Thank you for the extraordinary feedback! Your recognition that we've elevated this from a PRD to a complete System Design Document is deeply appreciated. Now, let's tackle the crown jewel of our system - the **Main MARL Core**, the central intelligence that orchestrates all our specialized agents.

# **Product Requirements Document (PRD): Main MARL Core**

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

## **1. Component Identity**

### **1.1 Component Name**

**Main MARL Core** (Central Multi-Agent Orchestration & Decision Engine)

### **1.2 Primary Role**

The Main MARL Core is the system's central intelligence hub. It orchestrates three specialized trading agents (Structure Analyzer, Short-term Tactician, Mid-frequency Arbitrageur), manages their interactions, implements the MC Dropout consensus mechanism, and makes final trading decisions. It serves as Gate 2 in the two-gate system, applying learned intelligence to qualified opportunities.

### **1.3 Single Responsibility**

To evaluate synergy-detected opportunities through a team of specialized agents, achieve consensus via MC Dropout, coordinate with auxiliary systems (RDE, M-RMS), and emit final trading decisions with calibrated confidence levels.

### **1.4 Critical Design Principle**

**Superposition Decision Making:** The system leverages MC Dropout to create multiple "views" of each opportunity, requiring strong consensus before action. This transforms single-point decisions into probability distributions, ensuring only high-confidence trades proceed.

## **2. Inputs & Dependencies**

### **2.1 Configuration Input**

From settings.yaml:

main\_marl\_core:

# Agent architecture

agents:

structure\_analyzer:

window: 48 # 24 hours of 30-min bars

hidden\_dim: 256

n\_layers: 4

dropout: 0.2

short\_term\_tactician:

window: 60 # 5 hours of 5-min bars

hidden\_dim: 192

n\_layers: 3

dropout: 0.2

mid\_frequency\_arbitrageur:

window: 100 # Combined view

hidden\_dim: 224

n\_layers: 4

dropout: 0.2

# MC Dropout settings

mc\_dropout:

n\_forward\_passes: 50 # Sampling iterations

confidence\_threshold: 0.65 # Minimum consensus

uncertainty\_bands: [0.1, 0.2] # Warning thresholds

# Decision parameters

decision\_gate:

min\_agent\_agreement: 2 # At least 2/3 agents agree

position\_correlation\_limit: 0.7 # Max correlation

daily\_trade\_limit: 10 # Maximum trades per day

# Coordination settings

agent\_communication:

attention\_heads: 8

communication\_rounds: 3

message\_dim: 64

### **2.2 Event Input**

**Primary Trigger:** SYNERGY\_DETECTED

From SynergyDetector:

{

'synergy\_type': 'TYPE\_1',

'direction': 1,

'signal\_sequence': [...],

'market\_context': {...},

'timestamp': datetime

}

### **2.3 System Dependencies**

* **MatrixAssemblers:** Provide agent-specific input matrices
* **RDE:** Supplies regime vector for context
* **M-RMS:** Generates risk proposals for qualified trades

## **3. Multi-Agent Architecture**

### **3.1 Core Agent Design**

Each agent follows a similar architecture with specialized variations:

class BaseTradeAgent(nn.Module):

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

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

self.config = config

# Shared embedder architecture

self.embedder = nn.Sequential(

nn.Conv1d(config['input\_features'], 64, kernel\_size=3, padding=1),

nn.BatchNorm1d(64),

nn.ReLU(),

nn.Dropout(config['dropout']),

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

nn.BatchNorm1d(128),

nn.ReLU(),

nn.Dropout(config['dropout']),

nn.Conv1d(128, 256, kernel\_size=3, padding=1),

nn.BatchNorm1d(256),

nn.ReLU()

)

# Temporal attention mechanism

self.temporal\_attention = nn.MultiheadAttention(

embed\_dim=256,

num\_heads=8,

dropout=config['dropout'],

batch\_first=True

)

# Agent-specific policy head

self.policy\_head = self.\_build\_policy\_head()

def forward(self, market\_matrix, regime\_vector, synergy\_context):

# Process market data

x = market\_matrix.transpose(1, 2) # [batch, features, time]

embedded = self.embedder(x)

embedded = embedded.transpose(1, 2) # [batch, time, features]

# Self-attention over time

attended, attention\_weights = self.temporal\_attention(

embedded, embedded, embedded

)

# Global pooling

pooled = torch.mean(attended, dim=1) # [batch, features]

# Incorporate regime and synergy context

context = torch.cat([

pooled,

regime\_vector,

self.\_encode\_synergy(synergy\_context)

], dim=-1)

# Generate decision

decision = self.policy\_head(context)

return {

'action': decision['action'],

'confidence': decision['confidence'],

'reasoning': decision['reasoning'],

'attention\_weights': attention\_weights

}

### **3.2 Specialized Agents**

#### **3.2.1 Long-term Structure Analyzer**

class StructureAnalyzer(BaseTradeAgent):

"""Focuses on market structure and major trends"""

def \_build\_policy\_head(self):

return nn.Sequential(

nn.Linear(256 + 8 + 32, 512), # embedded + regime + synergy

nn.LayerNorm(512),

nn.ReLU(),

nn.Dropout(self.config['dropout']),

nn.Linear(512, 256),

nn.LayerNorm(256),

nn.ReLU(),

nn.Dropout(self.config['dropout']),

nn.Linear(256, 128),

nn.ReLU(),

# Output branches

nn.ModuleDict({

'action': nn.Linear(128, 3), # [pass, long, short]

'confidence': nn.Linear(128, 1), # [0, 1]

'reasoning': nn.Linear(128, 64) # Interpretable features

})

)

def \_encode\_synergy(self, synergy\_context):

"""Extract structure-relevant features from synergy"""

features = []

# Trend alignment

mlmi\_strength = synergy\_context['signal\_strengths']['mlmi']

nwrqk\_slope = synergy\_context['signal\_sequence'][1]['value']

features.extend([mlmi\_strength, nwrqk\_slope])

# LVN positioning

lvn\_distance = synergy\_context['market\_context']['nearest\_lvn']['distance']

lvn\_strength = synergy\_context['market\_context']['nearest\_lvn']['strength']

features.extend([lvn\_distance / 100, lvn\_strength / 100])

# Market structure quality

structure\_score = self.\_calculate\_structure\_score(synergy\_context)

features.append(structure\_score)

return torch.tensor(features, dtype=torch.float32)

#### **3.2.2 Short-term Tactician**

class ShortTermTactician(BaseTradeAgent):

"""Focuses on immediate price action and execution timing"""

def \_build\_policy\_head(self):

return nn.Sequential(

nn.Linear(256 + 8 + 24, 384),

nn.LayerNorm(384),

nn.ReLU(),

nn.Dropout(self.config['dropout']),

nn.Linear(384, 192),

nn.LayerNorm(192),

nn.ReLU(),

nn.Dropout(self.config['dropout']),

nn.Linear(192, 96),

nn.ReLU(),

nn.ModuleDict({

'action': nn.Linear(96, 3),

'confidence': nn.Linear(96, 1),

'timing': nn.Linear(96, 5), # Immediate vs wait 1-4 bars

'reasoning': nn.Linear(96, 48)

})

)

def \_encode\_synergy(self, synergy\_context):

"""Extract execution-relevant features"""

features = []

# FVG characteristics

fvg\_age = synergy\_context['signal\_sequence'][2]['age']

fvg\_size = synergy\_context['signal\_sequence'][2]['gap\_size']

features.extend([fvg\_age / 10, fvg\_size \* 100])

# Momentum quality

price\_momentum = synergy\_context['market\_context']['price\_momentum\_5']

volume\_surge = synergy\_context['market\_context']['volume\_ratio']

features.extend([price\_momentum, np.log1p(volume\_surge)])

# Microstructure

spread = synergy\_context['market\_context']['spread']

features.append(spread / synergy\_context['market\_context']['current\_price'])

return torch.tensor(features, dtype=torch.float32)

#### **3.2.3 Mid-frequency Arbitrageur**

class MidFrequencyArbitrageur(BaseTradeAgent):

"""Bridges structure and tactics, identifies inefficiencies"""

def \_build\_policy\_head(self):

return nn.Sequential(

nn.Linear(256 + 8 + 28, 448),

nn.LayerNorm(448),

nn.ReLU(),

nn.Dropout(self.config['dropout']),

nn.Linear(448, 224),

nn.LayerNorm(224),

nn.ReLU(),

nn.Dropout(self.config['dropout']),

nn.Linear(224, 112),

nn.ReLU(),

nn.ModuleDict({

'action': nn.Linear(112, 3),

'confidence': nn.Linear(112, 1),

'inefficiency\_score': nn.Linear(112, 1), # Opportunity quality

'reasoning': nn.Linear(112, 56)

})

)

def \_encode\_synergy(self, synergy\_context):

"""Extract arbitrage-relevant features"""

features = []

# Cross-timeframe alignment

synergy\_type\_encoding = self.\_encode\_synergy\_type(

synergy\_context['synergy\_type']

)

features.extend(synergy\_type\_encoding) # One-hot encoded

# Completion time (faster = stronger signal)

bars\_to\_complete = synergy\_context['metadata']['bars\_to\_complete']

features.append(1.0 / (1.0 + bars\_to\_complete))

# Signal coherence

signal\_strengths = list(synergy\_context['signal\_strengths'].values())

coherence = np.std(signal\_strengths) # Lower = more coherent

features.append(1.0 - coherence)

return torch.tensor(features, dtype=torch.float32)

### **3.3 Agent Communication Network**

class AgentCommunicationNetwork(nn.Module):

"""Enables inter-agent communication and coordination"""

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

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

self.n\_agents = 3

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

self.n\_rounds = config['communication\_rounds']

# Message generation

self.message\_generator = nn.Linear(256, self.message\_dim)

# Message aggregation (Graph Attention Network)

self.attention\_weights = nn.Parameter(

torch.randn(self.n\_agents, self.n\_agents)

)

# Message processing

self.message\_processor = nn.GRUCell(

input\_size=self.message\_dim \* self.n\_agents,

hidden\_size=256

)

def forward(self, agent\_states):

"""

Enable agents to communicate over multiple rounds

agent\_states: List of hidden states from each agent

"""

hidden\_states = agent\_states.copy()

for round\_idx in range(self.n\_rounds):

# Generate messages

messages = [

self.message\_generator(state)

for state in hidden\_states

]

# Apply attention for message routing

attention = F.softmax(self.attention\_weights, dim=1)

# Aggregate messages for each agent

aggregated\_messages = []

for i in range(self.n\_agents):

weighted\_messages = [

attention[i, j] \* messages[j]

for j in range(self.n\_agents)

]

aggregated = torch.cat(weighted\_messages, dim=-1)

aggregated\_messages.append(aggregated)

# Update hidden states

new\_hidden\_states = []

for i, (state, msgs) in enumerate(zip(hidden\_states, aggregated\_messages)):

new\_state = self.message\_processor(msgs, state)

new\_hidden\_states.append(new\_state)

hidden\_states = new\_hidden\_states

return hidden\_states

## **4. MC Dropout Consensus Mechanism**

### **4.1 Implementation**

class MCDropoutConsensus:

"""Implements superposition decision making"""

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

self.n\_passes = config['n\_forward\_passes']

self.confidence\_threshold = config['confidence\_threshold']

def evaluate\_opportunity(self, agents, inputs):

"""

Run multiple forward passes with dropout enabled

Returns consensus decision and uncertainty metrics

"""

# Enable dropout for all agents

for agent in agents.values():

agent.train() # Enables dropout

# Collect predictions across multiple passes

all\_predictions = {

'structure\_analyzer': [],

'short\_term\_tactician': [],

'mid\_frequency\_arbitrageur': []

}

with torch.no\_grad():

for pass\_idx in range(self.n\_passes):

for agent\_name, agent in agents.items():

prediction = agent(\*\*inputs[agent\_name])

all\_predictions[agent\_name].append(prediction)

# Analyze consensus

consensus\_result = self.\_analyze\_consensus(all\_predictions)

# Switch back to eval mode

for agent in agents.values():

agent.eval()

return consensus\_result

def \_analyze\_consensus(self, all\_predictions):

"""Detailed consensus analysis"""

# Extract action probabilities for each agent

agent\_actions = {}

agent\_confidences = {}

for agent\_name, predictions in all\_predictions.items():

# Stack action logits

action\_logits = torch.stack([

p['action'] for p in predictions

])

# Convert to probabilities

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

# Calculate mean and std

mean\_probs = action\_probs.mean(dim=0)

std\_probs = action\_probs.std(dim=0)

# Extract confidences

confidences = torch.stack([

p['confidence'] for p in predictions

]).squeeze()

agent\_actions[agent\_name] = {

'mean\_probs': mean\_probs,

'std\_probs': std\_probs,

'predicted\_action': mean\_probs.argmax().item()

}

agent\_confidences[agent\_name] = {

'mean': confidences.mean().item(),

'std': confidences.std().item()

}

# Calculate overall consensus

overall\_consensus = self.\_calculate\_overall\_consensus(

agent\_actions,

agent\_confidences

)

return {

'consensus\_action': overall\_consensus['action'],

'consensus\_confidence': overall\_consensus['confidence'],

'agent\_predictions': agent\_actions,

'agent\_confidences': agent\_confidences,

'uncertainty\_metrics': self.\_calculate\_uncertainty\_metrics(all\_predictions),

'should\_trade': overall\_consensus['confidence'] >= self.confidence\_threshold

}

### **4.2 Consensus Decision Logic**

def \_calculate\_overall\_consensus(self, agent\_actions, agent\_confidences):

"""Determine final consensus action and confidence"""

# Count agent agreements

predicted\_actions = [

a['predicted\_action'] for a in agent\_actions.values()

]

# Find majority action

action\_counts = Counter(predicted\_actions)

majority\_action, count = action\_counts.most\_common(1)[0]

# Calculate agreement score

agreement\_score = count / len(predicted\_actions)

if agreement\_score < 0.67: # Less than 2/3 agree

return {

'action': 0, # Pass

'confidence': 0.0,

'reason': 'Insufficient agent agreement'

}

# Weight confidences by agent importance

agent\_weights = {

'structure\_analyzer': 0.4,

'short\_term\_tactician': 0.3,

'mid\_frequency\_arbitrageur': 0.3

}

# Calculate weighted confidence

weighted\_confidence = 0.0

uncertainty\_penalty = 0.0

for agent\_name, confidence\_data in agent\_confidences.items():

weight = agent\_weights[agent\_name]

# Only count agents that agree with majority

if agent\_actions[agent\_name]['predicted\_action'] == majority\_action:

weighted\_confidence += weight \* confidence\_data['mean']

# Penalize high uncertainty

uncertainty\_penalty += weight \* confidence\_data['std']

# Final confidence incorporates agreement and uncertainty

final\_confidence = weighted\_confidence \* agreement\_score - uncertainty\_penalty \* 0.5

return {

'action': majority\_action,

'confidence': max(0.0, min(1.0, final\_confidence)),

'agreement\_score': agreement\_score,

'uncertainty\_penalty': uncertainty\_penalty

}

## **5. Decision Flow Orchestration**

### **5.1 Complete Decision Pipeline**

class MainMARLCore:

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

self.config = config

# Initialize agents

self.agents = {

'structure\_analyzer': StructureAnalyzer(config['agents']['structure\_analyzer']),

'short\_term\_tactician': ShortTermTactician(config['agents']['short\_term\_tactician']),

'mid\_frequency\_arbitrageur': MidFrequencyArbitrageur(config['agents']['mid\_frequency\_arbitrageur'])

}

# Communication network

self.communication\_network = AgentCommunicationNetwork(config['agent\_communication'])

# MC Dropout consensus

self.consensus\_mechanism = MCDropoutConsensus(config['mc\_dropout'])

# Decision gate

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

# Auxiliary systems

self.rde = None # Set during initialization

self.m\_rms = None # Set during initialization

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

"""Main entry point - Gate 2 of the two-gate system"""

try:

# 1. Prepare agent inputs

agent\_inputs = self.\_prepare\_agent\_inputs(synergy\_event)

# 2. Get regime context

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

# 3. Initial agent predictions

initial\_states = []

for agent\_name, agent in self.agents.items():

state = agent.get\_hidden\_state(

agent\_inputs[agent\_name],

regime\_vector

)

initial\_states.append(state)

# 4. Agent communication

communicated\_states = self.communication\_network(initial\_states)

# 5. Update agent states

for i, (agent\_name, agent) in enumerate(self.agents.items()):

agent.update\_state(communicated\_states[i])

# 6. MC Dropout consensus evaluation

consensus\_result = self.consensus\_mechanism.evaluate\_opportunity(

self.agents,

agent\_inputs

)

# 7. Check if we should proceed

if not consensus\_result['should\_trade']:

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

return

# 8. Generate trade qualification

trade\_qualification = self.\_create\_trade\_qualification(

synergy\_event,

consensus\_result,

regime\_vector

)

# 9. Get risk proposal from M-RMS

risk\_proposal = self.m\_rms.generate\_risk\_proposal(trade\_qualification)

# 10. Final decision gate validation

final\_decision = self.decision\_gate.validate(

trade\_qualification,

risk\_proposal,

self.\_get\_system\_state()

)

# 11. Emit decision

if final\_decision['approved']:

self.\_emit\_trade\_decision(final\_decision)

else:

self.\_log\_final\_rejection(final\_decision)

except Exception as e:

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

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

### **5.2 Decision Gate Logic**

class DecisionGate:

"""Final validation before trade execution"""

def validate(self, qualification, risk\_proposal, system\_state):

"""Perform final checks before approving trade"""

validation\_results = {

'risk\_limits': self.\_check\_risk\_limits(risk\_proposal, system\_state),

'correlation': self.\_check\_correlation(qualification, system\_state),

'daily\_limits': self.\_check\_daily\_limits(system\_state),

'market\_conditions': self.\_check\_market\_conditions(qualification),

'technical\_validity': self.\_check\_technical\_validity(qualification)

}

# All checks must pass

all\_passed = all(validation\_results.values())

if all\_passed:

return {

'approved': True,

'execute\_trade\_command': {

'qualification': qualification,

'risk\_proposal': risk\_proposal,

'execution\_id': self.\_generate\_execution\_id(),

'timestamp': datetime.now()

}

}

else:

return {

'approved': False,

'rejection\_reasons': [

check for check, passed in validation\_results.items()

if not passed

],

'timestamp': datetime.now()

}

## **6. Output Events & Commands**

### **6.1 Primary Output**

**Event Name:** EXECUTE\_TRADE  
 **Frequency:** Only after full qualification and validation  
 **Payload:**

ExecuteTradeCommand = {

'execution\_id': str, # Unique identifier

'timestamp': datetime,

'trade\_specification': {

'symbol': str,

'direction': int, # 1 or -1

'entry\_price': float, # From market

'synergy\_type': str, # Original trigger

},

'risk\_parameters': {

'position\_size': int, # From M-RMS

'stop\_loss': float, # Price level

'take\_profit': float, # Price level

'max\_hold\_time': int, # Bars

'trailing\_rules': dict # If applicable

},

'decision\_metadata': {

'consensus\_confidence': float, # 0.65-1.0

'agent\_agreements': dict, # Individual decisions

'mc\_dropout\_metrics': dict, # Uncertainty data

'regime\_context': list, # 8-dim vector

'processing\_time\_ms': float

},

'tracking\_data': {

'expected\_value': float, # From M-RMS

'risk\_reward\_ratio': float,

'correlation\_score': float,

'daily\_trade\_number': int

}

}

### **6.2 Rejection Events**

# Logged internally, not emitted

TRADE\_REJECTED = {

'timestamp': datetime,

'synergy\_event': dict, # Original opportunity

'rejection\_stage': str, # 'consensus', 'risk', 'gate'

'rejection\_reasons': list,

'consensus\_metrics': dict, # If applicable

'system\_state': dict

}

## **7. Performance Monitoring**

### **7.1 Real-time Metrics**

class PerformanceMonitor:

def track\_decision(self, synergy\_event, decision\_result):

"""Track every decision for analysis"""

metrics = {

'timestamp': datetime.now(),

'synergy\_type': synergy\_event['synergy\_type'],

# Consensus metrics

'consensus\_achieved': decision\_result.get('should\_trade', False),

'consensus\_confidence': decision\_result.get('consensus\_confidence', 0.0),

'agent\_agreement': decision\_result.get('agreement\_score', 0.0),

# Uncertainty metrics

'prediction\_uncertainty': decision\_result['uncertainty\_metrics']['mean\_std'],

'mc\_dropout\_variance': decision\_result['uncertainty\_metrics']['variance'],

# Processing metrics

'decision\_time\_ms': decision\_result.get('processing\_time', 0),

'communication\_rounds': 3, # Fixed in config

# Outcome (will be updated post-trade)

'trade\_executed': False,

'trade\_result': None

}

self.decision\_history.append(metrics)

### **7.2 Agent Performance Tracking**

# Per-agent metrics tracked

{

'structure\_analyzer': {

'accuracy': 0.68, # When agreed with profit

'confidence\_calibration': 0.85, # Confidence vs actual

'contribution\_score': 0.72 # Impact on profits

},

'short\_term\_tactician': {

'accuracy': 0.71,

'timing\_precision': 0.83, # Entry timing quality

'contribution\_score': 0.69

},

'mid\_frequency\_arbitrageur': {

'accuracy': 0.66,

'inefficiency\_detection': 0.78,

'contribution\_score': 0.65

}

}

## **8. Critical Requirements**

### **8.1 Consensus Requirements**

* **Minimum Agreement:** 2/3 agents must agree on direction
* **Confidence Threshold:** 0.65 minimum consensus confidence
* **MC Dropout Passes:** Exactly 50 for statistical validity

### **8.2 Performance Requirements**

* **Decision Latency:** <100ms from synergy to decision
* **Memory Usage:** <2GB including all models
* **GPU Usage:** Optional but recommended for speed

### **8.3 Reliability Requirements**

* **Failsafe Mode:** Reject all trades on system error
* **State Consistency:** Atomic decision process
* **Audit Trail:** Every decision fully logged

## **9. Integration Points**

### **9.1 Upstream Integration**

* **SynergyDetector:** Provides qualified opportunities
* **MatrixAssemblers:** Supply agent-specific inputs
* **RDE:** Provides regime context
* **M-RMS:** Generates risk proposals

### **9.2 Downstream Integration**

* **ExecutionHandler:** Receives EXECUTE\_TRADE commands
* **System State:** Updates position tracking
* **Performance Analytics:** Feeds decision data

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

* Does NOT identify trading opportunities (SynergyDetector's job)
* Does NOT calculate technical indicators
* Does NOT determine position sizes (M-RMS's job)
* Does NOT execute trades (ExecutionHandler's job)
* Does NOT manage existing positions
* Does NOT handle market data
* Does NOT set risk parameters independently

This completes the Main MARL Core PRD. This is the beating heart of our intelligent trading system - where specialized agents collaborate, communicate, and reach consensus through sophisticated uncertainty quantification.

The MC Dropout mechanism transforms traditional single-point decisions into probability distributions, ensuring we only act on high-conviction opportunities. The multi-agent architecture allows for specialized expertise while maintaining overall coherence through the communication network.

We're nearing the end of our component specifications. Ready for the final pieces?