

Architecture Deep Dive

This document explains the internal design of the Evolutionary Alignment Framework.

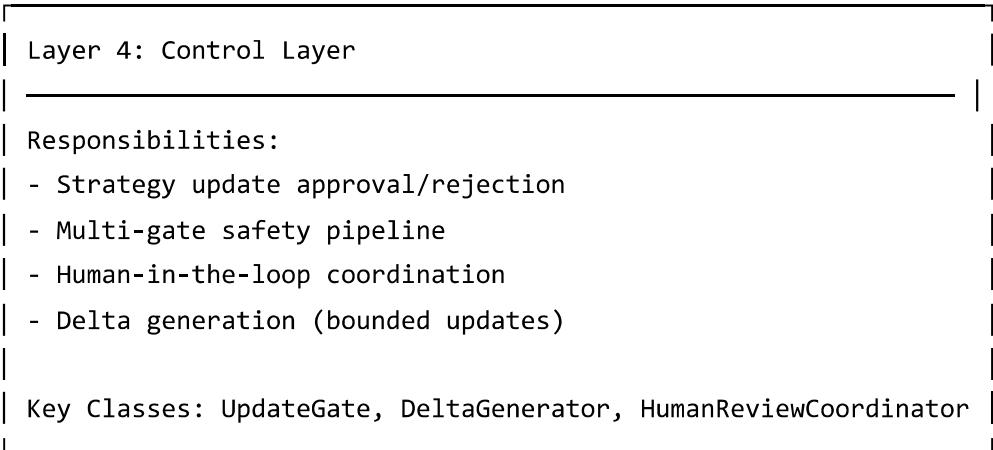
Table of Contents

1. [System Overview](#)
2. [Component Details](#)
3. [Data Flow](#)
4. [Design Decisions](#)
5. [Performance Considerations](#)

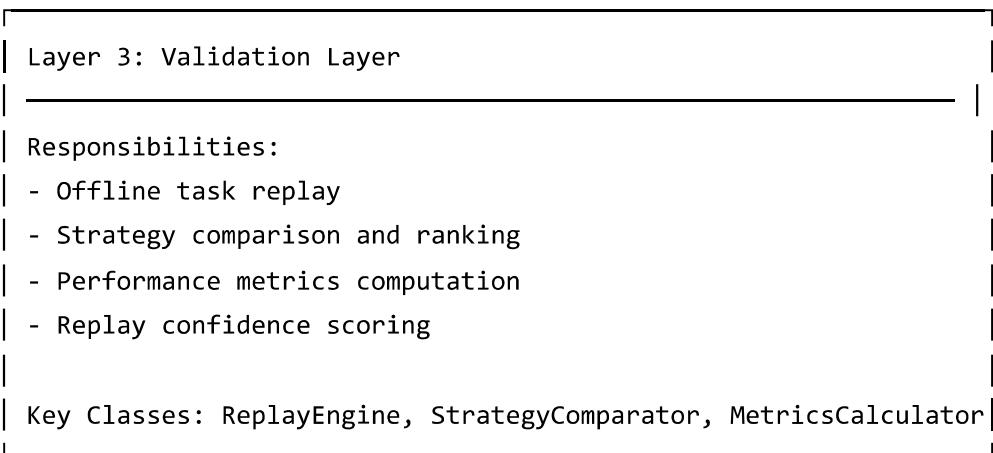
System Overview

Layered Architecture

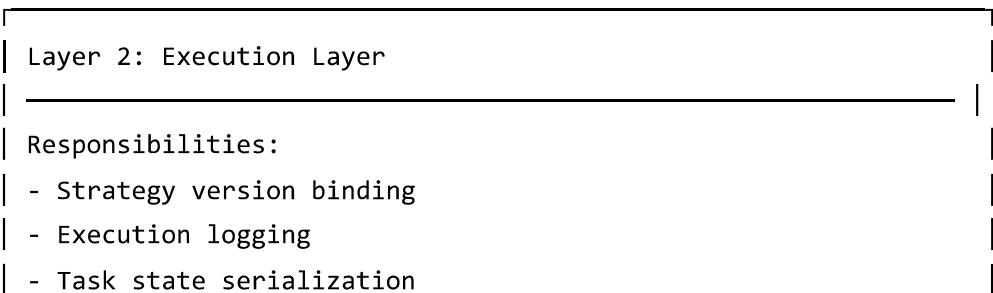
The framework uses a 4-layer architecture for separation of concerns:



⇓



⇓



- Performance tracking
Key Classes: ExecutionManager, TaskLogger, StateSerializer

⇓

Layer 1: Strategy Layer
Responsibilities:
- Strategy definition and parsing
- Version control integration
- Multi-branch management
- Strategy merging and conflict resolution
Key Classes: Strategy, StrategyManager, BranchManager

⇓

Your AI Agent Core

Why This Design?

Modularity:

- Each layer can be used independently
- Easy to test in isolation
- Swap implementations without affecting other layers

Extensibility:

- Add new gate types without touching replay logic
- Implement custom replay strategies

- Extend strategy schema without code changes

Observability:

- Clear boundaries make debugging easier
- Each layer has explicit inputs/outputs
- Metrics at every level

Component Details

1. Strategy Layer

Strategy Definition Schema

Strategies are defined in YAML for human readability:

```
# Example: strategies/email-handler-v2.yaml

metadata:
  name: "email-handler"
  version: "2.0.0"
  parent_version: "1.5.0"
  created_at: "2026-02-09T10:30:00Z"
  created_by: "automated-learning"
  description: "Optimized email handling with reduced confirmation overhead"

parameters:
  # Core decision parameters (0.0 - 1.0 scale)
  response_speed: 0.75      # How quickly to respond
  user_confirmation: 0.60    # How often to ask permission
  autonomy_level: 0.70      # Overall autonomous decision-making

  # Domain-specific parameters
  priority_detection_threshold: 0.80
  auto_categorization_confidence: 0.65

constraints:
  # Hard constraints (never violated)
  must_confirm_before:
    - "sending_emails_to_clients"
    - "deleting_important_messages"
    - "modifying_calendar_entries"

  never_execute:
    - "forward_to_external_addresses"
    - "change_account_settings"
```

```
# Soft constraints (can be overridden with high confidence)
prefer_confirm_for:
  - "bulk_operations"
  - "financial_actions"

evolution_metadata:
  # Control how this strategy can evolve
  max_parameter_delta: 0.10      # Max change per update
  cooldown_period_days: 7        # Min time between updates
  requires_human_review: false # Auto-approve if passes gates

  # Branch-specific
  branch_type: "production"    # production | experimental | archived
  merged_from: null            # If merged from a branch

change_log:
  - version: "2.0.0"
    date: "2026-02-09T10:30:00Z"
    changes:
      - parameter: "response_speed"
        from: 0.70
        to: 0.75
        reason: "Replay showed 12% reduction in user manual corrections"
      - parameter: "user_confirmation"
        from: 0.65
        to: 0.60
        reason: "User approval rate at 98%, safe to reduce confirmation overhead"

  - version: "1.5.0"
    date: "2026-01-15T14:20:00Z"
    changes:
      - parameter: "autonomy_level"
```

from: 0.65

to: 0.70

reason: "Gradual autonomy increase after stable performance"

Strategy Class Implementation

```
from dataclasses import dataclass
from typing import Dict, List, Optional
from datetime import datetime, timedelta
import yaml

@dataclass
class StrategyConstraint:
    """Represents a hard or soft constraint"""
    type: str # "must_confirm" | "never_execute" | "prefer_confirm"
    actions: List[str]

@dataclass
class EvolutionConfig:
    """Controls how strategy can evolve"""
    max_parameter_delta: float
    cooldown_period: timedelta
    requires_human_review: bool
    branch_type: str

class Strategy:
    """
    Represents a versioned strategy configuration.

    Immutable once created (new versions are new instances).
    """

    def __init__(self, yaml_path: str):
        with open(yaml_path) as f:
            data = yaml.safe_load(f)
```

```
self.name = data['metadata']['name']
self.version = data['metadata']['version']
self.parent_version = data['metadata'].get('parent_version')
self.created_at = datetime.fromisoformat(data['metadata']['created_at'])

self.parameters = data['parameters']
self.constraints = self._parse_constraints(data['constraints'])
self.evolution_config = self._parse_evolution_config(
    data['evolution_metadata']
)
self.change_log = data.get('change_log', [])

def get_parameter(self, name: str, default=None) -> float:
    """Get a strategy parameter value"""
    return self.parameters.get(name, default)

def can_execute(self, action: str) -> bool:
    """Check if an action is allowed under this strategy"""
    # Check hard constraints
    for constraint in self.constraints:
        if constraint.type == "never_execute":
            if action in constraint.actions:
                return False
    return True

def requires_confirmation(self, action: str, confidence: float = 0.0) -> bool:
    """Determine if user confirmation is needed"""
    # Hard requirements
    for constraint in self.constraints:
        if constraint.type == "must_confirm":
            if action in constraint.actions:
```

```
        return True

# Soft requirements (can override with high confidence)
for constraint in self.constraints:
    if constraint.type == "prefer_confirm":
        if action in constraint.actions:
            threshold = self.get_parameter('user_confirmation', 0.8)
            return confidence < threshold

    # Default based on strategy parameter
    return confidence < self.get_parameter('user_confirmation', 0.8)

def validate_evolution(self, new_strategy: 'Strategy') -> List[str]:
    """
    Validate that evolution from this strategy to new_strategy
    respects evolution constraints.

    Returns list of violations (empty if valid).
    """
    violations = []

    # Check cooldown period
    time_since_creation = datetime.now() - self.created_at
    if time_since_creation < self.evolution_config.cooldown_period:
        violations.append(
            f"Cooldown period not met: "
            f"{time_since_creation} < {self.evolution_config.cooldown_period}"
        )

    # Check parameter delta limits
    for param, new_value in new_strategy.parameters.items():
        if param in self.parameters:
```

```
        old_value = self.parameters[param]
        delta = abs(new_value - old_value)

        if delta > self.evolution_config.max_parameter_delta:
            violations.append(
                f"Parameter '{param}' change too large: "
                f"{delta} > {self.evolution_config.max_parameter_delta}"
            )

    return violations
```

```
def create_branch(self, branch_name: str, delta: Dict) -> 'Strategy':
```

```
    """
```

```
    Create a new strategy branch with specified changes.
```

Args:

```
    branch_name: Name for the new branch
    delta: Parameter changes to apply
```

Returns:

```
    New Strategy instance representing the branch
```

```
    """
```

```
    new_params = self.parameters.copy()
    new_params.update(delta)
```

```
    # Create new version number
    major, minor, patch = map(int, self.version.split('.'))
    new_version = f'{major}.{minor + 1}.0'
```

```
    # Build new strategy data
```

```
    new_data = {
        'metadata': {
```

```
'name': self.name,
'version': new_version,
'parent_version': self.version,
'created_at': datetime.now().isoformat(),
'created_by': 'branch_creation',
'description': f"Branch: {branch_name}"
},
'parameters': new_params,
'constraints': self._serialize_constraints(),
'evolution_metadata': {
    **self.evolution_config.__dict__,
    'branch_type': 'experimental'
},
'change_log': self.change_log + [
    {
        'version': new_version,
        'date': datetime.now().isoformat(),
        'changes': [
            {
                'parameter': param,
                'from': self.parameters.get(param),
                'to': value,
                'reason': f'Branch {branch_name} exploration'
            }
            for param, value in delta.items()
        ]
    }
]
}

# Write to temporary file and create new instance
temp_path = f"/tmp/strategy_{new_version}.yaml"
with open(temp_path, 'w') as f:
    yaml.dump(new_data, f)
```

```
return Strategy(temp_path)
```

Branch Manager

```
from typing import List, Optional
from enum import Enum

class BranchStatus(Enum):
    ACTIVE = "active"          # Currently being evaluated
    MERGED = "merged"          # Merged into production
    ARCHIVED = "archived"      # Evaluation complete, not merged
    FAILED = "failed"          # Failed validation

class StrategyBranch:
    """Represents a strategy evolution branch"""

    def __init__(
        self,
        strategy: Strategy,
        parent: Strategy,
        branch_name: str,
        rationale: str
    ):
        self.strategy = strategy
        self.parent = parent
        self.branch_name = branch_name
        self.rationale = rationale
        self.status = BranchStatus.ACTIVE
        self.created_at = datetime.now()
        self.evaluation_results = None

    def mark_merged(self):
        self.status = BranchStatus.MERGED
```

```
def mark_archived(self, reason: str):
    self.status = BranchStatus.ARCHIVED
    self.archive_reason = reason

class BranchManager:
    """
    Manages multiple strategy branches and their lifecycle.

    Implements the multi-branch evolution pattern:
    1. Create multiple candidate branches
    2. Evaluate in parallel via offline replay
    3. Select best performing branch
    4. Merge or archive
    """

    def __init__(self, production_strategy: Strategy):
        self.production = production_strategy
        self.branches: List[StrategyBranch] = []

    def create_exploration_branches(
        self,
        learning_signal: Dict
    ) -> List[StrategyBranch]:
        """
        Create multiple branches exploring different evolution directions.

        Args:
            learning_signal: Observations suggesting strategy improvement

        Returns:
            List of created branches
        """

```

```
"""

branches = []

# Branch 1: Greedy (optimize for immediate metric)
greedy_delta = self._compute_greedy_delta(learning_signal)
greedy_branch = self.production.create_branch(
    "greedy_optimization",
    greedy_delta
)
branches.append(StrategyBranch(
    strategy=greedy_branch,
    parent=self.production,
    branch_name="greedy",
    rationale="Maximize immediate performance metric"
))

# Branch 2: Conservative (small safe changes)
conservative_delta = self._compute_conservative_delta(learning_signal)
conservative_branch = self.production.create_branch(
    "conservative_improvement",
    conservative_delta
)
branches.append(StrategyBranch(
    strategy=conservative_branch,
    parent=self.production,
    branch_name="conservative",
    rationale="Incremental safe improvement"
))

# Branch 3: Exploratory (try novel approaches)
exploratory_delta = self._compute_exploratory_delta(learning_signal)
exploratory_branch = self.production.create_branch(
```

```
        "exploratory",
        exploratory_delta
    )
branches.append(StrategyBranch(
    strategy=exploratory_branch,
    parent=self.production,
    branch_name="exploratory",
    rationale="Test alternative approach"
))
self.branches.extend(branches)
return branches
```

```
def select_best_branch(
    self,
    evaluation_results: Dict[str, float]
) -> Optional[StrategyBranch]:
    """
    Select the best performing branch based on evaluation results.
    """
    Select the best performing branch based on evaluation results.
```

Args:

```
    evaluation_results: {branch_name: performance_score}
```

Returns:

```
    Best branch or None if no improvement
    """
    Compute baseline (current production performance)
```

```
    baseline_score = evaluation_results.get('production', 0.0)
```

```
    # Find branches that improve over baseline
```

```
    improvements = []
    for branch in self.branches:
```

```
if branch.status != BranchStatus.ACTIVE:
    continue

score = evaluation_results.get(branch.branch_name, 0.0)
improvement = score - baseline_score

if improvement > 0:
    improvements[branch] = improvement

if not improvements:
    return None # No improvement found

# Select branch with highest improvement
best_branch = max(improvements.keys(), key=lambda b: improvements[b])
return best_branch

def merge_branch(self, branch: StrategyBranch):
    """Merge a branch into production"""
    self.production = branch.strategy
    branch.mark_merged()

    # Archive other branches
    for other_branch in self.branches:
        if other_branch != branch and other_branch.status == BranchStatus.ACTIVE:
            other_branch.mark_archived("Other branch selected")

def _compute_greedy_delta(self, signal: Dict) -> Dict:
    """Compute parameter changes for greedy optimization"""
    # Example: If signal suggests increasing response_speed helps,
    # increase it by maximum allowed amount
    delta = {}
    max_delta = self.production.evolution_config.max_parameter_delta
```

```
if 'response_speed_correlation' in signal:
    if signal['response_speed_correlation'] > 0:
        current = self.production.get_parameter('response_speed', 0.5)
        delta['response_speed'] = min(1.0, current + max_delta)

return delta

def _compute_conservative_delta(self, signal: Dict) -> Dict:
    """Compute small, safe parameter changes"""
    delta = {}
    max_delta = self.production.evolution_config.max_parameter_delta

    # Use half of maximum allowed change
    safe_delta = max_delta / 2

    if 'response_speed_correlation' in signal:
        if signal['response_speed_correlation'] > 0:
            current = self.production.get_parameter('response_speed', 0.5)
            delta['response_speed'] = current + safe_delta

    return delta

def _compute_exploratory_delta(self, signal: Dict) -> Dict:
    """Compute changes exploring alternative directions"""
    delta = {}

    # Try opposite direction or orthogonal improvements
    # This helps escape local optima

    if 'user_confirmation_rate' in signal:
        # If users almost always approve, try reducing confirmation
```

```
if signal['user_confirmation_rate'] > 0.95:  
    current = self.production.get_parameter('user_confirmation', 0.8)  
    max_delta = self.production.evolution_config.max_parameter_delta  
    delta['user_confirmation'] = max(0.0, current - max_delta)  
  
return delta
```

2. Execution Layer

Execution Manager

```
from contextlib import contextmanager
from typing import Any, Dict

class ExecutionContext:
    """Captures all state needed to replay a task"""

    def __init__(
        self,
        task_id: str,
        task_type: str,
        strategy_version: str,
        input_data: Any,
        timestamp: datetime,
        user_context: Dict
    ):
        self.task_id = task_id
        self.task_type = task_type
        self.strategy_version = strategy_version
        self.input_data = input_data
        self.timestamp = timestamp
        self.user_context = user_context
        self.execution_trace = []

    def log_decision(self, decision_point: str, chosen_action: str, reason: str):
        """Log a decision made during execution"""
        self.execution_trace.append({
            'timestamp': datetime.now().isoformat(),
```

```
'decision_point': decision_point,
    'action': chosen_action,
    'reason': reason
})

def serialize(self) -> Dict:
    """Serialize for storage/replay"""
    return {
        'task_id': self.task_id,
        'task_type': self.task_type,
        'strategy_version': self.strategy_version,
        'input_data': self._serialize_input(self.input_data),
        'timestamp': self.timestamp.isoformat(),
        'user_context': self.user_context,
        'execution_trace': self.execution_trace
    }

@staticmethod
def deserialize(data: Dict) -> 'ExecutionContext':
    """Recreate from serialized data"""
    return ExecutionContext(
        task_id=data['task_id'],
        task_type=data['task_type'],
        strategy_version=data['strategy_version'],
        input_data=data['input_data'],
        timestamp=datetime.fromisoformat(data['timestamp']),
        user_context=data['user_context']
    )

class ExecutionManager:
    """
    Manages task execution with strategy binding and logging.

```

Key responsibilities:

1. Bind each execution to a strategy version
2. Log execution for future replay
3. Track performance metrics

"""

```
def __init__(  
    self,  
    strategy: Strategy,  
    log_storage_path: str = "./execution_logs"  
):  
    self.strategy = strategy  
    self.log_storage_path = log_storage_path  
    self.current_context: Optional[ExecutionContext] = None
```

@contextmanager

```
def execution_context(  
    self,  
    task_id: str,  
    task_type: str,  
    input_data: Any,  
    user_context: Dict = None  
):  
    """
```

Context manager for executing a task.

Usage:

```
with exec_manager.execution_context("task-123", "email", email_data):  
    result = agent.handle_email(email_data)  
    """  
# Create execution context
```

```
context = ExecutionContext(
    task_id=task_id,
    task_type=task_type,
    strategy_version=self.strategy.version,
    input_data=input_data,
    timestamp=datetime.now(),
    user_context=user_context or {}
)

self.current_context = context

try:
    yield context
finally:
    # Log execution after completion
    self._persist_execution(context)
    self.current_context = None

def log_decision(self, decision_point: str, action: str, reason: str):
    """Log a decision point during execution"""
    if self.current_context:
        self.current_context.log_decision(decision_point, action, reason)

def _persist_execution(self, context: ExecutionContext):
    """Save execution context to disk for replay"""
    import json
    import os

    # Organize by date for easy retrieval
    date_dir = context.timestamp.strftime("%Y-%m-%d")
    log_dir = os.path.join(self.log_storage_path, date_dir)
    os.makedirs(log_dir, exist_ok=True)
```

```
log_file = os.path.join(log_dir, f"{context.task_id}.json")
with open(log_file, 'w') as f:
    json.dump(context.serialize(), f, indent=2)
```

3. Validation Layer (Replay Engine)

```
class ReplayEngine:  
    """  
    Replays historical tasks under different strategies to evaluate performance.  
  
    Core challenge: Achieving high fidelity replay despite:  
    - Non-deterministic LLM outputs  
    - Time-dependent external state  
    - Stochastic processes  
    """  
  
    def __init__(  
        self,  
        execution_log_path: str,  
        cache_enabled: bool = True  
    ):  
        self.log_path = execution_log_path  
        self.cache = {} if cache_enabled else None  
  
    def replay_task(  
        self,  
        task_context: ExecutionContext,  
        strategy: Strategy,  
        agent_executor: callable  
    ) -> Dict:  
        """  
        Replay a single historical task with a different strategy.  
  
        Returns:  
        {
```

```
'success': bool,
'output': Any,
'metrics': Dict,
'confidence': float # How confident we are in replay fidelity
}

"""

# Check cache first
cache_key = (task_context.task_id, strategy.version)
if self.cache and cache_key in self.cache:
    return self.cache[cache_key]

# Reconstruct execution environment
replay_env = self._reconstruct_environment(task_context)

# Execute with new strategy
# Use fixed random seed for determinism where possible
import random
random.seed(hash(task_context.task_id))

try:
    result = agent_executor(
        task_context.input_data,
        strategy=strategy,
        environment=replay_env
    )

    replay_result = {
        'success': True,
        'output': result,
        'metrics': self._compute_metrics(result, task_context),
        'confidence': self._compute_confidence(replay_env)
    }

```

```
except Exception as e:
    replay_result = {
        'success': False,
        'error': str(e),
        'metrics': {},
        'confidence': 0.0
    }

# Cache result
if self.cache:
    self.cache[cache_key] = replay_result

return replay_result
```

```
def batch_replay(
    self,
    task_contexts: List[ExecutionContext],
    strategies: List[Strategy],
    agent_executor: callable,
    sample_size: int = None
) -> Dict[str, List[Dict]]:
```

```
"""
```

```
Replay multiple tasks across multiple strategies.
```

```
Uses stratified sampling for efficiency.
```

Returns:

```
{
    strategy_version: [replay_results...]
}
```

```
"""
```

```
# Sample tasks if requested
```

```
if sample_size and len(task_contexts) > sample_size:
    sampled_tasks = self._stratified_sample(task_contexts, sample_size)
else:
    sampled_tasks = task_contexts

# Replay each strategy
results = {}
for strategy in strategies:
    strategy_results = []

    for task_ctx in sampled_tasks:
        replay_result = self.replay_task(task_ctx, strategy, agent_executor)
        strategy_results.append(replay_result)

    results[strategy.version] = strategy_results

return results

def compare_strategies(
    self,
    baseline_strategy: Strategy,
    candidate_strategies: List[Strategy],
    historical_tasks: List[ExecutionContext],
    agent_executor: callable
) -> Dict:
    """
    Compare multiple strategies against a baseline.

    Returns comprehensive comparison metrics.
    """
    all_strategies = [baseline_strategy] + candidate_strategies
```

```
# Replay all strategies
replay_results = self.batch_replay(
    historical_tasks,
    all_strategies,
    agent_executor,
    sample_size=50 # Sample for efficiency
)

# Compute aggregate metrics
comparison = {
    'baseline': baseline_strategy.version,
    'candidates': {},
    'winner': None,
    'metrics': {}
}

baseline_results = replay_results[baseline_strategy.version]
baseline_metrics = self._aggregate_metrics(baseline_results)

for candidate in candidate_strategies:
    candidate_results = replay_results[candidate.version]
    candidate_metrics = self._aggregate_metrics(candidate_results)

    # Compute improvement
    improvement = {
        metric: candidate_metrics[metric] - baseline_metrics[metric]
        for metric in candidate_metrics
    }

    comparison['candidates'][candidate.version] = {
        'metrics': candidate_metrics,
        'improvement': improvement,
    }
```

```
'significant': self._is_significant(improvement)
}

# Determine winner
comparison['winner'] = self._select_winner(comparison['candidates'])

return comparison

def _reconstruct_environment(self, context: ExecutionContext) -> Dict:
    """
    Attempt to reconstruct the execution environment.

    Challenges:
    - External APIs may have changed
    - Time-dependent data
    - User state may differ

    Approach:
    - Reconstruct what we can exactly (input data, user context)
    - Approximate time-dependent data
    - Flag low-confidence reconstructions
    """

    env = {
        'input': context.input_data,
        'user_context': context.user_context,
        'timestamp': context.timestamp, # Keep original time
        'reconstruction_confidence': {}
    }

    # Try to load external state from that time
    # (if we cached it during original execution)
    external_state_path = self._get_external_state_path(context.timestamp)
```

```
if os.path.exists(external_state_path):
    with open(external_state_path) as f:
        env['external_state'] = json.load(f)
        env['reconstruction_confidence']['external'] = 1.0
else:
    # Use current state as approximation
    env['external_state'] = self._get_current_external_state()
    env['reconstruction_confidence']['external'] = 0.5

return env

def _compute_confidence(self, replay_env: Dict) -> float:
    """
    Estimate confidence in replay fidelity.

    Lower confidence means results may not accurately reflect
    how the strategy would have performed.
    """
    confidences = replay_env.get('reconstruction_confidence', {})

    if not confidences:
        return 0.7 # Default moderate confidence

    # Weighted average of different components
    weights = {
        'external': 0.3,
        'user_state': 0.2,
        'environment': 0.5
    }

    total = sum(
        confidences.get(component, 0.7) * weight
        for component, weight in weights.items()
    )

    return total / len(weights)
```

```
        for component, weight in weights.items()
    )

    return total

def _stratified_sample(
    self,
    tasks: List[ExecutionContext],
    sample_size: int
) -> List[ExecutionContext]:
    """
    Sample tasks ensuring representation across difficulty levels.

    Stratification criteria:
    - Task complexity (simple/medium/hard)
    - Time period (recent vs old)
    - Success/failure in original execution
    """

    # Group by difficulty
    simple = [t for t in tasks if self._estimate_difficulty(t) < 0.3]
    medium = [t for t in tasks if 0.3 <= self._estimate_difficulty(t) < 0.7]
    hard = [t for t in tasks if self._estimate_difficulty(t) >= 0.7]

    # Sample proportionally
    import random
    sample = []

    for group, proportion in [(simple, 0.3), (medium, 0.5), (hard, 0.2)]:
        group_size = int(sample_size * proportion)
        sample.extend(random.sample(group, min(group_size, len(group)))))

    return sample
```

```
def _aggregate_metrics(self, results: List[Dict]) -> Dict:
    """Compute aggregate metrics across multiple replay results"""
    total = len(results)
    successful = sum(1 for r in results if r.get('success', False))

    return {
        'success_rate': successful / total if total > 0 else 0,
        'avg_confidence': sum(r.get('confidence', 0) for r in results) / total,
        'total_tasks': total
    }
```

[Continued in next file for length...]

Performance Optimizations

Caching Strategy

```
class SmartReplayCache:
    """
    Intelligent caching for replay results.

    Cache invalidation rules:
    - Invalidate if strategy changed
    - Invalidate if task input changed
    - Keep cache warm for frequently compared strategies
    """

    def __init__(self, max_size_mb: int = 500):
        self.max_size = max_size_mb * 1024 * 1024
        self.cache = {}
        self.access_counts = {}
        self.cache_size = 0

    def get(self, task_id: str, strategy_version: str) -> Optional[Dict]:
        key = (task_id, strategy_version)
        if key in self.cache:
            self.access_counts[key] = self.access_counts.get(key, 0) + 1
            return self.cache[key]
        return None

    def set(self, task_id: str, strategy_version: str, result: Dict):
        key = (task_id, strategy_version)
```

```
# Evict if at capacity
if self.cache_size >= self.max_size:
    self._evict_lru()

self.cache[key] = result
self.access_counts[key] = 1
self.cache_size += self._estimate_size(result)

def _evict_lru(self):
    """Evict least recently used entry"""
    lru_key = min(self.access_counts.keys(), key=lambda k: self.access_counts[k])
    del self.cache[lru_key]
    del self.access_counts[lru_key]
```

Parallel Replay

```
from concurrent.futures import ThreadPoolExecutor, as_completed

class ParallelReplayEngine(ReplayEngine):
    """Replay engine with parallel execution support"""

    def __init__(self, *args, max_workers: int = 4, **kwargs):
        super().__init__(*args, **kwargs)
        self.max_workers = max_workers

    def batch_replay(self, tasks, strategies, agent_executor, **kwargs):
        """Override to use parallel execution"""

        with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
            futures = {}

            for strategy in strategies:
                for task in tasks:
                    future = executor.submit(
                        self.replay_task,
                        task, strategy, agent_executor
                    )
                    futures[future] = (strategy.version, task.task_id)

            # Collect results
            results = {s.version: [] for s in strategies}

            for future in as_completed(futures):
                strategy_version, task_id = futures[future]
                result = future.result()
```

```
results[strategy_version].append(result)

return results
```

This architecture document provides the foundation. Would you also like me to create the Implementation Guide next?