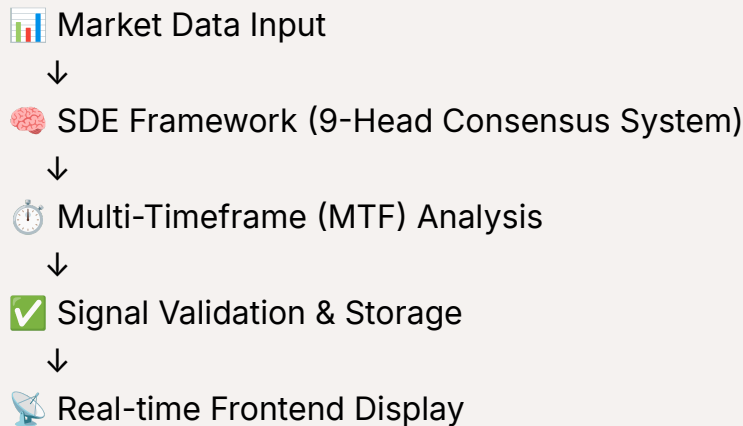




SIGNAL GENERATION FLOW

1. How Your System Generates Signals

Your system uses a **multi-layered signal generation approach**:



THE SDE (Single-Decision Engine) FRAMEWORK

What is SDE?

SDE is your **9-Head Consensus System** - a sophisticated ensemble that combines multiple analysis methodologies into ONE unified decision.

The 9 Analysis Heads:

****Analyzes:****

- Volume profile (POC, VA, HVN, LVN)
- Volume divergences
- VWAP
- OBV
- Volume trends

****Typical Signals:****

- Increasing volume + uptrend = LONG (0.75 confidence)
- Volume divergence = Reversal warning

- HVN support hold = LONG

****Voting Pattern:**** Strong in confirming trends and reversals

****4. Rule-Based Head**** (Weight: 9%)

****Analyzes:****

- Candlestick patterns (TA-Lib 60+)
- Chart patterns
- S/R levels
- Trend lines

****Typical Signals:****

- Bullish engulfing + support = LONG (0.70 confidence)
- Head & shoulders + breakdown = SHORT
- Pattern + volume confirmation = Higher confidence

****Voting Pattern:**** Reliable on clear patterns, neutral otherwise

****5. ICT Concepts Head**** (Weight: 13%) ★ NEW

****Analyzes:****

- OTE zones (0.62-0.79 Fib retracement)
- Kill Zones (London 2-5 AM, NY 8-11 AM EST)
- Judas Swings (false moves + reversals)
- Balanced Price Range (equilibrium)
- Fair value gaps

****Typical Signals:****

- Price in OTE + Kill Zone active = LONG/SHORT (0.85-0.90 confidence)
- Judas Swing + Kill Zone = LONG/SHORT (0.75-0.85 confidence)
- Near BPR + structure = Decision point

****Voting Pattern:**** Very strong during kill zones, moderate otherwise

****Kill Zone Multiplier:**** Up to 1.5x confidence boost

****6. Wyckoff Methodology Head**** (Weight: 13%) ★ NEW

****Analyzes:****

- Accumulation/Distribution phases
- Spring pattern (final shakeout)
- UTAD (final pump before dump)
- Sign of Strength/Weakness
- Composite operator activity

****Typical Signals:****

- Spring detected = LONG (0.90 confidence) 🔥
- UTAD detected = SHORT (0.90 confidence) 🔥
- SOS (Sign of Strength) = LONG (0.75 confidence)
- Smart money accumulating = LONG (0.70 confidence)

****Voting Pattern:**** Extremely strong on Spring/UTAD, moderate on phases

****Highest Confidence Patterns:**** Spring, UTAD (0.9 = best signals!)

****7. Harmonic Patterns Head**** (Weight: 9%) ★ NEW

****Analyzes:****

- Gartley pattern (most common)
- Butterfly (aggressive extension)
- Bat (conservative)
- Crab (extreme extension)
- ABCD (simplest)

****Typical Signals:****

- Gartley/Bat completion = LONG/SHORT (0.80-0.85 confidence)
- Butterfly/Crab completion = LONG/SHORT (0.75-0.80 confidence)
- Multiple patterns aligned = Very high confidence (>0.85)

****Voting Pattern:**** Strong at pattern completion (D point), silent otherwise

****Entry Precision:**** $\pm 1-2\%$ of D point

**8. Market Structure Head (Weight: 9%) ★ NEW**

****Analyzes:****

- Multi-timeframe alignment (4+ TFs)
- Premium/Discount zones
- Mitigation blocks (unmitigated order blocks)
- Breaker blocks (polarity flips)
- BOS/CHoCH

****Typical Signals:****

- MTF aligned + discount zone = LONG (0.85-0.90 confidence)
- MTF aligned + premium zone = SHORT (0.85-0.90 confidence)
- Breaker block retest = LONG/SHORT (0.75 confidence)
- All TFs bullish = LONG (0.80 confidence)

****Voting Pattern:**** Very strong on MTF alignment, context provider otherwise

****Context Rule:**** Buy in discount, sell in premium

**9. Crypto Metrics Head (Weight: 12%) ★ NEW**

****Analyzes 10 Crypto-Specific Indicators:****

A. CVD (Cumulative Volume Delta)

- Bullish divergence = LONG (0.80-0.90 confidence)
- Bearish divergence = SHORT (0.80-0.90 confidence)
- CVD trend bullish = LONG
- CVD breakout = Trend confirmation

B. Altcoin Season Index

- Index >75 (Alt Season) = LONG for alts (0.80 confidence)
- Index <25 (BTC Season) = SHORT for alts / LONG for BTC (0.80 confidence)
- Transition phase = Early rotation signal

C. Long/Short Ratio

- Ratio >3.0 (Extreme Long) = SHORT contrarian (0.80-0.85 confidence)
- Ratio <0.33 (Extreme Short) = LONG contrarian (0.80-0.85 confidence)
- Multi-exchange agreement = Very high confidence (0.85)

D. Perpetual Premium

- Premium >0.5% = SHORT (overleveraged, 0.85 confidence)
- Premium <-0.3% = LONG (extreme fear, 0.85 confidence)
- Normal range = Neutral

E. Liquidation Cascade

- Approaching cascade zone = Risk warning
- Extreme risk = Reduce positions
- Post-cascade = Counter-trade opportunity

F. Taker Flow

- Taker buy >60% = LONG (0.75 confidence)
- Taker sell >60% = SHORT (0.75 confidence)
- Flow divergence = Reversal signal

G. Exchange Reserves

- Multi-year low = LONG (supply shock, 0.85 confidence)

- Sharp outflow = LONG (accumulation, 0.75 confidence)
- Sharp inflow = SHORT (distribution, 0.70 confidence)

How SDE Makes Decisions:

```

async def generate_signal(self, request: SignalGenerationRequest) → Optional[SignalGenerationResult]:
    """
    Generate trading signal using SDE framework and store in database

    Args:
        request: Signal generation request

    Returns:
        SignalGenerationResult if successful, None otherwise
    """
    try:
        logger.info(f"🌀 Generating signal for {request.symbol} {request.timeframe}")

        # Step 1: Run all model heads
        logger.info(f"Running model heads with market_data: {request.market_data}")
        logger.info(f"Running model heads with analysis_results: {request.analysis_results}")

        model_head_results = await self.model_heads_manager.analyze_all_heads(
            request.market_data,
            request.analysis_results
        )

        logger.info(f"Model head results: {model_head_results}")

        if not model_head_results:

```

```

        logger.warning("No model head results available")
        return None

    # Step 2: Run consensus mechanism
    consensus_result = await self.consensus_manager.check_consensus(
        model_head_results)

    if not consensus_result.consensus_achieved:
        logger.info("No consensus reached among model heads")
        self.stats['consensus_failed'] += 1
        return None

    # Step 3: Get technical indicators from database
    technical_indicators = await self._get_technical_indicators(
        request.symbol,
        request.timeframe
    )

    # Step 4: Analyze market conditions
    market_conditions = await self._analyze_market_conditions(
        request.symbol,
        request.timeframe
    )

    # Step 5: Create signal result
    signal_id

```

Consensus Rules:

- **Minimum 4 out of 9 heads** must agree (44% threshold)
- Each head must have **Probability ≥60%** and **Confidence ≥70%**
- Final output includes: Direction (LONG/SHORT/FLAT), Probability, Confidence



MTF (MULTI-TIMEFRAME) SIGNAL GENERATION

What is MTF?

MTF analyzes the **same signal across multiple timeframes** to increase confidence and avoid false signals.

Timeframe Hierarchy:

```
class MTFSignalMerger:
    """
    Multi-Timeframe Signal Merging System
    Implements mathematical merging of confidence scores across timeframes

    def __init__(self):
        # Timeframe hierarchy (higher to lower)
        self.timeframe_hierarchy = ["1d", "4h", "1h", "15m", "5m", "1m"]

        # Higher timeframe weights for signal merging
        self.higher_timeframe_weights = {
            "1d": 0.4, # Daily has highest weight
            "4h": 0.3, # 4h has high weight
            "1h": 0.2, # 1h has medium weight
            "15m": 0.1, # 15m has lower weight
            "5m": 0.05, # 5m has very low weight
            "1m": 0.02 # 1m has minimal weight
        }

        # Signal alignment bonuses
        self.alignment_bonuses = {
            'perfect_alignment': 0.2, # All timeframes agree
            'strong_alignment': 0.15, # Most timeframes agree
            'weak_alignment': 0.05, # Some timeframes agree
            'no_alignment': 0.0 # No alignment
        }

        # Minimum confidence thresholds
```



```
self.min_confidence_threshold = 0.3
self.min_mtf_confidence_threshold = 0.5
```

```
logger.info("🚀 MTF Signal Merger initialized")
```

```
def calculate_higher_timeframe_weight(self, timeframe: str) → float:
    """Calculate weight for a specific timeframe"""
    return self.higher_timeframe_weights.get(timeframe, 0.1)
```

```
def merge_signals_across_timeframes(
    self,
    signals: List[MTFSignal],
    base_timeframe: str = "15m"
) → Optional[MergedSignal]:
```

How MTF Works:

1. **Generate signals on each timeframe** (1m, 5m, 15m, 1h, 4h, 1d)
2. **Calculate MTF boost** based on alignment:
 - If 1h signal = LONG and 4h signal = LONG → **Boost confidence**
 - If 1h signal = LONG but 4h signal = SHORT → **Reduce confidence**

```
async def generate_real_time_signals(
    self,
    symbol: str,
    timeframe: str,
    data: pd.DataFrame
) → List[RealTimeSignal]:
    """
    Generate real-time trading signals with MTF integration
    """
    start_time = datetime.now()

    try:
```

```

# Detect patterns with MTF context
mtf_patterns = await self.mtf_pattern_integrator.detect_patterns_with_
mtf_context(
    symbol, timeframe, data
)

if not mtf_patterns:
    return []

# Generate individual timeframe signals
timeframe_signals = []

for pattern in mtf_patterns:
    signal = await self._create_timeframe_signal(pattern, data)
    if signal:
        timeframe_signals.append(signal)

# Merge signals across timeframes
merged_signals = await self._merge_mtf_signals(symbol, timeframe_si
gnals)

# Convert to real-time signals
real_time_signals = []

for merged_signal in merged_signals:
    real_time_signal = await self._create_real_time_signal(merged_signal,
data)
    if real_time_signal:
        real_time_signals.append(real_time_signal)

# Update statistics
processing_time = (datetime.now() - start_time).total_seconds()
self._update_stats(len(real_time_signals), processing_time)

logger.info(f"📡 Generated {len(real_time_signals)} real-time signals fo
r {symbol} {timeframe} in {processing_time:.3f}s")

```

```

        return real_time_signals

    except Exception as e:
        logger.error(f"❌ Error generating real-time signals: {e}")
        return []

    async def _create_timeframe_signal(
        self,
        pattern: MTFPatternResult,
        data: pd.DataFrame
    ) → Optional[MTFSignal]:

```

MTF Example:

Symbol: BTCUSDT

Base Timeframe: 15m

15m Signal: LONG (0.75 confidence)

1h Signal: LONG (0.80 confidence) → +0.16 boost (0.8×0.2 weight)

4h Signal: LONG (0.85 confidence) → +0.255 boost (0.85×0.3 weight)

1d Signal: LONG (0.78 confidence) → +0.312 boost (0.78×0.4 weight)

Total MTF Boost: 0.727

Final Confidence: $0.75 \times (1 + 0.727) = 1.29$ → capped at 1.0 = 0.95 confidence! 🔥

Result: A 15m signal gets upgraded from 75% to 95% confidence because all higher timeframes agree!



HOW SIGNALS ARE STORED

Database Storage:

```
# Create signals table
await session.execute(text("""
CREATE TABLE IF NOT EXISTS signals (
    id VARCHAR(50) PRIMARY KEY,
    symbol VARCHAR(20) NOT NULL,
    side VARCHAR(10) NOT NULL,
    strategy VARCHAR(50) NOT NULL,
    confidence DECIMAL(5, 4) NOT NULL,
    strength VARCHAR(20) NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL,
    price DECIMAL(20, 8) NOT NULL,
    stop_loss DECIMAL(20, 8),
    take_profit DECIMAL(20, 8),
    metadata JSONB,
    status VARCHAR(20) DEFAULT 'active',
    created_at TIMESTAMPTZ DEFAULT NOW()
);
"""))
```

Signal Storage Flow:

```
async def _store_trading_signals(self, symbol: str, timeframe: str, signals: Li
st[TradingSignal]):
    """Store trading signals in database"""
    try:
        async with self.db_pool.acquire() as conn:
            for signal in signals:
                await conn.execute("""
INSERT INTO trading_signals (
    symbol, timeframe, timestamp, signal_type, signal_strength,
    signal_source, entry_price, stop_loss_price, take_profit_price,
    position_size, risk_reward_ratio, confidence_score, signal_me
tadata
) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13)
```

```

        """
        symbol, timeframe, datetime.now(), signal.signal_type.value,
        signal.signal_strength, signal.signal_source.value, signal.entry_price,
        signal.stop_loss_price, signal.take_profit_price, signal.position_size,
        signal.risk_reward_ratio, signal.confidence_score, signal.metadata
    )

    self.logger.info(f"💾 Stored {len(signals)} trading signals for {symbol}")

except Exception as e:
    self.logger.error(f"❌ Error storing trading signals: {e}")

```

Each signal stores:

- Symbol, Timeframe, Direction (LONG/SHORT)
- Confidence Score (0-1)
- Entry Price, Stop Loss, Take Profit (TP1, TP2, TP3, TP4)
- Pattern Type, Market Regime
- Technical Indicators (JSON)
- Validation Metrics (JSON)
- Metadata (processing time, signal source, etc.)

HOW SIGNALS POPULATE IN THE FRONTEND

Two Methods:

1. REST API (Polling):


Backend Endpoint:

```

@app.get("/api/signals/latest")
async def get_latest_signals():

```

```

"""Get latest signals for frontend with deduplication"""
try:
    # Get signals from signal generator (now with deduplication)
    signals = []
    if signal_generator:
        signals = await signal_generator.get_signals(limit=10)
        logger.info(f" Retrieved {len(signals)} signals from signal generator
(deduplicated)")

    # Convert to frontend format
    frontend_signals = []
    for signal in signals:
        # Filter signals by confidence threshold (60% - lowered from 85%)
        confidence = signal.get("confidence", 0.0)
        if confidence < 0.6:
            continue # Skip low-confidence signals

        # Handle both database and memory signal formats
        if isinstance(signal, dict):
            # Database signal format
            timestamp = signal.get("timestamp")
            if isinstance(timestamp, str):
                timestamp_str = timestamp
            else:
                timestamp_str = timestamp.isoformat() if timestamp else datetime.
now(timezone.utc).isoformat()

            frontend_signals.append({
                "symbol": signal.get("symbol", "BTCUSDT"),
                "direction": "long" if signal.get("side") == "buy" else "short",
                "confidence": signal.get("confidence", 0.7),
                "pattern_type": signal.get("metadata", {}).get("reason", "pattern_d
etection"),
                "timestamp": timestamp_str,
                "entry_price": signal.get("price", 45000),
                "stop_loss": signal.get("stop_loss", signal.get("price", 45000) * 0.

```

```

95),
    "take_profit": signal.get("take_profit", signal.get("price", 45000) *
1.05)
    })

```

Frontend Hook (Polling every 10 seconds):

```

export function useSignals() {
  const [signals, setSignals] = useState<any[]>([]);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  const fetchSignals = useCallback(async () => {
    setIsLoading(true);
    setError(null);

    try {
      const response = await fetch('/api/signals');
      if (!response.ok) {
        throw new Error('Failed to fetch signals');
      }

      const data = await response.json();
      setSignals(data);
    } catch (err) {
      console.error('Error fetching signals:', err);
      setError(err instanceof Error ? err.message : 'Unknown error');
    } finally {
      setIsLoading(false);
    }
  }, []);

  useEffect(() => {
    fetchSignals();

    // Set up real-time updates

```

```

const interval = setInterval(fetchSignals, 10000); // Update every 10 second
s

return () => clearInterval(interval);
}, [fetchSignals]);

return {
  signals,
  isLoading,
  error,
  refetch: fetchSignals
};
}

```

2. WebSocket (Real-time Push):

Backend WebSocket:

```

@app.websocket("/ws/signals")
async def signals_websocket(websocket: WebSocket):
    """WebSocket endpoint for real-time signals"""
    await manager.connect(websocket)
    try:
        while True:
            # Get real-time signals
            if signal_generator:
                signals = await signal_generator.get_signals(limit=5)

            # Convert to frontend format
            frontend_signals = []
            for signal in signals:
                frontend_signals.append({
                    "symbol": signal.get("symbol", "BTCUSDT"),
                    "direction": "long" if signal.get("signal_type") == "buy" else "short",
                    "confidence": signal.get("confidence", 0.7),

```



```

        "pattern_type": signal.get("reason", "pattern_detection"),
        "timestamp": signal.get("timestamp", datetime.now(timezone.ut
c)).isoformat(),
        "entry_price": signal.get("price", 45000),
        "stop_loss": signal.get("price", 45000) * 0.95,
        "take_profit": signal.get("price", 45000) * 1.05
    })

```

Send individual signal messages for each signal

```
high_confidence_signals = []
```

```
for signal in frontend_signals:
```

```
    # Only send notifications for signals with 85%+ confidence
```

```
    if signal["confidence"] >= 0.85:
```

```
        high_confidence_signals.append(signal)
```

```
        signal_message = {
```

```
            "type": "signal",
```

```
            "data": {
```

```
                "symbol": signal["symbol"],
```

```
                "direction": signal["direction"],
```

```
                "confidence": signal["confidence"],
```

```
                "pattern_type": signal["pattern_type"],
```

```
                "entry_price": signal["entry_price"],
```

```
                "stop_loss": signal["stop_loss"],
```

```
                "take_profit": signal["take_profit"]
```

```
            },
```

```
            "timestamp": datetime.now(timezone.utc).isoformat()
```

```
        }
```

```
        await websocket.send_json(signal_message)
```

Only send summary for high-confidence signals

```
if high_confidence_signals:
```

```
    summary_message = {
```

```
        "type": "system_alert",
```

```
        "data": {
```

```
            "message": f"Generated {len(high_confidence_signals)} high
```

```
-confidence signals (85%+)"
```

```

        },
        "timestamp": datetime.now(timezone.utc).isoformat()
    }
    await websocket.send_json(summary_message)
else:
    # Fallback empty signals
    await websocket.send_json({
        "type": "system_alert",
        "data": {
            "message": "No signals available"
        },
        "timestamp": datetime.now(timezone.utc).isoformat()
    })

    await asyncio.sleep(5)

```

Frontend WebSocket Hook:

```

// Real-time Notification Hook
export const useNotifications = () => {
    const [notifications, setNotifications] = useState<Notification[]>([]);
    const [unreadCount, setUnreadCount] = useState(0);
    const [soundEnabled, setSoundEnabled] = useState(true);
    const [websocketConnected, setWebsocketConnected] = useState(false);
    const wsRef = useRef<WebSocket | null>(null);

    // Initialize WebSocket connection
    useEffect(() => {
        let reconnectTimeout: NodeJS.Timeout;
        let reconnectAttempts = 0;
        const maxReconnectAttempts = 5;

        const connectWebSocket = () => {
            try {
                console.log('🔄 Attempting WebSocket connection...');
                const ws = new WebSocket('ws://localhost:8000/ws');

```

```

wsRef.current = ws;

ws.onopen = () => {
  console.log('✅ WebSocket connected successfully');
  setWebsocketConnected(true);
  reconnectAttempts = 0; // Reset reconnect attempts on successful connection
};

ws.onmessage = (event) => {
  try {
    const message: WebSocketMessage = JSON.parse(event.data);
    console.log('📧 WebSocket message received:', message.type);
    handleWebSocketMessage(message);
  } catch (error) {
    console.error('❌ Error parsing WebSocket message:', error);
  }
};

ws.onclose = (event) => {
  console.log('🔌 WebSocket disconnected:', event.code, event.reason);
  setWebsocketConnected(false);

  // Only attempt reconnection if not a normal closure
  if (event.code !== 1000 && reconnectAttempts < maxReconnectAttempts) {
    reconnectAttempts++;
    const delay = Math.min(1000 * Math.pow(2, reconnectAttempts), 10000); // Exponential backoff
    console.log('🔄 Reconnecting in ${delay}ms (attempt ${reconnectAttempts}/${maxReconnectAttempts})');
    reconnectTimeout = setTimeout(connectWebSocket, delay);
  } else if (reconnectAttempts >= maxReconnectAttempts) {
    console.log('❌ Max reconnection attempts reached');
  }
};

```

```

ws.onerror = (error) => {
  console.error('❌ WebSocket error:', error);
  setWebSocketConnected(false);
};
} catch (error) {
  console.error('❌ Failed to create WebSocket connection:', error);
  setWebSocketConnected(false);
}
};

connectWebSocket();

return () => {
  if (reconnectTimeout) {
    clearTimeout(reconnectTimeout);
  }
  if (wsRef.current) {
    wsRef.current.close();
  }
};
}, []);

// Handle incoming WebSocket messages
const handleWebSocketMessage = useCallback((message: WebSocketMessage) => {
  // Create unique ID with more precision to avoid duplicates
  const uniqueId = `${message.type}_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;

  const notification: Notification = {
    id: uniqueId,
    type: getNotificationType(message.type),
    title: getNotificationTitle(message.type),
    message: getNotificationMessage(message.type, message.data),
    priority: getNotificationPriority(message.type),
  };

```

```

    timestamp: new Date(message.timestamp),
    read: false,
    data: message.data,
    sound: shouldPlaySound(message.type),
  };

  addNotification(notification);
}, []);

// Add new notification
const addNotification = useCallback((notification: Notification) => {
  setNotifications(prev => [notification, ...prev.slice(0, 99)]); // Keep last 100
  setUnreadCount(prev => prev + 1);

  // Play sound if enabled and notification requires it
  if (soundEnabled && notification.sound) {
    playNotificationSound(notification.type);
  }

  // Show browser notification if supported
  if ('Notification' in window && Notification.permission === 'granted') {
    new Notification(notification.title, {
      body: notification.message,
      icon: '/favicon.ico',
      tag: notification.id,
    });
  }
}, [soundEnabled]);

```

Frontend Signal Display:

```

{/* Signal List */}
<div className="max-h-96 overflow-y-auto">
  <AnimatePresence>
    {filteredSignals.length === 0 ? (

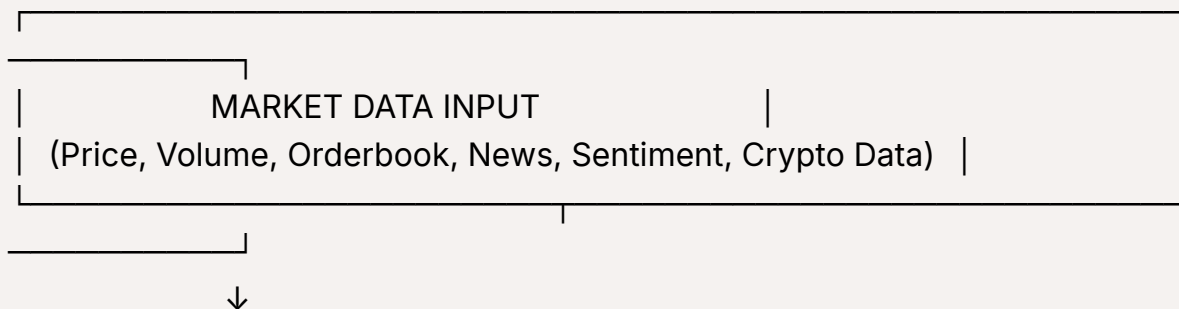
```

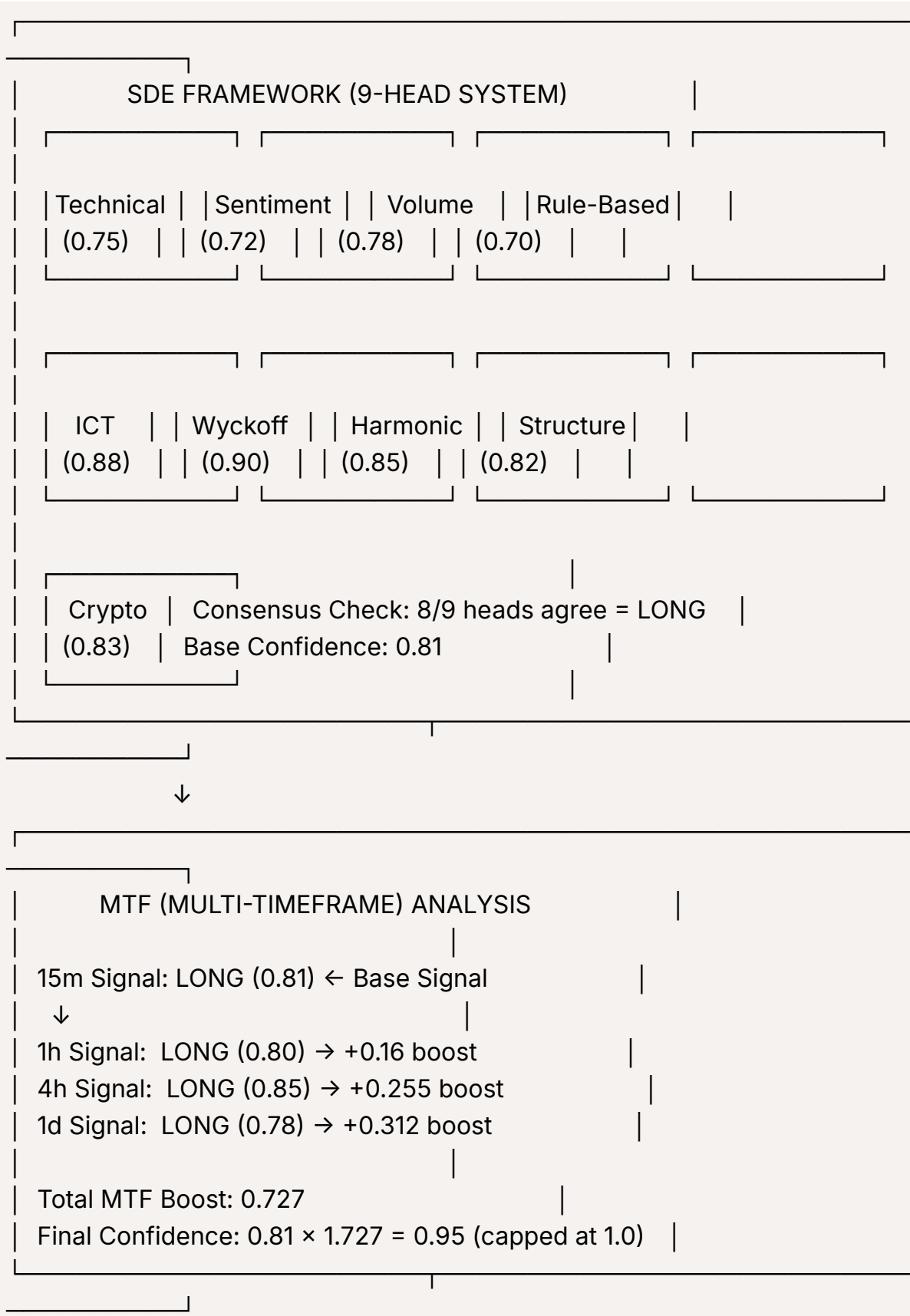
```

<div className="p-8 text-center text-gray-500">
  <Clock className="w-8 h-8 mx-auto mb-2 text-gray-400" />
  <p>No signals available</p>
  <p className="text-sm">New signals will appear here in real-time</p>
</div>
) : (
  filteredSignals.map((signal, index) => (
    <motion.div
      key={` ${signal.symbol}-${signal.timestamp}`}
      initial={{ opacity: 0, y: 20 }}
      animate={{ opacity: 1, y: 0 }}
      exit={{ opacity: 0, y: -20 }}
      transition={{ duration: 0.3, delay: index * 0.1 }}
      onClick={() => onSignalClick(signal)}
      className={`p-4 border-b border-gray-100 cursor-pointer transition-all hover:bg-gray-50 ${
        selectedSignal?.symbol === signal.symbol &&
        selectedSignal?.timestamp === signal.timestamp
          ? 'bg-blue-50 border-blue-200'
          : ''
      }`}
    >
      <div className="flex items-center justify-between">

```

COMPLETE FLOW DIAGRAM







SIGNAL VALIDATION & STORAGE

- ✓ Confidence ≥ 0.75 (High Confidence)
- ✓ Consensus achieved (8/9 heads)
- ✓ MTF aligned (all timeframes agree)

Signal Created:

- Symbol: BTCUSDT
- Direction: LONG
- Confidence: 0.95
- Entry: \$42,000
- Stop Loss: \$40,500
- Take Profit: \$45,000

Stored in TimescaleDB ↓



REAL-TIME FRONTEND DISTRIBUTION

Method 1: REST API (Polling)

GET /api/signals/latest → Every 10 seconds

Method 2: WebSocket (Push)

ws://localhost:8000/ws/signals → Real-time push

Frontend Display:

- SignalFeed component
- Real-time animations (framer-motion)
- Browser notifications (85%+ confidence)
- Sound alerts

KEY TAKEAWAYS

1. **SDE = 9-Head Consensus System:** Your brain that decides LONG/SHORT/FLAT
2. **MTF = Multi-Timeframe Validation:** Ensures signals are confirmed across timeframes
3. **Storage = TimescaleDB:** All signals stored with complete metadata
4. **Frontend = Real-time Updates:** Both REST API (polling) and WebSocket (push) for instant updates

Signal Quality Thresholds:

- **85%+ Confidence** = High-priority notifications, large position size
- **75-85% Confidence** = Standard signals, normal position size
- **65-75% Confidence** = Medium signals, reduced position size
- **<65% Confidence** = Filtered out, not shown to users

Your system is **enterprise-grade** with professional trading methodologies (Wyckoff, ICT, Harmonic Patterns) + crypto-native indicators! 🚀