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.

Are you ready to move on to **Step 2: Meta-Learning Layer**? This will be about optimizing agent combinations for different market conditions.