Unleashing the Swarm: Implementing Advanced Enhancements

Yes, we are absolutely ready to unleash this swarm. Your roadmap for advanced enhancements is brilliant and will push RAWE's capabilities even further.
We will start with the first item on your list: **Weighted Voting by Performance.** This is a critical step in ensuring that the collective intelligence learns from its successes and failures, giving more influence to agents that prove their predictive accuracy.

Phase 1: Implementing Weighted Voting by Performance

Objective: Adjust each agent's influence within the ConsensusEngine based on their historical trading performance (wins/losses), giving more weight to consistently profitable agents.
Implementation Plan:

1. **Modify ConsensusEngine.__init__:** Add a defaultdict to track each agent's performance (wins, losses, weight).
2. **Add update_agent_weight method:** This method will be called after a trade's result is known (successful execution and eventual P&L). It will adjust the agent's weight based on its win rate.
3. **Modify ConsensusEngine._evaluate_consensus:** When evaluating proposals, instead of a simple 3/5 vote, we'll sum the weights of the proposing agents. A consensus will be reached if the sum of weights exceeds a threshold.

Step 1: Update rawe_system/src/core/consensus_engine.py

We will modify the ConsensusEngine class.

- **Add update_agent_weight and agent_performance tracking:** Find the __init__ method and add the agent_performance dictionary. Then, add the update_agent_weight method.

```
# rawe_system/src/core/consensus_engine.py
import asyncio
import json
import logging
from collections import defaultdict
from datetime import datetime, timedelta
from typing import Dict, Any, List

import redis.asyncio as redis

from src.modules.alpaca_broker import AlpacaBroker
from src.core.unified_arbitrage_system import ArbitrageSignal

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

class ConsensusEngine:
    """
    Manages collective decision-making for RAWE agents.
    Listens for proposed signals, applies consensus logic, and
executes final trades.
```

```
    """
    def __init__(self, redis_client: redis.Redis, broker:
AlpacaBroker,
                 subscribe_channel: str = "rawe_signals",
                 consensus_threshold: int = 3, # Initial: 3 out of
5 agents agree
                 consensus_window_seconds: int = 10):
        self.redis_client = redis_client
        self.broker = broker
        self.subscribe_channel = subscribe_channel
        self.consensus_threshold = consensus_threshold # Used as a
raw count for initial proposals
        self.consensus_window =
timedelta(seconds=consensus_window_seconds)

        self.proposed_signals: Dict[str, List[Dict[str, Any]]] =
defaultdict(list)
        self.last_consensus_check_time = datetime.now()

        # NEW: Agent performance tracking for weighted voting
        self.agent_performance: Dict[str, Dict[str, Any]] =
defaultdict(lambda: {'wins': 0, 'losses': 0, 'weight': 1.0})
        logger.info("ConsensusEngine initialized with
performance-based weighting.")

    def update_agent_weight(self, agent_name: str, trade_result:
Dict[str, Any]):
        """
        Adjusts agent voting weight based on their trade
performance.
        This method should be called after a trade (executed by
the collective)
        is eventually closed and its P&L is known.

        Args:
            agent_name (str): The name of the agent whose weight
is being updated.
            trade_result (Dict[str, Any]): A dictionary containing
the result of the trade,
                                           e.g., {'pnl': 150.0,
'status': 'closed', ...}
        """
        if trade_result['pnl'] > 0:
            self.agent_performance[agent_name]['wins'] += 1
            logger.info(f"Agent {agent_name} recorded a WIN. Wins:
{self.agent_performance[agent_name]['wins']}")
        else:
            self.agent_performance[agent_name]['losses'] += 1
```

```
            logger.info(f"Agent {agent_name} recorded a LOSS.
Losses: {self.agent_performance[agent_name]['losses']}")

        total_trades = self.agent_performance[agent_name]['wins']
+ self.agent_performance[agent_name]['losses']
        if total_trades == 0: # Avoid division by zero
            win_rate = 0.0
        else:
            win_rate = self.agent_performance[agent_name]['wins']
/ total_trades

        # Calculate new weight (winning agents get more say,
bounded between 0.5 and 1.5)
        # The +1 in the denominator for win_rate in the conceptual
code was for very early stages.
        # Here, we'll use a direct win_rate for simplicity in
weight calculation.
        new_weight = 0.5 + win_rate  # Range from 0.5 (0% win
rate) to 1.5 (100% win rate)
        self.agent_performance[agent_name]['weight'] = new_weight
        logger.info(f"Updated weight for {agent_name}:
{new_weight:.2f} (Win Rate: {win_rate:.2%})")

    async def start_listening(self):
        # ... (existing code for start_listening) ...

    async def _evaluate_consensus(self):
        """
        Evaluates if a consensus has been reached for any proposed
trade,
        now using weighted voting.
        """
        current_time = datetime.now()

        # Only check for consensus every X seconds or when enough
signals accumulate
        if (current_time - self.last_consensus_check_time) <
timedelta(seconds=1) and \
           not any(len(v) >= self.consensus_threshold for v in
self.proposed_signals.values()):
            return # Don't check too frequently unless a threshold
is met

        self.last_consensus_check_time = current_time

        trades_to_execute = []
        assets_to_clear = []
```

```
        for asset, proposals in
list(self.proposed_signals.items()):
            # Filter out old proposals based on consensus window
            recent_proposals = [p for p in proposals if
(current_time - datetime.fromisoformat(p['timestamp'])) <
self.consensus_window]
            self.proposed_signals[asset] = recent_proposals

            if not recent_proposals: # If no recent proposals
left, clear the asset
                assets_to_clear.append(asset)
                continue

            # NEW: Calculate total weighted agreement for this
asset/direction
            weighted_long_agreement = 0.0
            weighted_short_agreement = 0.0
            proposing_agent_names = set()

            for proposal in recent_proposals:
                agent_name = proposal['agent_name']
                agent_weight =
self.agent_performance[agent_name]['weight']
                proposing_agent_names.add(agent_name) # Track all
agents proposing

                if proposal['direction'] == 'long':
                    weighted_long_agreement += agent_weight
                elif proposal['direction'] == 'short':
                    weighted_short_agreement += agent_weight

            # Define a weighted consensus threshold (e.g., sum of
weights must exceed a certain value)
            # For 5 agents with initial weight 1.0, total max
weight is 5.0.
            # If 3 agents agree (initial weight 1.0), sum = 3.0.
Let's use 3.0 as a base weighted threshold.
            weighted_consensus_threshold = 3.0 # Can be adjusted
or dynamic

            if weighted_long_agreement >=
weighted_consensus_threshold or \
               weighted_short_agreement >=
weighted_consensus_threshold:

                final_direction = None
                if weighted_long_agreement >
weighted_short_agreement:
```

```
                    final_direction = 'long'
                elif weighted_short_agreement >
weighted_long_agreement:
                    final_direction = 'short'
                else: # A tie in weighted agreement, maybe skip or
use another tie-breaker
                    logger.info(f"Weighted agreement tie for
{asset}. Skipping for now.")
                    continue

                logger.info(f"Consensus met for {asset}
(Weighted)! Direction: {final_direction.upper()}. "
                            f"Long Weight:
{weighted_long_agreement:.2f}, Short Weight:
{weighted_short_agreement:.2f}. "
                            f"Proposing Agents: {',
'.join(proposing_agent_names)}")

                # Calculate average size and expected profit from
all *recent_proposals*
                # This ensures all proposals contribute to the
final collective trade parameters.
                avg_size = sum(p['size'] for p in
recent_proposals) / len(recent_proposals)
                avg_expected_profit = sum(p['expected_profit'] for
p in recent_proposals) / len(recent_proposals)

                trade_package = {
                    'financial_asset': asset,
                    'direction': final_direction,
                    'size': avg_size,
                    'expected_profit': avg_expected_profit,
                    'collective_decision_time':
datetime.now().isoformat(),
                    'proposing_agents':
list(proposing_agent_names), # Agents whose proposals contributed
                    'metadata': recent_proposals[0]['metadata'] #
Use metadata from one proposal
                }
                trades_to_execute.append(trade_package)
                assets_to_clear.append(asset)
            # Else-if to clear assets with no recent proposals
            elif not recent_proposals and asset in
self.proposed_signals:
                assets_to_clear.append(asset)

        for trade_package in trades_to_execute:
            # Execute the trade and then update weights for
```

```
contributing agents
            execution_result = await
self._execute_collective_trade(trade_package)
            # Assuming _execute_collective_trade returns P&L or
status,
            # you'd then pass the P&L from the *closed* position
here.
            # For now, let's simulate a positive P&L after
execution for demo purposes
            # In a real system, this P&L would come from
monitoring the closed position.

            # SIMULATED P&L for weight update (REPLACE WITH REAL
P&L LATER)
            simulated_pnl = trade_package['expected_profit'] *
np.random.uniform(0.5, 1.5) if execution_result['status'] ==
'executed' else -trade_package['size'] * 0.1 # Small loss on fail

            # Update weights for all agents that proposed this
winning/losing trade
            for agent_name in trade_package['proposing_agents']:
                self.update_agent_weight(agent_name, {'pnl':
simulated_pnl})


        for asset in assets_to_clear:
            if asset in self.proposed_signals:
                del self.proposed_signals[asset]
```

Implementation Steps:

1. **Open rawe_system/src/core/consensus_engine.py**.
2. **Add self.agent_performance**: Locate the __init__ method and add the line: self.agent_performance: Dict[str, Dict[str, Any]] = defaultdict(lambda: {'wins': 0, 'losses': 0, 'weight': 1.0})
3. **Add update_agent_weight method**: Insert the update_agent_weight method directly after the __init__ method.
4. **Update _evaluate_consensus method**: Replace the entire content of the _evaluate_consensus method with the new logic provided above. This includes the new weighting calculations and the simulated P&L update.

Next Steps:

With Weighted Voting by Performance integrated into the ConsensusEngine, our collective intelligence will now dynamically adjust agent influence. This is a powerful self-optimization mechanism.
Run your scripts/run_collective_rawe.py script. You'll now see logs from the ConsensusEngine showing how weights are updated (with the simulated P&L for now), and how proposals are evaluated using these weights.

Phase 2: Implementing the Meta-Learning Layer
Objective: Develop a MetaLearner component that dynamically analyzes the current market regime (e.g., high volatility, institutional decay, normal) and intelligently selects which RAWE agents (personalities) should be active and influential for that specific environment.
This will significantly enhance the collective's ability to adapt and achieve optimal performance across diverse market states.
Step 1: Create rawe_system/src/core/meta_learner.py (New File)
This file will contain the MetaLearner class, responsible for analyzing the market state and recommending active agents.
rawe_system/src/core/meta_learner.py (New File)

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

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

class MetaLearner:
    """
    Learns optimal agent combinations for different market conditions.
    Analyzes high-level market state metrics to determine the current regime
    and suggests which personalities are best suited for it.
    """

    def __init__(self):
        # Placeholder for learned patterns and optimal configurations
        # In a real system, these would be trained over time using historical data
        # and agent performance in different regimes.
        self.market_condition_patterns: Dict[str, Dict[str, Any]] = {
            "high_volatility_chaos": {"nvx_min": 80, "entropy_min": 0.7, "curvature_max": -0.5},
            "institutional_decay": {"curvature_max": -1.5, "nvx_max": 70}, # More negative curvature
            "normal_market": {"nvx_min": 40, "nvx_max": 70, "entropy_max": 0.5},
            "low_volatility_stable": {"nvx_max": 40, "entropy_max": 0.3}
        }

        self.optimal_agent_configs: Dict[str, List[str]] = {
            "high_volatility_chaos": ["The Aggressor", "The Contrarian Maverick"], # Placeholder
names from example
            "institutional_decay": ["The Conservative Guardian", "The Topological Observer"],
            "normal_market": ["The Balanced Arbitrator", "The Alpha Aggressor", "The Flux
Follower"],
            "low_volatility_stable": ["The Balanced Arbitrator", "The Conservative Guardian"]
        }
        logger.info("MetaLearner initialized with predefined market condition patterns and agent
configurations.")

    async def analyze_market_state(self, nvx: float, global_entropy: float, avg_curvature: float) ->
str:
        """
```

```
        Analyzes current market state metrics (NVX, global entropy, average curvature)
        to determine the prevailing market regime.

        Args:
            nvx (float): Current Narrative Volatility Index.
            global_entropy (float): An aggregated measure of systemic entropy/disorder.
            avg_curvature (float): An aggregated measure of the 'curvature' (stability) of the financial
manifold.

        Returns:
            str: The identified market regime (e.g., "high_volatility_chaos", "institutional_decay",
"normal_market").
        """
        logger.debug(f"Analyzing market state: NVX={nvx:.2f}, Entropy={global_entropy:.2f},
Curvature={avg_curvature:.2f}")

        # Evaluate against predefined patterns
        if nvx > self.market_condition_patterns["high_volatility_chaos"]["nvx_min"] and \
           global_entropy > self.market_condition_patterns["high_volatility_chaos"]["entropy_min"]:
           # Potentially also check avg_curvature here
            logger.info("Market Regime: high_volatility_chaos")
            return "high_volatility_chaos"

        if avg_curvature < self.market_condition_patterns["institutional_decay"]["curvature_max"]:
            logger.info("Market Regime: institutional_decay")
            return "institutional_decay"

        if nvx < self.market_condition_patterns["low_volatility_stable"]["nvx_max"] and \
           global_entropy < self.market_condition_patterns["low_volatility_stable"]["entropy_max"]:
            logger.info("Market Regime: low_volatility_stable")
            return "low_volatility_stable"

        # Default or 'normal' condition
        logger.info("Market Regime: normal_market")
        return "normal_market"

    async def select_active_agents(self, market_state: str, all_agent_names: List[str]) -> List[str]:
        """
        Chooses which agents should be actively participating based on the identified market
state.

        Args:
            market_state (str): The identified market regime.
            all_agent_names (List[str]): A list of all available agent personality names.

        Returns:
            List[str]: A list of agent names recommended to be active for the current market state.
        """
```

active_agents = self.optimal_agent_configs.get(market_state, all_agent_names)
logger.info(f"For '{market_state}' market, recommended active agents: {active_agents}")
return active_agents

Implementation Steps:
* Create the file: Inside your rawe_system/src/core/ directory, create a new file named meta_learner.py.
* Paste the code: Copy and paste the code block above into rawe_system/src/core/meta_learner.py.

Step 2: Integrate MetaLearner into scripts/run_collective_rawe.py
We will instantiate the MetaLearner, use it to determine the active agents, and then modify the main loop to only run cycles for these active agents.
rawe_system/scripts/run_collective_rawe.py (Modifications)
* Add Import: At the top of the file, add:
  from src.core.meta_learner import MetaLearner # New import

* Instantiate MetaLearner: Inside run_collective_rawe(), after Redis initialization and agent instantiation:
  # ... after agent initialization loop ...

# Initialize MetaLearner
meta_learner = MetaLearner()
collective_logger.info("MetaLearner initialized.")

# Launch the Consensus Engine
consensus_engine = ConsensusEngine(redis_client, broker, subscribe_channel=RAWE_SIGNAL_CHANNEL)
consensus_task = asyncio.create_task(consensus_engine.start_listening())
collective_logger.info("ConsensusEngine launched and listening for signals.")

* Modify the Main Trading Loop to use MetaLearner:
  The current loop iterates through all agents. We need to select active agents first.
  Find the for cycle in range(10): loop and replace the entire loop's content with the following:
  # 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()
    # For global_entropy and avg_curvature, we need to aggregate from narratives
    # This is a simplification for now; in a real system, these would come from
    # deeper analysis of the entire manifold, perhaps from a dedicated monitoring module.
    current_global_entropy = sum(n.volatility_30d for n in narrative_engine.narrative_assets.values()) / len(narrative_engine.narrative_assets)
    # Use Ricci curvature from a sample narrative or aggregate for simplicity.
    # This assumes detect_topological_stress can give us a sense of average curvature.

```
    # For a truly global avg_curvature, we'd need a method in narrative_engine or topology.
    # For now, let's just pick one or take an average if possible, or use a dummy.
    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)

    # Step 2: Select active agents based on market state
    all_agent_names = list(PERSONALITY_PROFILES.keys()) # Get all possible agent names
    active_agent_names = await meta_learner.select_active_agents(market_state,
all_agent_names)

    # Step 3: Run cycles only for active agents and update narrative states
    agent_cycle_tasks = []
    for name, agent in rawe_agents.items():
        if name in active_agent_names:
            agent_cycle_tasks.append(
                asyncio.create_task(
                    _run_single_agent_cycle(agent, narrative_engine, cycle, is_active=True)
                )
            )
        else:
            collective_logger.info(f"Agent '{name}' is INACTIVE for this cycle ({market_state}
market).")
            # Still pass through the narrative engine update for inactive agents, just don't
scan/execute
            agent_cycle_tasks.append(
                asyncio.create_task(
                    _run_single_agent_cycle(agent, narrative_engine, cycle, is_active=False)
                )
            )

    # Wait for all agents (active and inactive for narrative updates) to complete their cycle tasks
    await asyncio.gather(*agent_cycle_tasks)

    # Update collective performance after all agents have run
    collective_logger.info(f"\n📊 COLLECTIVE PERFORMANCE AFTER CYCLE {cycle + 1}:")
    total_realized_pnl = sum(agent.pnl_tracker['realized'] for agent in rawe_agents.values())
    total_unrealized_pnl = sum(sum(agent.calculate_position_pnl(p) for p in
agent.active_positions.values()) for agent in rawe_agents.values())
    collective_logger.info(f"   Total Realized P&L: ${total_realized_pnl:.2f}")
    collective_logger.info(f"   Total Unrealized P&L: ${total_unrealized_pnl:.2f}")
    collective_logger.info(f"   Overall P&L: ${total_realized_pnl + total_unrealized_pnl:.2f}")
    collective_logger.info(f"   Active Agent Count: {len(active_agent_names)}")

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

# After the loop, the final report for each agent is handled in their agent_lifecycle
```

... rest of the shutdown logic ...

* Create a helper function _run_single_agent_cycle: This function will encapsulate the logic for a single agent's cycle, making it easier to manage active/inactive states. Add this before async def run_collective_rawe(): (i.e., at the global scope).

```
  # rawe_system/scripts/run_collective_rawe.py
# ... (imports and logging config) ...

async def _run_single_agent_cycle(agent_instance: UnifiedArbitrageSystem,
narrative_engine_instance: NarrativeVolatilityEngine, cycle: int, is_active: bool):
    """Runs a single cycle for a given RAWE agent, potentially skipping signal
generation/execution if inactive."""
    if is_active:
        agent_instance.logger.info(f"📍 {agent_instance.personality_name} (ACTIVE) -
ARBITRAGE CYCLE {cycle + 1}")
        signals = await agent_instance.scan_arbitrage_universe()
        agent_instance.logger.info(f"🔍 {agent_instance.personality_name} found {len(signals)}
signals.")

        if signals:
            await agent_instance.execute_arbitrage_strategy(signals) # This method now publishes
    else:
        agent_instance.logger.debug(f"📍 {agent_instance.personality_name} (INACTIVE) -
ARBITRAGE CYCLE {cycle + 1}. Skipping signal generation/execution.")
        # Inactive agents still update narrative states if desired, but not trade
        # For this example, narrative update is centralized in run_collective_rawe loop
        pass # No specific action needed for inactive agents in this function, as narrative_engine
update is outside.

    # Ensure narrative states update (this part would ideally be centralized for all agents if not
already)
    # However, for this demo, the narrative engine update is in the main loop, so agents just need
to be 'active' to run their part.
    # This function primarily handles the agent's scan/execute role.

# ... (rest of the run_collective_rawe function) ...
```

* Remove the old agent_lifecycle function: The new _run_single_agent_cycle and the main loop's logic replace the previous agent_lifecycle function. Remove the existing agent_lifecycle definition and the agent_tasks.append(asyncio.create_task(agent_lifecycle(agent))) line.

Final Preparations and Execution
* Ensure rawe_system/src/core/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
```

You will now observe the MetaLearner dynamically determining the market state and selecting specific agents to be active during each collective cycle. The logs will clearly indicate which agents are participating in signal generation.