Evolving the Swarm: Implementing Agent Evolution via Genetic Programming

Yes, we are absolutely ready to implement the next enhancement! We'll start with **Agent Evolution via Genetic Programming**. This will allow RAWE to automatically generate and refine agent personalities over time, naturally selecting for the most profitable trading strategies and continuously improving the collective's performance.

Phase 1: Creating the AgentEvolver Module

First, we'll create a new module agent_evolver.py that encapsulates the genetic programming logic.

rawe_system/src/core/agent_evolver.py (New File)

```
# rawe_system/src/core/agent_evolver.py
import random
import logging
from typing import Dict, Any, List
from collections import defaultdict

logger = logging.getLogger('rawe_system.AgentEvolver')

class AgentEvolver:
    """
    Evolves new agent personalities through genetic algorithms.
    This class handles crossover (combining traits) and mutation
(random changes)
    to create new, potentially more effective, agent configurations.
    """

    def __init__(self, personality_traits_config: Dict[str,
List[str]]):
        # Configuration for which traits to evolve and their
type/ranges
        self.evolvable_traits_config = {
            # Map conceptual traits to actual personality profile keys
            "risk_aversion": {"type": float, "min": 0.1, "max": 0.9,
"mutation_scale": 0.1},
            "min_expected_profit_multiplier": {"type": float, "min":
0.5, "max": 2.0, "mutation_scale": 0.2},
            "position_size_multiplier": {"type": float, "min": 0.5,
"max": 1.5, "mutation_scale": 0.1},
            "kelly_win_loss_ratio": {"type": float, "min": 1.0, "max":
3.0, "mutation_scale": 0.2},
            # Signal type preference is a list, needs special handling
for crossover/mutation
            "signal_type_preference": {"type": list, "options":
["narrative_leads", "capital_leads", "divergence"]}
```

```
        }
        logger.info("AgentEvolver initialized with evolvable traits
configuration.")

    def _mutate_trait(self, trait: str, value: Any) -> Any:
        """Applies a mutation to a single trait based on its type and
configured scale."""
        config = self.evolvable_traits_config.get(trait)
        if not config:
            return value # Don't mutate if not configured

        if config["type"] == float:
            mutation_amount =
random.uniform(-config["mutation_scale"], config["mutation_scale"])
            new_value = value + mutation_amount
            # Ensure within bounds
            return max(config["min"], min(config["max"], new_value))
        elif config["type"] == list:
            # For list traits (like signal_type_preference), randomly
add/remove/swap elements
            options = config["options"]
            new_list = list(value) # Create a mutable copy

            if random.random() < 0.5 and len(new_list) < len(options):
# Add
                new_list.append(random.choice([o for o in options if o
not in new_list]))
            elif random.random() < 0.3 and len(new_list) > 1: # Remove
                new_list.remove(random.choice(new_list))

            random.shuffle(new_list) # Shuffle to reorder
            return new_list
        # Add other types of mutations here as needed
        return value

    def crossover_personalities(self, parent1: Dict, parent2: Dict) ->
Dict:
        """
        Creates an offspring personality by combining traits from two
parent personalities.

        Args:
            parent1 (Dict): The personality profile of the first
parent.
            parent2 (Dict): The personality profile of the second
parent.

        Returns:
```

```
            Dict: The personality profile of the offspring.
        """
        offspring = {}
        for trait_name in self.evolvable_traits_config.keys():
            if trait_name in parent1 and trait_name in parent2:
                # Randomly inherit trait from one parent
                if random.random() > 0.5:
                    offspring[trait_name] = parent1[trait_name]
                else:
                    offspring[trait_name] = parent2[trait_name]
            elif trait_name in parent1: # If only in parent1
(shouldn't happen with fixed schema)
                offspring[trait_name] = parent1[trait_name]
            elif trait_name in parent2: # If only in parent2
                offspring[trait_name] = parent2[trait_name]

            # Apply mutation after crossover
            if random.random() < 0.2: # 20% chance of mutation
                offspring[trait_name] = self._mutate_trait(trait_name,
offspring.get(trait_name))

        # Handle 'coherence_threshold_for_trade' - usually constant or
needs complex crossover
        # For simplicity, copy from one parent, or use a default
        offspring["coherence_threshold_for_trade"] =
random.choice([parent1, parent2]).get("coherence_threshold_for_trade",
{})

        # Add any other non-evolvable but required traits
        # Example: 'coherence_threshold_for_trade' can be set to a
default or copied.
        # For this demo, let's just make it a fixed default.
        if "coherence_threshold_for_trade" not in offspring:
             offspring["coherence_threshold_for_trade"] = {'AAA': 0.0,
'AA': 0.0, 'A': 0.0, 'BBB': 0.0, 'BB': 0.0, 'B': 0.0, 'C': 0.0, 'D':
0.0}

        return offspring

    def evolve_population(self,
                          current_personalities: Dict[str, Dict[str,
Any]],
                          performance_data: Dict[str, Dict[str, Any]],
                          num_offspring: int = 2) -> Dict[str,
Dict[str, Any]]:
        """
        Evolves the agent population based on performance data.
```

```
        Args:
            current_personalities (Dict[str, Dict[str, Any]]):
Dictionary of agent names to their personality profiles.
            performance_data (Dict[str, Dict[str, Any]]): Dictionary
of agent names to their total P&L.
            num_offspring (int): Number of new offspring to generate.

        Returns:
            Dict[str, Dict[str, Any]]: The new, evolved population of
personalities.
        """
        logger.info("Starting agent population evolution...")

        # Filter out agents without performance data or with no trades
(if applicable)
        eligible_agents = {name: profile for name, profile in
current_personalities.items()
                           if name in performance_data and
performance_data[name].get('total_pnl') is not None}

        if not eligible_agents:
            logger.warning("No eligible agents with performance data
for evolution. Returning current personalities.")
            return current_personalities

        # Select top performers based on total P&L (simplified for
demo)
        # In a real system, you'd use a more robust fitness function
(Sharpe, Drawdown, etc.)
        sorted_agents = sorted(eligible_agents.items(),
                               key=lambda x:
performance_data.get(x[0], {}).get('total_pnl', -float('inf')),
                               reverse=True)

        # Keep top X% as survivors (elitism)
        # For simplicity, let's keep at least 1 and at most half.
        num_survivors = max(1, len(sorted_agents) // 2)
        survivors = dict(sorted_agents[:num_survivors])

        new_personalities = survivors.copy() # Start new population
with survivors

        # Create offspring from survivors
        offspring_count = 0
        while offspring_count < num_offspring:
            # Select two random parents from survivors
            if len(survivors) < 2:
                logger.warning("Not enough survivors for crossover.
```

```
Cannot generate more offspring.")
                break # Cannot perform crossover if less than 2
parents

            parent1_name, parent1_profile =
random.choice(list(survivors.items()))
            parent2_name, parent2_profile =
random.choice(list(survivors.items()))

            # Ensure parents are distinct for meaningful crossover
            while parent1_name == parent2_name and len(survivors) > 1:
                parent2_name, parent2_profile =
random.choice(list(survivors.items()))

            offspring_personality =
self.crossover_personalities(parent1_profile, parent2_profile)

            # Generate a unique name for the new agent
            offspring_name =
f"Evolved_Agent_{datetime.now().strftime('%H%M%S')}_{offspring_count}"
            new_personalities[offspring_name] = offspring_personality
            logger.info(f"Generated new agent: {offspring_name} from
{parent1_name} and {parent2_name}.")
            offspring_count += 1

        logger.info(f"Evolution complete. New population size:
{len(new_personalities)}")
        return new_personalities
```

Implementation Steps:

1. **Create the file:** Inside your rawe_system/src/core/ directory, create a new file named agent_evolver.py.
2. **Paste the code:** Copy and paste the entire AgentEvolver class code into this new file.

Phase 2: Integrate AgentEvolver into scripts/run_collective_rawe.py

Now, we'll modify the main script to periodically evolve the agent population. This involves keeping track of rawe_agents as personality_profiles, collecting performance data, and calling the evolve_population method.

rawe_system/scripts/run_collective_rawe.py (Modifications)

1. **Add Import:** At the top of the file, add:
   ```
   from src.core.agent_evolver import AgentEvolver # New import
   ```
2. **Instantiate AgentEvolver:** After MetaLearner initialization:
   ```
   # ... after MetaLearner initialization ...

   # Initialize Agent Evolver
   ```

```
agent_evolver = AgentEvolver(PERSONALITY_PROFILES) # Pass initial
profiles for trait configuration
collective_logger.info("AgentEvolver initialized.")
```

3. **Modify Agent Management for Evolution:** We need to keep the rawe_agents as a dictionary of UnifiedArbitrageSystem instances, but the PERSONALITY_PROFILES dictionary will be the one that AgentEvolver modifies.Change this block:
```
# Instantiate multiple RAWE agents with different personalities
rawe_agents = {}
agent_tasks = []
for name in PERSONALITY_PROFILES.keys():
    agent = UnifiedArbitrageSystem(narrative_engine,
                                   personality_name=name,
                                   redis_client=redis_client,

signal_channel=RAWE_SIGNAL_CHANNEL)
    agent.set_broker(broker)
    rawe_agents[name] = agent
    collective_logger.info(f"Agent '{name}' initialized.")
```
To initially populate rawe_agents based on PERSONALITY_PROFILES:
```
# Use a mutable copy of PERSONALITY_PROFILES so AgentEvolver can
modify it
current_personality_profiles = PERSONALITY_PROFILES.copy()

def _initialize_agents(current_profiles: Dict[str, Dict[str,
Any]], narrative_engine_instance, redis_client_instance,
signal_channel_name, broker_instance):
    agents = {}
    for name, profile in current_profiles.items():
        agent = UnifiedArbitrageSystem(narrative_engine_instance,
                                       personality_name=name,

redis_client=redis_client_instance,

signal_channel=signal_channel_name)
        agent.set_broker(broker_instance)
        agents[name] = agent
        collective_logger.info(f"Agent '{name}' initialized with
profile: {profile.get('risk_aversion', 'N/A')}")
    return agents

rawe_agents = _initialize_agents(current_personality_profiles,
narrative_engine, redis_client, RAWE_SIGNAL_CHANNEL, broker)
```

4. **Integrate Evolution into the Main Trading Loop:** We'll add a condition to trigger evolution periodically (e.g., every few cycles). When evolution occurs, we'll replace the old agents with new ones.Find the for cycle in range(10): loop. Inside this loop, after the await asyncio.gather(*agent_cycle_tasks) and performance logging, add the evolution logic:

```
    # ... (Collective performance logging) ...

    # Trigger agent evolution periodically (e.g., every 3 cycles)
    if (cycle + 1) % 3 == 0 and cycle > 0: # Evolve after cycle 3,
6, 9
        collective_logger.info(f"\n🧬 TRIGGERING AGENT EVOLUTION
after Cycle {cycle + 1}!")

        # Gather performance data for evolution (Total P&L from
each agent)
        agent_pnl_data = {}
        for name, agent in rawe_agents.items():
            report = agent.generate_performance_report()
            agent_pnl_data[name] = {'total_pnl':
report['pnl']['total']}

        # Evolve the personalities
        new_personality_profiles =
agent_evolver.evolve_population(current_personality_profiles,
agent_pnl_data, num_offspring=2)

        # Update the global personality profiles
        current_personality_profiles.clear()

current_personality_profiles.update(new_personality_profiles)

        # Re-initialize agents with the new/evolved personalities
        # This is a simplification; in a real system, you might
manage existing positions gracefully
        # or allow agents to be 'retired' rather than fully
replaced.
        collective_logger.info("Re-initializing RAWE agents with
evolved personalities...")
        rawe_agents =
_initialize_agents(current_personality_profiles, narrative_engine,
redis_client, RAWE_SIGNAL_CHANNEL, broker)
        collective_logger.info("Agents re-initialized. Evolution
cycle complete.")

    await asyncio.sleep(5) # Wait between collective cycles
```

5. **Modify _run_single_agent_cycle to ensure agent_instance.profile is up-to-date (optional but good practice):** The _initialize_agents recreates agents, but if we were to allow in-place evolution, an agent might need to refresh its profile. For our current setup, recreation handles it.

Final Preparations and Execution

1. **Ensure rawe_system/src/core/agent_evolver.py is created** and populated.
2. **Replace rawe_system/scripts/run_collective_rawe.py** with the updated content.
3. **Ensure Redis Server is Running.**
4. **Run the Collective RAWE System:**
   `python scripts/run_collective_rawe.py`

Phase 3: Implementing Adversarial Meta-Learning
Objective: Develop an AdversarialMetaLearner that can generate perturbed or "adversarial" market states to test the robustness of the primary MetaLearner's regime detection, ensuring it doesn't get fooled by unusual combinations of indicators.
This will enhance the adaptive nature of RAWE by making its foundational understanding of market conditions more resilient.
Step 1: Create rawe_system/src/core/adversarial_meta_learner.py (New File)
This file will contain the AdversarialMetaLearner class, inheriting from MetaLearner and adding adversarial capabilities.
rawe_system/src/core/adversarial_meta_learner.py (New File)

```
# rawe_system/src/core/adversarial_meta_learner.py
import logging
import random
from typing import Dict, Any, List

# Import the base MetaLearner
from src.core.meta_learner import MetaLearner

logger = logging.getLogger('rawe_system.AdversarialMetaLearner')

class AdversarialMetaLearner(MetaLearner):
    """
    Extends MetaLearner to generate adversarial market states and test
    the robustness of market regime detection.
    """

    def __init__(self):
        super().__init__() # Initialize the base MetaLearner
        logger.info("AdversarialMetaLearner initialized.")

    async def generate_adversarial_market_state(self, current_state: Dict[str, float]) -> Dict[str,
float]:
        """
        Creates an adversarial market state by subtly perturbing or creating
        contradictory conditions based on the current market state.

        Args:
            current_state (Dict[str, float]): The current market state with 'nvx', 'entropy', 'curvature'.

        Returns:
            Dict[str, float]: A new market state designed to challenge regime detection.
        """
```

```
        adversarial_state = current_state.copy()

        # Perturbation strategies:
        # 1. High NVX but stable curvature (unusual combo: high narrative noise but underlying
stability)
        if current_state['nvx'] > 70 and current_state['curvature'] < 0:
            adversarial_state['curvature'] = random.uniform(-0.1, 0.2) # Force a more stable
curvature
            logger.debug(f"Generated adversarial state: High NVX, forced stable curvature:
{adversarial_state}")
            return adversarial_state

        # 2. Negative curvature but low entropy (contradiction: systemic decay but low chaos)
        if current_state['curvature'] < -1.0 and current_state['entropy'] > 0.5:
            adversarial_state['entropy'] = random.uniform(0.1, 0.3) # Force lower entropy
            logger.debug(f"Generated adversarial state: Negative Curvature, forced low entropy:
{adversarial_state}")
            return adversarial_state

        # 3. Apply a small random perturbation to all values
        for key in adversarial_state:
            adversarial_state[key] += random.uniform(-5.0, 5.0) # Small random shift
            adversarial_state[key] = max(0.0, adversarial_state[key]) # Ensure non-negative for
some metrics

        logger.debug(f"Generated adversarial state: Random perturbation: {adversarial_state}")
        return adversarial_state

    async def calculate_regime_confidence(self, actual_state: Dict[str, float], predicted_regime:
str) -> float:
        """
        Simulates calculating confidence in a predicted market regime.
        In a real system, this would involve a complex model (e.g., probability distribution
        over regimes, or a similarity score to known regime patterns).
        For this demo, it's a placeholder.
        """
        # A simple confidence based on how 'far' the state is from a clear pattern
        # This is highly conceptual and would need a rigorous definition.
        # For demo, higher NVX/entropy mismatch with 'normal' -> lower confidence

        confidence = 1.0 # Start with high confidence
        if predicted_regime == "normal_market":
            if actual_state['nvx'] > 70 or actual_state['entropy'] > 0.6:
                confidence = 0.5 # Low confidence if predicted normal but seems volatile
        if predicted_regime == "high_volatility_chaos":
            if actual_state['nvx'] < 50 and actual_state['entropy'] < 0.4:
                confidence = 0.4 # Low confidence if predicted chaotic but seems stable
```

```
        return confidence * random.uniform(0.8, 1.0) # Add some randomness

    async def test_regime_detection_robustness(self, current_nvx: float, current_global_entropy: float, current_avg_curvature: float):
        """
        Tests the market regime detection robustness by generating adversarial states
        and evaluating the MetaLearner's predictions on them.
        """
        logger.info("Initiating Adversarial Meta-Learning: Testing regime detection robustness...")

        original_state = {
            'nvx': current_nvx,
            'entropy': current_global_entropy,
            'curvature': current_avg_curvature
        }

        num_tests = 3 # Number of adversarial tests per cycle
        for i in range(num_tests):
            adversarial_state = await self.generate_adversarial_market_state(original_state)

            # Use the base MetaLearner's analyze_market_state method
            predicted_regime = await super().analyze_market_state(
                adversarial_state['nvx'],
                adversarial_state['entropy'],
                adversarial_state['curvature']
            )

            confidence = await self.calculate_regime_confidence(adversarial_state, predicted_regime)

            if confidence < 0.7: # If confidence is low, warn
                logger.warning(f"Adversarial Test {i+1}: Low confidence ({confidence:.2f}) regime detection! "
                               f"Input: {adversarial_state}, Predicted: {predicted_regime}")
            else:
                logger.debug(f"Adversarial Test {i+1}: Robust detection ({confidence:.2f}). "
                             f"Input: {adversarial_state}, Predicted: {predicted_regime}")

        logger.info("Adversarial Meta-Learning tests completed for this cycle.")
```

Implementation Steps:
 * Create the file: Inside your rawe_system/src/core/ directory, create a new file named adversarial_meta_learner.py.
 * Paste the code: Copy and paste the entire AdversarialMetaLearner class code into this new file.

Step 2: Integrate AdversarialMetaLearner into scripts/run_collective_rawe.py

We will replace the instantiation of MetaLearner with AdversarialMetaLearner and call its testing

method periodically.
rawe_system/scripts/run_collective_rawe.py (Modifications)
* Update Import: Change the import from meta_learner to adversarial_meta_learner:

```
# from src.core.meta_learner import MetaLearner # Old import
from src.core.adversarial_meta_learner import AdversarialMetaLearner # New import
```

* Instantiate AdversarialMetaLearner: Replace the meta_learner instantiation with the new class:

```
# ... after AgentEvolver initialization ...

# Initialize Adversarial MetaLearner
meta_learner = AdversarialMetaLearner() # Now using the adversarial version
collective_logger.info("AdversarialMetaLearner initialized.")
```

* Integrate Adversarial Testing into the Main Trading Loop:
  Inside the for cycle in range(10): loop, after market_state = await meta_learner.analyze_market_state(...), we'll add the call to the adversarial testing method.

```
# Main trading loop for the collective
for cycle in range(10): # 10 cycles for demo
    collective_logger.info(f"\n⚡ COLLECTIVE ARBITRAGE CYCLE {cycle + 1}")
    collective_logger.info("=" * 80)

    # Step 1: Analyze current market state using MetaLearner
    current_nvx = narrative_engine.calculate_nvx_index()
    current_global_entropy = sum(n.volatility_30d for n in
narrative_engine.narrative_assets.values()) / len(narrative_engine.narrative_assets)
    dummy_avg_curvature = -0.5 # Placeholder for a global aggregated curvature

    market_state = await meta_learner.analyze_market_state(current_nvx,
current_global_entropy, dummy_avg_curvature)

    # NEW: Step 1.5: Perform Adversarial Meta-Learning tests periodically
    if (cycle + 1) % 2 == 0: # Run adversarial tests every 2 cycles
        await meta_learner.test_regime_detection_robustness(current_nvx,
current_global_entropy, dummy_avg_curvature)

    # Step 2: Select active agents based on market state
    all_agent_names = list(current_personality_profiles.keys()) # Use current_personality_profiles
for dynamic agent list
    active_agent_names = await meta_learner.select_active_agents(market_state,
all_agent_names)

    # ... rest of the loop (running agents, performance logging, evolution trigger) ...
```

Final Preparations and Execution
* Ensure rawe_system/src/core/adversarial_meta_learner.py is created and populated.
* Replace rawe_system/scripts/run_collective_rawe.py with the updated content.
* Ensure Redis Server is Running.

* Run the Collective RAWE System:
  python scripts/run_collective_rawe.py