Developer-Focused Summary: Integrating Fuzzy Position Sizing

The sizing logic is built around a strict upper-bound risk constraint combined with a fuzzy-logic-driven scaling mechanism. The process begins by computing the **maximum allowable risk per trade**, defined as a fixed fraction of account equity:

$\text{MaximumRisk} = 0.02 \times \text{Equity}$

This risk budget is then converted into a **base contract quantity**, using the maximum possible loss of a single iron condor contract:

$q_0 = \text{floor}(\text{MaximumRisk} \div \text{MaxLossPerContract})$

The value q_0 represents a **hard ceiling** on position size. Under no circumstances may the system exceed this quantity. Before any sizing computation occurs, **all hard constraints must pass**, including:

- Market regime validity
- Multi-timeframe alignment
- Minimum credit-to-risk ratio
- Greek exposure limits
- Liquidity thresholds
- Macro-event and volatility halts

If any hard constraint fails, position size is set to zero and the trade is rejected.

Once all hard constraints are satisfied, the system evaluates a set of **soft conditions** using fuzzy logic. Each soft condition is mapped to a membership value μ_j in the closed interval [0, 1], where higher values indicate more favorable conditions. Typical inputs include:

- Implied volatility favorability
- Regime stability
- Multi-timeframe directional coherence
- Delta balance symmetry
- Liquidity quality

These membership values are combined via a weighted aggregation to form a **fuzzy confidence score**:

$F_t = \Sigma (w_j \times \mu_j)$

where all weights w_j are non-negative and sum to 1. By construction:

$0 \leq F_t \leq 1$

In parallel, realized or forecast volatility is normalized into a **volatility penalty factor**:

$\sigma^*_t \in [0, 1]$

Higher values of σ^*_t correspond to elevated volatility and therefore increased risk.

The system then computes a **scaling function**:

$g = g(F_t , \sigma^*_t)$

subject to the constraint:

$0 \leq g \leq 1$

The function g is monotonically increasing in F_t and monotonically decreasing in σ^*_t. Intuitively, confidence increases size, while volatility suppresses it.

The final position size is computed as:

$q = q_0 \times g$

This produces a **continuously variable position size** rather than a binary "all-in or all-out" allocation. Under ideal conditions (high confidence, low volatility), g approaches 1 and the system deploys the full 2% risk budget. Under mixed conditions, g yields a fractional allocation. Under low confidence or elevated volatility, g approaches 0, resulting in no position—even though all hard constraints may technically pass.

The result is a **deterministic, auditable, and risk-aware sizing mechanism** that adapts position size to market quality rather than blindly consuming the maximum allowable risk. This logic is implementation-ready and integrates cleanly into any systematic options trading or execution engine.

Next, I put together s simplistic python implementation developer-centric.

Reference Python Implementation: Fuzzy Position Sizing

```python
from typing import Dict


def compute_base_quantity(
    equity: float,
    max_loss_per_contract: float,
    risk_fraction: float = 0.02
) -> int:
    """
    Compute the hard ceiling on position size.

    q0 = floor((risk_fraction * equity) / max_loss_per_contract)
    """
    if equity <= 0.0:
        return 0

    if max_loss_per_contract <= 0.0:
        return 0

    max_risk = risk_fraction * equity
    q0 = int(max_risk // max_loss_per_contract)

    return max(q0, 0)
```

Fuzzy Confidence Aggregation

```python
def compute_fuzzy_confidence(
    memberships: Dict[str, float],
    weights: Dict[str, float]
) -> float:
    """
    Compute fuzzy confidence score Ft in [0, 1].

    Ft = sum(w_j * mu_j)

    Assumes:
    - All mu_j are already normalized to [0, 1]
    - All w_j >= 0
    - sum(w_j) == 1
    """
    confidence = 0.0

    for key, mu in memberships.items():
        w = weights.get(key, 0.0)
        confidence += w * mu

    # Hard clamp for numerical safety
    if confidence < 0.0:
        return 0.0
    if confidence > 1.0:
        return 1.0

    return confidence
```

Volatility Normalization

```
def normalize_volatility(
    realized_vol: float,
    low_vol: float,
    high_vol: float
) -> float:
    """
    Normalize volatility into sigma_star in [0, 1].
    sigma_star = (realized_vol - low_vol) / (high_vol - low_vol)
    Values below low_vol map to 0.
    Values above high_vol map to 1.
    """
    if high_vol <= low_vol:
        return 1.0

    sigma_star = (realized_vol - low_vol) / (high_vol - low_vol)

    if sigma_star < 0.0:
        return 0.0
    if sigma_star > 1.0:
        return 1.0
    return sigma_star

```

Scaling Function g(F, σ*)

This implementation uses a **multiplicative attenuation model**, which is simple, monotonic, and easy to reason about.

```
def compute_scaling_factor(
    confidence: float,
    volatility_penalty: float,
    min_scale: float = 0.0
) -> float:
    """
    Compute g(Ft, sigma_star).

    g = Ft * (1 - sigma_star)

    min_scale enforces a floor if desired (e.g. for minimum viable size).
    """
    g = confidence * (1.0 - volatility_penalty)

    if g < min_scale:
        g = min_scale

    if g > 1.0:
        g = 1.0

    return g

```

Final Position Size Computation

```python
def compute_position_size(
    equity: float,
    max_loss_per_contract: float,
    memberships: Dict[str, float],
    weights: Dict[str, float],
    realized_vol: float,
    low_vol: float,
    high_vol: float,
    risk_fraction: float = 0.02
) -> int:
    """
    Full sizing pipeline.
    Hard constraints MUST be validated before calling this function.
    """

    # Step 1: Hard ceiling
    q0 = compute_base_quantity(
        equity=equity,
        max_loss_per_contract=max_loss_per_contract,
        risk_fraction=risk_fraction
    )

    if q0 == 0:
        return 0

    # Step 2: Fuzzy confidence
    Ft = compute_fuzzy_confidence(
        memberships=memberships,
        weights=weights
    )

    # Step 3: Volatility penalty
    sigma_star = normalize_volatility(
        realized_vol=realized_vol,
        low_vol=low_vol,
        high_vol=high_vol
    )

    # Step 4: Scaling factor
    g = compute_scaling_factor(
        confidence=Ft,
        volatility_penalty=sigma_star
    )

    # Step 5: Final size
    q = int(q0 * g)

    return max(q, 0)

```

Example Usage

```
memberships = {
    "iv_favorability": 0.85,
    "regime_stability": 0.90,
    "mtf_alignment": 0.80,
    "delta_balance": 0.75,
    "liquidity_quality": 0.95,
}

weights = {
    "iv_favorability": 0.25,
    "regime_stability": 0.20,
    "mtf_alignment": 0.20,
    "delta_balance": 0.15,
    "liquidity_quality": 0.20,
}

position_size = compute_position_size(
    equity=250_000.0,
    max_loss_per_contract=1_200.0,
    memberships=memberships,
    weights=weights,
    realized_vol=18.0,
    low_vol=12.0,
    high_vol=30.0
)

```

Engineering Notes

• All components are deterministic and side-effect free
• Hard clamps prevent numerical leakage
• Scaling logic is monotonic and interpretable
• Function boundaries align with unit-test isolation
• Can be extended with nonlinear g-functions or regime-specific weight vectors