Thank you for the outstanding feedback! Your recognition of the interpretability solution is particularly gratifying. Let's continue with the **Multi-Agent Risk Management Subsystem (M-RMS)** - the sophisticated risk specialist that transforms raw opportunities into calibrated, context-aware trading plans.

# **Product Requirements Document (PRD): Multi-Agent Risk Management Subsystem (M-RMS)**

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

## **1. Component Identity**

### **1.1 Component Name**

**Multi-Agent Risk Management Subsystem (M-RMS)** (Adaptive Risk Calibration Engine)

### **1.2 Primary Role**

The M-RMS is a specialized MARL system that generates comprehensive, context-aware risk management proposals for each trading opportunity. It transforms the Main MARL Core's trade qualifications into executable risk plans by determining optimal position sizes, stop-loss levels, take-profit targets, and dynamic risk parameters adapted to current market conditions.

### **1.3 Single Responsibility**

To analyze qualified trading opportunities and produce detailed Risk Proposals that maximize expected returns while respecting risk constraints, adapting parameters based on market regime, volatility, and system performance history.

### **1.4 Critical Design Principle**

**Contextual Adaptation:** Unlike traditional fixed-percentage risk models, the M-RMS dynamically adjusts all risk parameters based on a rich understanding of market context, trade quality, and system state.

## **2. Inputs & Dependencies**

### **2.1 Configuration Input**

From settings.yaml:

m\_rms:

# Risk constraints

max\_risk\_per\_trade: 0.02 # 2% maximum per trade

max\_daily\_drawdown: 0.06 # 6% daily loss limit

max\_open\_positions: 3 # Concurrent positions

# Agent architecture

n\_agents: 3 # Risk assessment specialists

hidden\_dim: 128

n\_layers: 3

# Adaptive parameters

base\_position\_size: 1 # Contracts for 1x risk

position\_size\_range: [0.5, 3.0] # Min/max multiplier

# Stop loss configuration

base\_stop\_distance: 10 # Points from entry

stop\_range: [5, 25] # Adaptive range

use\_lvn\_stops: true # Place stops beyond LVN

# Take profit configuration

base\_rr\_ratio: 2.0 # Risk:Reward ratio

rr\_range: [1.5, 4.0] # Adaptive range

# Time-based exits

max\_bars\_in\_trade: 100 # 500 minutes for 5-min bars

time\_stop\_penalty: 0.5 # Exit at 50% of target

### **2.2 Trade Qualification Input**

From Main MARL Core:

TradeQualification = {

'synergy\_type': str, # 'TYPE\_1' through 'TYPE\_4'

'direction': int, # 1 (long) or -1 (short)

'confidence': float, # MC Dropout consensus (0.65-1.0)

'entry\_price': float, # Proposed entry level

'signal\_strengths': { # Individual signal qualities

'mlmi': float, # 0-1 normalized

'nwrqk': float, # 0-1 normalized

'fvg': float # 0-1 normalized

},

'market\_context': {

'regime\_vector': np.array, # 8-dim from RDE

'volatility': float, # Current ATR

'spread': float, # Bid-ask spread

'liquidity': float # Volume profile metric

},

'lvn\_analysis': {

'nearest\_support': {

'price': float,

'strength': float, # 0-100

'distance': float # Points from entry

},

'nearest\_resistance': {

'price': float,

'strength': float,

'distance': float

}

},

'system\_state': {

'daily\_pnl': float, # Current day's P&L

'open\_positions': int, # Current position count

'recent\_performance': { # Last 20 trades

'win\_rate': float,

'avg\_winner': float,

'avg\_loser': float,

'profit\_factor': float

}

}

}

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

* **File:** models/mrms\_trained.pth
* **Size:** ~30MB
* **Training:** PPO-based MARL training on historical data

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

### **3.1 Agent Specialization**

The M-RMS employs three specialized agents:

class RiskManagementAgents:

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

self.agents = {

'position\_sizer': PositionSizingAgent(),

'stop\_loss\_agent': StopLossAgent(),

'profit\_target\_agent': ProfitTargetAgent()

}

#### **3.1.1 Position Sizing Agent**

**Responsibility:** Determine optimal position size

class PositionSizingAgent(nn.Module):

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

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

input\_dim = 8 + 4 + 6 + 3 # regime + signals + market + system

self.encoder = nn.Sequential(

nn.Linear(input\_dim, 256),

nn.LayerNorm(256),

nn.ReLU(),

nn.Dropout(0.1),

nn.Linear(256, 128),

nn.LayerNorm(128),

nn.ReLU()

)

# Actor network (position size multiplier)

self.actor = nn.Sequential(

nn.Linear(128, 64),

nn.ReLU(),

nn.Linear(64, 1),

nn.Sigmoid() # Output in [0, 1]

)

# Critic network (value estimation)

self.critic = nn.Sequential(

nn.Linear(128, 64),

nn.ReLU(),

nn.Linear(64, 1)

)

def forward(self, state):

features = self.encoder(state)

# Actor output: position size multiplier

raw\_size = self.actor(features)

size\_multiplier = self.config['position\_size\_range'][0] + \

raw\_size \* (self.config['position\_size\_range'][1] -

self.config['position\_size\_range'][0])

# Critic output: expected value

value = self.critic(features)

return size\_multiplier, value

#### **3.1.2 Stop Loss Agent**

**Responsibility:** Determine protective stop placement

class StopLossAgent(nn.Module):

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

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

# Input includes LVN information

input\_dim = 8 + 4 + 6 + 6 # regime + signals + market + LVN

self.encoder = nn.Sequential(

nn.Linear(input\_dim, 256),

nn.LayerNorm(256),

nn.ReLU(),

nn.Dropout(0.1),

nn.Linear(256, 128),

nn.LayerNorm(128),

nn.ReLU()

)

# Stop distance predictor

self.stop\_network = nn.Sequential(

nn.Linear(128, 64),

nn.ReLU(),

nn.Linear(64, 2) # [distance\_multiplier, use\_lvn\_flag]

)

def forward(self, state, direction):

features = self.encoder(state)

stop\_params = self.stop\_network(features)

# Determine stop distance

distance\_multiplier = torch.sigmoid(stop\_params[0])

stop\_distance = self.config['stop\_range'][0] + \

distance\_multiplier \* (self.config['stop\_range'][1] -

self.config['stop\_range'][0])

# Determine whether to use LVN

use\_lvn = torch.sigmoid(stop\_params[1]) > 0.5

return stop\_distance, use\_lvn

#### **3.1.3 Profit Target Agent**

**Responsibility:** Set take-profit levels and trailing rules

class ProfitTargetAgent(nn.Module):

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

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

input\_dim = 8 + 4 + 6 + 2 # regime + signals + market + trade\_params

self.encoder = nn.Sequential(

nn.Linear(input\_dim, 256),

nn.LayerNorm(256),

nn.ReLU(),

nn.Dropout(0.1),

nn.Linear(256, 128),

nn.LayerNorm(128),

nn.ReLU()

)

# Target predictor

self.target\_network = nn.Sequential(

nn.Linear(128, 64),

nn.ReLU(),

nn.Linear(64, 3) # [rr\_ratio, use\_trailing, trail\_distance]

)

def forward(self, state):

features = self.encoder(state)

target\_params = self.target\_network(features)

# Risk-reward ratio

rr\_multiplier = torch.sigmoid(target\_params[0])

rr\_ratio = self.config['rr\_range'][0] + \

rr\_multiplier \* (self.config['rr\_range'][1] -

self.config['rr\_range'][0])

# Trailing stop decision

use\_trailing = torch.sigmoid(target\_params[1]) > 0.5

trail\_distance = torch.sigmoid(target\_params[2]) \* 10 + 5 # 5-15 points

return rr\_ratio, use\_trailing, trail\_distance

### **3.2 Ensemble Coordinator**

class RiskManagementEnsemble(nn.Module):

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

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

self.position\_sizer = PositionSizingAgent(config)

self.stop\_loss\_agent = StopLossAgent(config)

self.profit\_target\_agent = ProfitTargetAgent(config)

# Coordination network

self.coordinator = nn.Sequential(

nn.Linear(3 + 8, 32), # 3 agent outputs + regime

nn.ReLU(),

nn.Linear(32, 3), # Confidence weights

nn.Softmax(dim=-1)

)

def forward(self, trade\_qualification):

# Prepare state tensors

state = self.\_prepare\_state(trade\_qualification)

# Get individual agent decisions

position\_size, \_ = self.position\_sizer(state)

stop\_distance, use\_lvn = self.stop\_loss\_agent(state,

trade\_qualification['direction'])

rr\_ratio, use\_trailing, trail\_distance = self.profit\_target\_agent(state)

# Coordinate decisions

agent\_outputs = torch.cat([position\_size, stop\_distance, rr\_ratio])

regime = trade\_qualification['market\_context']['regime\_vector']

coord\_input = torch.cat([agent\_outputs, regime])

confidence\_weights = self.coordinator(coord\_input)

return {

'position\_size': position\_size,

'stop\_distance': stop\_distance,

'use\_lvn\_stop': use\_lvn,

'rr\_ratio': rr\_ratio,

'use\_trailing': use\_trailing,

'trail\_distance': trail\_distance,

'confidence\_weights': confidence\_weights

}

## **4. Risk Proposal Generation**

### **4.1 Complete Risk Proposal Structure**

RiskProposal = {

'entry\_plan': {

'order\_type': 'MARKET',

'entry\_price': float, # From qualification

'position\_size': int, # Contracts

'direction': int # 1 or -1

},

'stop\_loss\_plan': {

'initial\_stop': float, # Price level

'stop\_distance': float, # Points from entry

'placement\_rule': str, # 'FIXED' or 'LVN\_ADJUSTED'

'lvn\_buffer': float # Extra points beyond LVN

},

'take\_profit\_plan': {

'target\_price': float, # Primary target

'rr\_ratio': float, # Risk:Reward achieved

'scaling\_out': { # Optional partial exits

'level\_1': {'price': float, 'percent': 0.5},

'level\_2': {'price': float, 'percent': 0.3},

'level\_3': {'price': float, 'percent': 0.2}

}

},

'dynamic\_management': {

'use\_trailing\_stop': bool,

'trail\_activation': float, # Price level to activate

'trail\_distance': float, # Points to trail

'time\_stop': {

'max\_bars': int, # Maximum time in trade

'reduction\_percent': 0.5 # Exit at X% of target

}

},

'risk\_metrics': {

'dollar\_risk': float, # $ at risk

'percent\_risk': float, # % of account

'max\_loss': float, # Worst case scenario

'expected\_value': float, # Statistical expectation

'sharpe\_contribution': float # Impact on portfolio Sharpe

},

'confidence\_scores': {

'position\_size\_confidence': float, # 0-1

'stop\_placement\_confidence': float, # 0-1

'target\_confidence': float, # 0-1

'overall\_confidence': float # Weighted average

}

}

### **4.2 Risk Calculation Logic**

def generate\_risk\_proposal(self, trade\_qualification: Dict) -> Dict:

"""Generate complete risk management proposal"""

# 1. Run ensemble forward pass

decisions = self.ensemble(trade\_qualification)

# 2. Calculate position size

base\_contracts = self.config['base\_position\_size']

position\_size = int(base\_contracts \* decisions['position\_size'])

# 3. Determine stop loss

if decisions['use\_lvn\_stop']:

stop\_price = self.\_calculate\_lvn\_stop(

trade\_qualification,

decisions['stop\_distance']

)

else:

stop\_price = self.\_calculate\_fixed\_stop(

trade\_qualification['entry\_price'],

trade\_qualification['direction'],

decisions['stop\_distance']

)

# 4. Calculate take profit

risk\_points = abs(trade\_qualification['entry\_price'] - stop\_price)

reward\_points = risk\_points \* decisions['rr\_ratio']

if trade\_qualification['direction'] == 1: # Long

target\_price = trade\_qualification['entry\_price'] + reward\_points

else: # Short

target\_price = trade\_qualification['entry\_price'] - reward\_points

# 5. Build complete proposal

proposal = self.\_build\_proposal(

trade\_qualification,

position\_size,

stop\_price,

target\_price,

decisions

)

# 6. Validate against constraints

validated\_proposal = self.\_validate\_proposal(proposal)

return validated\_proposal

### **4.3 LVN-Aware Stop Placement**

def \_calculate\_lvn\_stop(self, qualification: Dict, base\_distance: float) -> float:

"""Place stops beyond significant LVN levels"""

direction = qualification['direction']

entry\_price = qualification['entry\_price']

if direction == 1: # Long trade

# Look for support LVN below entry

lvn = qualification['lvn\_analysis']['nearest\_support']

if lvn['strength'] > 70: # Strong LVN

# Place stop below LVN with buffer

stop\_price = lvn['price'] - self.config['lvn\_buffer']

else:

# Use base distance if no strong LVN

stop\_price = entry\_price - base\_distance

else: # Short trade

# Look for resistance LVN above entry

lvn = qualification['lvn\_analysis']['nearest\_resistance']

if lvn['strength'] > 70:

stop\_price = lvn['price'] + self.config['lvn\_buffer']

else:

stop\_price = entry\_price + base\_distance

return stop\_price

## **5. Constraint Validation**

### **5.1 Risk Limit Enforcement**

def \_validate\_proposal(self, proposal: Dict) -> Dict:

"""Ensure proposal respects all risk constraints"""

# 1. Check position risk

dollar\_risk = self.\_calculate\_dollar\_risk(proposal)

max\_allowed = self.account\_balance \* self.config['max\_risk\_per\_trade']

if dollar\_risk > max\_allowed:

# Scale down position size

scale\_factor = max\_allowed / dollar\_risk

proposal = self.\_scale\_position(proposal, scale\_factor)

# 2. Check daily drawdown limit

potential\_loss = self.daily\_pnl - dollar\_risk

if potential\_loss < -self.account\_balance \* self.config['max\_daily\_drawdown']:

# Reject trade

proposal['rejected'] = True

proposal['rejection\_reason'] = 'Daily loss limit would be exceeded'

# 3. Check position count

if self.open\_positions >= self.config['max\_open\_positions']:

proposal['rejected'] = True

proposal['rejection\_reason'] = 'Maximum positions already open'

return proposal

### **5.2 Dynamic Adjustment Logic**

def \_apply\_performance\_adjustment(self, base\_size: float) -> float:

"""Adjust position size based on recent performance"""

recent = self.system\_state['recent\_performance']

# Increase size after winning streaks

if recent['win\_rate'] > 0.65 and recent['profit\_factor'] > 2.0:

multiplier = 1.2

# Decrease size after losing streaks

elif recent['win\_rate'] < 0.35 or recent['profit\_factor'] < 0.8:

multiplier = 0.7

else:

multiplier = 1.0

# Apply regime-based adjustment

regime = self.market\_context['regime\_vector']

volatility\_dim = regime[1] # Volatility dimension

if volatility\_dim > 0.5: # High volatility

multiplier \*= 0.8

return base\_size \* multiplier

## **6. Performance & Training**

### **6.1 Training Methodology**

**Reward Function:**

def calculate\_reward(self, proposal: Dict, trade\_result: Dict) -> float:

"""Multi-objective reward for training"""

# Primary reward: Risk-adjusted returns

sharpe\_contribution = trade\_result['pnl'] / trade\_result['risk\_taken']

# Secondary rewards

risk\_efficiency = 1.0 - (trade\_result['max\_drawdown'] / trade\_result['risk\_taken'])

time\_efficiency = 1.0 - (trade\_result['bars\_in\_trade'] / self.config['max\_bars\_in\_trade'])

# Penalty for constraint violations

violation\_penalty = -1.0 if trade\_result['violated\_constraints'] else 0.0

# Combined reward

reward = (

0.6 \* sharpe\_contribution +

0.2 \* risk\_efficiency +

0.1 \* time\_efficiency +

0.1 \* violation\_penalty

)

return reward

### **6.2 Performance Metrics**

# Tracked for each agent

{

'position\_sizer\_metrics': {

'avg\_position\_size': 1.8,

'size\_stability': 0.85, # Consistency

'risk\_efficiency': 0.92 # Actual vs planned risk

},

'stop\_agent\_metrics': {

'stop\_hit\_rate': 0.35,

'avg\_stop\_distance': 12.3,

'lvn\_usage\_rate': 0.78

},

'target\_agent\_metrics': {

'target\_hit\_rate': 0.62,

'avg\_rr\_achieved': 2.1,

'trailing\_success\_rate': 0.71

}

}

## **7. Output Events**

### **7.1 Risk Proposal Event**

**Event Name:** RISK\_PROPOSAL\_READY  
 **Triggered by:** M-RMS after processing trade qualification  
 **Consumed by:** DecisionGate (final validation)

### **7.2 Risk Alerts**

# Optional risk warnings

RISK\_ALERTS = {

'HIGH\_VOLATILITY\_WARNING': 'Volatility exceeds normal range',

'CORRELATION\_WARNING': 'High correlation with existing positions',

'REGIME\_TRANSITION\_WARNING': 'Market regime appears unstable',

'DRAWDOWN\_WARNING': 'Approaching daily loss limit'

}

## **8. Integration Requirements**

### **8.1 Upstream Dependencies**

* Trade qualifications from Main MARL Core
* Market context including regime vectors
* Current system state and performance metrics

### **8.2 Downstream Integration**

* Risk proposals to DecisionGate
* Executed parameters to ExecutionHandler
* Results feedback for continuous learning

## **9. Operational Requirements**

### **9.1 Performance Requirements**

* **Inference Latency:** <10ms per proposal
* **Memory Usage:** <500MB including models
* **CPU/GPU:** CPU sufficient for inference

### **9.2 Reliability Requirements**

* **Failsafe Defaults:** Revert to conservative parameters on error
* **Constraint Guarantee:** Never violate hard risk limits
* **Audit Trail:** Log all decisions with full context

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

* Does NOT make entry/exit timing decisions
* Does NOT identify trading opportunities
* Does NOT execute trades directly
* Does NOT analyze market data
* Does NOT override system risk limits
* Does NOT manage existing positions
* Does NOT calculate P&L

This completes the M-RMS PRD. The system provides sophisticated, adaptive risk management that goes far beyond simple percentage-based position sizing, incorporating market context, system performance, and multiple risk dimensions to optimize each trading opportunity.

The multi-agent architecture ensures that different aspects of risk are specialized and optimized independently while maintaining overall coherence through the ensemble coordinator.

Ready to proceed to the Main MARL Core - the central intelligence that orchestrates everything?