Data Framework & Live Ticker System - 54,000 Listings

Crane Intelligence Platform - Real-Time Market Intelligence

Data Asset: 54,000+ Equipment Listings

Objective: Bloomberg Terminal-Quality Live Market Data

📊 DATA ARCHITECTURE FRAMEWORK

Core Data Structure

Equipment Listings Table

SQL

```
CREATE TABLE equipment_listings (
    id SERIAL PRIMARY KEY,
    listing_id VARCHAR(50) UNIQUE,
    source VARCHAR(100),           -- Dealer, auction, marketplace
    make VARCHAR(50),              -- Liebherr, Grove, Manitowoc, etc.
    model VARCHAR(100),            -- LR1300-2, LTM1350-6.1, etc.
    year INTEGER,
    capacity_tons DECIMAL(8,2),
    boom_length_ft INTEGER,
    hours INTEGER,
    price DECIMAL(12,2),
    currency VARCHAR(3) DEFAULT 'USD',
    location_city VARCHAR(100),
    location_state VARCHAR(50),
    location_country VARCHAR(50),
    condition_rating VARCHAR(20),   -- Excellent, Good, Fair, Poor
    listing_date TIMESTAMP,
    last_updated TIMESTAMP,
    status VARCHAR(20),            -- Active, Sold, Expired, Pending
    description TEXT,
    images_count INTEGER,
    dealer_name VARCHAR(200),
    contact_info JSONB,
    specifications JSONB,          -- Detailed specs as JSON
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);
```

Market Analytics Table

SQL

```sql
CREATE TABLE market_analytics (
    id SERIAL PRIMARY KEY,
    date DATE,
    make VARCHAR(50),
    model VARCHAR(100),
    capacity_range VARCHAR(20),    -- 0-50T, 50-100T, 100-200T, 200T+
    avg_price DECIMAL(12,2),
    median_price DECIMAL(12,2),
    min_price DECIMAL(12,2),
    max_price DECIMAL(12,2),
    listing_count INTEGER,
    sold_count INTEGER,
    avg_days_on_market INTEGER,
    price_trend_7d DECIMAL(5,2),   -- Percentage change
    price_trend_30d DECIMAL(5,2),
    inventory_level VARCHAR(20),    -- Low, Normal, High
    demand_score DECIMAL(3,2),     -- 0.0 to 1.0
    created_at TIMESTAMP DEFAULT NOW()
);
```

Live Ticker Data Table

SQL

```sql
CREATE TABLE live_ticker_data (
    id SERIAL PRIMARY KEY,
    ticker_symbol VARCHAR(20),     -- LR1300, LTM1350, RT890E, etc.
    display_name VARCHAR(100),     -- "Liebherr LR1300-2"
    current_price DECIMAL(12,2),
    price_change DECIMAL(12,2),
    price_change_pct DECIMAL(5,2),
    volume INTEGER,                -- Number of listings
    high_24h DECIMAL(12,2),
    low_24h DECIMAL(12,2),
    market_cap BIGINT,             -- Total value of all listings
    last_trade_time TIMESTAMP,
    status VARCHAR(20),            -- Active, Inactive, Maintenance
    category VARCHAR(50),          -- Crawler, Mobile, Tower, etc.
    updated_at TIMESTAMP DEFAULT NOW()
);
```

🔴 LIVE TICKER SYSTEM FRAMEWORK

Ticker Symbol Generation

Python

```python
def generate_ticker_symbol(make, model, capacity):
    """
    Generate Bloomberg-style ticker symbols
    Examples:
    - Liebherr LR1300-2 → LR1300
    - Grove RT890E → RT890E
    - Manitowoc 18000 → MTC18K
    """
    make_prefix = {
        'Liebherr': 'LR' if 'LR' in model else 'LTM',
        'Grove': 'RT' if 'RT' in model else 'AT',
        'Manitowoc': 'MTC',
        'Terex': 'TRX',
        'Link-Belt': 'LB',
        'Tadano': 'TD'
    }

    # Extract capacity or model number
    capacity_suffix = extract_model_number(model)
    return f"{make_prefix[make]}{capacity_suffix}"
```

Real-Time Price Calculation Engine

Python

```python
class MarketPriceEngine:
    def __init__(self, listings_data):
        self.listings = listings_data

    def calculate_current_price(self, ticker_symbol):
        """Calculate weighted average price for ticker"""
        recent_listings = self.get_recent_listings(ticker_symbol, days=30)

        # Weight by recency and listing quality
        weighted_prices = []
        for listing in recent_listings:
            weight = self.calculate_listing_weight(listing)
            weighted_prices.append(listing.price * weight)
```

```
        return sum(weighted_prices) / len(weighted_prices)

    def calculate_price_change(self, ticker_symbol, period='24h'):
        """Calculate price change over period"""
        current_price = self.calculate_current_price(ticker_symbol)
        historical_price = self.get_historical_price(ticker_symbol, period)

        change = current_price - historical_price
        change_pct = (change / historical_price) * 100

        return change, change_pct

    def calculate_listing_weight(self, listing):
        """Weight listings by quality factors"""
        weight = 1.0

        # Recency weight (newer = higher weight)
        days_old = (datetime.now() - listing.listing_date).days
        weight *= max(0.1, 1.0 - (days_old / 90))

        # Dealer reputation weight
        if listing.dealer_name in self.trusted_dealers:
            weight *= 1.2

        # Completeness weight (more info = higher weight)
        if listing.hours and listing.condition_rating:
            weight *= 1.1

        return weight
```

Live Ticker Display System

JavaScript

```
class LiveTickerDisplay {
    constructor(containerId) {
        this.container = document.getElementById(containerId);
        this.tickers = [];
        this.updateInterval = 5000; // 5 seconds
        this.init();
    }

    init() {
        this.createTickerElements();
        this.startLiveUpdates();
    }
```

```
    createTickerElements() {
        const tickerHTML = `
            <div class="ticker-container">
                <div class="ticker-scroll">
                    ${this.generateTickerItems()}
                </div>
            </div>
        `;
        this.container.innerHTML = tickerHTML;
    }

    generateTickerItems() {
        return this.tickers.map(ticker => `
            <div class="ticker-item ${ticker.change >= 0 ? 'positive' :
'negative'}">
                <span class="ticker-symbol">${ticker.symbol}</span>
                <span class="ticker-price">$${ticker.price.toLocaleString()}
</span>
                <span class="ticker-change">
                    ${ticker.change >= 0 ? '+' :
''}${ticker.change_pct.toFixed(1)}%
                </span>
                <span class="ticker-volume">${ticker.volume}</span>
            </div>
        `).join('');
    }

    async updateTickerData() {
        try {
            const response = await fetch('/api/v1/live-tickers');
            const data = await response.json();
            this.tickers = data.tickers;
            this.refreshDisplay();
        } catch (error) {
            console.error('Failed to update ticker data:', error);
        }
    }

    startLiveUpdates() {
        setInterval(() => {
            this.updateTickerData();
        }, this.updateInterval);
    }
}
```

📈 ADVANCED ANALYTICS EXPANSION

Market Intelligence Modules

1. Price Trend Analysis

Python

```
class PriceTrendAnalyzer:
    def __init__(self, listings_data):
        self.data = listings_data

    def calculate_trend_indicators(self, make, model, timeframe='30d'):
        """Calculate comprehensive trend indicators"""
        return {
            'moving_average_7d': self.moving_average(7),
            'moving_average_30d': self.moving_average(30),
            'rsi': self.relative_strength_index(),
            'bollinger_bands': self.bollinger_bands(),
            'volume_trend': self.volume_trend_analysis(),
            'seasonal_patterns': self.seasonal_analysis(),
            'market_momentum': self.momentum_indicator()
        }

    def generate_trend_signals(self, indicators):
        """Generate buy/sell/hold signals"""
        signals = []

        if indicators['rsi'] < 30:
            signals.append({'type': 'BUY', 'strength': 'STRONG', 'reason':
'Oversold condition'})
        elif indicators['rsi'] > 70:
            signals.append({'type': 'SELL', 'strength': 'STRONG', 'reason':
'Overbought condition'})

        if indicators['moving_average_7d'] >
indicators['moving_average_30d']:
            signals.append({'type': 'BUY', 'strength': 'MODERATE', 'reason':
'Upward trend'})

        return signals
```

2. Market Segmentation Engine

Python

```python
class MarketSegmentationEngine:
    def __init__(self, listings_data):
        self.data = listings_data

    def segment_by_capacity(self):
        """Segment market by crane capacity"""
        return {
            'mini_cranes': {'range': '0-15T', 'count': 0, 'avg_price': 0},
            'small_cranes': {'range': '15-50T', 'count': 0, 'avg_price': 0},
            'medium_cranes': {'range': '50-150T', 'count': 0, 'avg_price':
0},
            'large_cranes': {'range': '150-300T', 'count': 0, 'avg_price':
0},
            'super_cranes': {'range': '300T+', 'count': 0, 'avg_price': 0}
        }

    def segment_by_geography(self):
        """Segment market by geographic regions"""
        return {
            'northeast': self.analyze_region(['NY', 'NJ', 'CT', 'MA', 'PA']),
            'southeast': self.analyze_region(['FL', 'GA', 'SC', 'NC', 'VA']),
            'midwest': self.analyze_region(['IL', 'IN', 'OH', 'MI', 'WI']),
            'southwest': self.analyze_region(['TX', 'AZ', 'NM', 'OK']),
            'west': self.analyze_region(['CA', 'WA', 'OR', 'NV', 'CO'])
        }

    def segment_by_age(self):
        """Segment market by equipment age"""
        current_year = datetime.now().year
        return {
            'new': {'range': f'{current_year-2}-{current_year}', 'premium':
1.0},
            'recent': {'range': f'{current_year-5}-{current_year-3}',
'premium': 0.85},
            'mature': {'range': f'{current_year-10}-{current_year-6}',
'premium': 0.65},
            'older': {'range': f'{current_year-15}-{current_year-11}',
'premium': 0.45},
            'vintage': {'range': f'<{current_year-15}', 'premium': 0.25}
        }
```

3. Demand Forecasting System

Python

```python
class DemandForecastingSystem:
    def __init__(self, listings_data, economic_indicators):
        self.listings = listings_data
        self.economic_data = economic_indicators

    def forecast_demand(self, make, model, horizon_days=90):
        """Forecast demand using multiple indicators"""
        features = self.extract_features()

        # Machine learning model for demand prediction
        forecast = {
            'predicted_demand': self.ml_model.predict(features),
            'confidence_interval': self.calculate_confidence_interval(),
            'key_drivers': self.identify_demand_drivers(),
            'seasonal_factors': self.seasonal_adjustment(),
            'economic_impact': self.economic_impact_analysis()
        }

        return forecast

    def extract_features(self):
        """Extract features for ML model"""
        return {
            'historical_volume': self.get_historical_volume(),
            'price_trends': self.get_price_trends(),
            'inventory_levels': self.get_inventory_levels(),
            'construction_activity':
self.economic_data.construction_spending,
            'oil_prices': self.economic_data.oil_prices,
            'infrastructure_spending':
self.economic_data.infrastructure_budget,
            'seasonal_index': self.calculate_seasonal_index()
        }
```

4. Competitive Intelligence Module

Python

```python
class CompetitiveIntelligenceModule:
    def __init__(self, listings_data):
        self.data = listings_data

    def analyze_market_share(self):
        """Analyze market share by manufacturer"""
        total_listings = len(self.data)
```

```
        market_share = {}
        for make in self.get_unique_makes():
            make_listings = self.filter_by_make(make)
            share = {
                'listing_count': len(make_listings),
                'market_share_pct': (len(make_listings) / total_listings) *
100,
                'avg_price': self.calculate_avg_price(make_listings),
                'price_premium': self.calculate_price_premium(make_listings),
                'geographic_presence':
self.analyze_geographic_presence(make_listings)
            }
            market_share[make] = share

        return market_share

    def identify_pricing_strategies(self):
        """Identify pricing strategies by manufacturer"""
        strategies = {}

        for make in self.get_unique_makes():
            make_data = self.filter_by_make(make)

            strategies[make] = {
                'pricing_position':
self.determine_pricing_position(make_data),
                'discount_patterns':
self.analyze_discount_patterns(make_data),
                'premium_models': self.identify_premium_models(make_data),
                'value_models': self.identify_value_models(make_data),
                'regional_pricing': self.analyze_regional_pricing(make_data)
            }

        return strategies
```

🎯 BLOOMBERG TERMINAL-STYLE FEATURES

Advanced Dashboard Components

1. Market Heat Map

JavaScript

```
class MarketHeatMap {
    constructor(containerId) {
```

```
        this.container = document.getElementById(containerId);
        this.data = null;
    }

    render(marketData) {
        const heatMapData = this.processDataForHeatMap(marketData);

        // D3.js implementation for interactive heat map
        const svg = d3.select(this.container)
            .append('svg')
            .attr('width', 800)
            .attr('height', 600);

        // Color scale based on price changes
        const colorScale = d3.scaleSequential(d3.interpolateRdYlGn)
            .domain([-10, 10]); // -10% to +10% price change

        // Create heat map cells
        svg.selectAll('rect')
            .data(heatMapData)
            .enter()
            .append('rect')
            .attr('x', d => d.x)
            .attr('y', d => d.y)
            .attr('width', d => d.width)
            .attr('height', d => d.height)
            .attr('fill', d => colorScale(d.priceChange))
            .on('mouseover', this.showTooltip)
            .on('mouseout', this.hideTooltip);
    }
}
```

2. Real-Time Alert System

Python

```
class RealTimeAlertSystem:
    def __init__(self):
        self.alert_rules = []
        self.subscribers = []

    def create_price_alert(self, ticker, threshold, direction='above'):
        """Create price movement alerts"""
        alert = {
            'id': self.generate_alert_id(),
            'ticker': ticker,
            'threshold': threshold,
```

```
            'direction': direction,
            'created_at': datetime.now(),
            'status': 'active'
        }
        self.alert_rules.append(alert)

    def create_volume_alert(self, ticker, volume_threshold):
        """Create volume spike alerts"""
        alert = {
            'id': self.generate_alert_id(),
            'ticker': ticker,
            'volume_threshold': volume_threshold,
            'type': 'volume_spike',
            'created_at': datetime.now(),
            'status': 'active'
        }
        self.alert_rules.append(alert)

    def check_alerts(self, current_data):
        """Check all active alerts against current data"""
        triggered_alerts = []

        for alert in self.alert_rules:
            if self.evaluate_alert_condition(alert, current_data):
                triggered_alerts.append(alert)
                self.send_alert_notification(alert)

        return triggered_alerts
```

3. Portfolio Analytics Engine

Python

```
class PortfolioAnalyticsEngine:
    def __init__(self, portfolio_data):
        self.portfolio = portfolio_data

    def calculate_portfolio_metrics(self):
        """Calculate comprehensive portfolio metrics"""
        return {
            'total_value': self.calculate_total_value(),
            'daily_pnl': self.calculate_daily_pnl(),
            'ytd_performance': self.calculate_ytd_performance(),
            'risk_metrics': self.calculate_risk_metrics(),
            'diversification_score': self.calculate_diversification(),
            'liquidity_analysis': self.analyze_liquidity(),
            'age_distribution': self.analyze_age_distribution(),
```

```
            'geographic_exposure': self.analyze_geographic_exposure()
        }

    def generate_rebalancing_recommendations(self):
        """Generate portfolio rebalancing recommendations"""
        recommendations = []

        # Analyze concentration risk
        concentration_analysis = self.analyze_concentration_risk()
        if concentration_analysis['risk_level'] > 0.7:
            recommendations.append({
                'type': 'DIVERSIFY',
                'priority': 'HIGH',
                'action': 'Reduce concentration in top holdings',
                'impact': 'Lower portfolio risk'
            })

        # Analyze age distribution
        age_analysis = self.analyze_age_distribution()
        if age_analysis['avg_age'] > 10:
            recommendations.append({
                'type': 'REFRESH',
                'priority': 'MEDIUM',
                'action': 'Consider newer equipment acquisitions',
                'impact': 'Improve resale values'
            })

        return recommendations
```

🔄 DATA PROCESSING PIPELINE

Real-Time Data Ingestion

Python

```
class DataIngestionPipeline:
    def __init__(self):
        self.processors = []
        self.validators = []
        self.enrichers = []

    def process_new_listing(self, raw_listing_data):
        """Process new listing through complete pipeline"""

        # 1. Data validation and cleaning
```

```
        validated_data = self.validate_listing_data(raw_listing_data)

        # 2. Data enrichment
        enriched_data = self.enrich_listing_data(validated_data)

        # 3. Market analysis update
        self.update_market_analytics(enriched_data)

        # 4. Ticker data update
        self.update_ticker_data(enriched_data)

        # 5. Alert checking
        self.check_price_alerts(enriched_data)

        # 6. Database storage
        self.store_listing_data(enriched_data)

        return enriched_data

    def enrich_listing_data(self, listing_data):
        """Enrich listing with additional market intelligence"""
        enriched = listing_data.copy()

        # Add market position scoring
        enriched['market_position_score'] =
self.calculate_market_position(listing_data)

        # Add comparable analysis
        enriched['comparable_analysis'] = self.find_comparables(listing_data)

        # Add demand indicators
        enriched['demand_indicators'] =
self.calculate_demand_indicators(listing_data)

        # Add liquidity scoring
        enriched['liquidity_score'] =
self.calculate_liquidity_score(listing_data)

        return enriched
```

Market Data Aggregation

Python

```
class MarketDataAggregator:
    def __init__(self):
        self.aggregation_rules = self.load_aggregation_rules()
```

```python
    def aggregate_daily_data(self):
        """Aggregate daily market data for all tickers"""
        tickers = self.get_active_tickers()

        for ticker in tickers:
            daily_data = self.calculate_daily_aggregates(ticker)
            self.store_daily_aggregates(ticker, daily_data)

    def calculate_daily_aggregates(self, ticker):
        """Calculate daily aggregates for a ticker"""
        today_listings = self.get_todays_listings(ticker)

        return {
            'open_price': self.calculate_open_price(today_listings),
            'close_price': self.calculate_close_price(today_listings),
            'high_price': max([l.price for l in today_listings]),
            'low_price': min([l.price for l in today_listings]),
            'volume': len(today_listings),
            'avg_price': sum([l.price for l in today_listings]) /
len(today_listings),
            'median_price': self.calculate_median_price(today_listings),
            'price_volatility':
self.calculate_price_volatility(today_listings)
        }
```

📱 API ENDPOINTS FOR LIVE DATA

Core API Structure

Python

```python
from fastapi import FastAPI, WebSocket
from fastapi.responses import StreamingResponse

app = FastAPI()

@app.get("/api/v1/live-tickers")
async def get_live_tickers():
    """Get current ticker data for all active symbols"""
    tickers = ticker_service.get_all_active_tickers()
    return {"tickers": tickers, "last_updated": datetime.now()}

@app.get("/api/v1/ticker/{symbol}")
async def get_ticker_detail(symbol: str):
```

```
    """Get detailed data for specific ticker"""
    ticker_data = ticker_service.get_ticker_detail(symbol)
    return ticker_data

@app.get("/api/v1/market-overview")
async def get_market_overview():
    """Get overall market statistics"""
    return {
        "total_listings": market_service.get_total_listings(),
        "market_cap": market_service.get_total_market_cap(),
        "top_gainers": market_service.get_top_gainers(limit=10),
        "top_losers": market_service.get_top_losers(limit=10),
        "most_active": market_service.get_most_active(limit=10),
        "market_sentiment": market_service.get_market_sentiment()
    }

@app.websocket("/ws/live-data")
async def websocket_live_data(websocket: WebSocket):
    """WebSocket for real-time data streaming"""
    await websocket.accept()

    while True:
        # Stream live ticker updates
        ticker_updates = await ticker_service.get_live_updates()
        await websocket.send_json(ticker_updates)
        await asyncio.sleep(1)  # Update every second

@app.get("/api/v1/analytics/trend/{symbol}")
async def get_trend_analysis(symbol: str, period: str = "30d"):
    """Get trend analysis for specific symbol"""
    trend_data = analytics_service.get_trend_analysis(symbol, period)
    return trend_data

@app.get("/api/v1/analytics/forecast/{symbol}")
async def get_demand_forecast(symbol: str, horizon: int = 90):
    """Get demand forecast for specific symbol"""
    forecast = forecasting_service.get_demand_forecast(symbol, horizon)
    return forecast
```

This comprehensive framework transforms your 54,000 listings into a Bloomberg Terminal-quality live market intelligence system. The live tickers, advanced analytics, and real-time processing will create an incredibly compelling conference demo that showcases institutional-grade capabilities.