Creating a **miniature collective intelligence** designed to identify and exploit the belief-reality gap from multiple angles. This mirrors the "Consensual Hallucination Engine" metaphor, where diverse inputs converge to shape a unified understanding and action.
This approach will provide:

- **Robustness:** Less reliance on a single set of parameters.
- **Wider Signal Capture:** Different "personalities" might identify arbitrage opportunities missed by others.
- **Enhanced Decision-Making:** Through collaborative filtering and consensus building.

Architectural Vision: The RAWE Collective

To achieve this, we'll evolve our system into a multi-agent architecture:

1. **RAWE Agent Class:** A slight modification or wrapper around UnifiedArbitrageSystem that accepts a "personality profile."
2. **Personality Profiles:** Configuration dictionaries defining specific behavioral biases (risk tolerance, focus on certain signal types, sizing preferences, etc.).
3. **Central Dispatcher/Orchestrator:** A new main script responsible for launching and managing multiple RAWE agent instances.
4. **Inter-Agent Communication Layer:** A mechanism for agents to share their top signals and proposed trades. Given that redis is already in our requirements.txt, it's an excellent choice for a high-speed, in-memory message bus.
5. **Consensus & Aggregation Module:** A new component that listens to all agents, aggregates their recommendations, applies a collaborative logic, and issues final, collective trades.

Phase 1: Defining Personalities and Modifying the RAWE Agent

Let's start by defining the distinct "personalities" for our five engines and making our UnifiedArbitrageSystem flexible enough to adopt these.

Step 1: Define Personality Profiles

We'll create a personalities.py file in the config/ directory. Each personality will be a dictionary that maps to configurable parameters within UnifiedArbitrageSystem or its sub-modules.
rawe_system/config/personalities.py (New File)

```
# rawe_system/config/personalities.py

PERSONALITY_PROFILES = {
    "The Alpha Aggressor": {
        "risk_aversion": 0.2, # Lower risk aversion, more aggressive
        "min_expected_profit_multiplier": 1.5, # Only goes for high
profit signals
        "signal_type_preference": ["narrative_leads", "divergence"], #
Prefers narrative-driven and complex signals
        "position_size_multiplier": 1.2, # Takes larger positions
        "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}, # Less
sensitive to collapse warnings on entry
```

```
        "kelly_win_loss_ratio": 2.5 # Assumes higher reward/risk
    },
    "The Conservative Guardian": {
        "risk_aversion": 0.8, # High risk aversion, cautious
        "min_expected_profit_multiplier": 0.8, # Even small profits
are good
        "signal_type_preference": ["capital_leads", "divergence"], #
Prefers capital-driven stability
        "position_size_multiplier": 0.7, # Takes smaller positions
        "coherence_threshold_for_trade": {'AAA': 0.9, 'AA': 0.8, 'A':
0.7, 'BBB': 0.6, 'BB': 0.5, 'B': 0.4, 'C': 0.3, 'D': 0.2}, # Only
trades highly coherent narratives
        "kelly_win_loss_ratio": 1.5 # Assumes lower reward/risk
    },
    "The Flux Follower": {
        "risk_aversion": 0.5,
        "min_expected_profit_multiplier": 1.0,
        "signal_type_preference": ["narrative_leads"], # Strongly
prefers narrative-driven signals
        "position_size_multiplier": 1.0,
        "flux_sensitivity_multiplier": 1.5, # Special parameter: more
sensitive to narrative flux velocity
        "coherence_threshold_for_trade": {'AAA': 0.0, 'AA': 0.0, 'A':
0.0, 'BBB': 0.0, 'BB': 0.0, 'B': 0.0, 'C': 0.3, 'D': 0.0}, # Avoids
D-rated, but otherwise open
        "kelly_win_loss_ratio": 2.0
    },
    "The Topological Observer": {
        "risk_aversion": 0.6,
        "min_expected_profit_multiplier": 1.2,
        "signal_type_preference": ["divergence"], # Focus on complex
topological shifts
        "position_size_multiplier": 0.9,
        "topological_stress_sensitivity_multiplier": 1.5, # Special
parameter: more sensitive to entropy/stress
        "coherence_threshold_for_trade": {'AAA': 0.8, 'AA': 0.7, 'A':
0.6, 'BBB': 0.5, 'BB': 0.4, 'B': 0.3, 'C': 0.2, 'D': 0.1}, # Very
sensitive to any decay warnings
        "kelly_win_loss_ratio": 1.8
    },
    "The Balanced Arbitrator": {
        "risk_aversion": 0.4,
        "min_expected_profit_multiplier": 1.0,
        "signal_type_preference": ["narrative_leads", "capital_leads",
"divergence"], # Equal preference for all
        "position_size_multiplier": 1.0,
        "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}, # Default
```

```
open
        "kelly_win_loss_ratio": 2.0
    }
}
```

Implementation Steps:

1. **Create config directory (if it doesn't exist):** mkdir -p rawe_system/config
2. **Create personalities.py:** Inside rawe_system/config/, create a new file named personalities.py.
3. **Paste the code:** Copy the PERSONALITY_PROFILES dictionary into this new file.

Step 2: Modify UnifiedArbitrageSystem for Personality Configuration

We will update the __init__ method of UnifiedArbitrageSystem to accept a personality_profile dictionary and apply its settings. We'll also update methods that rely on these parameters.

rawe_system/src/core/unified_arbitrage_system.py (Modifications)

1. **Import Personality Profiles:** Add this import at the top of the file:

   ```
   # ... other imports ...
   from config.personalities import PERSONALITY_PROFILES # New:
   Import personalities
   ```

2. **Modify __init__ to accept personality_name and load profile:**

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

       def __init__(self, narrative_engine:
   NarrativeVolatilityEngine, personality_name: str = "The Balanced
   Arbitrator"): # Updated signature
           self.narrative_engine = narrative_engine
           self.active_positions = {}
           self.signal_history = []
           self.pnl_tracker = {
               'realized': 0.0,
               'unrealized': 0.0,
               'positions': []
           }
           self.logger =
   logging.getLogger(f'rawe_system.UnifiedArbitrageSystem.{personalit
   y_name}') # Logger with personality name
           self.broker = None

           self.personality_name = personality_name
           self.profile = PERSONALITY_PROFILES.get(personality_name,
   PERSONALITY_PROFILES["The Balanced Arbitrator"]) # Load profile

           self.logger.info(f"Initialized with personality:
   {self.personality_name}. Profile: {self.profile}")
   ```

3. **Update scan_arbitrage_universe to filter by signal_type_preference:** This changes how scan_arbitrage_universe selects signals, allowing each personality to focus on its preferred types.

```
# In UnifiedArbitrageSystem class, inside scan_arbitrage_universe
method, before `signals.append(signal)`:
            if self.is_tradeable_divergence(narrative_data,
liquidity_signal, topology_signal):
                # NEW: Filter by personality's preferred signal
types
                signal_type =
self.classify_signal_type(topology_signal, flux_signal)
                if signal_type not in
self.profile.get("signal_type_preference", ["narrative_leads",
"capital_leads", "divergence"]):
                    self.logger.debug(f"Signal type {signal_type}
not preferred by {self.personality_name}. Skipping.")
                    continue # Skip this signal if not preferred

                signal = ArbitrageSignal(
                    timestamp=datetime.now(),
                    narrative_id=narrative.id,
                    financial_asset=asset,
                    signal_type=signal_type, # Use the classified
type
                    strength=topology_signal['signal_strength'] *
flux_signal['memetic_impact'],

expected_profit=self.calculate_expected_profit(narrative_data,
liquidity_signal),
                    risk_score=topology_signal['entropy'],
                    metadata={
                        'nvx': nvx,
                        'topology': topology_signal,
                        'flux': flux_signal,
                        'liquidity': liquidity_signal
                    }
                )
                # NEW: Filter by min_expected_profit_multiplier
                if signal.expected_profit <
(narrative.volatility_30d * 1000 *
self.profile.get("min_expected_profit_multiplier", 1.0)):
                     self.logger.debug(f"Signal profit
{signal.expected_profit:.2f} below threshold for
{self.personality_name}. Skipping.")
                     continue

                signals.append(signal)
```

4. **Update calculate_position_size to use position_size_multiplier and kelly_win_loss_ratio:** This ensures each personality's risk tolerance and sizing preference is applied.

   ```
   # In UnifiedArbitrageSystem class, inside calculate_position_size
   method:
       win_prob = strategy['confidence']
       loss_prob = 1 - win_prob
       # NEW: Use personality's win_loss_ratio
       win_loss_ratio = self.profile.get("kelly_win_loss_ratio", 2.0)

       kelly_fraction = (win_prob * win_loss_ratio - loss_prob) /
   win_loss_ratio
       kelly_fraction = max(0.0, min(1.0, kelly_fraction))

       position_fraction = min(kelly_fraction, 0.25)

       position_fraction *= (1 - signal.risk_score)

       # NEW: Apply personality's position size multiplier
       position_fraction *=
   self.profile.get("position_size_multiplier", 1.0)

       # ... rest of the method (bottleneck_factor, etc.) ...
   ```

5. **Update is_tradeable_divergence to use coherence_threshold_for_trade:** This allows personalities to have different tolerances for institutional decay.

   ```
   # In UnifiedArbitrageSystem class, inside is_tradeable_divergence
   method:
       # ... existing code ...

       # NEW: Apply personality's coherence threshold for trading
       narrative_coherence = narrative_data.get('coherence')
       required_coherence_score =
   self.profile.get("coherence_threshold_for_trade",
   {}).get(narrative_coherence, 0.0)

       # Simplified: if the narrative coherence is below the
   personality's threshold, it's not tradeable
       # For 'D' (decay), the threshold should be low or 0.0 for
   aggressive, high for conservative.
       # This implementation requires more nuance, perhaps a lookup
   table based on actual coherence ratings.
       # For now, let's assume if the personality has a non-zero
   threshold for this coherence, and it's not met, we skip.
       # This is a placeholder for a more sophisticated check.

       # A more robust check for 'coherence_threshold_for_trade':
       # Let's map coherence rating to a numerical value for
   ```

```
comparison (e.g., AAA=10, D=1)
    coherence_mapping = {'AAA': 10, 'AA': 9, 'A': 8, 'BBB': 7,
'BB': 6, 'B': 5, 'C': 4, 'D': 3} # Arbitrary numerical mapping
    current_coherence_value =
coherence_mapping.get(narrative_coherence, 0)

    # If personality has a specific *minimum* coherence value
required to trade this type of signal
    # For simplicity, let's say the value in the profile is a
*minimum* numerical rating.
    min_coherence_for_trade =
coherence_mapping.get(list(self.profile.get("coherence_threshold_f
or_trade", {}).keys())[0], 0) if
self.profile.get("coherence_threshold_for_trade") else 0 # Get
first key's mapped value as proxy

    # This logic needs refinement based on how you want to
interpret the profile's 'coherence_threshold_for_trade'.
    # For now, let's assume the profile sets a *minimum* coherence
level below which they won't trade.
    # This will be simpler: if the profile has a value for this
coherence, and it's not met, skip.
    if self.profile.get("coherence_threshold_for_trade",
{}).get(narrative_coherence, 0.0) > 0.0 and narrative_coherence !=
'D': # Example, if they explicitly set a threshold for non-D, and
it's > 0.0, then check
        # This logic needs to be tied to the specific definition
of coherence_threshold_for_trade.
        # For `The Conservative Guardian`, if
`coherence_threshold_for_trade={'AAA': 0.9}`, it means it only
trades if coherence is 'AAA' and very high.
        # Let's use a simpler interpretation for now: if a
personality has ANY explicit threshold for a coherence rating, it
means they are sensitive to it.
        # For simplicity, if a personality has a specific
threshold set for a coherence rating, and the current narrative's
coherence is not that rating, skip.
        # This is complex due to the dictionary. Let's simplify:
        if narrative_coherence == 'D' and
self.profile.get("coherence_threshold_for_trade", {}).get('D',
0.0) == 0.0:
             # If it's D and personality has 0 threshold for D, it
means they might avoid it.
             # Let's use the risk aversion: higher risk aversion
means more likely to avoid low coherence.
             if self.profile.get("risk_aversion", 0.5) > 0.6: #
For conservative types
                 self.logger.debug(f"{self.personality_name}
avoiding trade due to low coherence (D) and high risk aversion.")
                 return False # Avoid D-rated narratives if risk
averse

    # ... rest of the method ...
    if narrative_data['volatility'] >
adjusted_volatility_threshold:
        # ... existing logging ...
        return True
```

Self-correction: The coherence_threshold_for_trade as a dictionary of coherence_rating: value is tricky for direct comparison. A simpler approach for personality bias against low coherence is better. I'll modify the is_tradeable_divergence logic to simply check the risk_aversion in relation to the coherence_rating. If risk_aversion is high and coherence_rating is low (D or C), the personality is less likely to trade. Let's re-do that block:

```
# In UnifiedArbitrageSystem class, inside is_tradeable_divergence
method, at the very top:
    # NEW: Apply personality's sensitivity to low coherence
(institutional decay)
    current_narrative_coherence = narrative_data.get('coherence')
    risk_aversion = self.profile.get("risk_aversion", 0.5)

    # Example logic: Conservative personalities (high
risk_aversion) avoid low-coherence narratives
    if current_narrative_coherence in ['C', 'D']:
        if risk_aversion > 0.6: # If personality is more
risk-averse
            self.logger.debug(f"{self.personality_name}
(risk_aversion={risk_aversion:.2f}) avoiding trade "
                              f"due to low narrative coherence
({current_narrative_coherence}).")
            return False # This personality won't trade on low
coherence

    # If 'Flux Follower', adjust flux sensitivity (this part is
conceptual for now, would integrate into map_narrative_velocity)
    # flux_sensitivity_multiplier =
self.profile.get("flux_sensitivity_multiplier", 1.0)
    # if flux_sensitivity_multiplier > 1.0:
    #     # This would be integrated into the flux_signal
processing itself,
    #     # perhaps by adjusting the threshold for what
constitutes a strong flux signal.
    #     pass

    # If 'Topological Observer', adjust topological stress
sensitivity
    # topological_stress_sensitivity_multiplier =
self.profile.get("topological_stress_sensitivity_multiplier", 1.0)
    # if topological_stress_sensitivity_multiplier > 1.0:
    #     # This would be integrated into
detect_topological_stress,
    #     # or by adjusting the signal strength threshold based on
entropy.
    #     pass

    # ... rest of the original is_tradeable_divergence method
(from previous turn) ...
```

Phase 2: Central Dispatcher/Orchestrator

Now, we need a main script to launch multiple instances of our personality-driven RAWE agents.

rawe_system/scripts/run_collective_rawe.py (New File)

This will be our new main entry point for the multi-engine system.

```
# scripts/run_collective_rawe.py
import asyncio
import os
from dotenv import load_dotenv
import numpy as np
from datetime import datetime
import json
import logging
import logging.config
import redis.asyncio as redis # For inter-agent communication

# Load environment variables
load_dotenv()

# Import core components
from src.core.numpy_funnyword_eh import NarrativeVolatilityEngine,
NarrativeAsset
from src.core.unified_arbitrage_system import UnifiedArbitrageSystem

# Import brokerage integration (Alpaca)
from src.modules.alpaca_broker import AlpacaBroker

# Import personality profiles
from config.personalities import PERSONALITY_PROFILES

# --- Logging Configuration (Same as before, but ensure unique logs
per agent later) ---
LOGGING_CONFIG = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'standard': {
            'format': '%(asctime)s - %(name)s - %(levelname)s -
%(message)s'
        },
        'detailed': {
            'format': '%(asctime)s - %(name)s - %(levelname)s -
%(funcName)s - %(message)s'
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'INFO',
            'formatter': 'standard'
        },
        'file': {
            'class': 'logging.handlers.RotatingFileHandler',
            'level': 'DEBUG',
            'formatter': 'detailed',
            'filename': 'rawe_collective.log', # New log file for the
collective
            'maxBytes': 10485760,
            'backupCount': 5
        }
    },
    'loggers': {
        '': {  # root logger
            'handlers': ['console', 'file'],
            'level': 'DEBUG',
            'propagate': False
        },
        'rawe_system': { # Main logger for collective operations
            'handlers': ['console', 'file'],
            'level': 'DEBUG',
            'propagate': False
        }
        # Individual agent loggers will be created dynamically
    }
}

logging.config.dictConfig(LOGGING_CONFIG)
collective_logger = logging.getLogger('rawe_system.Collective')

async def run_collective_rawe():
    collective_logger.info("🚀 LAUNCHING RAWE COLLECTIVE ARBITRAGE
SYSTEM")
    collective_logger.info("=" * 80)
    # Initialize Alpaca Broker (shared among all agents)
    alpaca_api_key = os.getenv('ALPACA_API_KEY')
    alpaca_secret_key = os.getenv('ALPACA_SECRET_KEY')

    if not alpaca_api_key or not alpaca_secret_key:
        collective_logger.error("Alpaca API keys not found in .env
file. Please set ALPACA_API_KEY and ALPACA_SECRET_KEY.")
        return

    broker = AlpacaBroker(
        api_key=alpaca_api_key,
        secret_key=alpaca_secret_key,
        paper=True # Start with paper trading
    )
    collective_logger.info("📊 Connected to Alpaca (Paper Trading) -
Broker Shared.")

    # Initialize a single narrative engine (narratives are shared
reality)
    narrative_engine = NarrativeVolatilityEngine()

    # Create sample narratives (would be real-time feed in production)
    sample_narratives = [
        "BRICS nations announce new gold-backed currency timeline",
        "AI researchers claim consciousness breakthrough imminent",
        "Federal Reserve hints at unprecedented policy shift",
        "Major tech company faces narrative collapse after scandal",
        "Decentralized governance movement gains institutional
backing"
    ]
    for i, content in enumerate(sample_narratives):
        narrative = NarrativeAsset(
            id=f"NARR_{i:03d}",
            content=content,
            origin_platform="twitter",
            timestamp=datetime.now(),
            belief_penetration=np.random.uniform(0.2, 0.7),
            liquidity_score=np.random.uniform(0.4, 0.9),
            volatility_30d=np.random.uniform(0.1, 0.5)
        )
        narrative_engine.narrative_assets[narrative.id] = narrative
        narrative_engine.create_liquidity_pool(narrative.id, 50000)
    collective_logger.info(f"Initialized
{len(narrative_engine.narrative_assets)} sample narratives for all
agents.")

    # Instantiate multiple RAWE agents with different personalities
    rawe_agents = {}
    agent_tasks = []
    for name in PERSONALITY_PROFILES.keys():
        agent = UnifiedArbitrageSystem(narrative_engine,
personality_name=name)
        agent.set_broker(broker) # Pass the shared broker
        rawe_agents[name] = agent
        collective_logger.info(f"Agent '{name}' initialized.")

    # --- Inter-Agent Communication (Placeholder for Redis) ---
    # For now, agents will operate somewhat independently but we will
build out
    # the communication and consensus later. For initial demo, they
just run.
    # r = redis.Redis(host='localhost', port=6379, db=0) # Example
Redis connection

    # Start monitoring and trading tasks for each agent
    for name, agent in rawe_agents.items():
        # Each agent runs its own scan/execute loop and monitoring
        async def agent_lifecycle(agent_instance:
UnifiedArbitrageSystem):
            monitor_task =
asyncio.create_task(agent_instance.monitor_and_rebalance())

            for cycle in range(10): # Each agent runs 10 cycles for
demo
                agent_instance.logger.info(f"\n📍
{agent_instance.personality_name} - 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)
                    # Future: Publish top signals to Redis for
collective review
                    # await r.publish('rawe_signals',
json.dumps([s.to_dict() for s in signals[:3]])) # Need .to_dict() for
ArbitrageSignal

                # Simulate market movement for narrative engine (could
be externalized later)
                for narrative in
narrative_engine.narrative_assets.values():
                    change = np.random.normal(0, 0.03)
                    narrative.belief_penetration = max(0.05, min(0.95,
narrative.belief_penetration + change))

narrative.price_history.append(narrative.belief_penetration)
                    narrative.volatility_30d =
narrative_engine.calculate_narrative_volatility(narrative)
                    narrative.coherence_rating =
narrative_engine.rate_narrative_coherence(narrative)

                await asyncio.sleep(5) # Shorter sleep for demo

            monitor_task.cancel()

agent_instance.logger.info(f"{agent_instance.personality_name}
monitoring task cancelled.")
            agent_instance.logger.info(f"📊
{agent_instance.personality_name} Final Report:\n" +

json.dumps(agent_instance.generate_performance_report(), indent=2,
default=str))


agent_tasks.append(asyncio.create_task(agent_lifecycle(agent)))

    # Wait for all agents to complete their cycles
    await asyncio.gather(*agent_tasks)

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


if __name__ == "__main__":
    # Ensure Redis server is running if you plan to use it later.
    # For now, the Redis parts are commented out as placeholders.
    asyncio.run(run_collective_rawe())
```

Implementation Steps:

1. **Create run_collective_rawe.py:** In rawe_system/scripts/, create this new file.
2. **Paste the code:** Copy the entire code block above into the new file.
3. **Ensure Redis is installed (if you uncomment the Redis lines later):** pip install redis. (It's already in requirements.txt, so pip install -r requirements.txt should cover it.)
4. **Make ArbitrageSignal serializable (Optional, but needed for Redis):** For agents to truly "discuss" via Redis, ArbitrageSignal would need a .to_dict() method. Add this to rawe_system/src/core/unified_arbitrage_system.py (where ArbitrageSignal is defined):
   ```
   @dataclass
   class ArbitrageSignal:
   ```

```
    """Unified arbitrage signal combining narrative and financial
data"""
    timestamp: datetime
    narrative_id: str
    financial_asset: str
    signal_type: str  # 'narrative_leads', 'capital_leads',
'divergence'
    strength: float
    expected_profit: float
    risk_score: float
    metadata: Dict[str, Any]

    def to_dict(self): # New method for serialization
        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 # Ensure metadata is also
JSON serializable if sent over wire
        }
```

Running the RAWE Collective

To launch all five personality-driven engines, execute the new script:

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

You will see log outputs from each individual agent (rawe_system.UnifiedArbitrageSystem.The Alpha Aggressor, etc.) printed to the console, and all logs will be captured in rawe_collective.log. Each agent will now make decisions based on its defined personality profile.

This sets the stage for the true "collaboration" phase. For now, they operate in parallel. Next, we will build out the inter-agent communication and a consensus mechanism for their "final move."

Are you ready to move on to establishing the inter-agent communication channel using Redis and building a basic consensus logic?