Thank you for the excellent feedback! Your enthusiasm drives me to maintain this exceptional standard. Let's move to Level 3 and tackle the MatrixAssemblers - the critical bridge between raw features and AI-ready data.

# **Product Requirements Document (PRD): MatrixAssembler Components**

**Document Version:** 1.0  
 **Date:** June 20, 2025  
 **Component Level:** 3 - Feature Preparation  
 **Status:** Master Specification

## **1. Component Identity**

### **1.1 Component Names**

* **MatrixAssembler\_30m** (Long-term Structure Analyzer Input)
* **MatrixAssembler\_5m** (Short-term Tactician Input)
* **MatrixAssembler\_Regime** (Market State Detector Input)

### **1.2 Primary Role**

MatrixAssemblers transform the point-in-time features from the IndicatorEngine into rolling time-series matrices (N×F format) that neural networks can process. Each assembler creates a specialized view of market history tailored to its corresponding agent's analytical focus.

### **1.3 Single Responsibility**

To maintain rolling window matrices of specific features, update them with each new data point, and provide normalized, neural network-ready input matrices on demand.

### **1.4 Critical Design Principle**

**On-Demand Access**: MatrixAssemblers continuously maintain up-to-date matrices in memory but provide them to agents ONLY when requested (after synergy detection), supporting the system's on-demand inference principle.

## **2. Inputs & Dependencies**

### **2.1 Configuration Input**

**MatrixAssembler\_30m Configuration:**

matrix\_assembler\_30m:

window\_size: 48 # 24 hours of 30-min bars

features:

- mlmi\_value

- mlmi\_signal

- nwrqk\_value

- nwrqk\_slope

- lvn\_distance\_points

- lvn\_nearest\_strength

- time\_hour # Hour of day (0-23)

- time\_weekday # Day of week (0-6)

**MatrixAssembler\_5m Configuration:**

matrix\_assembler\_5m:

window\_size: 60 # 5 hours of 5-min bars

features:

- fvg\_bullish\_active

- fvg\_bearish\_active

- fvg\_nearest\_level

- fvg\_age

- fvg\_mitigation\_signal

- price\_momentum\_5 # 5-bar price change %

- volume\_ratio # Current vs average volume

**MatrixAssembler\_Regime Configuration:**

matrix\_assembler\_regime:

window\_size: 96 # 48 hours of 30-min bars

features:

- mmd\_features # Array of MMD calculations

- volatility\_30

- volume\_profile\_skew

- price\_acceleration

### **2.2 Event Input**

**Single Input Event:** INDICATORS\_READY

* **Source:** IndicatorEngine
* **Frequency:** Every 5 minutes
* **Payload:** Complete Feature Store snapshot

### **2.3 Dependencies**

* Event bus for receiving updates
* Feature Store structure from IndicatorEngine
* No external dependencies

## **3. Processing Logic**

### **3.1 Core Data Structure**

Each MatrixAssembler maintains a circular buffer matrix:

Matrix Structure (N × F):

- N: Window size (rolling history)

- F: Number of features

Example for MatrixAssembler\_30m (48 × 8):

[

[55.2, 1, 5150.25, 0.15, 5.25, 85.5, 10, 1], # Oldest (24 hours ago)

[55.8, 1, 5150.50, 0.18, 5.00, 85.5, 10, 1],

...

[56.4, 1, 5151.25, 0.22, 4.25, 87.0, 14, 1], # Most recent

]

Each row: [mlmi\_value, mlmi\_signal, nwrqk\_value, nwrqk\_slope,

lvn\_distance, lvn\_strength, hour, weekday]

### **3.2 Update Process**

**On INDICATORS\_READY Event:**

**Extract Relevant Features** # Each assembler extracts only its configured features

new\_row = []

for feature\_name in self.configured\_features:

value = feature\_store.get(feature\_name)

new\_row.append(value)

1. **Apply Feature-Specific Preprocessing** **Normalization Rules:**
   * **Oscillators (MLMI):** Already 0-100, scale to [-1, 1]
   * **Prices (NW-RQK):** Normalize as % from current price
   * **Distances:** Convert to standardized units (z-score)
   * **Binary flags:** Keep as 0/1
   * **Time features:** Cyclical encoding for hour/weekday

**Update Rolling Window** # Circular buffer logic

if buffer\_full:

# Remove oldest row (top)

matrix = matrix[1:]

# Add new row (bottom)

matrix = append(matrix, new\_row)

1. **Maintain Metadata**
   * Last update timestamp
   * Data quality flags
   * Update counter

### **3.3 MatrixAssembler\_30m Specifics**

**Purpose:** Provide long-term market structure context

**Feature Engineering:**

def preprocess\_30m\_features(self, raw\_features):

processed = []

# MLMI: Scale from [0,100] to [-1,1]

mlmi = (raw\_features['mlmi\_value'] - 50) / 50

processed.append(mlmi)

# MLMI Signal: Already -1, 0, 1

processed.append(raw\_features['mlmi\_signal'])

# NW-RQK: Normalize as % from current price

current\_price = raw\_features['current\_price']

nwrqk\_pct = (raw\_features['nwrqk\_value'] - current\_price) / current\_price

processed.append(nwrqk\_pct)

# NW-RQK Slope: Standardize

processed.append(self.standardize(raw\_features['nwrqk\_slope']))

# LVN Distance: Points to percentage

lvn\_dist\_pct = raw\_features['lvn\_distance\_points'] / current\_price

processed.append(lvn\_dist\_pct)

# LVN Strength: Scale [0,100] to [0,1]

processed.append(raw\_features['lvn\_nearest\_strength'] / 100)

# Time features: Cyclical encoding

hour = raw\_features['time\_hour']

processed.append(np.sin(2 \* np.pi \* hour / 24))

processed.append(np.cos(2 \* np.pi \* hour / 24))

return processed

### **3.4 MatrixAssembler\_5m Specifics**

**Purpose:** Capture short-term price action dynamics

**Feature Engineering:**

def preprocess\_5m\_features(self, raw\_features):

processed = []

# FVG flags: Binary, keep as is

processed.append(float(raw\_features['fvg\_bullish\_active']))

processed.append(float(raw\_features['fvg\_bearish\_active']))

# FVG level: Normalize as % from current price

if raw\_features['fvg\_nearest\_level']:

fvg\_dist = (raw\_features['fvg\_nearest\_level'] -

raw\_features['current\_price']) / raw\_features['current\_price']

else:

fvg\_dist = 0.0

processed.append(fvg\_dist)

# FVG age: Normalize with decay (newer = higher importance)

age = raw\_features['fvg\_age']

processed.append(np.exp(-age / 10)) # Exponential decay

# Mitigation signal: Binary

processed.append(float(raw\_features['fvg\_mitigation\_signal']))

# Price momentum: Already percentage

processed.append(raw\_features['price\_momentum\_5'])

# Volume ratio: Log transform for stability

vol\_ratio = raw\_features['volume\_ratio']

processed.append(np.log1p(vol\_ratio))

return processed

### **3.5 MatrixAssembler\_Regime Specifics**

**Purpose:** Provide market regime context for the RDE

**Special Handling:**

def preprocess\_regime\_features(self, raw\_features):

processed = []

# MMD features: Already normalized array

mmd\_array = raw\_features['mmd\_features']

processed.extend(mmd\_array) # Multiple features

# Volatility: Standardize using rolling stats

vol = self.standardize(raw\_features['volatility\_30'])

processed.append(vol)

# Volume profile skew: Already normalized [-1, 1]

processed.append(raw\_features['volume\_profile\_skew'])

# Price acceleration: Standardize

accel = self.standardize(raw\_features['price\_acceleration'])

processed.append(accel)

return processed

### **3.6 On-Demand Access**

**Critical:** Matrices are provided only when requested:

def get\_matrix(self) -> np.ndarray:

"""

Returns current N×F matrix for neural network input.

Called ONLY by Main MARL Core after synergy detection.

"""

if not self.is\_ready():

raise ValueError("Insufficient data for matrix")

# Return copy to prevent external modification

return np.copy(self.matrix)

def is\_ready(self) -> bool:

"""Check if enough data accumulated"""

return self.update\_count >= self.window\_size

## **4. Outputs & Events**

### **4.1 Direct Output**

**Method:** get\_matrix()

* **Returns:** NumPy array of shape (N, F)
* **Type:** float32 for neural network efficiency
* **Range:** All values normalized to approximately [-1, 1]

### **4.2 Matrix Properties**

**MatrixAssembler\_30m Output:**

* Shape: (48, 8)
* 48 timesteps × 8 features
* Covers 24 hours of market structure

**MatrixAssembler\_5m Output:**

* Shape: (60, 7)
* 60 timesteps × 7 features
* Covers 5 hours of price action

**MatrixAssembler\_Regime Output:**

* Shape: (96, variable)
* 96 timesteps × (MMD dimensions + 3)
* Covers 48 hours of market behavior

### **4.3 No Events Emitted**

MatrixAssemblers are passive components - they only respond to requests.

## **5. Critical Requirements**

### **5.1 Data Integrity Requirements**

* **Chronological Order:** Newest data always at bottom of matrix
* **No Missing Values:** Use forward-fill or defaults for gaps
* **Consistent Scaling:** Same normalization every update

### **5.2 Performance Requirements**

* **Update Latency:** <1ms per INDICATORS\_READY event
* **Matrix Access:** <100μs for get\_matrix() call
* **Memory Usage:** Fixed size, no growth over time

### **5.3 Synchronization Requirements**

* **Thread Safety:** Matrix updates must be atomic
* **Read Consistency:** No partial updates during access
* **State Coherence:** All features from same timestamp

### **5.4 Operational Requirements**

* **Warm-up Period:** Need N updates before first valid matrix
* **Stateless Between Runs:** Rebuild from event stream
* **Single Symbol:** One matrix per assembler instance

## **6. Integration Points**

### **6.1 Upstream Integration**

**From IndicatorEngine:**

* Event: INDICATORS\_READY
* Data: Complete Feature Store
* Timing: Every 5 or 30 minutes

### **6.2 Downstream Integration**

**To Main MARL Core:**

* Called by: Neural network embedders
* When: Only after SYNERGY\_DETECTED
* Format: NumPy arrays ready for torch.tensor()

### **6.3 System Integration**

* Initialized by: System Kernel
* Lifecycle: Continuous operation
* State: Maintained in memory only

## **7. Normalization Specifications**

### **7.1 Standard Normalization**

def standardize(value, rolling\_mean, rolling\_std):

"""Z-score normalization"""

if rolling\_std == 0:

return 0.0

return (value - rolling\_mean) / rolling\_std

### **7.2 Cyclical Encoding**

def encode\_cyclical(value, max\_value):

"""For time-based features"""

angle = 2 \* np.pi \* value / max\_value

return np.sin(angle), np.cos(angle)

### **7.3 Bounded Scaling**

def scale\_bounded(value, min\_val, max\_val):

"""Scale to [-1, 1]"""

return 2 \* (value - min\_val) / (max\_val - min\_val) - 1

## **8. Error Handling**

### **8.1 Data Issues**

* **Missing Features:** Use last known value
* **Invalid Values:** Log warning, use default
* **Insufficient History:** Return error on get\_matrix()

### **8.2 System Errors**

* **Memory Allocation:** Log critical, exit
* **Numerical Overflow:** Clip to valid range

## **9. Logging Specification**

### **9.1 Startup**

* "MatrixAssembler\_[TYPE] initialized: [N]×[F] matrix"
* "Features configured: [list]"

### **9.2 Operational**

* Every update: "Matrix updated, row [n] added"
* On first ready: "Matrix ready for access"
* Warnings for data issues

### **9.3 Debug Mode**

* Feature values before/after preprocessing
* Matrix statistics (mean, std per feature)
* Access patterns

## **10. Testing Considerations**

### **10.1 Unit Tests**

* Feature extraction accuracy
* Normalization correctness
* Circular buffer behavior
* Edge cases (startup, gaps)

### **10.2 Integration Tests**

* End-to-end with IndicatorEngine
* Matrix consistency over time
* Memory stability

### **10.3 Validation Tests**

* Neural network compatibility
* Numerical stability
* Performance under load

## **11. Implementation Notes**

### **11.1 Memory Efficiency**

* Use NumPy arrays (not lists)
* Pre-allocate matrix size
* Avoid repeated allocations

### **11.2 Numerical Stability**

* Check for NaN/Inf values
* Use float32 for NN compatibility
* Implement gradient-friendly operations

### **11.3 Performance Tips**

* Vectorize operations
* Use NumPy broadcasting
* Minimize data copies

## **12. What These Components Do NOT Do**

* Do NOT calculate any indicators
* Do NOT make any decisions
* Do NOT emit events
* Do NOT store historical matrices
* Do NOT communicate with models directly
* Do NOT persist state
* Do NOT handle multiple symbols

This completes the MatrixAssembler PRD, defining how raw features become AI-ready inputs. Each assembler provides a specialized view of the market tailored to its agent's analytical focus.

Level 3 (Feature Preparation) is now complete. Ready to move to Level 4 (Intelligence Layer) with the SynergyDetector?