# **Safety Sigma MVP Evolution: Feature-Sized Phases**

**Owner**: Assaf (Solo Developer)  
 **Audience**: Implementation Guide  
 **Status**: Ready for Development  
 **Version**: MVP Evolution 1.0

## **Overview: 6 Feature-Sized Evolution Steps**

Each step is a complete, testable feature that builds on your existing Phase 1 script. Every step maintains backward compatibility while adding new agent-based capabilities.

**Core Principle**: Never break what works, always prove value before proceeding.

## **Step 1: Tool Abstraction Layer**

**Duration**: 3-4 days  
 **Goal**: Wrap existing functions as tools without changing behavior

### **1.1 Feature Description**

Transform your existing Phase 1 script functions into a tool-based architecture while maintaining identical functionality. This creates the foundation for agent orchestration.

### **1.2 User Story**

*"As a developer, I need to refactor my script into tools so that agents can orchestrate them intelligently, but with zero risk to existing functionality."*

### **1.3 Technical Deliverables**

**Day 1-2: Core Tool Infrastructure**

# File: tools/base\_tool.py

class BaseTool:

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

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

# File: tools/pdf\_tool.py

class PDFTool(BaseTool):

def \_\_init\_\_(self):

from phase1\_script import process\_pdf\_with\_spans

self.\_process\_pdf = process\_pdf\_with\_spans

def execute(self, pdf\_path: str) -> ToolResult:

return self.\_process\_pdf(pdf\_path) # Your existing function

# File: tools/extraction\_tool.py

class ExtractionTool(BaseTool):

def \_\_init\_\_(self):

from phase1\_script import extract\_ir\_from\_text

self.\_extract\_ir = extract\_ir\_from\_text

def execute(self, text\_data: Dict) -> ToolResult:

return self.\_extract\_ir(text\_data) # Your existing function

**Day 3: Tool Orchestration**

# File: orchestration/tool\_orchestrator.py

class ToolOrchestrator:

def register\_tool(self, tool: BaseTool)

def execute\_pipeline(self, input\_data, pipeline: List[str]) -> Dict

**Day 4: Integration Testing**

* Test that tool pipeline produces identical output to Phase 1 script
* Validate all existing functionality works through tool interface

### **1.4 Acceptance Criteria**

* [ ] **Functional Parity**: Tool pipeline produces identical outputs to Phase 1 script
* [ ] **Tool Registry**: All existing functions wrapped as tools
* [ ] **Pipeline Execution**: Sequential tool execution with data chaining
* [ ] **Error Handling**: Graceful tool error handling and reporting
* [ ] **Audit Trail**: Basic logging of tool execution

### **1.5 Success Validation**

**Test Command**: python test\_tool\_abstraction.py

* Run same FBI report through both Phase 1 script and tool pipeline
* Assert outputs are byte-for-byte identical
* Validate processing time within 10% of original

### **1.6 Rollback Plan**

If tool abstraction fails:

* Phase 1 script continues to work unchanged
* Remove tool wrapper files
* Continue with Phase 1 script until issue resolved

## **Step 2: Simple Agent Decision Logic**

**Duration**: 3-4 days  
 **Goal**: Add basic agent decision-making without changing core functionality

### **2.1 Feature Description**

Create a simple agent that makes basic decisions about which tools to use based on input analysis. Agent uses hardcoded logic initially but provides foundation for intelligent decision-making.

### **2.2 User Story**

*"As a user, I want the system to automatically choose the right processing workflow based on my input type, so I don't have to specify processing options manually."*

### **2.3 Technical Deliverables**

**Day 1-2: Simple Agent Logic**

# File: agents/simple\_agent.py

class SimpleAgent:

def process(self, input\_data) -> Dict

def analyze\_input(self, input\_data) -> Dict

def select\_workflow(self, analysis: Dict) -> List[str]

# Basic input analysis

def analyze\_input(self, input\_data):

return {

'input\_type': 'pdf' if input\_data.endswith('.pdf') else 'text',

'estimated\_size': 'large' if len(str(input\_data)) > 1000 else 'small'

}

# Simple workflow selection

def select\_workflow(self, analysis):

if analysis['input\_type'] == 'pdf':

return ['pdf\_processor', 'ir\_extractor', 'rule\_generator', 'validator']

else:

return ['ir\_extractor', 'rule\_generator', 'validator'] # Skip PDF

**Day 3: Decision Logic Enhancement**

* Add support for URL inputs
* Add complexity-based processing decisions
* Add basic performance optimization logic

**Day 4: Agent Testing**

* Test agent with different input types (PDF, text, URL)
* Validate correct workflow selection
* Ensure output quality maintained

### **2.4 Acceptance Criteria**

* [ ] **Input Analysis**: Correctly identifies input type and characteristics
* [ ] **Workflow Selection**: Chooses appropriate tool sequence based on input
* [ ] **Decision Logging**: Records decision rationale for audit
* [ ] **Performance Parity**: Output quality matches or exceeds tool orchestrator
* [ ] **Multiple Input Types**: Handles PDF, text, and URL inputs

### **2.5 Success Validation**

* Agent correctly processes FBI PDF (PDF workflow)
* Agent correctly processes extracted text (text workflow)
* Agent correctly handles different input types in same session
* Decision log shows clear rationale for each choice

## **Step 3: Claude Integration for Enhancement**

**Duration**: 4-5 days  
 **Goal**: Add Claude for enhancement tasks while maintaining compliance

### **3.1 Feature Description**

Integrate Claude API for enhancement tasks (better documentation, explanations) while keeping your proven extraction logic unchanged. Claude is used to improve output quality, not replace core functionality.

### **3.2 User Story**

*"As a user, I want enhanced rule documentation and explanations so that I can better understand and maintain the generated rules, but with zero risk of hallucinated data."*

### **3.3 Technical Deliverables**

**Day 1-2: Safe Claude Driver**

# File: drivers/claude\_driver.py

class ClaudeDriver:

def \_\_init\_\_(self, compliance\_validator):

self.client = anthropic.Anthropic()

self.compliance\_validator = compliance\_validator

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

response = self.client.messages.create(...)

if not validator(response):

raise ComplianceError("Output failed validation")

return response

def enhance\_rule\_documentation(self, rule\_data: Dict) -> Dict:

# Use Claude to improve rule comments/documentation only

# Rule logic remains unchanged from your proven extraction

**Day 3-4: Enhancement Integration**

# File: tools/enhancement\_tool.py

class EnhancementTool(BaseTool):

def execute(self, rule\_data: Dict) -> ToolResult:

try:

enhanced = self.claude\_driver.enhance\_rule\_documentation(rule\_data)

return ToolResult(success=True, data=enhanced)

except ComplianceError:

# Enhancement failed - return original rules unchanged

return ToolResult(success=True, data=rule\_data)

**Day 5: Validation & Testing**

* Test Claude enhancement adds value without changing rule logic
* Validate compliance checks block any synthetic data
* Test fallback to original rules when enhancement fails

### **3.4 Acceptance Criteria**

* [ ] **Safe Integration**: Claude outputs pass compliance validation 100% of time
* [ ] **Enhancement Value**: Claude improvements make rules more understandable
* [ ] **Compliance Preservation**: Core extraction logic unchanged
* [ ] **Graceful Fallback**: System works when Claude enhancement fails
* [ ] **Zero Hallucinations**: No synthetic data in any Claude outputs

### **3.5 Success Validation**

* Enhanced rules have better documentation than original
* Rule logic remains identical to Phase 1 script output
* Compliance validator blocks any synthetic Claude content
* System continues working if Claude API is unavailable

## **Step 4: Dynamic Workflow Selection**

**Duration**: 4-5 days  
 **Goal**: Agent intelligently selects workflows based on document analysis

### **4.1 Feature Description**

Add document analysis capabilities so the agent can choose specialized processing workflows based on threat type, complexity, and document characteristics.

### **4.2 User Story**

*"As a threat analyst, I want the system to automatically use specialized processing for different threat types so that I get better extraction quality for romance scams vs phishing vs malware reports."*

### **4.3 Technical Deliverables**

**Day 1-2: Document Analysis**

# File: analysis/document\_analyzer.py

class DocumentAnalyzer:

def analyze(self, document\_data: Dict) -> Dict:

return {

'threat\_type': self.\_classify\_threat\_type(document\_data),

'complexity\_level': self.\_assess\_complexity(document\_data),

'processing\_requirements': self.\_determine\_requirements(document\_data)

}

def \_classify\_threat\_type(self, document\_data: Dict) -> str:

text = document\_data.get('text', '').lower()

if 'romance' in text or 'dating' in text:

return 'romance\_scam'

elif 'investment' in text or 'bitcoin' in text:

return 'investment\_fraud'

elif 'phishing' in text or 'email' in text:

return 'phishing'

return 'general\_fraud'

**Day 3-4: Specialized Workflows**

# File: workflow/workflow\_selector.py

class WorkflowSelector:

def select\_workflow(self, analysis: Dict) -> List[str]:

threat\_type = analysis['threat\_type']

if threat\_type == 'romance\_scam':

return ['pdf\_processor', 'ir\_extractor', 'behavioral\_extractor', 'rule\_generator']

elif threat\_type == 'investment\_fraud':

return ['pdf\_processor', 'ir\_extractor', 'financial\_extractor', 'rule\_generator']

# etc.

**Day 5: Integration & Testing**

* Test workflow selection accuracy on golden dataset
* Validate specialized workflows produce better results
* Test fallback to general workflow for unknown threat types

### **4.4 Acceptance Criteria**

* [ ] **Threat Classification**: ≥90% accuracy on test dataset threat type identification
* [ ] **Workflow Selection**: Correct workflow chosen based on analysis
* [ ] **Specialized Processing**: Specialized workflows outperform general workflow
* [ ] **Fallback Handling**: Unknown threat types use general workflow
* [ ] **Performance Improvement**: Overall extraction quality measurably improved

### **4.5 Success Validation**

* Romance scam reports processed with romance-specific workflow
* Investment fraud reports processed with financial-specific workflow
* Specialized processing shows measurable improvement over general workflow
* Unknown document types gracefully handled with general workflow

## **Step 5: Multi-Agent Collaboration**

**Duration**: 5-6 days  
 **Goal**: Specialized agents working together for better results

### **5.1 Feature Description**

Break processing into specialized agents (extraction, validation, compilation) coordinated by a master agent. Each agent focuses on their specialty for better overall results.

### **5.2 User Story**

*"As a system, I want specialized agents handling different aspects of processing so that each component can be optimized for its specific task while maintaining overall coordination."*

### **5.3 Technical Deliverables**

**Day 1-2: Specialized Agent Creation**

# File: agents/extraction\_agent.py

class ExtractionAgent:

def extract\_patterns(self, document: Dict, strategy: str = "default") -> Dict:

# Specialized extraction logic

# File: agents/validation\_agent.py

class ValidationAgent:

def validate\_extraction(self, extraction\_result: Dict, source: Dict) -> Dict:

# Specialized validation logic

# File: agents/compilation\_agent.py

class CompilationAgent:

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

# Specialized rule compilation

**Day 3-4: Master Agent Coordination**

# File: agents/master\_agent.py

class MasterAgent:

def process\_document(self, document: Dict) -> Dict:

# Coordinate specialized agents

extraction = self.extraction\_agent.extract\_patterns(document)

validation = self.validation\_agent.validate\_extraction(extraction, document)

if validation['passed']:

compilation = self.compilation\_agent.compile\_rules(extraction)

return {'extraction': extraction, 'validation': validation, 'compilation': compilation}

**Day 5-6: Agent Integration & Testing**

* Test multi-agent coordination works reliably
* Validate specialized agents outperform general agents
* Test error handling between agents

### **5.4 Acceptance Criteria**

* [ ] **Agent Specialization**: Each agent focuses on specific task optimally
* [ ] **Coordination**: Master agent successfully orchestrates specialized agents
* [ ] **Quality Improvement**: Multi-agent results exceed single-agent results
* [ ] **Error Handling**: Graceful handling when one agent fails
* [ ] **Performance**: Overall processing time reasonable despite multiple agents

### **5.5 Success Validation**

* Extraction agent produces higher quality extractions than general extractor
* Validation agent catches more issues than simple validation
* Compilation agent produces better rules than generic compiler
* Multi-agent coordination works reliably across different document types

## **Step 6: Self-Improvement Loop**

**Duration**: 5-6 days  
 **Goal**: System learns from performance data and improves over time

### **6.1 Feature Description**

Add performance tracking and learning capabilities so the system can identify its own weaknesses and improve processing strategies based on historical data and user feedback.

### **6.2 User Story**

*"As a system, I want to learn from my processing results and user feedback so that I can continuously improve my performance without manual intervention."*

### **6.3 Technical Deliverables**

**Day 1-2: Performance Tracking**

# File: learning/performance\_tracker.py

class PerformanceTracker:

def track\_processing\_run(self, input\_doc: Dict, result: Dict, feedback: Dict = None):

# Store processing performance data

def analyze\_performance\_trends(self) -> List[Dict]: