# **Safety Sigma MVP Evolution: Technical Requirements**

**Owner**: Assaf (Solo Developer)  
 **Audience**: Self + Future Reference  
 **Status**: Ready for Implementation  
 **Version**: MVP Evolution 1.0

## **1. Context: What We're Building On**

### **1.1 Phase 1 Script MVP (Current State)**

**Working System**:

* PDF processing with PyPDF2 ✅
* Zero-inference extraction with GPT-4 ✅
* Multi-format rule generation (SQL, Python, JSON) ✅
* Compliance validation (blocks synthetic data) ✅
* Basic audit trail ✅

**Phase 1 Key Learnings Applied**:

* **Compliance must be architectural, not prompt-based**
* **User feedback integration is essential**
* **Clear separation of extraction vs analysis**
* **Audit trail must be comprehensive**

### **1.2 Evolution Goal**

**Transform existing script into agent-based system** through 6 incremental steps, each provable and reversible.

## **2. Evolution Requirements by Step**

### **Step 1: Tool Abstraction (Week 1)**

**Requirement**: Wrap existing functions as tools without changing behavior

#### **2.1.1 Functional Requirements**

* **Tool Interface**: Standard interface for all processing functions
* **Backward Compatibility**: Existing script behavior must be preserved exactly
* **Tool Registry**: Simple way to register and discover tools
* **Error Handling**: Tools must handle errors gracefully

#### **2.1.2 Technical Specifications**

# Base tool interface

class BaseTool:

def execute(self, input\_data: Any) -> ToolResult

def validate\_input(self, input\_data: Any) -> bool

def get\_name(self) -> str

#### **2.1.3 Success Criteria**

* [ ] Same PDF input produces identical output to Phase 1 script
* [ ] All existing functionality accessible through tool interface
* [ ] Zero behavior changes from user perspective

### **Step 2: Simple Agent Logic (Week 2)**

**Requirement**: Add basic decision-making without changing core functionality

#### **2.2.1 Functional Requirements**

* **Decision Logic**: Simple if/then logic for tool selection
* **Input Analysis**: Basic input type detection (PDF vs text vs URL)
* **Workflow Execution**: Sequential tool execution with data passing
* **State Management**: Track processing state between tool calls

#### **2.2.2 Technical Specifications**

class SimpleAgent:

def process(self, input\_data: Any) -> ProcessingResult

def analyze\_input(self, input\_data: Any) -> InputAnalysis

def select\_workflow(self, analysis: InputAnalysis) -> WorkflowPlan

#### **2.2.3 Success Criteria**

* [ ] Agent produces identical outputs to tool orchestrator
* [ ] Correct workflow selection for different input types
* [ ] Clear decision audit trail

### **Step 3: Claude Integration (Week 3)**

**Requirement**: Add Claude for enhancement tasks while maintaining compliance

#### **2.3.1 Functional Requirements**

* **Safe Claude Integration**: Claude driver with compliance validation
* **Enhancement Tasks**: Use Claude for non-critical improvements (documentation, explanations)
* **Compliance Preservation**: Core extraction logic unchanged
* **Validation Pipeline**: All Claude outputs validated before use

#### **2.3.2 Technical Specifications**

class ClaudeDriver:

def safe\_completion(self, prompt: str, validator: Callable) -> str

def enhance\_output(self, base\_output: Any, enhancement\_type: str) -> Any

#### **2.3.3 Success Criteria**

* [ ] Zero compliance violations in Claude outputs
* [ ] Enhanced outputs provide clear value over baseline
* [ ] Core extraction accuracy maintained

### **Step 4: Dynamic Workflows (Week 4)**

**Requirement**: Agent selects processing strategy based on document analysis

#### **2.4.1 Functional Requirements**

* **Document Analysis**: Classify threat type, complexity, format
* **Workflow Selection**: Choose specialized processing based on analysis
* **Specialized Processing**: Different workflows for different threat types
* **Performance Tracking**: Measure workflow effectiveness

#### **2.4.2 Technical Specifications**

class DocumentAnalyzer:

def analyze(self, document: Any) -> DocumentAnalysis

class WorkflowSelector:

def select\_workflow(self, analysis: DocumentAnalysis) -> WorkflowConfig

#### **2.4.3 Success Criteria**

* [ ] Correct threat type identification (≥90% accuracy on test set)
* [ ] Specialized workflows outperform generic workflow
* [ ] Processing time appropriate for complexity level

### **Step 5: Multi-Agent System (Week 5)**

**Requirement**: Specialized agents collaborate for better results

#### **2.5.1 Functional Requirements**

* **Agent Specialization**: Separate agents for extraction, validation, compilation
* **Agent Communication**: Clean interfaces between specialized agents
* **Coordination Logic**: Master agent orchestrates specialized agents
* **Performance Comparison**: Multi-agent vs single-agent metrics

#### **2.5.2 Technical Specifications**

class ExtractionAgent:

def extract\_patterns(self, document: Document) -> ExtractionResult

class ValidationAgent:

def validate\_extraction(self, extraction: ExtractionResult, source: Document) -> ValidationResult

class CompilationAgent:

def compile\_rules(self, validated\_extraction: ExtractionResult) -> RuleSet

#### **2.5.3 Success Criteria**

* [ ] Each specialized agent outperforms general-purpose equivalent
* [ ] Multi-agent coordination works reliably
* [ ] Overall system quality improvement measurable

### **Step 6: Self-Improvement (Week 6)**

**Requirement**: System learns from performance data and improves over time

#### **2.6.1 Functional Requirements**

* **Performance Tracking**: Detailed metrics on processing quality
* **Pattern Recognition**: Identify recurring issues or improvement opportunities
* **Strategy Adjustment**: Modify processing based on learned patterns
* **Feedback Integration**: Incorporate user feedback into improvement loop

#### **2.6.2 Technical Specifications**

class PerformanceTracker:

def track\_processing(self, input: Any, output: Any, feedback: Any) -> None

def analyze\_patterns(self) -> List[ImprovementOpportunity]

class AdaptiveSystem:

def adjust\_processing(self, opportunities: List[ImprovementOpportunity]) -> None

#### **2.6.3 Success Criteria**

* [ ] System demonstrates measurable improvement over time
* [ ] Feedback integration improves relevant metrics
* [ ] Self-adjustment doesn't compromise compliance

## **3. Non-Functional Requirements**

### **3.1 Performance Requirements**

**MVP-Appropriate Targets** (not enterprise-scale):

* Processing time: ≤2× Phase 1 script performance
* Memory usage: Reasonable for single-machine deployment
* Concurrent processing: Not required for MVP
* Throughput: 10-20 documents per day sufficient

### **3.2 Reliability Requirements**

**MVP-Appropriate Standards**:

* Graceful error handling with clear error messages
* Basic retry logic for transient failures
* No silent failures (all errors logged and reported)
* Rollback capability to previous evolution step

### **3.3 Compliance Requirements**

**Non-Negotiable** (from Phase 1 learnings):

* Zero synthetic data generation
* Complete source traceability for all extractions
* Audit trail for all processing decisions
* Validation that blocks non-compliant outputs

### **3.4 Usability Requirements**

**Solo Developer Focused**:

* Simple command-line interface
* Clear configuration files
* Comprehensive logging for debugging
* Easy deployment (single script/container)

## **4. Technical Architecture Requirements**

### **4.1 Modular Design**

* **Pluggable Tools**: Easy to add new tools without changing core system
* **Configurable Workflows**: Workflows defined in config files, not code
* **Extensible Agents**: New agent types can be added with minimal changes
* **Clean Interfaces**: Well-defined contracts between components

### **4.2 State Management**

* **Processing State**: Track state between tool calls
* **Configuration State**: Runtime configuration management
* **Performance State**: Historical performance data storage
* **Recovery State**: Checkpoints for error recovery

### **4.3 Data Flow**

Input → Analysis → Workflow Selection → Tool Execution → Validation → Output

↓ ↓ ↓ ↓ ↓ ↓

Audit Audit Audit Audit Audit Audit

## **5. Integration Requirements**

### **5.1 Backward Compatibility**

* **Script Interface**: Original script interface must continue working
* **Output Format**: Existing output formats preserved
* **Configuration**: Existing config files continue to work
* **Dependencies**: Minimal new dependencies added

### **5.2 Development Environment**

* **Python Environment**: Continue using existing Python setup
* **Testing Framework**: Simple testing that validates each evolution step
* **Version Control**: Git workflow for tracking evolution steps
* **Documentation**: Living documentation updated with each step

### **5.3 Deployment**

* **Single Machine**: Designed for solo developer single-machine deployment
* **Simple Dependencies**: Minimize external service dependencies
* **Configuration**: Environment variables and config files
* **Monitoring**: Basic logging and health checks

## **6. Quality Requirements**

### **6.1 Testing Strategy**

**Per Evolution Step**:

* **Regression Tests**: Validate previous functionality still works
* **Feature Tests**: Validate new functionality works as expected
* **Compliance Tests**: Validate zero-hallucination guarantee maintained
* **Performance Tests**: Validate performance doesn't degrade significantly

### **6.2 Code Quality**

**Solo Developer Standards** (practical, not perfect):

* **Readable Code**: Clear variable names, logical structure
* **Error Handling**: Comprehensive error catching and reporting
* **Logging**: Detailed logging for debugging and audit
* **Documentation**: Inline comments and README updates

### **6.3 Validation Criteria**

**Each Evolution Step Must**:

1. Maintain existing functionality (backward compatibility)
2. Add new capability that provides measurable value
3. Pass all compliance tests (zero hallucinations)
4. Complete within planned timeframe (1 week per step)
5. Be demonstrable to potential customers

## **7. Success Metrics**

### **7.1 Technical Metrics**

* **Functional Parity**: All Phase 1 script capabilities preserved
* **Performance Parity**: Processing time within 2× of original script
* **Compliance Parity**: Zero regression in compliance validation
* **Quality Improvement**: Measurable improvement in output quality

### **7.2 Evolution Metrics**

* **Step Completion**: Each evolution step completed on schedule
* **Rollback Capability**: Ability to revert to previous step if needed
* **Integration Success**: Each step integrates cleanly with previous steps
* **Customer Validation**: Customer feedback validates evolution direction

### **7.3 Learning Metrics**

* **Technical Learning**: Understanding of agent architectures improved
* **Customer Learning**: Better understanding of customer needs
* **Product Learning**: Clearer path to product-market fit
* **Market Learning**: Validation of agent-based approach with customers

## **8. Risk Mitigation**

### **8.1 Technical Risks**

* **Complexity Creep**: Keep each step simple and focused
* **Performance Degradation**: Benchmark and validate each step
* **Integration Failures**: Thorough testing between steps
* **Compliance Regression**: Continuous compliance validation

### **8.2 Product Risks**

* **Over-Engineering**: Stay focused on MVP-level functionality
* **Feature Creep**: Only build what's needed for current evolution step
* **Customer Disconnect**: Regular customer feedback throughout evolution
* **Timeline Risk**: Fixed 1-week iterations with clear success criteria

### **8.3 Business Risks**

* **Technology Risk**: Agent approach may not provide value
* **Market Risk**: Customers may prefer simpler script-based approach
* **Resource Risk**: Solo developer capacity limitations
* **Opportunity Risk**: Time spent on evolution vs customer acquisition

This requirements document provides the technical foundation for evolving your working Phase 1 script into an agent-based system through proven, incremental steps that maintain all existing functionality while adding new capabilities.