SPL Technical Architecture v3.1

Implementation Guide for Three-Layer Hierarchical Agent Systems

Author: Pamela Cuce
Email: pamela@dasein.works
Phone: 914-240-3564
Version: 3.1 (Corrected)
Date: December 3, 2025

Architecture Overview

SPL implements a three-layer hierarchical decision system where each layer can suppress upper layers based on cost, confidence, and safety constraints.

```
┌─────────────────────────────────────────────────┐
│ INPUT (User Request) │
└────────────────────┬────────────────────────────┘
↓
┌─────────────────────────────────────────────────┐
│ LAYER 0: REACTIVE (Validation) │
├─────────────────────────────────────────────────┤
│ - Format validation │
│ - Permission checks │
│ - Rate limiting │
│ - Blocklist/allowlist │
│ Cost: $0 | Speed: <1ms │
│ Can suppress: Layer 1, 2 │
└────────────────────┬────────────────────────────┘
↓ (if passes)
┌─────────────────────────────────────────────────┐
│ LAYER 1: TACTICAL (Patterns) │
├─────────────────────────────────────────────────┤
│ - Pattern matching (regex) │
│ - Rule engine execution │
│ - Cache lookup │
│ - Learned patterns (>0.85 confidence) │
│ Cost: $0.0001 | Speed: <10ms │
│ Can suppress: Layer 2 │
```

```
└──────────────────────┬──────────────────────────────┘
↓ (if no match)
┌─────────────────────────────────────────────────────┐
│ LAYER 2: DELIBERATIVE (LLM) │
├─────────────────────────────────────────────────────┤
│ - Full LLM reasoning │
│ - Context processing │
│ - Complex analysis │
│ - Pattern learning │
│ Cost: $0.01+ | Speed: 100-500ms │
└──────────────────────┬──────────────────────────────┘
↓
┌─────────────────────────────────────────────────────┐
│ OUTPUT (Decision + Metadata) │
└─────────────────────────────────────────────────────┘
```

Layer 0: Reactive Layer

Purpose
Fast, deterministic validation. Rejects invalid inputs before processing.

Layer 0 Reactive Validation

Purpose: Zero-cost structural validation before expensive operations

This layer acts as first-pass filter to reject invalid requests immediately

class Layer0Reactive:
"""
Layer 0 handles all structural validation:
- Format checking (is input well-formed?)
- Permission verification (is user allowed?)
- Rate limiting (has user exceeded quotas?)
- Blocklist/allowlist enforcement (is user authorized?)

```
### Implementation

All operations here are O(1) or O(log n), guaranteed <1ms
Cost is always $0 - no API calls or expensive operations
"""

def __init__(self):
    # blocklist: set of user IDs that are blocked (fast O(1) lookup)
    self.blocklist = set()
    # allowlist: if non-empty, only these users are allowed
    self.allowlist = set()
    # rate_limits: dict tracking request counts per user per minute
    self.rate_limits = {}

def validate(self, request):
    """
    Validate request structure

    Args:
        request: dict with 'user_id' and 'content' keys

    Returns:
        tuple: (is_valid: bool, error_message: str or None)

    Checks:
    - Is request a dict? (not None, string, list, etc.)
    - Does it have 'user_id'? (required for authentication)
    - Does it have 'content'? (required for processing)
    """
    # Type check: request must be dict
    if not isinstance(request, dict):
        return False, "Invalid request format"

    # Check for required user_id field
    if 'user_id' not in request:
        return False, "Missing user_id"
```

```
    # Check for required content field
    if 'content' not in request:
        return False, "Missing content"

    # All checks passed
    return True, None

def check_permissions(self, user_id):
    """
    Check if user has permission to use system

    Args:
        user_id: string identifier for user

    Returns:
        tuple: (has_permission: bool, error_message: str or None)

    Logic:
    1. If user in blocklist → REJECT (explicitly denied)
    2. If allowlist exists and user not in it → REJECT (only specific users)
    3. Otherwise → ALLOW (user authorized)
    """
    # Blocklist check: if user explicitly blocked, deny immediately
    if user_id in self.blocklist:
        return False, "User blocked"

    # Allowlist check: if allowlist configured and user not in it, deny
    if self.allowlist and user_id not in self.allowlist:
        return False, "User not allowed"

    # User passed all permission checks
    return True, None

def check_rate_limit(self, user_id):
    """
    Check if user has exceeded rate limit

    Args:
```

```
        user_id: string identifier for user

    Returns:
        tuple: (within_limit: bool, error_message: str or None)

    Implementation:
    - Track requests per user per minute
    - Limit: 100 requests/minute (configurable)
    - Time window: 60 seconds (1 minute)
    - After 60s elapses, counter resets
    """
    limit = 100  # Max 100 requests per minute per user

    # Initialize tracking for new user
    if user_id not in self.rate_limits:
        self.rate_limits[user_id] = {
            'count': 0,
            'timestamp': __import__('time').time()
        }

    # Get current tracking entry
    entry = self.rate_limits[user_id]
    # Calculate elapsed time since window started
    elapsed = __import__('time').time() - entry['timestamp']

    # If more than 60 seconds passed, reset window
    if elapsed > 60:
        entry['count'] = 0
        entry['timestamp'] = __import__('time').time()

    # Check if user hit rate limit
    if entry['count'] >= limit:
        return False, f"Rate limit exceeded ({limit}/min)"

    # User within limit - increment counter and allow
    entry['count'] += 1
    return True, None

def execute(self, request):
```

```
"""
Execute all Layer 0 checks in sequence

Args:
    request: dict with user_id and content

Returns:
    dict with keys:
    - halt: bool (True = stop processing)
    - reason: str error message if halted
    - cost: float (always 0.0 for Layer 0)

Execution flow:
1. Validate structure
2. Check permissions
3. Check rate limit

If any check fails, return halt=True immediately
"""

# Check 1: Structure validation
valid, error = self.validate(request)
if not valid:
    return {'halt': True, 'reason': error, 'cost': 0.0}

# Check 2: Permission check
valid, error = self.check_permissions(request['user_id'])
if not valid:
    return {'halt': True, 'reason': error, 'cost': 0.0}

# Check 3: Rate limit check
valid, error = self.check_rate_limit(request['user_id'])
if not valid:
    return {'halt': True, 'reason': error, 'cost': 0.0}

# All checks passed - safe to proceed to Layer 1
return {'halt': False, 'reason': None, 'cost': 0.0}
```

Layer 1: Tactical Layer

Purpose
Pattern matching and simple rules. Suppress Layer 2 if high-confidence match found.

Layer 1 Tactical Pattern Matching

Purpose: Match against learned patterns before calling expensive LLM

This layer can SUPPRESS Layer 2 (LLM) if confidence is high enough

```
import re

class Layer1Tactical:
"""
Layer 1 handles pattern-based decision-making:
- Pattern matching using regex (extremely fast)
- Rule engine evaluation (simple if-then logic)
- Cache lookup (avoid reprocessing same content)
- High-confidence pattern suppression (skip LLM if confident)


Cost: $0.0001 per match (just cache lookup)
Speed: <10ms typical
Confidence threshold: 0.85 (configurable)
"""

def __init__(self):
    # patterns: dict mapping pattern names to their configurations
    # Example: {'urgent': {'regex': r'urgent|asap', 'category': 'urgent', 'confidence': 0.95}}
    self.patterns = {}
    # confidence_threshold: minimum confidence needed to suppress Layer 2
    self.confidence_threshold = 0.85
```

```
    # cache: dict mapping content hashes to previous results
    self.cache = {}

def add_pattern(self, name, regex, category, confidence=0.92):
    """
    Register new pattern for matching

    Args:
        name: string identifier for pattern (e.g., 'urgent')
        regex: compiled or raw regex pattern (e.g., r'urgent|asap')
        category: string category to assign if pattern matches
        confidence: float 0.0-1.0 indicating pattern accuracy
    """
    self.patterns[name] = {
        'regex': regex,
        'category': category,
        'confidence': confidence
    }

def match_patterns(self, content):
    """
    Try to match content against all known patterns

    Args:
        content: string text to classify

    Returns:
        dict with:
        - matched: bool (True if pattern found)
        - pattern: str pattern name that matched
        - category: str assigned category
        - confidence: float confidence of match
        - method: str 'pattern' (for tracking)
        - cost: float 0.0 (no API calls)

    Algorithm:
    1. Convert content to lowercase (case-insensitive)
    2. For each registered pattern:
       a. Check if pattern's regex matches
```

```
      b. If matches AND confidence >= threshold:
        - Return match immediately
    3. If no pattern matches, return matched=False
    """
    # Convert to lowercase for case-insensitive matching
    content_lower = content.lower()

    # Iterate through all registered patterns
    for pattern_name, pattern_config in self.patterns.items():
        regex = pattern_config['regex']

        # Try to match pattern against content
        if re.search(regex, content_lower):
            confidence = pattern_config['confidence']

            # Check if confidence high enough to suppress LLM
            if confidence >= self.confidence_threshold:
                # Return successful match
                return {
                    'matched': True,
                    'pattern': pattern_name,
                    'category': pattern_config['category'],
                    'confidence': confidence,
                    'method': 'pattern',
                    'cost': 0.0
                }

    # No patterns matched
    return {'matched': False}

def check_cache(self, content_hash):
    """
    Check if we've already processed this exact content

    Args:
        content_hash: int hash of the content string

    Returns:
        dict with:
```

```
            - matched: bool (True if found in cache)
            - (plus all fields from original result if cached)

        Purpose:
        - Avoid re-processing identical content
        - Reuse previous results without re-running LLM
        - Reduce latency on duplicate requests
        """
        if content_hash in self.cache:
            cached = self.cache[content_hash]
            # Mark as coming from cache for tracking
            cached['method'] = 'cache'
            cached['cost'] = 0.0
            return {'matched': True, **cached}

        # Not in cache
        return {'matched': False}

    def execute(self, request):
        """
        Execute Layer 1 matching in optimal order

        Args:
            request: dict with 'content' key

        Returns:
            dict with:
            - matched: bool (True if pattern/cache found result)
            - category: str assigned category (if matched)
            - confidence: float confidence (if matched)
            - method: str 'cache' | 'pattern'
            - cost: float 0.0 (Layer 1 never costs money)

        Execution order:
        1. Cache check first (fastest: hash lookup O(1))
        2. Pattern matching second (fast: regex matching)
        3. If neither works, return to Layer 2 (LLM call)
        """
        content = request['content']
```

```
    # Create hash of content for cache lookup
    content_hash = hash(content)

    # Strategy 1: Check cache first (fastest path)
    cache_result = self.check_cache(content_hash)
    if cache_result['matched']:
        # Cache hit! Return immediately
        return cache_result

    # Strategy 2: Try pattern matching (second fastest path)
    pattern_result = self.match_patterns(content)
    if pattern_result['matched']:
        # Pattern matched! Cache this result for future
        self.cache[content_hash] = pattern_result
        return pattern_result

    # Strategy 3: No cache or pattern match found
    return {'matched': False, 'cost': 0.0}
```

Layer 2: Deliberative Layer

Purpose
Complex reasoning using LLM. Called only when Layer 1 cannot resolve.

Implementation

Layer 2 Deliberative LLM Reasoning

Purpose: Call LLM for complex reasoning when simpler layers can't decide

```
class Layer2Deliberative:
"""
Layer 2 handles complex decision-making:
- Full LLM reasoning with context
```

```
- Novel decision-making (cases Layers 0-1 couldn't handle)
- Pattern learning (extract new patterns from LLM decisions)
- Context integration (use conversation history)

Cost: $0.01-0.05 per call (expensive!)
Speed: 100-500ms typical
Confidence: 85-95% typical

This layer only reached when:
1. Layer 0 passed validation
2. Layer 1 found no high-confidence pattern match
"""

def __init__(self, llm_model='claude-3-5-sonnet-20241022', api_key=None):
    """
    Initialize Layer 2 with LLM configuration

    Args:
        llm_model: string LLM model ID
        api_key: string API key (optional)
    """
    self.llm_model = llm_model
    self.api_key = api_key
    # cost_per_call: average cost of one LLM call
    self.cost_per_call = 0.01
    # learned_patterns: patterns discovered by this agent
    self.learned_patterns = {}

def call_llm(self, prompt, context=None):
    """
    Call LLM for reasoning and categorization

    Args:
        prompt: string question/task for LLM
        context: optional dict with conversation history

    Returns:
        string: LLM's response/decision
```

```
        """
        # In production, this would call actual Anthropic API:
        # client = anthropic.Anthropic(api_key=self.api_key)
        # response = client.messages.create(...)

        # For demo, use heuristic rules
        prompt_lower = prompt.lower()

        if 'urgent' in prompt_lower:
            return 'urgent'
        elif 'invoice' in prompt_lower:
            return 'billing'
        elif 'unsubscribe' in prompt_lower:
            return 'spam'
        else:
            return 'other'

    def learn_pattern(self, content, category, confidence):
        """
        Extract and save new pattern from LLM decision

        Args:
            content: string original text that was categorized
            category: string category assigned by LLM
            confidence: float 0.0-1.0 confidence in decision

        Purpose:
        - Save high-confidence patterns to Layer 1
        - Next time we see similar content, Layer 1 handles it
        - Gradually shift expensive decisions to cheap layer
        """
        # Only learn from high-confidence decisions
        if confidence >= 0.90:
            # Extract key features from content
            words = content.lower().split()
            if words:
                # Use first word as pattern keyword
                key_word = words
```

```
        # Save pattern
        self.learned_patterns[category] = {
            'keyword': key_word,
            'confidence': confidence,
            'learned_at': __import__('datetime').datetime.now().isoformat()
        }

def execute(self, request):
    """
    Execute Layer 2 LLM reasoning

    Args:
        request: dict with 'content' key

    Returns:
        dict with:
        - category: str assigned category
        - method: str 'llm'
        - confidence: float 0.85-0.95 typical
        - cost: float $0.01 (the expensive part!)
        - layer: int 2 (for tracking)
    """

    content = request['content']
    # Create prompt for LLM
    prompt = f"Categorize this text: {content}"

    # Call LLM (expensive operation!)
    category = self.call_llm(prompt)

    # Learn pattern if high confidence
    self.learn_pattern(content, category, 0.92)

    # Return decision with full metadata
    return {
        'category': category,
        'method': 'llm',
        'confidence': 0.92,
        'cost': self.cost_per_call,
```

'layer': 2
}

Integration: Three-Layer Pipeline

SPL Agent: Full Three-Layer Pipeline

Orchestrates all three layers in correct sequence

```
class SPLAgent:
"""
SPL Agent integrates all three layers:
- Layer 0: Validation (fast, free)
- Layer 1: Patterns (fast, cheap)
- Layer 2: LLM (slow, expensive)

Execution flow:
request → Layer 0 (halt?) → Layer 1 (match?) → Layer 2 (LLM) → result

Cost optimization:
- Most requests stop at Layer 0 or 1
- Only ~5-15% reach expensive Layer 2
- This achieves 10-50x cost reduction
"""

def __init__(self):
    # Initialize all three layers
    self.layer0 = Layer0Reactive()
    self.layer1 = Layer1Tactical()
    self.layer2 = Layer2Deliberative()

    # Pre-load common patterns (bootstrapping)
    self.layer1.add_pattern(
```

```
        'urgent_keyword',
        r'urgent|asap|immediately',
        'urgent',
        0.92
    )

def process(self, request):
    """
    Process request through all three layers

    Args:
        request: dict with 'user_id' and 'content'

    Returns:
        dict with result and metadata

    Execution stages:
    ===============
    Stage 1: Layer 0 Validation
        - If invalid → REJECT ($0 cost)
        - If valid → continue to Layer 1

    Stage 2: Layer 1 Pattern Matching
        - If pattern matches → RETURN ($0 cost)
        - If no match → continue to Layer 2

    Stage 3: Layer 2 LLM
        - Full reasoning → RETURN ($0.01 cost)
    """

    # STAGE 1: Layer 0 Validation
    layer0_result = self.layer0.execute(request)
    if layer0_result['halt']:
        # Request rejected at validation stage
        return {
            'result': None,
            'reason': layer0_result['reason'],
            'cost': 0.0,
            'layer': 0
```

```
        }

    # STAGE 2: Layer 1 Pattern Matching
    layer1_result = self.layer1.execute(request)
    if layer1_result['matched']:
        # Pattern matched - suppress Layer 2!
        return {
            'result': layer1_result['category'],
            'confidence': layer1_result['confidence'],
            'method': layer1_result['method'],
            'cost': layer1_result['cost'],
            'layer': 1,
            'suppressed': True
        }

    # STAGE 3: Layer 2 LLM Reasoning
    layer2_result = self.layer2.execute(request)

    return {
        'result': layer2_result['category'],
        'confidence': layer2_result['confidence'],
        'method': layer2_result['method'],
        'cost': layer2_result['cost'],
        'layer': 2
    }
```

Cost Optimization Strategies

Strategy 1: Pattern Preloading

***Load high-confidence patterns at start**

```
def preload_patterns(agent):
"""Pre-populate agent with high-accuracy patterns"""
```

patterns = [
('urgent', r'urgent|asap|emergency', 'urgent', 0.95),
('billing', r'invoice|payment|bill|receipt', 'billing', 0.93),
('spam', r'unsubscribe|viagra|lottery|winner', 'spam', 0.98),
]

for name, regex, category, conf in patterns:
    agent.layer1.add_pattern(name, regex, category, conf)

Strategy 2: Confidence Threshold Tuning

***Adjust threshold based on domain**

def set_confidence_threshold(agent, domain):
"""Configure confidence threshold for domain"""
if domain == 'email_filtering':
# High cost of false positives
agent.layer1.confidence_threshold = 0.90 # Strict
elif domain == 'content_moderation':
# Balance between cost and accuracy
agent.layer1.confidence_threshold = 0.85 # Moderate
else:
# Prioritize cost savings
agent.layer1.confidence_threshold = 0.75 # Permissive

Strategy 3: Timeout Protection

***Prevent Layer 2 from running indefinitely**

```
def execute_with_timeout(agent, request, timeout_ms=500):
"""Execute Layer 2 with timeout protection"""

import time

start = time.time()
result = agent.layer2.execute(request)
elapsed = (time.time() - start) * 1000

if elapsed > timeout_ms:
    # Timeout occurred - fall back to Layer 1
    return agent.layer1.execute(request)

return result
```

Performance Benchmarks

Single Request Processing

Layer	Operation	Time	Cost
0	Validation	<1ms	$0
1	Pattern match (hit)	2ms	$0
1	Pattern match (miss)	2ms	$0
2	LLM call	200ms	$0.01

1000 Email Batch

Scenario	Layer 0	Layer 1	Layer 2	Total Cost	Avg/Email
No optimization	0	0	1000	$10.00	$0.01
50% patterns	0	500	500	$5.00	$0.005
80% patterns	10	800	190	$1.90	$0.0019
95% patterns	50	950	0	$0.00	$0.0000

Monitoring & Observability

***Track performance metrics for optimization**

```
class SPLMonitor:
"""Collect metrics for performance monitoring"""

def __init__(self):
    # Metrics dictionary
    self.metrics = {
        'layer0_halts': 0,
        'layer1_hits': 0,
        'layer2_calls': 0,
        'total_cost': 0.0,
        'total_requests': 0
    }

def record_execution(self, result):
    """Record metrics from one execution"""
    self.metrics['total_requests'] += 1
    self.metrics['total_cost'] += result['cost']

    if result['layer'] == 0:
        self.metrics['layer0_halts'] += 1
    elif result['layer'] == 1:
        self.metrics['layer1_hits'] += 1
    elif result['layer'] == 2:
        self.metrics['layer2_calls'] += 1
```

```
def report(self):
    """Generate performance report"""
    total = self.metrics['total_requests']
    return {
        'total_requests': total,
        'layer0_halt_rate': self.metrics['layer0_halts'] / total if total > 0 else 0,
        'layer1_hit_rate': self.metrics['layer1_hits'] / total if total > 0 else 0,
        'layer2_call_rate': self.metrics['layer2_calls'] / total if total > 0 else 0,
        'total_cost': self.metrics['total_cost'],
        'avg_cost_per_request': self.metrics['total_cost'] / total if total > 0 else 0
    }
```

Deployment Checklist

- [ ] Layer 0 validation rules configured
- [ ] Layer 1 patterns loaded and tested
- [ ] Layer 2 LLM API credentials configured
- [ ] Rate limiting configured
- [ ] Monitoring/logging enabled
- [ ] Cost tracking enabled
- [ ] Timeout protection in place
- [ ] Error handling for all layers
- [ ] Fallback mechanisms configured
- [ ] Performance baselines established

Pamela Cuce | pamcuce@alum.vassar.edu | Version 3.1 (Corrected) | December 3, 2025

Implementation

```
class Layer1Tactical:
def init(self):
self.patterns = {}
self.confidence_threshold = 0.85
self.cache = {}

def add_pattern(self, name, regex, category, confidence=0.92):
    """Register a pattern"""
    self.patterns[name] = {
        'regex': regex,
        'category': category,
        'confidence': confidence
    }

def match_patterns(self, content):
    """Try to match against known patterns"""
    content_lower = content.lower()

    for pattern_name, pattern_config in self.patterns.items():
        regex = pattern_config['regex']

        if re.search(regex, content_lower):
            confidence = pattern_config['confidence']
```

```
        if confidence >= self.confidence_threshold:
            return {
                'matched': True,
                'pattern': pattern_name,
                'category': pattern_config['category'],
                'confidence': confidence,
                'method': 'pattern',
                'cost': 0.0
            }

    return {'matched': False}

def check_cache(self, content_hash):
    """Check if result already cached"""
    if content_hash in self.cache:
        cached = self.cache[content_hash]
        cached['method'] = 'cache'
        cached['cost'] = 0.0
        return {'matched': True, **cached}

    return {'matched': False}

def execute(self, request):
    """Execute Layer 1 matching"""
    content = request['content']
    content_hash = hash(content)

    # Try cache first
    cache_result = self.check_cache(content_hash)
    if cache_result['matched']:
        return cache_result

    # Try pattern matching
    pattern_result = self.match_patterns(content)
    if pattern_result['matched']:
        # Cache this result
        self.cache[content_hash] = pattern_result
        return pattern_result
```

```
return {'matched': False, 'cost': 0.0}
```