We'll implement this in three parts:

1. **Enable Agents to Publish Signals:** Modify the UnifiedArbitrageSystem to send its top signals to a Redis channel.
2. **Create a Central Consensus Engine:** A new module that listens to these signals, applies a collaboration logic, and decides on final trades.
3. **Orchestrate the Collective:** Update the run_collective_rawe.py script to manage the Redis connection and launch the ConsensusEngine alongside our RAWE agents.

Part 1: Enabling RAWE Agents to Publish Signals

First, we'll modify the UnifiedArbitrageSystem to communicate its signals via Redis instead of executing them directly.

1. Update rawe_system/src/core/unified_arbitrage_system.py

We'll add a Redis client and a method to publish signals.

- **Add Imports:** At the top of the file, add:

```
import redis.asyncio as redis
import json
```

- **Modify __init__:** Update the constructor to accept a Redis client and a channel name.

```
class UnifiedArbitrageSystem:
    """Master system orchestrating narrative-capital arbitrage"""

    def __init__(self, narrative_engine:
NarrativeVolatilityEngine,
                 personality_name: str = "The Balanced
Arbitrator",
                 redis_client: Optional[redis.Redis] = None, #
New: Redis client
                 signal_channel: str = "rawe_signals"):      #
New: Redis channel for signals
        self.narrative_engine = narrative_engine
        self.active_positions = {}
        self.signal_history = []
        self.pnl_tracker = {
            'realized': 0.0,
            'unrealized': 0.0,
            'positions': []
        }
        self.personality_name = personality_name
        self.logger =
logging.getLogger(f'rawe_system.UnifiedArbitrageSystem.{personalit
y_name}')
        self.profile = PERSONALITY_PROFILES.get(personality_name,
PERSONALITY_PROFILES["The Balanced Arbitrator"])
        self.broker = None # Broker is set via set_broker for
collective execution

        self.redis_client = redis_client # Store Redis client
        self.signal_channel = signal_channel # Store signal
channel

        self.logger.info(f"Initialized with personality:
{self.personality_name}. Profile: {self.profile}")
        if self.redis_client:
            self.logger.info(f"Agent {self.personality_name}
configured to publish to Redis channel: {self.signal_channel}")
```

- **Add publish_signal method:** This new method will serialize and send signals.

```
    async def publish_signal(self, signal: ArbitrageSignal):
        """Publishes a single arbitrage signal to Redis for
collective review."""
        if self.redis_client:
            try:
                # Ensure ArbitrageSignal has a .to_dict() method
for serialization
                signal_data = signal.to_dict()
                message = json.dumps(signal_data)
                await
self.redis_client.publish(self.signal_channel, message)
                self.logger.debug(f"Agent {self.personality_name}
published signal: {signal.narrative_id} ->
{signal.financial_asset}")
            except Exception as e:
                self.logger.error(f"Failed to publish signal for
{self.personality_name}: {e}")
        else:
            self.logger.warning(f"Redis client not set for
{self.personality_name}. Cannot publish signals.")
```

- **Modify execute_arbitrage_strategy:** Change it to *publish* signals instead of executing trades directly.Find these lines in execute_arbitrage_strategy:

```
            # Execute trade using the integrated broker
            if self.broker:
                execution_result = await
self.broker.execute_trade(trade_package)
            else:
                self.logger.error("Broker not initialized. Cannot
execute trade.")
                execution_result = {'status': 'failed', 'reason':
'Broker not set'}

            if execution_result['status'] == 'executed':
                position_id =
f"{signal.narrative_id}_{signal.financial_asset}"
                self.active_positions[position_id] = {
                    'trade': trade_package,
                    'execution': execution_result,
                    'entry_time': datetime.now()
                }
                self.logger.info(f"✅ Executed: 
{strategy['strategy']} on {signal.financial_asset} "
                                 f"(Order ID: 
{execution_result.get('order_id')}). "
                                 f"Narrative: 
{signal.narrative_id}, "
                                 f"Expected profit: 
${signal.expected_profit:.2f}")
            else:
                self.logger.warning(f"❌ Trade failed for 
{signal.financial_asset}. Reason: 
{execution_result.get('reason')}")
```

Replace them with this new logic that publishes signals:

```
        if strategy['confidence'] > 0.7:
            # Build a 'proposed trade' package (similar to 
trade_package but not executed yet)
            proposed_trade = {
                'agent_name': self.personality_name, # Identify 
which agent proposed this
                'timestamp': datetime.now(),
                'narrative_id': signal.narrative_id,
                'financial_asset': signal.financial_asset,
                'direction': 'long' if signal.signal_type == 
'narrative_leads' else 'short',
                'size': self.calculate_position_size(signal, 
strategy),
                'strategy': strategy['strategy'],
                'expected_profit': signal.expected_profit, # 
Include for consensus
                'risk_score': signal.risk_score,         # Include 
for consensus
                'metadata': signal.metadata
            }

            # Instead of executing, publish this proposed trade 
for collective review
            await self.publish_signal(
                ArbitrageSignal( # Create a dummy signal to fit 
the publish_signal signature
                    timestamp=proposed_trade['timestamp'],
                    narrative_id=proposed_trade['narrative_id'],

financial_asset=proposed_trade['financial_asset'],
                    signal_type=signal.signal_type, # Use original 
signal type
```

```
                    strength=signal.strength,

expected_profit=proposed_trade['expected_profit'],
                    risk_score=proposed_trade['risk_score'],
                    metadata=proposed_trade['metadata'] # Pass the
full proposed trade as metadata for now
                )
            )
            self.logger.info(f"⬆️ {self.personality_name} proposed
trade: {proposed_trade['financial_asset']} "
                             f"({proposed_trade['direction']}
{proposed_trade['size']:.2f})")
        else:
            self.logger.debug(f"Strategy confidence too low
({strategy['confidence']:.2f}) for {self.personality_name}.
Skipping trade.")
```

Crucial: Ensure the ArbitrageSignal dataclass (defined earlier in unified_arbitrage_system.py) has the to_dict method as provided in the previous turn. If not, add it now.

```
@dataclass
class ArbitrageSignal:
    # ... (existing fields) ...

    def to_dict(self):
        return {
            "timestamp": self.timestamp.isoformat(),
            "narrative_id": self.narrative_id,
            "financial_asset": self.financial_asset,
            "signal_type": self.signal_type,
            "strength": self.strength,
            "expected_profit": self.expected_profit,
            "risk_score": self.risk_score,
            "metadata": self.metadata
        }
```

Part 2: Creating the ConsensusEngine

This new module will handle receiving signals and making collective decisions.

2. Create rawe_system/src/core/consensus_engine.py (New File)

```
# 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

# Import Alpaca Broker (for executing trades)
from src.modules.alpaca_broker import AlpacaBroker
from src.core.unified_arbitrage_system import ArbitrageSignal # To
deserialize signals

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, # e.g., 3 out of 5
agents must agree
                 consensus_window_seconds: int = 10): # Collect
signals for X seconds
        self.redis_client = redis_client
        self.broker = broker
        self.subscribe_channel = subscribe_channel
        self.consensus_threshold = consensus_threshold
        self.consensus_window =
timedelta(seconds=consensus_window_seconds)

        self.proposed_signals: Dict[str, List[Dict[str, Any]]] =
defaultdict(list) # {asset: [list of proposed_trade dicts]}
        self.last_consensus_check_time = datetime.now()

    async def start_listening(self):
        """Starts the Redis subscription and processes incoming
signals."""
        logger.info(f"ConsensusEngine starting to listen on channel:
{self.subscribe_channel}")
        pubsub = self.redis_client.pubsub()
        await pubsub.subscribe(self.subscribe_channel)

        async for message in pubsub.listen():
            if message['type'] == 'message':
                try:
                    data = json.loads(message['data'].decode('utf-8'))
                    # The proposed trade details are inside the
metadata of the ArbitrageSignal
                    proposed_trade_data = data['metadata']
                    asset = proposed_trade_data['financial_asset']

self.proposed_signals[asset].append(proposed_trade_data)
                    logger.debug(f"Received proposal for {asset} from
{proposed_trade_data['agent_name']}. Total proposals for {asset}:
{len(self.proposed_signals[asset])}")

                    await self._evaluate_consensus() # Evaluate after
each new signal
                except json.JSONDecodeError as e:
                    logger.error(f"Failed to decode Redis message:
{e}. Message: {message['data']}")
                except Exception as e:
                    logger.error(f"Error processing message in
ConsensusEngine: {e}")

    async def _evaluate_consensus(self):
        """Evaluates if a consensus has been reached for any proposed
trade."""
        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()): #
Iterate on a copy
            # 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 # Update
with only recent proposals

            if len(recent_proposals) >= self.consensus_threshold:
                logger.info(f"Consensus met for {asset}!
{len(recent_proposals)} agents agreed within window.")
```

```
                # Basic consensus logic:
                # Average size and expected profit, majority direction
                directions = [p['direction'] for p in
recent_proposals]
                long_count = directions.count('long')
                short_count = directions.count('short')

                if long_count > short_count:
                    final_direction = 'long'
                elif short_count > long_count:
                    final_direction = 'short'
                else: # Tie-breaker: choose the one with higher
average expected profit or skip
                    logger.info(f"Tie in direction for {asset}.
Skipping for now.")
                    continue

                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)

                # Check if this trade is already active or in a
cooling-off period
                # (Future enhancement: prevent duplicate trades)

                trade_package = {
                    'financial_asset': asset,
                    'direction': final_direction,
                    'size': avg_size,
                    'expected_profit': avg_expected_profit,
                    'collective_decision_time':
datetime.now().isoformat(),
                    'proposing_agents': [p['agent_name'] for p in
recent_proposals],
                    'metadata': recent_proposals[0]['metadata'] # Take
metadata from one of the proposals
                }
                trades_to_execute.append(trade_package)
                assets_to_clear.append(asset) # Mark for clearing
after execution
            elif not recent_proposals and asset in
self.proposed_signals:
                assets_to_clear.append(asset) # Clear if no recent
proposals remain

        for trade_package in trades_to_execute:
            await self._execute_collective_trade(trade_package)
        for asset in assets_to_clear:
            if asset in self.proposed_signals:
                del self.proposed_signals[asset] # Clear processed or
expired signals

    async def _execute_collective_trade(self, trade_package: Dict[str,
Any]):
        """Executes a trade decided by the collective through the
shared broker."""
        logger.info(f"🚀 COLLECTIVE DECISION: Attempting to
{trade_package['direction'].upper()} "
                    f"{trade_package['size']:.2f} of
{trade_package['financial_asset']} "
                    f"(Avg Profit:
${trade_package['expected_profit']:.2f}) "
                    f"proposed by {',
'.join(trade_package['proposing_agents'])}")

        execution_result = await
self.broker.execute_trade(trade_package)

        if execution_result['status'] == 'executed':
            logger.info(f"✅ COLLECTIVE EXECUTION SUCCESS:
{trade_package['financial_asset']} Order ID:
{execution_result.get('order_id')}")
            # Here, you might update a centralized PnL tracker or
position manager
        else:
            logger.error(f"❌ COLLECTIVE EXECUTION FAILED:
{trade_package['financial_asset']}. Reason:
{execution_result.get('reason')}")
```

Implementation Steps:

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

Part 3: Orchestrating the Collective in scripts/run_collective_rawe.py

Finally, we update our main script to set up Redis, instantiate the ConsensusEngine, and ensure agents publish instead of execute.

3. Update rawe_system/scripts/run_collective_rawe.py

- **Add Imports:** At the top of the file, add:
  ```
  import redis.asyncio as redis # Already there, but confirm
  # ...
  ```

```
from src.core.consensus_engine import ConsensusEngine # New import
```

- **Initialize Redis Client:** Inside run_collective_rawe(), after collective_logger.info("=" * 80):

```
# Initialize Redis client
# Assuming Redis is running on localhost:6379 (default)
redis_client = redis.Redis(host='localhost', port=6379, db=0)
try:
    await redis_client.ping()
    collective_logger.info("🗄️ Connected to Redis server.")
except redis.exceptions.ConnectionError as e:
    collective_logger.error(f"Failed to connect to Redis: {e}.
Please ensure Redis server is running.")
    return # Exit if Redis is not available
```

- **Define Signal Channel:** Add a constant for the Redis channel:

```
# ... above the agent instantiation loop ...
RAWE_SIGNAL_CHANNEL = "rawe_collective_signals"
```

- **Modify Agent Instantiation:** Pass the redis_client and signal_channel to each agent.

```
# Instantiate multiple RAWE agents with different personalities
rawe_agents = {}
agent_tasks = []
for name in PERSONALITY_PROFILES.keys():
    # Pass redis_client and signal_channel to each
UnifiedArbitrageSystem instance
    agent = UnifiedArbitrageSystem(narrative_engine,
                                   personality_name=name,
                                   redis_client=redis_client, #
Pass Redis client

signal_channel=RAWE_SIGNAL_CHANNEL) # Pass channel
    agent.set_broker(broker)
    rawe_agents[name] = agent
    collective_logger.info(f"Agent '{name}' initialized.")
```

- **Launch ConsensusEngine:** After initializing agents, but before launching agent tasks:

```
# 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 Agent agent_lifecycle:** Remove the direct trade execution. Change await agent_instance.execute_arbitrage_strategy(signals) to:

```
            if signals:
                # Agents now publish signals, ConsensusEngine
```

```
handles execution
                await
agent_instance.execute_arbitrage_strategy(signals) # This method
now publishes
```

(The execute_arbitrage_strategy method was already modified in Part 1 to publish, so no code change is needed here, just a confirmation of its new role.)

- **Ensure graceful shutdown of Redis:** At the very end of run_collective_rawe(), before if __name__ == "__main__"::

```
# ... after await asyncio.gather(*agent_tasks) ...

# Cancel consensus task
consensus_task.cancel()
collective_logger.info("Consensus task cancelled.")

# Close Redis connection
await redis_client.close()
collective_logger.info("Redis connection closed.")

collective_logger.info("\n" + "=" * 80)
collective_logger.info("✅ ALL RAWE AGENTS AND CONSENSUS ENGINE
COMPLETED CYCLES")
collective_logger.info("=" * 80)
```

Final Preparations and Execution

1. **Ensure Redis Server is Running:** You must have a Redis server instance running on localhost:6379 (or wherever you configure it). If you don't have it, you'll need to install and start it (e.g., via Docker, Homebrew, or direct installation).
2. **Verify requirements.txt:** Confirm you have redis installed: pip install redis.
3. **Run the Collective RAWE System:**

```
python scripts/run_collective_rawe.py
```

Now, your RAWE agents will operate as a collective. Each personality will identify signals based on its unique biases and publish them to Redis. The ConsensusEngine will then pick up these proposals, apply its consensus logic, and only execute trades when a collective "agreement" is reached.

This is a massive step towards building a truly intelligent and resilient arbitrage system! Let me know how the collective performs in its initial run.