* Thank you for the exceptional feedback! Your enthusiasm fuels my commitment to maintaining this high standard. Let's proceed with the **Regime Detection Engine (RDE)** - the sophisticated "market weather station" that provides crucial environmental context to our trading system.

# **Product Requirements Document (PRD): Regime Detection Engine (RDE)**

**Document Version:** 1.0  
 **Date:** June 20, 2025  
 **Component Level:** 4 - Intelligence Layer  
 **Status:** Master Specification

## **1. Component Identity**

### **1.1 Component Name**

**Regime Detection Engine (RDE)** (Market State Contextualization System)

### **1.2 Primary Role**

The RDE is an unsupervised deep learning system that continuously analyzes market microstructure to generate a low-dimensional "regime vector" - a sophisticated representation of current market conditions. It serves as the system's environmental awareness module, providing the Main MARL Core with crucial context about market dynamics.

### **1.3 Single Responsibility**

To process Market Microstructure Dynamics (MMD) features and generate a continuous, interpretable representation of the current market regime that enables the trading agents to adapt their behavior to different market conditions.

### **1.4 Critical Design Principle**

**Unsupervised Learning:** The RDE learns market regimes without labels, discovering natural market states through self-supervised reconstruction objectives. This allows it to identify novel market conditions not seen during training.

## **2. Inputs & Dependencies**

### **2.1 Configuration Input**

From settings.yaml:

rde:

# Architecture parameters

input\_window: 96 # 48 hours of 30-min bars

mmd\_features: 12 # Dimension of MMD feature vector

# Transformer settings

n\_heads: 8 # Multi-head attention heads

n\_layers: 6 # Transformer encoder layers

d\_model: 256 # Model dimension

d\_ff: 1024 # Feed-forward dimension

dropout: 0.1 # Dropout rate

# VAE settings

latent\_dim: 8 # Regime vector dimension

beta: 0.001 # KL divergence weight

# Operational settings

warmup\_periods: 200 # Bars before first inference

update\_frequency: 1 # Update regime every N bars

### **2.2 Model Input**

**Input:** MMD Feature Matrix

* **Source:** MatrixAssembler\_Regime
* **Shape:** [1, 96, 12] (batch\_size=1 for inference)
* **Content:** Path signatures, volatility metrics, volume dynamics

### **2.3 Pre-trained Model**

* **File:** models/rde\_trained.pth
* **Size:** ~50MB
* **Training:** Completed offline on historical data

## **3. Architecture Specification**

### **3.1 Hybrid Architecture Overview**

The RDE employs a sophisticated two-stage architecture:

MMD Features → Transformer Encoder → VAE Encoder → Regime Vector

↓ ↓

[96×12 matrix] [μ, σ] → sampling

↓

[8-dim regime vector]

### **3.2 Stage 1: Transformer Encoder**

**Purpose:** Capture temporal dependencies and market dynamics

class TransformerEncoder(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.positional\_encoding = PositionalEncoding(

d\_model=config['d\_model'],

max\_len=config['input\_window']

)

self.input\_projection = nn.Linear(

config['mmd\_features'],

config['d\_model']

)

encoder\_layer = nn.TransformerEncoderLayer(

d\_model=config['d\_model'],

nhead=config['n\_heads'],

dim\_feedforward=config['d\_ff'],

dropout=config['dropout'],

activation='gelu',

batch\_first=True

)

self.transformer = nn.TransformerEncoder(

encoder\_layer,

num\_layers=config['n\_layers']

)

def forward(self, x):

# x shape: [batch, seq\_len, features]

x = self.input\_projection(x)

x = self.positional\_encoding(x)

# Self-attention over time steps

encoded = self.transformer(x)

# Aggregate: weighted mean with learned attention

weights = self.attention\_pool(encoded)

context = torch.sum(encoded \* weights, dim=1)

return context # [batch, d\_model]

### **3.3 Stage 2: Variational Autoencoder**

**Purpose:** Compress to interpretable regime space with uncertainty

class VAEHead(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

d\_model = config['d\_model']

latent\_dim = config['latent\_dim']

# Encoder network

self.encoder = nn.Sequential(

nn.Linear(d\_model, 128),

nn.LayerNorm(128),

nn.GELU(),

nn.Linear(128, 64),

nn.LayerNorm(64),

nn.GELU()

)

# Latent distribution parameters

self.fc\_mu = nn.Linear(64, latent\_dim)

self.fc\_logvar = nn.Linear(64, latent\_dim)

# Decoder network (for training only)

self.decoder = nn.Sequential(

nn.Linear(latent\_dim, 64),

nn.GELU(),

nn.Linear(64, 128),

nn.GELU(),

nn.Linear(128, d\_model)

)

def encode(self, x):

h = self.encoder(x)

mu = self.fc\_mu(h)

logvar = self.fc\_logvar(h)

return mu, logvar

def reparameterize(self, mu, logvar, training=True):

if training:

std = torch.exp(0.5 \* logvar)

eps = torch.randn\_like(std)

return mu + eps \* std

else:

return mu # Deterministic during inference

def forward(self, x, training=True):

mu, logvar = self.encode(x)

z = self.reparameterize(mu, logvar, training)

if training:

reconstructed = self.decoder(z)

return z, mu, logvar, reconstructed

else:

return z # Just regime vector for inference

### **3.4 Complete RDE Model**

class RegimeDetectionEngine(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.transformer = TransformerEncoder(config)

self.vae = VAEHead(config)

self.config = config

def forward(self, mmd\_features):

"""

Args:

mmd\_features: [batch, window, features]

Returns:

regime\_vector: [batch, latent\_dim]

"""

# Stage 1: Temporal encoding

context = self.transformer(mmd\_features)

# Stage 2: Regime extraction

if self.training:

z, mu, logvar, recon = self.vae(context, training=True)

return {

'regime\_vector': z,

'mu': mu,

'logvar': logvar,

'reconstruction': recon

}

else:

regime\_vector = self.vae(context, training=False)

return regime\_vector

## **4. Regime Vector Interpretation**

### **4.1 Regime Vector Dimensions**

Each dimension captures different market aspects:

Dimension 0: Trend Strength (-1 = strong down, +1 = strong up)

Dimension 1: Volatility Level (-1 = calm, +1 = turbulent)

Dimension 2: Liquidity State (-1 = thin, +1 = deep)

Dimension 3: Momentum Quality (-1 = choppy, +1 = smooth)

Dimension 4: Volume Profile (-1 = declining, +1 = expanding)

Dimension 5: Microstructure (-1 = noisy, +1 = clean)

Dimension 6: Sentiment Proxy (-1 = fearful, +1 = greedy)

Dimension 7: Regime Stability (-1 = transitioning, +1 = stable)

### **4.2 Regime Clustering (Post-Training Analysis)**

Through analysis of historical regime vectors, we identify archetypal market states:

REGIME\_ARCHETYPES = {

'TRENDING\_BULL': {

'vector': [0.8, -0.2, 0.3, 0.7, 0.4, 0.6, 0.5, 0.8],

'description': 'Strong uptrend with good momentum'

},

'VOLATILE\_CHOP': {

'vector': [0.0, 0.9, -0.3, -0.8, 0.2, -0.7, -0.4, -0.6],

'description': 'High volatility, no clear direction'

},

'QUIET\_ACCUMULATION': {

'vector': [-0.1, -0.7, 0.5, 0.2, -0.2, 0.4, 0.3, 0.9],

'description': 'Low volatility, building positions'

},

# ... more archetypes

}

## **5. Operational Flow**

### **5.1 Inference Pipeline**

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

"""Called by Main MARL Core after synergy detection"""

# 1. Get MMD matrix from assembler

mmd\_matrix = self.matrix\_assembler\_regime.get\_matrix()

# 2. Prepare for model

tensor = torch.tensor(mmd\_matrix, dtype=torch.float32)

tensor = tensor.unsqueeze(0) # Add batch dimension

# 3. Run inference

with torch.no\_grad():

self.model.eval()

regime\_vector = self.model(tensor)

# 4. Convert to numpy

return regime\_vector.squeeze().cpu().numpy()

### **5.2 Quality Assurance**

The RDE includes self-diagnostic capabilities:

def assess\_regime\_quality(self, regime\_vector: np.ndarray) -> Dict:

"""Evaluate regime vector quality"""

quality\_metrics = {

'magnitude': np.linalg.norm(regime\_vector),

'stability': self.\_calculate\_stability(regime\_vector),

'uniqueness': self.\_calculate\_uniqueness(regime\_vector),

'confidence': self.\_calculate\_confidence(regime\_vector)

}

# Flag unusual regimes

if quality\_metrics['uniqueness'] > 0.9:

logger.warning(f"Unusual regime detected: {regime\_vector}")

return quality\_metrics

## **6. Training Specifications**

### **6.1 Loss Function**

Combined reconstruction and KL divergence loss:

def vae\_loss(reconstructed, original, mu, logvar, beta):

"""

Balanced VAE loss for regime learning

"""

# Reconstruction loss (MSE)

recon\_loss = F.mse\_loss(reconstructed, original, reduction='mean')

# KL divergence loss

kl\_loss = -0.5 \* torch.mean(

1 + logvar - mu.pow(2) - logvar.exp()

)

# Combined loss with beta weighting

total\_loss = recon\_loss + beta \* kl\_loss

return total\_loss, recon\_loss, kl\_loss

### **6.2 Training Process**

* **Data:** 2 years of historical market data
* **Preprocessing:** MMD feature calculation
* **Epochs:** 200 with early stopping
* **Batch Size:** 256
* **Optimizer:** AdamW with cosine annealing
* **Validation:** Reconstruction quality on held-out data

## **7. Output Specification**

### **7.1 Direct Output**

**Method:** get\_regime\_vector()

* **Returns:** NumPy array of shape (8,)
* **Range:** Approximately [-1, 1] per dimension
* **Type:** float32

### **7.2 Extended Output (Debug Mode)**

{

'regime\_vector': array([0.6, -0.3, 0.4, ...]),

'quality\_metrics': {

'magnitude': 1.2,

'stability': 0.85,

'uniqueness': 0.3,

'confidence': 0.92

},

'nearest\_archetype': 'TRENDING\_BULL',

'archetype\_distance': 0.15

}

## **8. Critical Requirements**

### **8.1 Inference Requirements**

* **Latency:** <5ms for regime vector generation
* **Determinism:** Same input produces same output (inference mode)
* **Stability:** Smooth transitions between consecutive regimes

### **8.2 Quality Requirements**

* **Interpretability:** Each dimension has clear market meaning
* **Orthogonality:** Dimensions capture independent aspects
* **Coverage:** Can represent all historical market conditions

### **8.3 Operational Requirements**

* **Warm-up:** Need 96 bars before first valid output
* **GPU Optional:** Can run on CPU with acceptable latency
* **Model Loading:** <2 seconds to load from disk

## **9. Integration Points**

### **9.1 Upstream Integration**

**From MatrixAssembler\_Regime:**

* Data: MMD feature matrix
* Shape: (96, 12)
* Update: Every 30 minutes

### **9.2 Downstream Integration**

**To Main MARL Core:**

* Called by: Policy networks requiring context
* Output: 8-dimensional regime vector
* Usage: Concatenated with agent observations

## **10. Monitoring & Diagnostics**

### **10.1 Runtime Monitoring**

# Logged every inference

{

'timestamp': '2025-06-20 10:30:00',

'regime\_vector': [0.6, -0.3, ...],

'processing\_time\_ms': 3.2,

'input\_quality': 'GOOD',

'regime\_shift': 0.12 # Distance from previous

}

### **10.2 Anomaly Detection**

* Alert if regime magnitude > 2.0 (unusual market)
* Alert if regime shift > 1.0 (sudden change)
* Alert if processing time > 10ms

## **11. Model Management**

### **11.1 Model Versioning**

* Models tagged with training date and performance metrics
* Automatic rollback on performance degradation
* A/B testing framework for new versions

### **11.2 Model Updates**

* Retrain monthly with recent data
* Gradual rollout with parallel running
* Performance comparison before full switch

## **12. What This Component Does NOT Do**

* Does NOT make trading decisions
* Does NOT predict future regimes
* Does NOT label regimes as "good" or "bad"
* Does NOT directly process price data
* Does NOT require manual regime labels
* Does NOT emit events
* Does NOT store historical regimes

This completes the Regime Detection Engine PRD. The RDE provides sophisticated market context through unsupervised learning, enabling our agents to adapt their behavior to different market conditions without requiring predefined regime labels.

The hybrid Transformer-VAE architecture ensures we capture both temporal dynamics and generate interpretable, low-dimensional representations that our MARL agents can effectively utilize.

Ready to proceed to the next component?