🧠 Memory & Learning System Documentation

Table of Contents

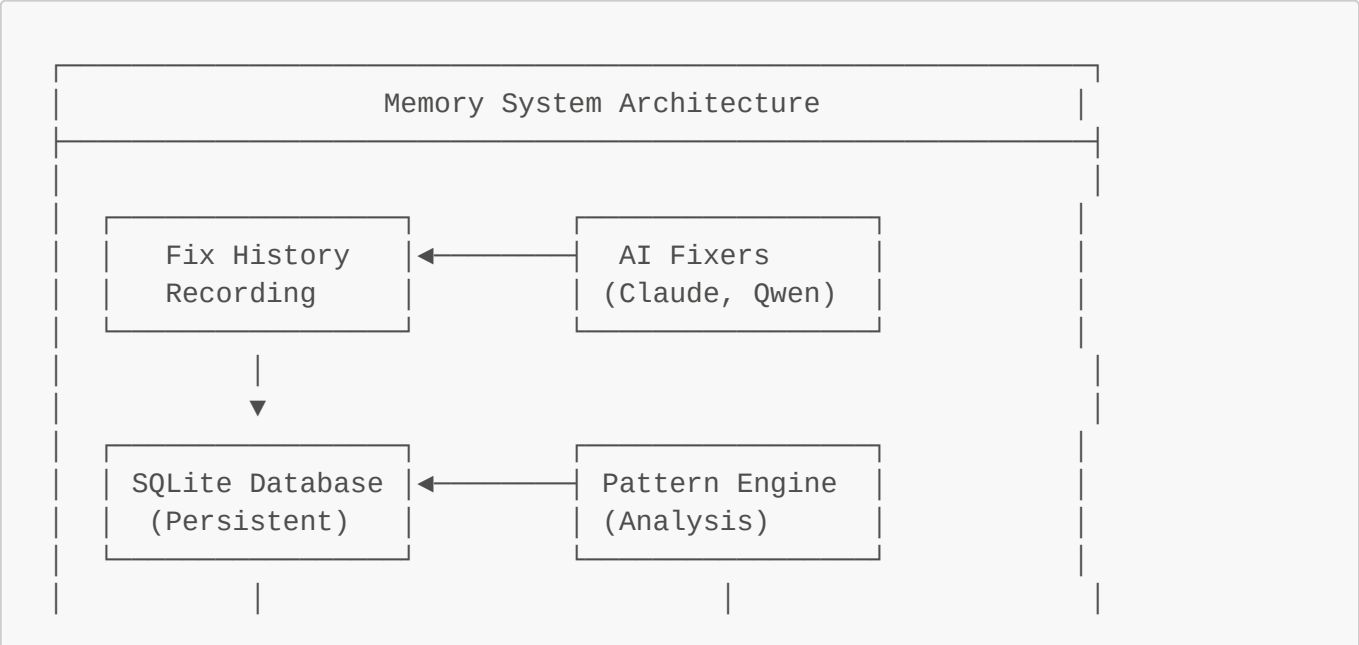
Overview

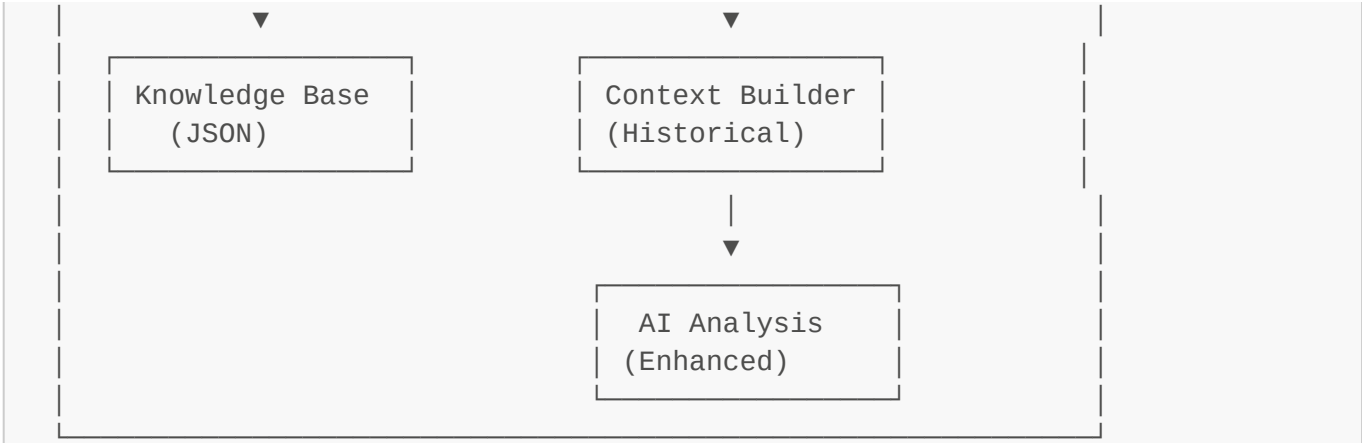
The Memory & Learning System is the brain of the AI Auto-Fix framework, providing persistent storage and intelligent learning capabilities that improve fix success rates over time. It stores every fix attempt, learns from patterns, and provides historical context to AI fixers.

Key Features

- 🗄️ **Persistent Storage**: SQLite database for fix history and patterns
- 🧠 **Pattern Recognition**: Automatic identification of recurring issues
- 📈 **Success Tracking**: Monitor improvement over time
- 🔍 **Historical Context**: Provide AI fixers with relevant past experiences
- 📊 **Analytics**: Detailed statistics and insights
- 🏗️ **Knowledge Base**: Accumulated wisdom from all fix attempts

Architecture





Database Schema

Tables Overview

1. fix_history

Primary table storing every fix attempt with complete context.

```
CREATE TABLE fix_history (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp TEXT NOT NULL,               -- ISO format timestamp
    model_name TEXT NOT NULL,              -- Target model being fixed
    issue_type TEXT NOT NULL,              -- Category of issue
    issue_signature TEXT NOT NULL,        -- Unique hash for issue type
    description TEXT NOT NULL,             -- Human-readable description
    fixer_type TEXT NOT NULL,             -- AI provider used (claude, qwen)
    claude_analysis TEXT NOT NULL,        -- AI analysis (naming kept for
compatibility)
    fix_commands TEXT NOT NULL,           -- JSON array of bash commands
    fix_success BOOLEAN NOT NULL,         -- Whether fix was applied
successfully
    verification_success BOOLEAN NOT NULL, -- Whether fix was verified
    execution_time_seconds REAL,          -- Time taken for complete fix
    system_info TEXT NOT NULL,            -- JSON system context
    notes TEXT                            -- Additional notes
);
```

2. issue_patterns

Aggregated pattern recognition for common issues.

```
CREATE TABLE issue_patterns (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    pattern_name TEXT UNIQUE NOT NULL,    -- Human-readable pattern name
    issue_signature TEXT NOT NULL,       -- Associated signature hash
```

```
    success_count INTEGER DEFAULT 0,     -- Number of successful fixes
    failure_count INTEGER DEFAULT 0,     -- Number of failed attempts
    best_fix TEXT,                       -- JSON of most successful fix
    last_seen TEXT,                      -- Last occurrence timestamp
    confidence_score REAL DEFAULT 0.0    -- Pattern confidence (0.0-1.0)
);
```

3. model_characteristics

Model-specific learned behaviors and patterns.

```
CREATE TABLE model_characteristics (
    model_name TEXT PRIMARY KEY,         -- Unique model identifier
    total_tests INTEGER DEFAULT 0,       -- Total number of tests run
    success_rate REAL DEFAULT 0.0,       -- Overall success rate (0.0-1.0)
    common_issues TEXT,                  -- JSON array of frequent issues
    effective_fixes TEXT,                -- JSON array of working solutions
    last_updated TEXT,                   -- Last update timestamp
    performance_notes TEXT               -- Free-form notes about model
);
```

Data Types and Constraints

Issue Types

Standard categories for classification:

- `MODEL_NOT_AVAILABLE` - Model needs installation
- `TIMEOUT` - Response took too long
- `UNEXPECTED_RESPONSE` - Output doesn't match expected pattern
- `NO_OUTPUT` - Model produced no response
- `ERROR` - System or runtime error
- `CUSTOM` - User-defined issue types

Fixer Types

AI providers currently supported:

- `claude` - Anthropic Claude 3.5 Sonnet
- `qwen` - Local Qwen 2.5 Coder via Ollama
- Future: `gpt`, `gemini`, `custom`

JSON Fields Format

fix_commands:

```
["ollama pull model-name", "systemctl restart ollama"]
```

system_info:

```
{
    "os": "Linux",
    "ram_gb": 16,
    "gpu_available": true,
    "ollama_version": "0.1.17"
}
```

common_issues:

```
["TIMEOUT", "MODEL_NOT_AVAILABLE", "UNEXPECTED_RESPONSE"]
```

Learning Mechanisms

1. Issue Signature Generation

Each issue gets a unique signature based on:

- Issue type category
- Model family (e.g., "qwen2.5", "claude")
- Prompt characteristics (first 50 characters)
- Response length category

```
def generate_issue_signature(issue_data):
    signature_parts = [
        issue_data.get('issue_type', ''),
        issue_data.get('model', '').split(':')[0],
        issue_data.get('test_prompt', '')[:50],
        str(len(issue_data.get('actual_response', '')))
    ]
    signature_text = "|".join(signature_parts)
    return hashlib.md5(signature_text.encode()).hexdigest()[:16]
```

2. Pattern Recognition

The system automatically identifies:

- **Exact Matches**: Identical issue signatures
- **Model Patterns**: Same model + issue type combinations
- **General Patterns**: Issue type across all models
- **Temporal Patterns**: Issues that appear at specific times

3. Success Rate Calculation

For each pattern, the system tracks:

```
success_rate = successful_fixes / total_attempts
confidence = min(1.0, total_attempts / 10.0)  # More data = higher
confidence
```

4. Historical Context Building

When an AI fixer analyzes a new issue, it receives:

```
context = build_context_for_claude(issue_data)
# Returns structured text with:
# - Model history and characteristics
# - Similar issue solutions
# - Success/failure patterns
# - Effective strategies
```

Memory Operations

Core Operations

1. Recording Fix Attempts

```
fix_id = memory.record_fix_attempt(
    issue_data={
        "model": "qwen2.5:7b",
        "issue_type": "TIMEOUT",
        "description": "Model response timeout after 30 seconds"
    },
    ai_response={
        "analysis": "Model needs more time due to complex prompt",
        "fix_commands": ["# Increase timeout to 60 seconds"],
        "confidence": 0.85
    },
    fix_success=True,
    verification_success=True,
    execution_time=45.2,
    notes="Timeout increased successfully",
    fixer_type="qwen"
)
```

2. Querying Similar Issues

```
similar_issues = memory.query_similar_issues(issue_data, limit=5)
# Returns:
```

```
# {
#     'exact_matches': [...],      # Same signature
#     'model_matches': [...],      # Same model + issue type
#     'type_matches': [...]        # Same issue type, any model
# }
```

3. Model Insights

```
insights = memory.get_model_insights("qwen2.5:7b")
# Returns comprehensive model performance data
```

Advanced Queries

1. Success Rate Analysis

```
SELECT
    fixer_type,
    COUNT(*) as total_attempts,
    SUM(CASE WHEN fix_success = 1 AND verification_success = 1 THEN 1 ELSE
0 END) as successes,
    ROUND(
        100.0 * SUM(CASE WHEN fix_success = 1 AND verification_success = 1
THEN 1 ELSE 0 END) / COUNT(*),
        2
    ) as success_rate
FROM fix_history
GROUP BY fixer_type;
```

2. Model Performance Trends

```
SELECT
    model_name,
    DATE(timestamp) as date,
    COUNT(*) as daily_attempts,
    AVG(CASE WHEN fix_success = 1 THEN 1.0 ELSE 0.0 END) as
daily_success_rate
FROM fix_history
WHERE timestamp > datetime('now', '-30 days')
GROUP BY model_name, DATE(timestamp)
ORDER BY date DESC;
```

3. Issue Type Distribution

```
SELECT
    issue_type,
    COUNT(*) as occurrences,
    AVG(execution_time_seconds) as avg_fix_time,
    ROUND(100.0 * SUM(fix_success) / COUNT(*), 2) as success_rate
FROM fix_history
GROUP BY issue_type
ORDER BY occurrences DESC;
```

CLI Interface

Command Overview

```
./memory.sh <command> [options]
```

Available Commands

1. Statistics

```
./memory.sh stats
```

Output:

```
🧠 Claude Memory Statistics:
  total_fixes_attempted: 47
  successful_fixes: 41
  success_rate: 0.872
  models_encountered: 8
  issue_types_seen: 5
  recent_activity_7days: 12
  knowledge_base_size: 15847
  memory_directory: /path/to/AIMemory
```

2. Model-Specific Insights

```
./memory.sh model qwen2.5:7b
```

Output:

```
{
  "model_name": "qwen2.5:7b",
```

```
  "total_tests": 15,
  "success_rate": 0.93,
  "common_issues": ["TIMEOUT", "UNEXPECTED_RESPONSE"],
  "effective_fixes": ["Increase timeout", "Adjust prompt format"],
  "recent_history": [...]
}
```

3. Export Complete Report

```
./memory.sh export insights_2025_01_15.json
```

Generates comprehensive report with:

- All statistics
- Model insights
- Knowledge base
- Historical trends

4. Recent Activity

```
./memory.sh recent 7
```

Shows last 7 days of fix attempts with status and outcomes.

5. Successful Strategies

```
./memory.sh successful
```

Lists most effective fix strategies by issue type.

6. Pattern Analysis

```
./memory.sh patterns
```

Identifies recurring issues and their success rates.

7. Cleanup

```
./memory.sh cleanup 30
```

Removes records older than 30 days (optional maintenance).

Data Export & Import

Export Formats

1. JSON Export (Default)

```
./memory.sh export full_report.json
```

Complete system state in structured JSON format.

2. CSV Export (Custom)

```
# Custom script for CSV export
import sqlite3
import csv

conn = sqlite3.connect('AIMemory/ai_memory.db')
cursor = conn.cursor()

cursor.execute("SELECT * FROM fix_history")
with open('fix_history.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow([description[0] for description in cursor.description])
    writer.writerows(cursor.fetchall())
```

3. Analytics Export

```
# Generate analytics-ready data
def export_analytics_data(output_file):
    memory = AIMemory()
    analytics_data = {
        'daily_stats': memory.get_daily_statistics(),
        'model_performance': memory.get_model_performance_matrix(),
        'issue_trends': memory.get_issue_trend_analysis(),
        'fixer_comparison': memory.get_fixer_effectiveness()
    }

    with open(output_file, 'w') as f:
        json.dump(analytics_data, f, indent=2)
```

Import Capabilities

1. Knowledge Base Import

```
# Import external knowledge
def import_knowledge_base(source_file):
    with open(source_file, 'r') as f:
        external_knowledge = json.load(f)

    memory = AIMemory()
    memory.knowledge_base.update(external_knowledge)
    memory._save_knowledge_base()
```

2. Historical Data Migration

```
# Migrate from legacy systems
def migrate_legacy_data(legacy_db_path):
    # Custom migration logic
    legacy_conn = sqlite3.connect(legacy_db_path)
    memory = AIMemory()

    # Transfer and transform data
    # ... migration logic
```

Performance & Scaling

Database Optimization

1. Indexes

```
-- Performance indexes for common queries
CREATE INDEX idx_fix_history_timestamp ON fix_history(timestamp);
CREATE INDEX idx_fix_history_model ON fix_history(model_name);
CREATE INDEX idx_fix_history_issue_type ON fix_history(issue_type);
CREATE INDEX idx_fix_history_signature ON fix_history(issue_signature);
CREATE INDEX idx_fix_history_fixer_type ON fix_history(fixer_type);
CREATE INDEX idx_fix_history_success ON fix_history(fix_success,
verification_success);
```

2. Query Optimization

```
# Efficient pattern matching
def query_similar_issues_optimized(self, issue_data, limit=5):
    signature = self.generate_issue_signature(issue_data)

    # Use prepared statements and proper indexing
    queries = [
        ("exact", "WHERE issue_signature = ? ORDER BY timestamp DESC LIMIT
```

```
?"),
        ("model", "WHERE model_name = ? AND issue_type = ? ORDER BY
fix_success DESC LIMIT ?"),
        ("type", "WHERE issue_type = ? ORDER BY fix_success DESC LIMIT ?")
    ]

    # Execute optimized queries...
```

Storage Management

1. Automatic Cleanup

```
# Configurable retention policies
def auto_cleanup():
    # Keep successful fixes longer than failed ones
    # Retain patterns even after individual records expire
    # Preserve high-value learning experiences
```

2. Data Compression

```
# Compress older records
def compress_historical_data():
    # Archive detailed logs to compressed format
    # Keep summary statistics accessible
    # Maintain pattern integrity
```

Scaling Considerations

1. Multi-User Environments

- Database locking strategies
- Concurrent access patterns
- Shared knowledge base management

2. Large Datasets

- Partitioning strategies for fix_history table
- Aggregation tables for faster analytics
- Background processing for pattern analysis

Maintenance

Regular Maintenance Tasks

1. Weekly Tasks

```
# Check database integrity
sqlite3 AIMemory/ai_memory.db "PRAGMA integrity_check;"

# Update statistics
./memory.sh stats

# Review recent patterns
./memory.sh patterns
```

2. Monthly Tasks

```
# Export backup
./memory.sh export monthly_backup_$(date +%Y_%m).json

# Clean old records (optional)
./memory.sh cleanup 90

# Analyze performance trends
./memory.sh recent 30
```

3. Database Maintenance

```
-- Vacuum database to reclaim space
VACUUM;

-- Analyze query plans
ANALYZE;

-- Update statistics
UPDATE STATISTICS;
```

Backup Strategy

1. Automated Backups

```
#!/bin/bash
# backup_memory.sh
DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_DIR="/backup/memory_system"

# Create backup directory
mkdir -p "$BACKUP_DIR"

# Export full system state
./memory.sh export "$BACKUP_DIR/memory_backup_$DATE.json"
```

```
# Copy database file
cp AIMemory/ai_memory.db "$BACKUP_DIR/database_backup_$DATE.db"

# Compress old backups
find "$BACKUP_DIR" -name "*.json" -mtime +7 -exec gzip {} \;
```

2. Recovery Procedures

```
# Restore from backup
cp /backup/memory_system/database_backup_YYYYMMDD.db AIMemory/ai_memory.db

# Verify integrity
sqlite3 AIMemory/ai_memory.db "PRAGMA integrity_check;"

# Test functionality
./memory.sh stats
```

API Reference

AIMemory Class

Initialization

```
from Scripts.ai_memory import AIMemory

# Default initialization
memory = AIMemory()

# Custom memory directory
memory = AIMemory(memory_dir="/custom/path")
```

Core Methods

record_fix_attempt()

```
def record_fix_attempt(
    self,
    issue_data: Dict,
    ai_response: Dict,
    fix_success: bool,
    verification_success: bool,
    execution_time: float,
    notes: str = None,
    fixer_type: str = "claude"
```

```
) -> int:
    """
    Record a fix attempt in the history database.

    Returns:
        int: Unique fix ID for the recorded attempt
    """
```

query_similar_issues()

```
def query_similar_issues(
    self,
    issue_data: Dict,
    limit: int = 5
) -> Dict:
    """
    Find similar issues from history.

    Returns:
        Dict: {
            'exact_matches': List[Dict],
            'model_matches': List[Dict],
            'type_matches': List[Dict]
        }
    """
```

get_model_insights()

```
def get_model_insights(self, model_name: str) -> Dict:
    """
    Get comprehensive insights for a specific model.

    Returns:
        Dict: Model performance data and characteristics
    """
```

build_context_for_claude()

```
def build_context_for_claude(self, issue_data: Dict) -> str:
    """
    Build rich historical context for AI analysis.

    Returns:
        str: Formatted context string for AI consumption
    """
```

get_memory_stats()

```
def get_memory_stats(self) -> Dict:
    """
    Get comprehensive memory system statistics.

    Returns:
        Dict: Statistical summary of system performance
    """
```

export_insights()

```
def export_insights(self, output_file: str) -> None:
    """
    Export complete system insights to JSON file.

    Args:
        output_file: Path to output JSON file
    """
```

Utility Functions

generate_issue_signature()

```
def generate_issue_signature(self, issue_data: Dict) -> str:
    """
    Generate unique signature for issue classification.

    Args:
        issue_data: Issue information dictionary

    Returns:
        str: 16-character hexadecimal signature
    """
```

Database Direct Access

```
import sqlite3

# Direct database operations
conn = sqlite3.connect('AIMemory/ai_memory.db')
cursor = conn.cursor()
```

```
# Custom queries
cursor.execute("SELECT COUNT(*) FROM fix_history WHERE fixer_type = ?",
("qwen",))
qwen_attempts = cursor.fetchone()[0]

conn.close()
```

🧠 **The Memory & Learning System is the foundation that makes the AI Auto-Fix framework increasingly intelligent over time. It learns from every interaction and continuously improves fix success rates.**