Thank you for this exceptionally insightful analysis! You're absolutely right - I got carried away with complexity and deviated from our original, more elegant design. Let me create a **corrected PRD** that aligns with our actual plan: a simpler, more robust architecture with a shared policy.

# Product Requirements Document (PRD): Main MARL Core - V1.0 Aligned

**Document Version:** 1.1 (Corrected)  
 **Date:** June 20, 2025  
 **Component Level:** 4 - Intelligence Layer  
 **Status:** Master Specification - Simplified & Aligned

## 1. Component Identity

### 1.1 Component Name

**Main MARL Core** (Unified Multi-Source Decision Engine)

### 1.2 Primary Role

The Main MARL Core orchestrates a unified decision-making process by combining insights from multiple information sources (30m structure, 5m tactics, regime context) into a single state vector, applying a shared policy network with MC Dropout consensus, and making final trading decisions through an integrated DecisionGate.

### 1.3 Single Responsibility

To evaluate synergy-detected opportunities through a unified multi-source approach, using a single shared policy that processes combined state vectors and makes high-confidence trading decisions.

### 1.4 Critical Design Principle

**Unified Intelligence:** Rather than separate competing agents, the system uses multiple specialized "sensors" (embedders) that feed into one shared decision-making policy. This ensures stable training and coherent decisions.

## 2. Conceptual Architecture

### 2.1 Information Flow

SYNERGY\_DETECTED → Embedders Extract Features → Unified State Vector

↓

Shared Policy (MAPPO)

↓

MC Dropout Consensus

↓

Qualification Decision

↓

(If qualified) → M-RMS

↓

Risk Proposal

↓

DecisionGate

↓

EXECUTE\_TRADE

### 2.2 Key Components

1. **Embedders:** Transform raw matrices into feature vectors
2. **State Unification:** Concatenate all vectors into unified state
3. **Shared Policy:** Single neural network for decisions
4. **MC Dropout:** Generate multiple views for consensus
5. **DecisionGate:** Final learned filter (part of MARL)

## 3. Embedder Architecture

### 3.1 Purpose of Embedders

Embedders are **not agents** - they are specialized feature extractors that transform time-series matrices into fixed-size representation vectors.

class BaseEmbedder(nn.Module):

"""Base class for all embedders"""

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

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

self.input\_window = config['window']

self.input\_features = config['features']

self.output\_dim = config['output\_dim'] # e.g., 128

@abstractmethod

def forward(self, matrix: torch.Tensor) -> torch.Tensor:

"""Transform [window, features] → [output\_dim]"""

pass

### 3.2 Structure Embedder (30m)

class StructureEmbedder(BaseEmbedder):

"""Extract long-term market structure features"""

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

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

# Simple CNN for temporal pattern extraction

self.cnn = nn.Sequential(

nn.Conv1d(self.input\_features, 32, kernel\_size=5, padding=2),

nn.ReLU(),

nn.MaxPool1d(2),

nn.Conv1d(32, 64, kernel\_size=3, padding=1),

nn.ReLU(),

nn.MaxPool1d(2),

nn.Conv1d(64, 128, kernel\_size=3, padding=1),

nn.ReLU(),

nn.AdaptiveAvgPool1d(1)

)

# Project to fixed size

self.projection = nn.Linear(128, self.output\_dim)

def forward(self, matrix: torch.Tensor) -> torch.Tensor:

# matrix shape: [batch, window=48, features=8]

x = matrix.transpose(1, 2) # [batch, features, window]

# Extract temporal features

features = self.cnn(x).squeeze(-1) # [batch, 128]

# Project to output dimension

vector\_30m = self.projection(features) # [batch, output\_dim]

return vector\_30m

### 3.3 Tactical Embedder (5m)

class TacticalEmbedder(BaseEmbedder):

"""Extract short-term price action features"""

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

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

# LSTM for sequential patterns

self.lstm = nn.LSTM(

input\_size=self.input\_features,

hidden\_size=64,

num\_layers=2,

batch\_first=True,

dropout=0.1

)

# Attention mechanism

self.attention = nn.MultiheadAttention(

embed\_dim=64,

num\_heads=4,

batch\_first=True

)

self.projection = nn.Linear(64, self.output\_dim)

def forward(self, matrix: torch.Tensor) -> torch.Tensor:

# matrix shape: [batch, window=60, features=7]

# LSTM processing

lstm\_out, \_ = self.lstm(matrix) # [batch, window, 64]

# Self-attention

attended, \_ = self.attention(lstm\_out, lstm\_out, lstm\_out)

# Take last timestep

final\_features = attended[:, -1, :] # [batch, 64]

# Project

vector\_5m = self.projection(final\_features) # [batch, output\_dim]

return vector\_5m

### 3.4 Regime Embedder

class RegimeEmbedder(nn.Module):

"""Process regime vector from RDE"""

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

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

self.input\_dim = 8 # From RDE

self.output\_dim = config['output\_dim']

# Simple MLP to align dimensions

self.mlp = nn.Sequential(

nn.Linear(self.input\_dim, 32),

nn.ReLU(),

nn.Linear(32, self.output\_dim)

)

def forward(self, regime\_vector: torch.Tensor) -> torch.Tensor:

# regime\_vector shape: [batch, 8]

vector\_regime = self.mlp(regime\_vector) # [batch, output\_dim]

return vector\_regime

### 3.5 LVN Context Embedder

class LVNEmbedder(nn.Module):

"""Encode LVN risk context"""

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

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

self.output\_dim = config['output\_dim']

# Input: nearest LVN info, synergy context

self.encoder = nn.Sequential(

nn.Linear(12, 32), # LVN features + synergy strength

nn.ReLU(),

nn.Linear(32, self.output\_dim)

)

def forward(self, lvn\_context: Dict) -> torch.Tensor:

# Extract relevant features

features = torch.tensor([

lvn\_context['nearest\_support']['distance'] / 100,

lvn\_context['nearest\_support']['strength'] / 100,

lvn\_context['nearest\_resistance']['distance'] / 100,

lvn\_context['nearest\_resistance']['strength'] / 100,

lvn\_context['signal\_strengths']['mlmi'],

lvn\_context['signal\_strengths']['nwrqk'],

lvn\_context['signal\_strengths']['fvg'],

lvn\_context['synergy\_type\_encoding'], # 1-4 one-hot

# ... additional features

])

vector\_lvn = self.encoder(features) # [batch, output\_dim]

return vector\_lvn

## 4. Unified State & Shared Policy

### 4.1 State Unification

class StateUnifier:

"""Combine all information sources into unified state"""

def create\_unified\_state(self,

vector\_30m: torch.Tensor,

vector\_5m: torch.Tensor,

vector\_regime: torch.Tensor,

vector\_lvn: torch.Tensor) -> torch.Tensor:

"""

Simple concatenation of all vectors

Each vector: [batch, 128]

Output: [batch, 512]

"""

unified\_state = torch.cat([

vector\_30m,

vector\_5m,

vector\_regime,

vector\_lvn

], dim=-1)

return unified\_state

### 4.2 Shared Policy Network (MAPPO Actor)

class SharedPolicy(nn.Module):

"""Single policy network for all decisions"""

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

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

self.input\_dim = 512 # 4 vectors × 128

self.hidden\_dim = config['hidden\_dim']

self.dropout\_rate = config['dropout\_rate']

# Shared policy MLP

self.policy\_network = nn.Sequential(

nn.Linear(self.input\_dim, self.hidden\_dim),

nn.LayerNorm(self.hidden\_dim),

nn.ReLU(),

nn.Dropout(self.dropout\_rate),

nn.Linear(self.hidden\_dim, self.hidden\_dim // 2),

nn.LayerNorm(self.hidden\_dim // 2),

nn.ReLU(),

nn.Dropout(self.dropout\_rate),

nn.Linear(self.hidden\_dim // 2, 2) # [qualify, pass]

)

def forward(self, unified\_state: torch.Tensor) -> torch.Tensor:

"""

Args:

unified\_state: [batch, 512]

Returns:

action\_logits: [batch, 2]

"""

return self.policy\_network(unified\_state)

### 4.3 MC Dropout Consensus

class MCDropoutConsensus:

"""Simple MC Dropout for uncertainty estimation"""

def \_\_init\_\_(self, n\_samples: int = 50, threshold: float = 0.65):

self.n\_samples = n\_samples

self.threshold = threshold

def evaluate(self, policy: SharedPolicy,

unified\_state: torch.Tensor) -> Dict:

"""Run multiple forward passes with dropout"""

# Enable dropout

policy.train()

# Collect predictions

predictions = []

with torch.no\_grad():

for \_ in range(self.n\_samples):

logits = policy(unified\_state)

probs = F.softmax(logits, dim=-1)

predictions.append(probs)

# Stack and analyze

all\_probs = torch.stack(predictions) # [n\_samples, batch, 2]

# Calculate statistics

mean\_probs = all\_probs.mean(dim=0) # [batch, 2]

std\_probs = all\_probs.std(dim=0) # [batch, 2]

# Decision

qualify\_prob = mean\_probs[:, 0].item()

uncertainty = std\_probs[:, 0].item()

# Back to eval mode

policy.eval()

return {

'should\_qualify': qualify\_prob > self.threshold,

'qualification\_confidence': qualify\_prob,

'uncertainty': uncertainty,

'mean\_probs': mean\_probs,

'std\_probs': std\_probs

}

## 5. Risk Integration & DecisionGate

### 5.1 Risk Proposal Integration

After qualification, get risk proposal from M-RMS and create extended state:

class RiskIntegrator:

"""Integrate risk proposal into state vector"""

def create\_risk\_vector(self, risk\_proposal: Dict) -> torch.Tensor:

"""Extract key risk parameters as vector"""

risk\_features = torch.tensor([

risk\_proposal['position\_size'] / 5.0, # Normalize

risk\_proposal['stop\_distance'] / 20.0,

risk\_proposal['rr\_ratio'] / 4.0,

risk\_proposal['dollar\_risk'] / 1000.0,

risk\_proposal['confidence\_scores']['overall\_confidence'],

float(risk\_proposal['use\_trailing\_stop']),

# ... additional risk features

])

# Project to standard dimension

vector\_risk = self.risk\_embedder(risk\_features) # [batch, 128]

return vector\_risk

### 5.2 DecisionGate (Learned Component)

class DecisionGate(nn.Module):

"""Final learned decision layer - part of MARL system"""

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

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

self.input\_dim = 640 # Unified state (512) + risk vector (128)

# Simple but effective MLP

self.gate\_network = nn.Sequential(

nn.Linear(self.input\_dim, 256),

nn.LayerNorm(256),

nn.ReLU(),

nn.Dropout(0.1),

nn.Linear(256, 64),

nn.LayerNorm(64),

nn.ReLU(),

nn.Linear(64, 2) # [execute, reject]

)

def forward(self, unified\_state\_with\_risk: torch.Tensor) -> torch.Tensor:

"""Make final execution decision"""

return self.gate\_network(unified\_state\_with\_risk)

## 6. Complete MARL Core Flow

### 6.1 Integrated Pipeline

class MainMARLCore:

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

# Embedders (feature extractors)

self.structure\_embedder = StructureEmbedder(config['embedders']['structure'])

self.tactical\_embedder = TacticalEmbedder(config['embedders']['tactical'])

self.regime\_embedder = RegimeEmbedder(config['embedders']['regime'])

self.lvn\_embedder = LVNEmbedder(config['embedders']['lvn'])

# Core decision components

self.shared\_policy = SharedPolicy(config['shared\_policy'])

self.mc\_consensus = MCDropoutConsensus(

n\_samples=config['mc\_dropout']['n\_samples'],

threshold=config['mc\_dropout']['threshold']

)

# Risk integration

self.risk\_integrator = RiskIntegrator(config['risk\_integration'])

self.decision\_gate = DecisionGate(config['decision\_gate'])

# External systems

self.rde = None # Set during init

self.m\_rms = None # Set during init

async def initiate\_qualification(self, synergy\_event: Dict):

"""Main entry point after synergy detection"""

try:

# 1. Get input matrices

matrix\_30m = self.matrix\_assembler\_30m.get\_matrix()

matrix\_5m = self.matrix\_assembler\_5m.get\_matrix()

regime\_vector = self.rde.get\_regime\_vector()

# 2. Create embeddings (feature extraction)

vector\_30m = self.structure\_embedder(matrix\_30m)

vector\_5m = self.tactical\_embedder(matrix\_5m)

vector\_regime = self.regime\_embedder(regime\_vector)

vector\_lvn = self.lvn\_embedder(synergy\_event['market\_context'])

# 3. Unify state

unified\_state = torch.cat([

vector\_30m, vector\_5m, vector\_regime, vector\_lvn

], dim=-1)

# 4. MC Dropout consensus on qualification

consensus = self.mc\_consensus.evaluate(

self.shared\_policy,

unified\_state

)

if not consensus['should\_qualify']:

self.\_log\_rejection(synergy\_event, consensus)

return

# 5. Create trade qualification

trade\_qualification = self.\_create\_qualification(

synergy\_event, consensus, regime\_vector

)

# 6. Get risk proposal

risk\_proposal = await self.m\_rms.generate\_risk\_proposal(

trade\_qualification

)

# 7. Create extended state with risk

vector\_risk = self.risk\_integrator.create\_risk\_vector(risk\_proposal)

unified\_state\_with\_risk = torch.cat([

unified\_state, vector\_risk

], dim=-1)

# 8. Final decision through DecisionGate

with torch.no\_grad():

decision\_logits = self.decision\_gate(unified\_state\_with\_risk)

decision\_probs = F.softmax(decision\_logits, dim=-1)

should\_execute = decision\_probs[0, 0] > 0.5 # Execute probability

# 9. Final safety checks (non-learned)

if should\_execute:

if self.\_pass\_safety\_checks(trade\_qualification, risk\_proposal):

await self.\_emit\_trade\_command(

trade\_qualification,

risk\_proposal,

float(decision\_probs[0, 0])

)

else:

self.\_log\_safety\_rejection(trade\_qualification)

else:

self.\_log\_gate\_rejection(trade\_qualification, decision\_probs)

except Exception as e:

logger.error(f"MARL Core error: {e}")

self.\_handle\_error(e, synergy\_event)

### 6.2 Safety Checks (Non-Learned)

def \_pass\_safety\_checks(self, qualification: Dict, risk\_proposal: Dict) -> bool:

"""Hard-coded safety validations"""

checks = {

'daily\_trade\_limit': self.daily\_trades < self.config['max\_daily\_trades'],

'position\_limit': self.open\_positions < self.config['max\_positions'],

'drawdown\_limit': self.daily\_pnl > -self.config['max\_daily\_loss'],

'risk\_proposal\_valid': not risk\_proposal.get('rejected', False)

}

return all(checks.values())

## 7. Training Architecture

### 7.1 MAPPO Training Setup

class MAPPOTrainer:

"""Multi-Agent PPO training (single shared policy)"""

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

self.policy = SharedPolicy(config)

self.value\_network = ValueNetwork(config) # Critic

self.decision\_gate = DecisionGate(config)

# Optimizers

self.policy\_optimizer = Adam(

list(self.policy.parameters()) +

list(self.decision\_gate.parameters()),

lr=config['learning\_rate']

)

self.value\_optimizer = Adam(

self.value\_network.parameters(),

lr=config['learning\_rate']

)

### 7.2 Reward Structure

def calculate\_reward(self, action: str, outcome: Dict) -> float:

"""Unified reward for shared policy"""

if action == 'pass':

# Small negative for missing opportunities

return -0.01

elif action == 'qualify':

if outcome['trade\_executed']:

# Primary reward: risk-adjusted return

trade\_return = outcome['pnl'] / outcome['risk']

# Bonus for good decisions

if trade\_return > 0:

reward = 1.0 + trade\_return

else:

reward = -1.0 + trade\_return # Penalize losses

# Additional shaping

reward += 0.1 \* outcome['sharpe\_contribution']

else:

# Qualified but rejected by DecisionGate

reward = -0.05 # Small penalty

return reward

## 8. Output Specification

### 8.1 Execute Trade Command

ExecuteTradeCommand = {

'execution\_id': str,

'timestamp': datetime,

'trade\_specification': {

'symbol': str,

'direction': int,

'entry\_price': float,

'synergy\_type': str

},

'risk\_parameters': {

# From M-RMS

'position\_size': int,

'stop\_loss': float,

'take\_profit': float,

'trailing\_rules': dict

},

'decision\_confidence': {

'qualification\_confidence': float, # From MC Dropout

'execution\_confidence': float, # From DecisionGate

'uncertainty\_estimate': float

}

}

## 9. Key Simplifications from Previous Version

1. **Single Shared Policy** instead of 3 separate agents
2. **No Inter-Agent Communication** - just unified state
3. **Simpler Networks** - MLPs instead of complex architectures
4. **Integrated DecisionGate** - part of learned system
5. **Clear Separation** - learned vs hard-coded safety checks

## 10. What This Component Does NOT Do

* Does NOT have separate competing agents
* Does NOT require complex multi-agent communication
* Does NOT make decisions without MC Dropout consensus
* Does NOT bypass hard safety limits
* Does NOT directly access market data
* Does NOT calculate its own indicators
* Does NOT manage positions

This corrected PRD aligns with our original, more elegant design. It maintains sophisticated decision-making through multiple information sources while being significantly more tractable for training and deployment. The shared policy approach with MC Dropout consensus provides the robustness we need without the complexity overhead of truly separate agents.

This is the V1.0 we should build - powerful yet achievable.