# **Safety Sigma MVP Evolution: System Design**

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

## **1. Evolution Architecture Overview**

### **1.1 Current State → Target State**

PHASE 1 SCRIPT → AGENT-BASED SYSTEM

[Single Script] → [Master Agent]

├── pdf\_processing() → ├── [PDF Tool]

├── extraction() → ├── [Extraction Agent]

├── rule\_generation() → ├── [Compilation Agent]

├── validation() → ├── [Validation Agent]

└── output() → └── [Output Tools]

### **1.2 Evolution Principles**

**Incremental Transformation**:

* Each step wraps existing functionality, doesn't replace it
* Backward compatibility maintained at every step
* New capabilities added on top of proven foundation
* Rollback to previous step always possible

**Risk-Free Evolution**:

* Existing Phase 1 script continues to work unchanged
* New agent layer can be toggled on/off
* All Phase 1 compliance guarantees preserved
* Customer can use either version during transition

## **2. Step-by-Step Design Evolution**

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

**Goal**: Wrap existing functions without changing behavior

#### **2.1 Tool Interface Design**

# tools/base\_tool.py

from abc import ABC, abstractmethod

from typing import Any, Dict

from dataclasses import dataclass

@dataclass

class ToolResult:

success: bool

data: Any

metadata: Dict[str, Any]

processing\_time: float

error\_message: str = None

class BaseTool(ABC):

"""Base interface for all processing tools"""

def \_\_init\_\_(self, name: str):

self.name = name

self.call\_count = 0

@abstractmethod

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

"""Execute the tool with input data"""

pass

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

"""Validate input before execution"""

return True

def \_log\_execution(self, input\_data: Any, result: ToolResult):

"""Log tool execution for audit trail"""

self.call\_count += 1

# Simple logging - expand in later steps

#### **2.2 Existing Function Wrappers**

# tools/pdf\_tool.py

class PDFTool(BaseTool):

"""Wrapper for existing PDF processing logic"""

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

super().\_\_init\_\_("pdf\_processor")

# Import your existing PDF processing function

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

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

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

"""Wrap existing PDF processing"""

start\_time = time.time()

try:

# Call your existing function unchanged

result\_data = self.\_process\_pdf(pdf\_path)

return ToolResult(

success=True,

data=result\_data,

metadata={"pdf\_path": pdf\_path},

processing\_time=time.time() - start\_time

)

except Exception as e:

return ToolResult(

success=False,

data=None,

metadata={"pdf\_path": pdf\_path},

processing\_time=time.time() - start\_time,

error\_message=str(e)

)

# tools/extraction\_tool.py

class ExtractionTool(BaseTool):

"""Wrapper for existing extraction logic"""

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

super().\_\_init\_\_("ir\_extractor")

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

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

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

"""Wrap existing extraction logic"""

start\_time = time.time()

try:

# Call your existing function unchanged

ir\_data = self.\_extract\_ir(text\_data)

return ToolResult(

success=True,

data=ir\_data,

metadata={"pattern\_count": len(ir\_data.get("patterns", []))},

processing\_time=time.time() - start\_time

)

except Exception as e:

return ToolResult(

success=False,

data=None,

metadata={},

processing\_time=time.time() - start\_time,

error\_message=str(e)

)

#### **2.3 Tool Orchestrator**

# orchestration/tool\_orchestrator.py

class ToolOrchestrator:

"""Simple sequential tool execution"""

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

self.tools = {}

self.execution\_log = []

def register\_tool(self, tool: BaseTool):

"""Register a tool for use"""

self.tools[tool.name] = tool

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

"""Execute tools in sequence"""

current\_data = input\_data

results = {}

for tool\_name in pipeline:

if tool\_name not in self.tools:

raise ValueError(f"Tool {tool\_name} not registered")

tool = self.tools[tool\_name]

result = tool.execute(current\_data)

# Log execution

self.execution\_log.append({

"tool": tool\_name,

"success": result.success,

"processing\_time": result.processing\_time,

"timestamp": datetime.now().isoformat()

})

if not result.success:

break

results[tool\_name] = result

current\_data = result.data

return results

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

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

#### **2.4 Simple Agent Design**

# agents/simple\_agent.py

class SimpleAgent:

"""Basic agent with hardcoded decision logic"""

def \_\_init\_\_(self, orchestrator: ToolOrchestrator):

self.orchestrator = orchestrator

self.decision\_log = []

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

"""Process input with simple decision logic"""

# Simple input analysis

analysis = self.analyze\_input(input\_data)

# Select workflow based on analysis

workflow = self.select\_workflow(analysis)

# Execute workflow

results = self.orchestrator.execute\_pipeline(input\_data, workflow)

# Log decision

self.\_log\_decision(analysis, workflow, results)

return results

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

"""Simple input analysis"""

analysis = {

"input\_type": self.\_detect\_input\_type(input\_data),

"estimated\_complexity": "medium", # Default for now

"processing\_strategy": "standard"

}

return analysis

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

"""Select tools based on analysis"""

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

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

elif analysis["input\_type"] == "text":

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

else:

return ["pdf\_processor", "ir\_extractor", "rule\_generator", "validator"] # Default

def \_detect\_input\_type(self, input\_data: Any) -> str:

"""Basic input type detection"""

if isinstance(input\_data, str):

if input\_data.endswith('.pdf'):

return "pdf"

elif input\_data.startswith('http'):

return "url"

else:

return "text"

return "unknown"

### **Step 3: Claude Integration**

**Goal**: Add Claude for enhancement without compromising core extraction

#### **2.5 Claude Driver Design**

# drivers/claude\_driver.py

class ClaudeDriver:

"""Safe Claude integration with compliance validation"""

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

self.client = anthropic.Anthropic()

self.compliance\_validator = compliance\_validator

self.call\_log = []

def safe\_completion(self, prompt: str, validation\_fn: Callable = None) -> str:

"""Claude completion with mandatory validation"""

# Pre-flight validation

if not self.\_validate\_prompt(prompt):

raise ComplianceError("Prompt failed pre-flight validation")

# Call Claude

response = self.\_make\_claude\_call(prompt)

# Post-flight validation

if validation\_fn and not validation\_fn(response):

raise ComplianceError("Response failed validation")

# Compliance check

if not self.compliance\_validator.validate\_output(response):

raise ComplianceError("Response failed compliance check")

# Log successful call

self.\_log\_call(prompt, response, "success")

return response

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

"""Use Claude to enhance rule documentation without changing logic"""

# Build enhancement prompt

prompt = self.\_build\_enhancement\_prompt(rule\_data, ir\_data)

# Validation function - ensures only documentation changes

def validate\_enhancement(response):

# Check that rule logic is unchanged

# Only documentation/comments should be different

return self.\_validate\_documentation\_only(response, rule\_data)

# Get enhancement

enhanced\_docs = self.safe\_completion(prompt, validate\_enhancement)

# Merge enhancement with original rule (logic unchanged)

enhanced\_rule = rule\_data.copy()

enhanced\_rule['documentation'] = enhanced\_docs

enhanced\_rule['enhanced'] = True

return enhanced\_rule

#### **2.6 Enhancement Integration**

# tools/enhancement\_tool.py

class EnhancementTool(BaseTool):

"""Tool for Claude-based enhancements"""

def \_\_init\_\_(self, claude\_driver: ClaudeDriver):

super().\_\_init\_\_("enhancer")

self.claude\_driver = claude\_driver

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

"""Enhance rules with better documentation"""

rule\_data = rule\_and\_ir\_data['rules']

ir\_data = rule\_and\_ir\_data['ir']

try:

enhanced\_rules = []

for rule in rule\_data['rules']:

enhanced\_rule = self.claude\_driver.enhance\_rule\_documentation(

rule, ir\_data

)

enhanced\_rules.append(enhanced\_rule)

enhanced\_data = rule\_data.copy()

enhanced\_data['rules'] = enhanced\_rules

return ToolResult(

success=True,

data=enhanced\_data,

metadata={"enhanced\_count": len(enhanced\_rules)},

processing\_time=time.time() - start\_time

)

except ComplianceError as e:

# Enhancement failed - return original rules

return ToolResult(

success=True, # Success with fallback

data=rule\_data, # Original rules unchanged

metadata={"enhancement\_failed": str(e)},

processing\_time=time.time() - start\_time

)

### **Step 4: Dynamic Workflows**

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

#### **2.7 Document Analysis Design**

# analysis/document\_analyzer.py

class DocumentAnalyzer:

"""Analyze documents to select optimal processing strategy"""

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

self.analysis\_history = []

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

"""Comprehensive document analysis"""

analysis = {

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

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

"document\_format": self.\_analyze\_format(document\_data),

"processing\_requirements": self.\_determine\_requirements(document\_data),

"recommended\_workflow": None # Filled by workflow selector

}

# Store analysis for learning

self.analysis\_history.append(analysis)

return analysis

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

"""Classify the type of threat described"""

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

# Simple keyword-based classification (improve over time)

if any(word in text for word in ['romance', 'dating', 'relationship']):

return 'romance\_scam'

elif any(word in text for word in ['bitcoin', 'cryptocurrency', 'investment']):

return 'investment\_fraud'

elif any(word in text for word in ['phishing', 'email', 'credential']):

return 'phishing'

elif any(word in text for word in ['ransomware', 'malware', 'infection']):

return 'malware'

else:

return 'general\_fraud'

def \_assess\_complexity(self, document\_data: Dict) -> str:

"""Assess processing complexity requirements"""

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

# Simple heuristics

if len(text) > 10000:

return 'high'

elif len(text) > 5000:

return 'medium'

else:

return 'low'

#### **2.8 Workflow Selection Design**

# workflow/workflow\_selector.py

class WorkflowSelector:

"""Select optimal workflow based on document analysis"""

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

self.workflow\_templates = self.\_load\_workflow\_templates()

self.performance\_history = {}

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

"""Select best workflow for document"""

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

complexity = analysis['complexity\_level']

# Select specialized workflow

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

workflow = self.\_get\_romance\_scam\_workflow(complexity)

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

workflow = self.\_get\_investment\_fraud\_workflow(complexity)

elif threat\_type == 'phishing':

workflow = self.\_get\_phishing\_workflow(complexity)

else:

workflow = self.\_get\_general\_workflow(complexity)

# Track workflow selection for performance analysis

self.\_track\_workflow\_selection(analysis, workflow)

return workflow

def \_get\_romance\_scam\_workflow(self, complexity: str) -> List[str]:

"""Specialized workflow for romance scams"""

base\_workflow = ["pdf\_processor", "ir\_extractor"]

# Add specialized extractors

base\_workflow.append("behavioral\_pattern\_extractor")

base\_workflow.append("financial\_pattern\_extractor")

# Add validation and compilation

base\_workflow.extend(["validator", "rule\_generator", "enhancer"])

return base\_workflow

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

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

#### **2.9 Specialized Agent Architecture**

# agents/extraction\_agent.py

class ExtractionAgent:

"""Specialized agent for data extraction"""

def \_\_init\_\_(self, tools: Dict[str, BaseTool]):

self.tools = tools

self.extraction\_strategies = self.\_load\_strategies()

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

"""Extract threat patterns using specialized strategy"""

# Select extraction strategy

extraction\_plan = self.extraction\_strategies[strategy]

# Execute extraction tools in optimized sequence

results = {}

for tool\_name in extraction\_plan:

if tool\_name in self.tools:

result = self.tools[tool\_name].execute(document)

if result.success:

results[tool\_name] = result.data

document = result.data # Chain outputs

else:

break

return results

# agents/validation\_agent.py

class ValidationAgent:

"""Specialized agent for compliance and quality validation"""

def \_\_init\_\_(self, validators: Dict[str, BaseTool]):

self.validators = validators

self.validation\_history = []

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

"""Comprehensive validation of extraction results"""

validation\_results = {}

overall\_passed = True

# Run all validation checks

for validator\_name, validator in self.validators.items():

validation\_data = {

'extraction': extraction\_result,

'source': source\_document

}

result = validator.execute(validation\_data)

validation\_results[validator\_name] = result

if not result.success:

overall\_passed = False

# Store validation history

self.validation\_history.append({

'results': validation\_results,

'overall\_passed': overall\_passed,

'timestamp': datetime.now().isoformat()

})

return {

'passed': overall\_passed,

'details': validation\_results,

'validation\_id': len(self.validation\_history)

}

#### **2.10 Master Agent Coordination**

# agents/master\_agent.py

class MasterAgent:

"""Coordinates specialized agents for optimal results"""

def \_\_init\_\_(self,

extraction\_agent: ExtractionAgent,

validation\_agent: ValidationAgent,

compilation\_agent: CompilationAgent):

self.extraction\_agent = extraction\_agent

self.validation\_agent = validation\_agent

self.compilation\_agent = compilation\_agent

self.coordination\_log = []

def process\_document(self, document: Dict, processing\_config: Dict = None) -> Dict:

"""Coordinate multi-agent processing"""

processing\_start = time.time()

coordination\_record = {

'document\_id': document.get('id', 'unknown'),

'start\_time': processing\_start,

'steps': []

}

try:

# Step 1: Extraction

extraction\_result = self.extraction\_agent.extract\_patterns(

document,

processing\_config.get('extraction\_strategy', 'default')

)

coordination\_record['steps'].append({

'agent': 'extraction',

'success': bool(extraction\_result),

'patterns\_found': len(extraction\_result.get('patterns', []))

})

# Step 2: Validation

validation\_result = self.validation\_agent.validate\_extraction(

extraction\_result, document

)

coordination\_record['steps'].append({

'agent': 'validation',

'success': validation\_result['passed'],

'validation\_details': validation\_result['details']

})

# Step 3: Compilation (only if validation passed)

if validation\_result['passed']:

compilation\_result = self.compilation\_agent.compile\_rules(

extraction\_result,

processing\_config.get('output\_formats', ['sql', 'python'])

)

coordination\_record['steps'].append({

'agent': 'compilation',

'success': bool(compilation\_result),

'rules\_generated': len(compilation\_result.get('rules', []))

})

else:

compilation\_result = None

coordination\_record['steps'].append({

'agent': 'compilation',

'success': False,

'reason': 'validation\_failed'

})

# Final result

final\_result = {

'extraction': extraction\_result,

'validation': validation\_result,

'compilation': compilation\_result,

'processing\_time': time.time() - processing\_start,

'coordination\_id': len(self.coordination\_log)

}

coordination\_record['end\_time'] = time.time()

coordination\_record['success'] = validation\_result['passed']

self.coordination\_log.append(coordination\_record)

return final\_result

except Exception as e:

coordination\_record['error'] = str(e)

coordination\_record['success'] = False

self.coordination\_log.append(coordination\_record)

raise

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

**Goal**: System learns from performance and improves over time

#### **2.11 Performance Tracking Design**

# learning/performance\_tracker.py

class PerformanceTracker:

"""Track system performance and identify improvement opportunities"""

def \_\_init\_\_(self, storage\_path: str = "performance\_data.json"):

self.storage\_path = storage\_path

self.performance\_data = self.\_load\_performance\_data()

self.metrics = {

'processing\_times': [],

'accuracy\_scores': [],

'user\_feedback': [],

'error\_patterns': [],

'workflow\_effectiveness': {}

}

def track\_processing\_run(self,

input\_document: Dict,

processing\_result: Dict,

user\_feedback: Dict = None) -> None:

"""Track a complete processing run"""

run\_data = {

'timestamp': datetime.now().isoformat(),

'document\_id': input\_document.get('id', 'unknown'),

'document\_type': input\_document.get('threat\_type', 'unknown'),

'processing\_time': processing\_result.get('processing\_time', 0),

'success': processing\_result.get('validation', {}).get('passed', False),

'patterns\_extracted': len(processing\_result.get('extraction', {}).get('patterns', [])),

'rules\_generated': len(processing\_result.get('compilation', {}).get('rules', [])),

'user\_feedback': user\_feedback or {},

'workflow\_used': processing\_result.get('workflow\_used', 'unknown')

}

# Store run data

self.performance\_data['runs'].append(run\_data)

# Update metrics

self.\_update\_metrics(run\_data)

# Persist data

self.\_save\_performance\_data()

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

"""Analyze performance data to identify improvement opportunities"""

opportunities = []

# Check processing time trends

recent\_times = [r['processing\_time'] for r in self.performance\_data['runs'][-20:]]

if recent\_times and np.mean(recent\_times) > self.\_get\_baseline\_time() \* 1.5:

opportunities.append({

'type': 'performance\_degradation',

'severity': 'medium',

'description': 'Processing times have increased significantly',

'suggested\_action': 'optimize\_workflow\_selection'

})

# Check accuracy trends

recent\_feedback = [r['user\_feedback'].get('accuracy', 5) for r in self.performance\_data['runs'][-10:] if r['user\_feedback']]

if recent\_feedback and np.mean(recent\_feedback) < 4.0:

opportunities.append({

'type': 'accuracy\_decline',

'severity': 'high',

'description': 'User feedback indicates accuracy issues',

'suggested\_action': 'review\_extraction\_strategies'

})

# Check workflow effectiveness

workflow\_performance = self.\_analyze\_workflow\_performance()

for workflow, perf in workflow\_performance.items():

if perf['success\_rate'] < 0.8:

opportunities.append({

'type': 'workflow\_ineffective',

'severity': 'medium',

'description': f'Workflow {workflow} has low success rate: {perf["success\_rate"]:.2f}',

'suggested\_action': f'optimize\_workflow\_{workflow}'

})

return opportunities

# learning/adaptive\_system.py

class AdaptiveSystem:

"""System that adapts based on performance analysis"""

def \_\_init\_\_(self,

master\_agent: MasterAgent,

performance\_tracker: PerformanceTracker):

self.master\_agent = master\_agent

self.performance\_tracker = performance\_tracker

self.adaptation\_log = []

def adapt\_system(self) -> Dict:

"""Analyze performance and adapt system accordingly"""

# Get improvement opportunities

opportunities = self.performance\_tracker.analyze\_performance\_trends()

adaptations\_made = []

for opportunity in opportunities:

if opportunity['severity'] == 'high':

adaptation = self.\_apply\_adaptation(opportunity)

if adaptation:

adaptations\_made.append(adaptation)

adaptation\_record = {

'timestamp': datetime.now().isoformat(),

'opportunities\_identified': len(opportunities),

'adaptations\_made': adaptations\_made

}

self.adaptation\_log.append(adaptation\_record)

return adaptation\_record

def \_apply\_adaptation(self, opportunity: Dict) -> Dict:

"""Apply specific adaptation based on opportunity"""

adaptation\_type = opportunity['suggested\_action']

if adaptation\_type == 'optimize\_workflow\_selection':

return self.\_optimize\_workflow\_selection()

elif adaptation\_type == 'review\_extraction\_strategies':

return self.\_review\_extraction\_strategies()

elif adaptation\_type.startswith('optimize\_workflow\_'):

workflow\_name = adaptation\_type.split('\_')[-1]

return self.\_optimize\_specific\_workflow(workflow\_name)

return None

def \_optimize\_workflow\_selection(self) -> Dict:

"""Optimize workflow selection logic based on performance data"""

# Analyze which workflows work best for which document types

workflow\_analysis = self.\_analyze\_workflow\_document\_type\_performance()

# Update workflow selection rules

improvements = []

for doc\_type, best\_workflow in workflow\_analysis.items():

if self.\_update\_workflow\_mapping(doc\_type, best\_workflow):

improvements.append(f"Updated {doc\_type} -> {best\_workflow}")

return {

'type': 'workflow\_selection\_optimization',

'improvements': improvements,

'success': len(improvements) > 0

}

## **3. Data Flow Architecture**

### **3.1 Processing Pipeline Evolution**

STEP 1: Tool Abstraction

Input -> [Tool Registry] -> [Tool Orchestrator] -> Output

↓

[Existing Functions Wrapped]

STEP 2: Simple Agent

Input -> [Simple Agent] -> [Decision Logic] -> [Tool Selection] -> Output

↓

[Basic Input Analysis]

STEP 3: Claude Integration

Input -> [Agent] -> [Tools + Claude Enhancement] -> Output

↓

[Compliance Validation]

STEP 4: Dynamic Workflows

Input -> [Document Analysis] -> [Workflow Selection] -> [Specialized Tools] -> Output

STEP 5: Multi-Agent

Input -> [Master Agent] -> [Extraction Agent] -> [Validation Agent] -> [Compilation Agent] -> Output

STEP 6: Self-Improvement

Input -> [Adaptive System] -> [Performance Analysis] -> [System Optimization] -> Improved Output

### **3.2 State Management**

# state/processing\_state.py

class ProcessingState:

"""Manage state throughout processing pipeline"""

def \_\_init\_\_(self, run\_id: str):

self.run\_id = run\_id

self.start\_time = time.time()

self.current\_step = None

self.step\_results = {}

self.metadata = {}

self.errors = []

def update\_step(self, step\_name: str, result: Any, metadata: Dict = None):

"""Update state with step completion"""

self.current\_step = step\_name

self.step\_results[step\_name] = result

if metadata:

self.metadata.update(metadata)

def add\_error(self, step\_name: str, error: Exception):

"""Record error for debugging"""

self.errors.append({

'step': step\_name,

'error': str(error),

'timestamp': time.time()

})

def get\_state\_summary(self) -> Dict:

"""Get current processing state summary"""

return {

'run\_id': self.run\_id,

'current\_step': self.current\_step,

'steps\_completed': list(self.step\_results.keys()),

'total\_time': time.time() - self.start\_time,

'error\_count': len(self.errors),

'metadata': self.metadata

}

## **4. Configuration Management**

### **4.1 Configuration Architecture**

# config/config\_manager.py

class ConfigManager:

"""Manage configuration for all system components"""

def \_\_init\_\_(self, config\_path: str = "config.yaml"):

self.config\_path = config\_path

self.config = self.\_load\_config()

def \_load\_config(self) -> Dict:

"""Load configuration from file"""

default\_config = {

'tools': {

'pdf\_processor': {'enabled': True},

'ir\_extractor': {'enabled': True},

'rule\_generator': {'enabled': True},

'validator': {'enabled': True},

'enhancer': {'enabled': False} # Disabled until Step 3

},

'agents': {

'simple\_agent': {'enabled': True},

'extraction\_agent': {'enabled': False}, # Disabled until Step 5

'validation\_agent': {'enabled': False},

'compilation\_agent': {'enabled': False}

},

'workflows': {

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

'romance\_scam': ['pdf\_processor', 'ir\_extractor', 'behavioral\_extractor', 'rule\_generator', 'validator'],

'investment\_fraud': ['pdf\_processor', 'ir\_extractor', 'financial\_extractor', 'rule\_generator', 'validator']

},

'performance': {

'tracking\_enabled': True,

'adaptation\_enabled': False # Disabled until Step 6

}

}

# Load from file if exists, otherwise use defaults

if os.path.exists(self.config\_path):

with open(self.config\_path, 'r') as f:

file\_config = yaml.safe\_load(f)

return {\*\*default\_config, \*\*file\_config}

return default\_config

## **5. Testing Strategy**

### **5.1 Evolution Step Testing**

# tests/test\_evolution\_steps.py

class EvolutionStepTest:

"""Base class for testing each evolution step"""

def \_\_init\_\_(self, test\_data\_path: str):

self.test\_data = self.\_load\_test\_data(test\_data\_path)

self.baseline\_results = None

def test\_backward\_compatibility(self):

"""Ensure new version produces same results as previous"""

for test\_case in self.test\_data:

old\_result = self.\_run\_previous\_version(test\_case)

new\_result = self.\_run\_current\_version(test\_case)

assert self.\_results\_equivalent(old\_result, new\_result)

def test\_new\_functionality(self):

"""Test that new functionality works as expected"""

# Override in specific step tests

pass

def test\_compliance\_preservation(self):

"""Ensure compliance guarantees are maintained"""

for test\_case in self.test\_data:

result = self.\_run\_current\_version(test\_case)

assert self.\_validate\_compliance(result)

def test\_performance\_acceptable(self):

"""Ensure performance hasn't degraded significantly"""

performance\_data = []

for test\_case in self.test\_data:

start\_time = time.time()

result = self.\_run\_current\_version(test\_case)

processing\_time = time.time() - start\_time

performance\_data.append(processing\_time)

avg\_time = np.mean(performance\_data)

assert avg\_time <= self.baseline\_performance \* 2.0 # Within 2x baseline

## **6. Deployment Architecture**

### **6.1 Evolution Step Deployment**

# deployment/evolution\_deployer.py

class EvolutionDeployer:

"""Manage deployment of evolution steps"""

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

self.current\_step = self.\_detect\_current\_step()

self.rollback\_data = {}

def deploy\_step(self, step\_number: int) -> bool:

"""Deploy specific evolution step"""

# Create rollback point

rollback\_id = self.\_create\_rollback\_point()

try:

# Enable new step components

self.\_enable\_step\_components(step\_number)

# Run validation tests

if self.\_validate\_step\_deployment(step\_number):

self.\_commit\_deployment(step\_number)

return True

else:

self.\_rollback\_deployment(rollback\_id)

return False

except Exception as e:

self.\_rollback\_deployment(rollback\_id)

raise

def rollback\_to\_step(self, step\_number: int) -> bool:

"""Rollback to previous evolution step"""

if step\_number >= self.current\_step:

return False

# Disable components from later steps

for step in range(step\_number + 1, self.current\_step + 1):

self.\_disable\_step\_components(step)

self.current\_step = step\_number

return True

## **7. Monitoring and Observability**

### **7.1 Simple Monitoring**

# monitoring/simple\_monitor.py

class SimpleMonitor:

"""Basic monitoring for MVP evolution"""

def \_\_init\_\_(self, log\_file: str = "system\_monitor.log"):

self.log\_file = log\_file

self.metrics = {

'processing\_count': 0,

'success\_count': 0,

'error\_count': 0,

'average\_processing\_time': 0,

'step\_performance': {}

}

def log\_processing\_event(self,

event\_type: str,

processing\_state: ProcessingState,

additional\_data: Dict = None):

"""Log processing event for monitoring"""

event\_data = {

'timestamp': datetime.now().isoformat(),

'event\_type': event\_type,

'run\_id': processing\_state.run\_id,

'current\_step': processing\_state.current\_step,

'processing\_time': time.time() - processing\_state.start\_time,

'step\_count': len(processing\_state.step\_results),

'error\_count': len(processing\_state.errors)

}

if additional\_data:

event\_data.update(additional\_data)

# Log to file

with open(self.log\_file, 'a') as f:

f.write(json.dumps(event\_data) + '\n')

# Update metrics

self.\_update\_metrics(event\_data)

def get\_health\_status(self) -> Dict:

"""Get current system health status"""

success\_rate = self.metrics['success\_count'] / max(self.metrics['processing\_count'], 1)

return {

'status': 'healthy' if success\_rate > 0.8 else 'degraded',

'success\_rate': success\_rate,

'average\_processing\_time': self.metrics['average\_processing\_time'],

'total\_processed': self.metrics['processing\_count'],

'error\_rate': self.metrics['error\_count'] / max(self.metrics['processing\_count'], 1)

}

This design document provides the complete technical architecture for evolving your Phase 1 script into an agent-based system through safe, incremental steps that maintain backward compatibility and prove value at each stage.