Implementation Guide

Step-by-step guide to integrating the Evolutionary Alignment Framework into your AI agent system.

Table of Contents

Prerequisites

System Requirements

```
# Python version
python >= 3.9

# Storage
- 500MB for framework
- 1GB+ for execution logs (scales with usage)

# Compute
- CPU: 2+ cores recommended
- RAM: 4GB+ recommended
```

Your Agent Requirements

Your AI agent must support:

✅ **Programmatic access** - Can be called from Python code
✅ **State serialization** - Can save/load task state
✅ **Structured output** - Returns consistent data structures

Nice to have:

- 🟡 Historical task logs (for replay)
- 🟡 Metrics/observability
- 🟡 User feedback collection

Installation

Option 1: pip install (Recommended)

```
pip install evolutionary-alignment
```

Option 2: From source

```
git clone https://github.com/kestiny18/Lydia.git
cd Lydia
pip install -e .
```

Verify installation

```
python -c "import evolutionary_alignment; print(evolutionary_alignment.__version__)"
# Should print: 1.0.0
```

Quick Start

Step 1: Define Your Strategy (5 minutes)

Create `config/strategy.yaml` :

```yaml
metadata:
  name: "my-agent"
  version: "1.0.0"
  description: "Initial production strategy"

parameters:
  # Adjust these to match your agent's behavior
  autonomy_level: 0.7      # 0.0 = always ask, 1.0 = never ask
  response_speed: 0.8       # 0.0 = slow/careful, 1.0 = fast
  user_confirmation: 0.6    # How often to ask permission

constraints:
  must_confirm_before:
    - "delete_data"
    - "send_external_message"

  never_execute:
    - "system_shutdown"
    - "credential_modification"

evolution_metadata:
  max_parameter_delta: 0.10
  cooldown_period_days: 7
  requires_human_review: false
```

Step 2: Wrap Your Agent (10 minutes)

Before (your existing code):

```python
class MyAgent:
    def handle_task(self, task_data):
        # Your logic here
        if self.should_execute(task_data):
            return self.execute(task_data)
        else:
            return self.ask_user(task_data)
```

After (with framework):

```python
from evolutionary_alignment import StrategyAwareAgent

class MyAgent(StrategyAwareAgent):
    def __init__(self):
        super().__init__(strategy_path="config/strategy.yaml")

    def handle_task(self, task_data):
        # Framework now provides self.strategy
        autonomy = self.strategy.get_parameter('autonomy_level')

        if autonomy > 0.8:
            return self.execute(task_data)
        else:
            return self.ask_user(task_data)
```

Step 3: Enable Execution Logging (5 minutes)

```python
from evolutionary_alignment import ExecutionLogger

agent = MyAgent()
logger = ExecutionLogger(log_dir="./logs")

# Wrap task execution
@logger.log_execution
def run_task(task):
    with logger.task_context(
        task_id=task.id,
        task_type=task.type,
        input_data=task.data
    ):
        result = agent.handle_task(task.data)
        return result

# Use as normal
result = run_task(my_task)
# Automatically logged to ./logs/ for future replay
```

Step 4: Enable Learning (10 minutes)

```python
from evolutionary_alignment import EvolutionaryLearner

learner = EvolutionaryLearner(
    agent=agent,
    strategy_path="config/strategy.yaml",
    log_dir="./logs"
)

# Runs in background
learner.start_learning_loop(
    check_interval_hours=24,  # Evaluate daily
    min_tasks_for_learning=100  # Need 100 tasks before learning
)

# Your agent continues working normally
# Framework will:
# 1. Observe patterns
# 2. Propose improvements
# 3. Validate via replay
# 4. Request your approval for changes
```

That's it! Your agent is now safely self-evolving.

Integration Patterns

Pattern 1: Minimal Integration (Best for Testing)

Use if you want to try the framework without code changes.

```python
from evolutionary_alignment import wrap_agent_minimal

# Your existing agent (no changes needed)
my_agent = ExistingAgent()

# Wrap it
wrapped = wrap_agent_minimal(
    agent=my_agent,
    strategy_yaml="config/strategy.yaml",
    execute_method_name="run"  # Method that executes tasks
)

# Use wrapped agent
result = wrapped.run(task)
```

Pros:

- Zero code changes
- Works with any Python class

Cons:

- Limited customization
- Can't control individual decisions

Pattern 2: Decorator-Based (Recommended)

Use for fine-grained control over which methods use strategy.

```python
from evolutionary_alignment import (
    strategy_aware,
    log_execution,
    require_confirmation
)

class MyAgent:
    def __init__(self):
        self.strategy = Strategy("config/strategy.yaml")

    @strategy_aware(parameter="response_speed")
    @log_execution
    def quick_reply(self, message):
        """Uses response_speed parameter"""
        speed = self.strategy.get_parameter('response_speed')

        if speed > 0.8:
            return self.instant_reply(message)
        else:
            return self.thoughtful_reply(message)

    @require_confirmation(
        action="delete_file",
        unless=lambda ctx: ctx.file_size < 1024  # Auto-ok for small files
    )
    def delete_file(self, filepath):
        """Automatically checks strategy constraints"""
        os.remove(filepath)

    @log_execution
    @strategy_aware(parameter="autonomy_level")
    def complex_decision(self, context):
        """Multiple decorators work together"""
        autonomy = self.strategy.get_parameter('autonomy_level')

        # Decision logged automatically
        if autonomy > 0.7:
            return self.autonomous_action(context)
        else:
            return self.ask_user_first(context)
```

Pros:

- Explicit control
- Easy to understand
- Gradual adoption (add decorators one at a time)

Cons:

- Requires code changes
- Need to identify decision points

Pattern 3: Inheritance-Based (Advanced)

Use when building a new agent from scratch.

```python
from evolutionary_alignment import EvolutionaryAgent

class MySmartAgent(EvolutionaryAgent):
    """
    Agent that inherits full evolutionary capabilities.

    Automatically gets:
    - Strategy binding
    - Execution logging
    - Replay support
    - Learning loop
    """

    def __init__(self, config_path):
        super().__init__(
            strategy_path=f"{config_path}/strategy.yaml",
            enable_logging=True,
            enable_replay=True,
            enable_learning=True
        )

        # Your initialization
        self.custom_setup()

    def execute_task(self, task):
        """
        Override this method to implement your agent logic.

        Framework automatically:
        - Binds current strategy version
        - Logs execution for replay
        - Tracks metrics
        """

        # Access strategy parameters
        if self.should_act_autonomously(task):
            return self.autonomous_execution(task)
        else:
            return self.supervised_execution(task)

    def should_act_autonomously(self, task):
        """Use strategy to decide autonomy level"""
        base_autonomy = self.strategy.get_parameter('autonomy_level')
        task_risk = self.assess_risk(task)

        # Higher risk → need more confirmation
        adjusted_autonomy = base_autonomy * (1 - task_risk)

        return adjusted_autonomy > 0.7

    def on_strategy_update_proposed(self, old_strategy, new_strategy, rationale):
        """
        Hook called when framework proposes a strategy update.

        Override to customize approval logic.
        """
        # Example: Auto-approve small changes, ask for large ones
        max_change = max(
            abs(new_strategy.get_parameter(p) - old_strategy.get_parameter(p))
            for p in new_strategy.parameters
        )

        if max_change < 0.05:
            return "auto_approve"
        else:
            return "request_human_review"
```

Pros:

- Full framework integration
- Cleanest code
- All features available

Cons:

- Requires agent redesign
- Steeper learning curve

Real-World Integration Examples

Example 1: LangChain Agent

```python
from langchain.agents import AgentExecutor, create_openai_functions_agent
from evolutionary_alignment import wrap_langchain_agent

# Your existing LangChain agent
agent = create_openai_functions_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools)

# Wrap with framework
evolved_executor = wrap_langchain_agent(
    agent_executor,
    strategy_path="config/strategy.yaml",
    # Map LangChain concepts to strategy parameters
    parameter_mapping={
        'max_iterations': 'autonomy_level',  # Higher autonomy → more iterations
        'early_stopping_method': 'response_speed'
    }
)

# Use as normal LangChain agent
result = evolved_executor.invoke({"input": "What's the weather?"})

# Framework automatically:
# - Adjusts max_iterations based on autonomy_level
# - Logs execution for replay
# - Learns from patterns
```

Example 2: AutoGPT-style Agent

```python
from evolutionary_alignment import EvolutionaryAgent

class AutoGPTEvolved(EvolutionaryAgent):
    def __init__(self):
        super().__init__(strategy_path="config/strategy.yaml")
        self.memory = []
        self.goals = []

    def execute_task(self, goal):
        """Override AutoGPT's main loop"""
        self.goals.append(goal)

        while not self.is_goal_achieved(goal):
            # Framework provides strategy-aware decision making
            if self.should_ask_for_guidance():
                user_input = self.ask_user("What should I do next?")
                action = self.parse_user_input(user_input)
            else:
                action = self.autonomous_decide_action()

            result = self.execute_action(action)
            self.memory.append((action, result))

            # Framework logs this iteration
            self.log_decision(
                decision_point="action_selection",
                chosen_action=action,
                reason=self.get_reasoning()
            )
```

```python
def should_ask_for_guidance(self):
    """Use strategy to decide when to ask user"""
    autonomy = self.strategy.get_parameter('autonomy_level')
    task_complexity = self.assess_complexity()

    # Complex tasks need more supervision
    threshold = autonomy * (1 - task_complexity)

    return threshold < 0.5
```

Example 3: Email Automation Agent

```python
from evolutionary_alignment import StrategyAwareAgent
import imaplib
import smtplib

class EmailAgent(StrategyAwareAgent):
    def __init__(self):
        super().__init__(strategy_path="config/email_strategy.yaml")
        self.imap = self.connect_imap()
        self.smtp = self.connect_smtp()

    def process_inbox(self):
        """Main loop: check emails and decide what to do"""
        emails = self.fetch_unread_emails()

        for email in emails:
            self.process_email(email)

    @log_execution
    @strategy_aware(parameter="response_speed")
    def process_email(self, email):
        """Process a single email using strategy"""

        # Classify email
        category = self.classify(email)
        priority = self.assess_priority(email)

        # Use strategy to decide action
        if self.should_auto_reply(category, priority):
            self.send_auto_reply(email)
        elif self.should_draft_for_review(category, priority):
            self.create_draft(email)
        else:
            self.notify_user(email)

    def should_auto_reply(self, category, priority):
        """Decision logic using strategy parameters"""

        # Get strategy parameters
        autonomy = self.strategy.get_parameter('autonomy_level')
        response_speed = self.strategy.get_parameter('response_speed')

        # High autonomy + low priority = auto-reply
        if category == "routine" and priority < 0.5:
            return autonomy > 0.7

        # Urgent emails: speed matters
        if priority > 0.8:
            return response_speed > 0.8

        # Default: ask user
        return False
```

Production Deployment Checklist

Pre-Deployment

- [ ] **Strategy validation**

```
python -m evolutionary_alignment.validate config/strategy.yaml
```

- [ ] **Collect baseline logs**

```
# Run in logging-only mode for 1 week
agent.enable_learning = False
agent.enable_logging = True
```

- [ ] **Set up monitoring**

```
from evolutionary_alignment.monitoring import setup_monitoring

setup_monitoring(
    agent=agent,
    metrics=['success_rate', 'user_satisfaction', 'error_rate'],
    alert_on_degradation=True
)
```

- [ ] **Configure human review**

```
# config/strategy.yaml
evolution_metadata:
    requires_human_review: true  # Start conservative
    notification_channel: "slack"
    notification_webhook: "https://hooks.slack.com/..."
```

Deployment Phases

Phase 1: Observation (Week 1-2)

```
# Logging only, no learning
agent = MyAgent()
agent.enable_learning = False
agent.enable_logging = True
```

Phase 2: Learning with Human Approval (Week 3-4)

```
# Enable learning, but all changes need approval
agent.enable_learning = True
agent.auto_approve_updates = False
```

Phase 3: Partial Autonomy (Week 5-8)

```
# Auto-approve small changes
agent.auto_approve_threshold = 0.05  # 5% max parameter change
```

Phase 4: Full Autonomy (Week 9+)

```
# Auto-approve most changes, human review for critical
agent.auto_approve_threshold = 0.10
agent.human_review_for_risk_level = "high"
```

Post-Deployment Monitoring

```python
from evolutionary_alignment.monitoring import EvolutionMonitor

monitor = EvolutionMonitor(agent)

# Check health daily
@daily_cron_job
def check_agent_health():
    health = monitor.get_health_report()

    if health['goal_drift_score'] > 0.3:
        alert("Agent may be drifting from goals")

    if health['error_rate_increase'] > 0.2:
        alert("Error rate increased by 20%")

    if health['strategy_update_frequency'] > 3:
        alert("Too many strategy updates this week")
```

Troubleshooting

Common Issues

Issue 1: "Strategy file not found"

Error:

```
FileNotFoundError: [Errno 2] No such file or directory: 'config/strategy.yaml'
```

Solution:

```python
# Use absolute path
import os
strategy_path = os.path.join(os.path.dirname(__file__), 'config/strategy.yaml')
agent = MyAgent(strategy_path=strategy_path)
```

Issue 2: "Replay confidence too low"

Warning:

```
ReplayConfidence: 0.45 (below threshold 0.70)
Unable to reliably validate strategy change
```

Causes:

- Not enough historical task logs
- Task inputs not serializable
- External dependencies changed

Solutions:

```
# Option 1: Lower confidence threshold (temporarily)
replay_engine = ReplayEngine(min_confidence=0.45)

# Option 2: Collect more logs before learning
learner = EvolutionaryLearner(
    min_tasks_before_learning=500  # Increase from 100
)

# Option 3: Improve serialization
class MyAgent(EvolutionaryAgent):
    def serialize_task_context(self, task):
        """Override to capture more context"""
        return {
            'input': task.data,
            'user_state': self.get_user_state(),
            'environment': self.capture_environment(),
            'timestamp': datetime.now().isoformat()
        }
```

Issue 3: "Strategy updates too frequent"

Observation:

```
5 strategy updates in 2 days (exceeds cooldown period)
```

Cause: Cooldown period not properly enforced

Solution:

```yaml
# config/strategy.yaml
evolution_metadata:
  cooldown_period_days: 7  # Enforce 1 week minimum
  max_updates_per_month: 4  # Hard limit
```

```python
# In code
from evolutionary_alignment import UpdateRateLimiter

rate_limiter = UpdateRateLimiter(
    max_per_day=1,
    max_per_week=2,
    max_per_month=4
)

learner = EvolutionaryLearner(
    agent=agent,
    rate_limiter=rate_limiter
)
```

Issue 4: "Memory/storage growing too large"

Problem:

```
Execution logs: 15GB and growing
Replay cache: 3GB
```

Solutions:

```python
# Option 1: Log rotation
from evolutionary_alignment import LogRotation

LogRotation(
    log_dir="./logs",
    max_age_days=90,  # Delete logs older than 90 days
    compress_after_days=30  # Compress logs older than 30 days
).enable()

# Option 2: Sampling for replay
replay_engine = ReplayEngine(
    sample_strategy="stratified",  # Don't replay every task
    sample_size=100,  # Only 100 tasks per evaluation
    cache_size_mb=500  # Limit cache to 500MB
)

# Option 3: Cloud storage archival
from evolutionary_alignment.storage import S3LogArchiver

archiver = S3LogArchiver(
    bucket="my-agent-logs",
    archive_after_days=60
)
archiver.start_background_archival()
```

Issue 5: "High replay cost (LLM API calls)"

Problem:

```
Monthly replay cost: $450
50 strategy evaluations × 100 tasks × $0.09/task
```

Solutions:

```python
# Solution 1: Aggressive caching
replay_engine = ReplayEngine(
    cache_enabled=True,
    cache_hit_target=0.80  # Aim for 80% cache hit rate
)

# Solution 2: Use cheaper model for replay
from evolutionary_alignment import ReplayEngine

replay_engine = ReplayEngine(
    llm_for_replay="gpt-3.5-turbo",  # Cheaper than gpt-4
    llm_for_production="gpt-4"        # Production still uses best
)

# Solution 3: Smarter sampling
replay_engine = ReplayEngine(
    sampling_strategy="adaptive",  # Start with 10 tasks, increase if uncertain
    max_sample_size=100
)

# Result: Cost reduced to ~$50/month
```

Advanced Configuration

Custom Delta Generation

Override how strategy improvements are proposed:

```python
from evolutionary_alignment import DeltaGenerator

class ConservativeDeltaGenerator(DeltaGenerator):
    """Generate very small, safe parameter changes"""

    def generate_delta(self, learning_signal):
        """
        learning_signal = {
            'metric': 'response_speed',
            'current_value': 0.7,
            'suggested_direction': 'increase',
            'confidence': 0.85
        }
        """
        current = learning_signal['current_value']
        confidence = learning_signal['confidence']

        # Scale change by confidence
        max_change = 0.05  # Very conservative: 5% max
        actual_change = max_change * confidence

        if learning_signal['suggested_direction'] == 'increase':
            new_value = min(1.0, current + actual_change)
        else:
            new_value = max(0.0, current - actual_change)

        return {
            learning_signal['metric']: new_value
        }

# Use custom generator
```

```
learner = EvolutionaryLearner(
    agent=agent,
    delta_generator=ConservativeDeltaGenerator()
)
```

Custom Update Gates

Add your own safety checks:

```python
from evolutionary_alignment import UpdateGate, GateResult

class BusinessHoursGate(UpdateGate):
    """Only allow updates during business hours"""

    def evaluate(self, strategy_update):
        from datetime import datetime

        now = datetime.now()

        # Only allow updates Mon-Fri, 9am-5pm
        if now.weekday() >= 5:  # Weekend
            return GateResult(
                status="REJECT",
                reason="No updates on weekends"
            )

        if now.hour < 9 or now.hour >= 17:
            return GateResult(
                status="REJECT",
                reason="Updates only during business hours"
            )

        return GateResult(status="PASS")

# Add to gate pipeline
learner = EvolutionaryLearner(
    agent=agent,
    additional_gates=[
        BusinessHoursGate(),
        # Your other custom gates...
    ]
)
```

Migration Guide

Migrating from Existing Agent

Step 1: Audit current behavior

```
# Run your agent for 1 week, log all decisions
from evolutionary_alignment import BehaviorAuditor

auditor = BehaviorAuditor(agent)
auditor.start_logging()

# After 1 week:
behavior_report = auditor.generate_report()
# Shows: What decisions are made, how often, patterns
```

Step 2: Extract implicit strategy

```python
# Automatically generate strategy YAML from observed behavior
strategy_yaml = auditor.extract_strategy(
    output_path="config/strategy.yaml"
)

# This creates a strategy that matches your current agent's behavior
```

Step 3: Gradual integration

```python
# Week 1: Just logging
agent = YourAgent()
wrap_with_logging_only(agent)

# Week 2: Add strategy binding (no changes yet)
agent = StrategyAwareAgent(your_agent)

# Week 3: Enable learning (observation only)
enable_learning(agent, dry_run=True)

# Week 4: Enable learning (with human approval)
enable_learning(agent, dry_run=False, auto_approve=False)
```

Next Steps

1. **Read case studies**: See README.md → Case Studies
2. **Join community**: GitHub Discussions
3. **Explore examples**: `examples/` directory has more integration patterns

4. **Watch tutorials**: YouTube playlist

Need help?

- Open an issue
- Ask in Discussions
- Email: kestiny.me@gmail.com