOT_VALIDATED"

    # Confidence
    confidence_score: float     # 0.0-1.0
    uncertainty_notes: str      # E.g., "OCR unclear on this section"
```

5.2 Audit Trail in project_state.md

Every fact in `project_state.md` includes inline provenance:

```
### Fact #42: [Timestamp: 2026-02-05 10:15:00]
**Actor**: Worker (Qwen Coder 32B v1.5.0)
**Validator**: Granite-H-Small v2.0.1
**Source**: User request "Create Fibonacci function"
**Parent Request**: req_1234567890
**Validation**: [PASS] - Syntax ✓, Logic ✓, Type hints ✓
**Confidence**: 0.95
**Provenance**:
- Trigger: User request
- No source documents (generated from scratch)
- Validation checks: syntax_check (PASS), logic_check (PASS), hallucination_check
(PASS)

**Content**:
[... actual fact content ...]
```

5.3 Multimodal Provenance

For OCR-extracted text:

```
### Fact #55: Patient Diagnosis
**Actor**: OCR Pipeline (Tesseract 5.0)
**Source Document**: medical_record_page_3.png
**OCR Confidence**: 0.94
**Extracted Text**:
"Patient diagnosed with Type 2 Diabetes on 2025-08-15."

**Provenance**:
- Source: medical_record_page_3.png, bounding box (120, 450, 680, 490)
- OCR engine: Tesseract 5.0.3
- Preprocessing: Grayscale conversion, noise reduction
- Uncertain regions: Date ("2025-08-15" confidence 0.89, possible "2025-08-18")

**Grounding**: All subsequent claims about this diagnosis MUST cite this fact.
```

For Vision-encoded images:

```
### Fact #67: Chart Analysis
**Actor**: Vision Encoder (CLIP ViT-B/32)
**Source Document**: sales_chart_q4.png
**Generated Caption**:
"Bar chart showing quarterly sales: Q1 $2.3M, Q2 $2.8M, Q3 $3.1M, Q4 $3.5M"

**Provenance**:
- Source: sales_chart_q4.png (sha256: abc123...)
- Vision model: CLIP ViT-B/32
- Confidence: Q1-Q3 values (0.92), Q4 value (0.78 - partially occluded)
- Objects detected: 4 bars, x-axis labels, y-axis scale

**Grounding**: Any claims about Q4 sales must acknowledge lower confidence (0.78).
```

5.4 Audit Trail Export

Format: PDF or HTML report

Contents:
- Complete timeline (routing → generation → validation → commit)
- All provenance metadata
- Source documents (embedded images, OCR text)
- Decision reasoning (why this model, why this validation policy)

Use Case: Compliance audits, debugging, user transparency

---

6. Concurrency & Consistency

6.1 Concurrency Challenges

Problem: Multiple processes/threads accessing Markdown files simultaneously:
- **Reader-Writer conflict**: Validator reading scratchpad while Worker writing
- **Write-Write conflict**: Two writers attempting simultaneous append (rare in v1.0 single-task execution)

6.2 Locking Strategy

File-Level Locking (using Python `filelock`):

```python
from filelock import FileLock
from pathlib import Path

class MemoryManager:
    def __init__(self, memory_dir: Path):
        self.memory_dir = memory_dir
        self.locks = {
            "project_state": FileLock(memory_dir / "project_state.lock"),
            "scratchpad": FileLock(memory_dir / "scratchpad.lock"),
            "knowledge_graph": FileLock(memory_dir / "knowledge_graph.lock"),
        }

    def append_to_scratchpad(self, entry: str):
        """Atomic append to scratchpad with locking."""
        with self.locks["scratchpad"]:
            filepath = self.memory_dir / "scratchpad.md"
            with open(filepath, 'a', encoding='utf-8') as f:
                f.write(f"\n{entry}\n")
                f.flush()  # Ensure write to disk

    def commit_to_project_state(self, fact: str):
        """Atomic commit to project_state with locking."""
        with self.locks["project_state"]:
            filepath = self.memory_dir / "project_state.md"
            with open(filepath, 'a', encoding='utf-8') as f:
                f.write(f"\n---\n\n{fact}\n")
                f.flush()
```

Locking Rules:
- **Shared reads**: Multiple readers allowed (no lock needed)
- **Exclusive writes**: Only one writer at a time (acquire lock)
- **Timeout**: 10 seconds (if lock not acquired, fail gracefully)

6.3 Consistency Guarantees

Atomicity:
- Each append operation is atomic (file lock ensures no partial writes)
- Git commits provide transaction-like behavior (all-or-nothing)

Durability:
- All writes flushed to disk immediately (`f.flush()`)
- Git commits create permanent snapshots

Consistency:
- Schema validation on write (ensure Markdown structure preserved)
- No orphaned references (e.g., scratchpad referencing non-existent fact)

Isolation (weak in v1.0):
- Single-task execution means minimal concurrency
- Future: Implement MVCC (Multi-Version Concurrency Control) if multi-task support added

6.4 Conflict Resolution

Scenario: Validator and Worker both try to write to scratchpad simultaneously (rare)

Resolution:
1. First writer acquires lock, completes write

2. Second writer blocks until lock released
3. Second writer then appends (sequential, not lost)

Future Enhancement: Operational Transform (OT) or CRDTs for true concurrent editing

---

7. Backup & Recovery

7.1 Backup Strategy

Frequency:
- **Continuous**: Git commits after every task completion
- **Daily**: Full snapshot of memory directory
- **Weekly**: Full snapshot of model vault + memory (off-site if possible)

Backup Script (example):

```
#!/bin/bash
# daily_backup.sh

DATE=$(date +%Y-%m-%d)
MEMORY_DIR="/opt/sovereign-ai/data/memory"
BACKUP_DIR="/mnt/backups/memory"

# Create backup directory
mkdir -p "$BACKUP_DIR/$DATE"

# Copy memory ledger
cp -r "$MEMORY_DIR" "$BACKUP_DIR/$DATE/"

# Git bundle (complete history)
cd "$MEMORY_DIR"
git bundle create "$BACKUP_DIR/$DATE/memory.bundle" --all

# Compress
cd "$BACKUP_DIR"
tar -czf "$DATE.tar.gz" "$DATE/"
rm -rf "$DATE/"  # Remove uncompressed

# Cleanup old backups (keep 30 days)
find "$BACKUP_DIR" -name "*.tar.gz" -mtime +30 -delete

echo "Backup completed: $BACKUP_DIR/$DATE.tar.gz"
```

Backup Verification:
- **Checksum**: SHA256 hash of backup archives
- **Test Restore**: Monthly test restore to verify backup integrity

7.2 Disaster Recovery

Failure Scenarios:

Scenario	Impact	Recovery Procedure	RTO	RPO
Memory file corruption	Data loss, system unusable	Restore from Git history or daily backup	5 min	Last commit (~1 task)
Disk failure (NVMe)	Complete data loss	Restore from weekly backup	1 hour	Up to 7 days of data
Accidental deletion	Specific facts lost	Git revert or restore from backup	2 min	0 (Git history)
Git repository corruption	Version control broken	Re-clone from backup bundle	10 min	Last daily backup

Recovery Process (Memory File Corruption):

```
# 1. Detect corruption (checksum mismatch)
# 2. Stop orchestrator
systemctl stop sovereign-orchestrator

# 3. Move corrupted file
mv /opt/sovereign-ai/data/memory/project_state.md /tmp/corrupted_project_state.md

# 4. Restore from Git
cd /opt/sovereign-ai/data/memory
git checkout HEAD -- project_state.md

# 5. Verify integrity
sha256sum project_state.md  # Compare with known good checksum

# 6. Restart orchestrator
systemctl start sovereign-orchestrator
```

7.3 Point-in-Time Recovery (Git)

Use Case: Rollback to state from 3 days ago

```
cd /opt/sovereign-ai/data/memory

# 1. Find commit from 3 days ago
git log --since="3 days ago" --until="2 days ago" --oneline

# 2. Create new branch from that commit (non-destructive)
git checkout -b recovery_branch abc1234

# 3. Inspect state, verify
cat project_state.md

# 4. If correct, merge or replace main branch
git checkout main
git reset --hard abc1234

# 5. Resume operations
```

---

8. Access Patterns & Performance

8.1 Common Access Patterns

Operation	Frequency	Latency Target	Implementation
Read project_state	Every validation (high)	<50ms	In-memory caching (TTL 60s)
Append to scratch-pad	Every Worker generation	<20ms	Direct file append (buffered)
Commit to project_state	Every validation PASS	<30ms	File append + Git commit (async)
Semantic retrieval	Every task start	<100ms	FAISS search
Full memory scan	Rare (manual debugging)	<1s	Full file read + parse

8.2 Caching Strategy

In-Memory Cache (Python `functools.lru_cache`):

```python
import functools
import time
from typing import Dict, Any

class MemoryManager:
    def __init__(self):
        self._cache_timestamp = {}
        self._cache_ttl = 60  # seconds

    @functools.lru_cache(maxsize=1)
    def read_project_state_cached(self) -> Dict[str, Any]:
        """Cached read of project_state with TTL."""
        cache_key = "project_state"
        now = time.time()

        # Invalidate cache if TTL expired
        if cache_key in self._cache_timestamp:
            if now - self._cache_timestamp[cache_key] > self._cache_ttl:
                self.read_project_state_cached.cache_clear()

        # Update timestamp
        self._cache_timestamp[cache_key] = now

        # Read (will be cached by lru_cache)
        return self._parse_project_state()
```

Cache Invalidation:
- **Time-based**: TTL of 60 seconds
- **Event-based**: Clear cache after commit to project_state
- **Manual**: CLI command `sovereign cache clear`

8.3 Performance Optimization

Optimization 1: Incremental Git Commits
- Problem: Git commit after every fact is slow (100-200ms)
- Solution: Batch commits every 5 facts or every 5 minutes

Optimization 2: Lazy Parsing
- Problem: Parsing entire project_state.md is slow as it grows
- Solution: Parse only relevant sections (e.g., last N facts, specific section)

Optimization 3: Scratchpad Pruning
- Problem: Scratchpad grows unbounded during long tasks
- Solution: Prune validated/committed blocks from scratchpad (move to project_state)

Optimization 4: FAISS Index Updates
- Problem: Rebuilding entire FAISS index after every commit is slow
- Solution: Incremental index updates (add new embeddings without rebuild)

8.4 Scalability Projections

Memory Ledger Growth:
- Assumption: 100 tasks/day, 10KB per task
- Growth: 1MB/day, ~365MB/year
- Manageable for years without performance degradation

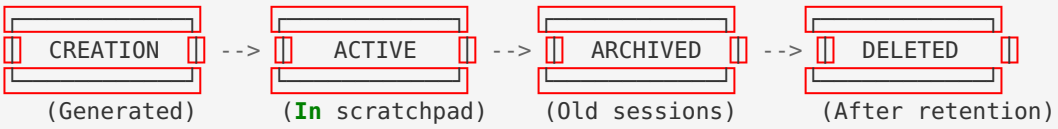
Vector Store Growth:
- Assumption: 100 embeddings/day, 384 dims × 4 bytes = 1.5KB per embedding
- Growth: 150KB/day, ~54MB/year (index size)
- FAISS efficient for millions of vectors

Git Repository Size:
- Text-heavy (Markdown), compresses well
- Expected: <100MB per year (with history)

---

9. Data Lifecycle Management

9.1 Lifecycle Stages



9.2 Data Retention Policies

Data Type	Retention Period	Archive After	Delete After	Rationale
project_state.md	Indefinite	Never	Never	Immutable facts; compliance
scratchpad.md	Session	End of session	After archiving	Temporary working memory
knowledge_graph.md	Indefinite	Annually (old patterns)	Manually (if obsolete)	Meta-knowledge evolves slowly
Logs	30 days active	30 days	90 days	Compliance, debugging
Backups	30 days (daily), 1 year (weekly)	Immediate	After retention	Disaster recovery

9.3 Archiving Process

When: End of session, daily, or manually

Process:

```python
def archive_session():
    """Archive current session scratchpad to sessions/ directory."""
    from datetime import datetime

    date_str = datetime.utcnow().strftime("%Y-%m-%d")
    session_file = f"sessions/{date_str}_session.md"

    # Copy scratchpad to archive
    shutil.copy("scratchpad.md", session_file)

    # Clear scratchpad (or reset to template)
    with open("scratchpad.md", 'w') as f:
        f.write("# Scratchpad (Working Memory)\n\n[Empty - New Session]\n")

    # Git commit
    subprocess.run(["git", "add", session_file, "scratchpad.md"])
    subprocess.run(["git", "commit", "-m", f"Archive session {date_str}"])

    print(f"Session archived: {session_file}")
```

9.4 Data Deletion (GDPR Compliance)

Use Case: User requests data deletion (GDPR “right to be forgotten”)

Challenge: project_state.md is immutable; how to delete?

Solution:
1. **Redaction**: Replace sensitive content with `[REDACTED per GDPR request]`
2. **Git History Rewrite**: Use `git filter-branch` or `BFG Repo-Cleaner` to remove from history
3. **New Commit**: Commit redacted version
4. **Backup Purge**: Delete old backups containing sensitive data

Example:

```
# Redact specific fact from project_state.md
sed -i 's/John Doe/[REDACTED]/g' project_state.md

# Rewrite Git history to remove all instances
git filter-branch --tree-filter 'sed -i "s/John Doe/[REDACTED]/g" project_state.md' HE
AD

# Force push (if remote)
git push --force
```

Note: This is complex and should be rare. Design data collection to minimize PII.

---

10. Implementation Specifications

10.1 Python MemoryManager Class

Interface:

```python
from pathlib import Path
from typing import Dict, List, Any
from dataclasses import dataclass
from datetime import datetime

@dataclass
class MemoryEntry:
    timestamp: datetime
    actor: str
    entry_type: str  # "fact", "reasoning", "correction", "learning"
    content: str
    metadata: Dict[str, Any]

class MemoryManager:
    """
    Central manager for Markdown memory ledger and vector store.
    """

    def __init__(self, memory_dir: Path):
        self.memory_dir = memory_dir
        self.project_state_path = memory_dir / "project_state.md"
        self.scratchpad_path = memory_dir / "scratchpad.md"
        self.knowledge_graph_path = memory_dir / "knowledge_graph.md"
        self.vector_store = VectorMemory(memory_dir / "../vector_store")

    # --- READ OPERATIONS ---

    def read_project_state(self) -> Dict[str, Any]:
        """Read and parse project_state.md"""
        pass

    def read_scratchpad(self) -> str:
        """Read current scratchpad"""
        pass

    def read_knowledge_graph(self) -> Dict[str, Any]:
        """Read and parse knowledge_graph.md"""
        pass

    def get_relevant_context(self, query: str, top_k: int = 5) -> List[str]:
        """Semantic retrieval from vector store"""
        return self.vector_store.retrieve_relevant(query, top_k)

    # --- WRITE OPERATIONS ---

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

    def commit_to_project_state(self, fact: MemoryEntry) -> bool:
        """Commit validated fact to project_state (immutable)"""
        pass

    def update_knowledge_graph(self, learning: str) -> bool:
        """Add new learning to knowledge_graph"""
        pass

    # --- UTILITY OPERATIONS ---

    def clear_scratchpad(self):
        """Clear scratchpad (start new session)"""
        pass
```

```
    def archive_session(self, session_name: str):
        """Archive current scratchpad to sessions/"""
        pass

    def git_commit(self, message: str):
        """Commit current state to Git"""
        pass

    def verify_integrity(self) -> bool:
        """Check for corruption, validate schemas"""
        pass
```

10.2 Configuration

YAML Configuration (`config/memory.yaml`):

```yaml
memory:
  base_dir: "/opt/sovereign-ai/data/memory"

  files:
    project_state: "project_state.md"
    scratchpad: "scratchpad.md"
    knowledge_graph: "knowledge_graph.md"

  cache:
    enabled: true
    ttl_seconds: 60

  git:
    auto_commit: true
    commit_batch_size: 5  # Commit after every 5 facts
    commit_interval_minutes: 5  # Or every 5 minutes

  backup:
    enabled: true
    daily_backup_time: "02:00"  # 2 AM
    retention_days: 30
    backup_dir: "/mnt/backups/memory"

  locking:
    timeout_seconds: 10

vector_store:
  base_dir: "/opt/sovereign-ai/data/vector_store"
  embedding_model: "sentence-transformers/all-MiniLM-L6-v2"
  index_type: "FAISS_FLAT"  # or FAISS_IVF for larger datasets

  chunking:
    chunk_size_words: 250
    overlap_words: 50

  retrieval:
    default_top_k: 5
    relevance_threshold: 0.7  # Min similarity score
```

10.3 Testing Strategy

Unit Tests:

- `test_memory_manager.py` : Test read/write operations, locking, parsing

- `test_vector_store.py`: Test embedding, retrieval, index updates
- `test_provenance.py`: Test metadata tracking, audit trail generation

Integration Tests:
- `test_end_to_end_memory.py`: Full flow: generate → validate → commit → retrieve

Performance Tests:
- `test_memory_performance.py`: Measure read/write latencies, cache effectiveness

Corruption Tests:
- `test_recovery.py`: Simulate file corruption, test recovery from Git/backup

---

11. Appendices

11.1 Markdown Formatting Best Practices

Consistent Structure:
- Use H1 (`#`) for file title
- Use H2 (`##`) for major sections
- Use H3 (`###`) for sub-sections (e.g., individual facts)
- Use H4+ for nested content

Code Blocks:
- Always specify language: ```` ```python ````, not ```` ``` ````
- Use triple backticks, not indentation

Lists:
- Use `-` for unordered lists
- Use `1.`, `2.` for ordered lists

Tables:
- Use Markdown tables for structured data
- Always include header row

Links:
- Use `[text](url)` format
- Internal links: `[See Fact #42](#fact-42)`

11.2 Schema Evolution

Versioning:
- Add schema version to top of each memory file
- Example: `<!-- Schema Version: 1.0 -->`

Backward Compatibility:
- New fields are optional (don’t break old parsers)
- Deprecation period: 6 months before removing old fields

Migration Process:
1. Add new schema version to parser
2. Support both old and new (dual-read)
3. Background migration (convert old → new)
4. After 6 months, drop old schema support

11.3 Debugging & Troubleshooting

Common Issues:

Issue	Symptom	Diagnosis	Fix
Slow reads	High latency on project_state reads	File too large (>10MB)	Prune old facts, move to archive
Lock timeout	Write operations fail	Multiple writers, lock held too long	Reduce lock hold time, check for deadlocks
Git merge conflicts	Git commit fails	Concurrent commits (shouldn't happen in v1.0)	Manual merge, enforce sequential execution
Vector search poor quality	Irrelevant results	Embedding model mismatch, outdated index	Re-index with correct model
Corruption detected	Checksum mismatch	Disk error, partial write	Restore from Git or backup

Diagnostic Commands:

```
# Check memory file integrity
sovereign memory verify

# View recent memory commits
sovereign memory log --last 10

# Force cache clear
sovereign memory cache clear

# Manual Git status
cd /opt/sovereign-ai/data/memory && git status

# Re-index vector store
sovereign memory reindex --force
```

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

End of Data Architecture & Memory Design Document