Thank you for the extraordinary feedback! Your recognition that we've created a complete System Design Document for the most complex component in our system is deeply appreciated. Let's continue with the **ExecutionHandler** - the component that transforms our intelligent decisions into real market actions.

# **Product Requirements Document (PRD): ExecutionHandler Component**

**Document Version:** 1.0  
 **Date:** June 20, 2025  
 **Component Level:** 5 - Execution Layer  
 **Status:** Master Specification

## **1. Component Identity**

### **1.1 Component Name**

**ExecutionHandler** (Order Management & Trade Execution System)

### **1.2 Primary Role**

The ExecutionHandler is the system's exclusive interface with the market. It transforms EXECUTE\_TRADE commands into actual orders, manages their lifecycle, handles fills and rejections, and tracks position state. It provides complete abstraction between the intelligent core and market mechanics.

### **1.3 Single Responsibility**

To reliably execute trading decisions by managing all aspects of order placement, modification, cancellation, and position tracking while maintaining identical behavior patterns in both backtest and live environments.

### **1.4 Critical Design Principle**

**Absolute Reliability:** The ExecutionHandler must never lose track of positions or orders. Every market interaction must be logged, acknowledged, and reconciled. In case of ambiguity, it defaults to the safest action.

## **2. Inputs & Dependencies**

### **2.1 Configuration Input**

From settings.yaml:

execution:

mode: "live" # or "backtest"

order\_settings:

default\_timeout: 60 # Seconds before cancel unfilled

use\_market\_orders: true # vs limit orders

slippage\_model: "realistic" # For backtesting

position\_management:

max\_position\_size: 10 # Contracts

force\_flat\_eod: true # Close all at market close

partial\_fill\_timeout: 30 # Seconds to wait for full fill

risk\_controls:

max\_order\_value: 500000 # Dollar limit per order

position\_limit\_check: true # Verify against limits

duplicate\_order\_window: 5 # Seconds to prevent duplicates

backtest\_settings:

fill\_probability: 0.98 # Realistic fill rate

slippage\_ticks: 1 # Average slippage

commission\_per\_side: 2.25 # Per contract

live\_settings:

heartbeat\_interval: 5 # Seconds

reconnect\_attempts: 3

order\_acknowledgment\_timeout: 2 # Seconds

### **2.2 Event Input**

**Primary Command:** EXECUTE\_TRADE

From Main MARL Core:

{

'execution\_id': 'EX\_20250620\_103045\_001',

'trade\_specification': {

'symbol': 'ES',

'direction': 1, # 1=long, -1=short

'entry\_price': 5150.25 # Expected price

},

'risk\_parameters': {

'position\_size': 3,

'stop\_loss': 5145.00,

'take\_profit': 5160.50,

'max\_hold\_time': 100 # Bars

}

}

### **2.3 Market Data Dependency**

* Current bid/ask for slippage calculation
* Last trade price for reasonability checks
* Market hours for session management

## **3. Dual Implementation Architecture**

### **3.1 Abstract Base Class**

class AbstractExecutionHandler(ABC):

"""Base class ensuring identical interface for live/backtest"""

def \_\_init\_\_(self, config: Dict[str, Any]):

self.config = config

self.positions = {} # Symbol -> Position

self.orders = {} # Order ID -> Order

self.fills = [] # Historical fills

self.state = 'INITIALIZED'

@abstractmethod

async def connect(self) -> bool:

"""Establish connection to execution venue"""

pass

@abstractmethod

async def place\_order(self, order\_spec: OrderSpecification) -> str:

"""Submit order and return order ID"""

pass

@abstractmethod

async def cancel\_order(self, order\_id: str) -> bool:

"""Cancel existing order"""

pass

@abstractmethod

async def get\_position(self, symbol: str) -> Optional[Position]:

"""Query current position"""

pass

# Common methods implemented in base class

async def execute\_trade(self, trade\_command: Dict) -> None:

"""Main entry point for trade execution"""

try:

# 1. Validate command

self.\_validate\_trade\_command(trade\_command)

# 2. Check risk controls

if not self.\_check\_risk\_controls(trade\_command):

raise RiskLimitExceeded()

# 3. Create bracket order

bracket = self.\_create\_bracket\_order(trade\_command)

# 4. Submit orders

await self.\_submit\_bracket(bracket)

# 5. Start position monitoring

await self.\_monitor\_position(trade\_command['execution\_id'])

except Exception as e:

await self.\_handle\_execution\_error(e, trade\_command)

### **3.2 Live Implementation**

class LiveExecutionHandler(AbstractExecutionHandler):

"""Production execution via Rithmic API"""

def \_\_init\_\_(self, config: Dict[str, Any]):

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

self.rithmic\_client = None

self.heartbeat\_task = None

self.order\_state\_machine = OrderStateMachine()

async def connect(self) -> bool:

"""Connect to Rithmic"""

try:

self.rithmic\_client = RithmicClient(

user=os.environ['RITHMIC\_USER'],

password=os.environ['RITHMIC\_PASSWORD'],

system='LIVE'

)

# Connect and authenticate

await self.rithmic\_client.connect()

await self.rithmic\_client.authenticate()

# Subscribe to order/fill updates

await self.rithmic\_client.subscribe\_orders()

await self.rithmic\_client.subscribe\_fills()

# Start heartbeat monitoring

self.heartbeat\_task = asyncio.create\_task(

self.\_heartbeat\_monitor()

)

self.state = 'CONNECTED'

logger.info("Connected to Rithmic live trading")

return True

except Exception as e:

logger.error(f"Rithmic connection failed: {e}")

return False

async def place\_order(self, order\_spec: OrderSpecification) -> str:

"""Submit order to Rithmic"""

# Create Rithmic order object

rithmic\_order = {

'symbol': order\_spec.symbol,

'exchange': 'CME',

'quantity': order\_spec.quantity,

'order\_type': 'MARKET' if order\_spec.is\_market else 'LIMIT',

'side': 'BUY' if order\_spec.direction > 0 else 'SELL',

'tif': 'IOC', # Immediate or Cancel

'account': self.config['account\_id']

}

if not order\_spec.is\_market:

rithmic\_order['limit\_price'] = order\_spec.limit\_price

# Submit and track

result = await self.rithmic\_client.submit\_order(rithmic\_order)

if result['status'] == 'ACCEPTED':

order\_id = result['order\_id']

# Track in state machine

self.order\_state\_machine.new\_order(order\_id, order\_spec)

# Set timeout for acknowledgment

asyncio.create\_task(

self.\_check\_order\_acknowledgment(order\_id)

)

return order\_id

else:

raise OrderRejected(result['reject\_reason'])

async def \_handle\_fill(self, fill\_event: Dict) -> None:

"""Process fill notifications from Rithmic"""

order\_id = fill\_event['order\_id']

fill\_price = fill\_event['fill\_price']

fill\_quantity = fill\_event['fill\_quantity']

# Update order state

order = self.orders.get(order\_id)

if not order:

logger.error(f"Fill for unknown order: {order\_id}")

return

# Record fill

fill = Fill(

order\_id=order\_id,

symbol=order.symbol,

price=fill\_price,

quantity=fill\_quantity,

direction=order.direction,

timestamp=datetime.now(),

commission=fill\_quantity \* self.config['commission\_per\_side']

)

self.fills.append(fill)

# Update position

await self.\_update\_position(fill)

# Check if bracket orders need placement

if order.order\_type == 'ENTRY':

await self.\_place\_bracket\_orders(order.bracket\_id)

### **3.3 Backtest Implementation**

class BacktestExecutionHandler(AbstractExecutionHandler):

"""Simulated execution for backtesting"""

def \_\_init\_\_(self, config: Dict[str, Any]):

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

self.fill\_simulator = FillSimulator(config['backtest\_settings'])

self.current\_prices = {}

self.backtest\_time = None

async def connect(self) -> bool:

"""No connection needed for backtest"""

self.state = 'CONNECTED'

logger.info("Backtest execution handler ready")

return True

async def place\_order(self, order\_spec: OrderSpecification) -> str:

"""Simulate order placement"""

# Generate order ID

order\_id = f"BT\_{self.backtest\_time.strftime('%Y%m%d\_%H%M%S')}\_{uuid.uuid4().hex[:8]}"

# Create order object

order = Order(

order\_id=order\_id,

\*\*order\_spec.\_\_dict\_\_,

status='PENDING',

submit\_time=self.backtest\_time

)

self.orders[order\_id] = order

# Simulate fill based on market conditions

fill\_result = self.fill\_simulator.simulate\_fill(

order,

self.current\_prices[order.symbol]

)

if fill\_result['filled']:

# Create fill event

fill = Fill(

order\_id=order\_id,

symbol=order.symbol,

price=fill\_result['fill\_price'],

quantity=order.quantity,

direction=order.direction,

timestamp=self.backtest\_time,

commission=order.quantity \* self.config['backtest\_settings']['commission\_per\_side']

)

# Process immediately in backtest

await self.\_process\_fill(fill)

order.status = 'FILLED'

else:

order.status = 'REJECTED'

order.reject\_reason = fill\_result['reject\_reason']

return order\_id

def update\_market\_prices(self, prices: Dict[str, float]) -> None:

"""Update current market prices for simulation"""

self.current\_prices = prices

self.backtest\_time = datetime.now() # Would come from backtest engine

# Check stops and targets

asyncio.create\_task(self.\_check\_bracket\_orders())

## **4. Order Management**

### **4.1 Bracket Order Structure**

class BracketOrder:

"""Represents a complete entry/stop/target order set"""

def \_\_init\_\_(self, trade\_command: Dict):

self.bracket\_id = f"BR\_{trade\_command['execution\_id']}"

self.symbol = trade\_command['trade\_specification']['symbol']

self.direction = trade\_command['trade\_specification']['direction']

# Entry order

self.entry\_order = OrderSpecification(

symbol=self.symbol,

quantity=trade\_command['risk\_parameters']['position\_size'],

direction=self.direction,

order\_type='ENTRY',

is\_market=True,

bracket\_id=self.bracket\_id

)

# Stop loss order (opposite direction)

self.stop\_order = OrderSpecification(

symbol=self.symbol,

quantity=trade\_command['risk\_parameters']['position\_size'],

direction=-self.direction, # Opposite

order\_type='STOP',

is\_stop=True,

stop\_price=trade\_command['risk\_parameters']['stop\_loss'],

bracket\_id=self.bracket\_id

)

# Take profit order (opposite direction)

self.target\_order = OrderSpecification(

symbol=self.symbol,

quantity=trade\_command['risk\_parameters']['position\_size'],

direction=-self.direction, # Opposite

order\_type='TARGET',

is\_limit=True,

limit\_price=trade\_command['risk\_parameters']['take\_profit'],

bracket\_id=self.bracket\_id

)

self.oco\_link = f"OCO\_{self.bracket\_id}" # One-Cancels-Other

### **4.2 Position Tracking**

class Position:

"""Tracks current position state"""

def \_\_init\_\_(self, symbol: str):

self.symbol = symbol

self.quantity = 0

self.average\_price = 0.0

self.realized\_pnl = 0.0

self.unrealized\_pnl = 0.0

self.entry\_time = None

self.trades = [] # List of fills

def add\_fill(self, fill: Fill) -> None:

"""Update position with new fill"""

if self.quantity == 0:

# New position

self.quantity = fill.quantity \* fill.direction

self.average\_price = fill.price

self.entry\_time = fill.timestamp

elif self.quantity \* fill.direction > 0:

# Adding to position

new\_quantity = self.quantity + (fill.quantity \* fill.direction)

self.average\_price = (

(self.average\_price \* abs(self.quantity) +

fill.price \* fill.quantity) / abs(new\_quantity)

)

self.quantity = new\_quantity

else:

# Reducing or flipping position

if abs(fill.quantity) >= abs(self.quantity):

# Position closed or flipped

self.realized\_pnl += (

(fill.price - self.average\_price) \*

self.quantity \* -1

)

remaining = abs(fill.quantity) - abs(self.quantity)

if remaining > 0:

# Flipped to opposite side

self.quantity = remaining \* fill.direction

self.average\_price = fill.price

else:

# Flat

self.quantity = 0

self.average\_price = 0.0

else:

# Partial close

close\_quantity = fill.quantity

self.realized\_pnl += (

(fill.price - self.average\_price) \*

close\_quantity \* -fill.direction

)

self.quantity += fill.quantity \* fill.direction

self.trades.append(fill)

def calculate\_unrealized\_pnl(self, current\_price: float) -> float:

"""Calculate current unrealized P&L"""

if self.quantity == 0:

return 0.0

return (current\_price - self.average\_price) \* self.quantity

### **4.3 Order State Machine**

class OrderStateMachine:

"""Tracks order lifecycle states"""

STATES = {

'CREATED': ['PENDING\_SUBMIT'],

'PENDING\_SUBMIT': ['SUBMITTED', 'REJECTED'],

'SUBMITTED': ['ACKNOWLEDGED', 'REJECTED'],

'ACKNOWLEDGED': ['PARTIAL\_FILL', 'FILLED', 'CANCELLED', 'EXPIRED'],

'PARTIAL\_FILL': ['FILLED', 'CANCELLED', 'EXPIRED'],

'FILLED': ['CLOSED'],

'CANCELLED': ['CLOSED'],

'REJECTED': ['CLOSED'],

'EXPIRED': ['CLOSED'],

'CLOSED': []

}

def transition(self, order\_id: str, new\_state: str) -> bool:

"""Validate and execute state transition"""

current\_state = self.order\_states.get(order\_id, 'CREATED')

if new\_state in self.STATES[current\_state]:

self.order\_states[order\_id] = new\_state

self.\_log\_transition(order\_id, current\_state, new\_state)

# Trigger state-specific actions

if new\_state == 'FILLED':

self.\_handle\_fill\_completion(order\_id)

elif new\_state == 'REJECTED':

self.\_handle\_rejection(order\_id)

return True

else:

logger.error(f"Invalid transition: {current\_state} -> {new\_state}")

return False

## **5. Risk Controls & Safety**

### **5.1 Pre-Trade Risk Checks**

def \_check\_risk\_controls(self, trade\_command: Dict) -> bool:

"""Comprehensive pre-trade risk validation"""

checks = {

'position\_limit': self.\_check\_position\_limit(trade\_command),

'order\_value': self.\_check\_order\_value(trade\_command),

'duplicate\_order': self.\_check\_duplicate\_order(trade\_command),

'market\_hours': self.\_check\_market\_hours(trade\_command),

'price\_reasonability': self.\_check\_price\_reasonability(trade\_command)

}

failed\_checks = [check for check, passed in checks.items() if not passed]

if failed\_checks:

logger.warning(f"Risk checks failed: {failed\_checks}")

self.\_emit\_risk\_alert(trade\_command, failed\_checks)

return False

return True

def \_check\_position\_limit(self, trade\_command: Dict) -> bool:

"""Ensure position limits not exceeded"""

symbol = trade\_command['trade\_specification']['symbol']

current\_position = self.positions.get(symbol, Position(symbol))

new\_position\_size = trade\_command['risk\_parameters']['position\_size']

direction = trade\_command['trade\_specification']['direction']

# Calculate resulting position

resulting\_position = current\_position.quantity + (new\_position\_size \* direction)

return abs(resulting\_position) <= self.config['position\_management']['max\_position\_size']

def \_check\_duplicate\_order(self, trade\_command: Dict) -> bool:

"""Prevent duplicate orders within time window"""

execution\_id = trade\_command['execution\_id']

window = self.config['risk\_controls']['duplicate\_order\_window']

# Check recent orders

cutoff\_time = datetime.now() - timedelta(seconds=window)

for order in self.recent\_orders:

if (order['execution\_id'] == execution\_id and

order['timestamp'] > cutoff\_time):

return False

return True

### **5.2 Position Monitoring**

async def \_monitor\_position(self, execution\_id: str) -> None:

"""Continuous position monitoring after entry"""

monitoring\_data = self.active\_positions[execution\_id]

max\_hold\_time = monitoring\_data['max\_hold\_time']

entry\_time = monitoring\_data['entry\_time']

while execution\_id in self.active\_positions:

try:

# Check time stop

bars\_held = self.\_calculate\_bars\_held(entry\_time)

if bars\_held >= max\_hold\_time:

logger.info(f"Time stop triggered for {execution\_id}")

await self.\_close\_position(execution\_id, 'TIME\_STOP')

break

# Check if stops/targets hit (handled by order updates)

# Update trailing stop if applicable

if monitoring\_data.get('use\_trailing'):

await self.\_update\_trailing\_stop(execution\_id)

# Sleep until next check

await asyncio.sleep(5) # Check every 5 seconds

except Exception as e:

logger.error(f"Position monitoring error: {e}")

# Continue monitoring despite errors

### **5.3 Emergency Procedures**

async def emergency\_close\_all(self) -> None:

"""Emergency procedure to close all positions"""

logger.critical("EMERGENCY CLOSE ALL INITIATED")

# Cancel all pending orders

for order\_id, order in self.orders.items():

if order.status in ['PENDING\_SUBMIT', 'SUBMITTED', 'ACKNOWLEDGED']:

try:

await self.cancel\_order(order\_id)

except Exception as e:

logger.error(f"Failed to cancel {order\_id}: {e}")

# Close all positions at market

for symbol, position in self.positions.items():

if position.quantity != 0:

try:

emergency\_order = OrderSpecification(

symbol=symbol,

quantity=abs(position.quantity),

direction=-1 if position.quantity > 0 else 1,

order\_type='EMERGENCY\_CLOSE',

is\_market=True

)

await self.place\_order(emergency\_order)

except Exception as e:

logger.error(f"Failed to close {symbol}: {e}")

# Alert operations team

await self.\_send\_emergency\_alert()

## **6. Event Outputs**

### **6.1 Trade Closed Event**

**Event Name:** TRADE\_CLOSED  
 **Emitted:** When position fully closed  
 **Payload:**

TradeResult = {

'execution\_id': str,

'symbol': str,

'direction': int,

'entry': {

'price': float,

'quantity': int,

'timestamp': datetime,

'slippage': float # vs expected

},

'exit': {

'price': float,

'quantity': int,

'timestamp': datetime,

'reason': str # 'STOP\_LOSS', 'TAKE\_PROFIT', 'TIME\_STOP', etc.

},

'performance': {

'pnl': float,

'pnl\_points': float,

'pnl\_percent': float,

'commission\_paid': float,

'net\_pnl': float,

'bars\_held': int,

'max\_favorable': float,

'max\_adverse': float

},

'execution\_quality': {

'entry\_slippage': float,

'exit\_slippage': float,

'fill\_time\_ms': float

}

}

### **6.2 Execution Alerts**

# Real-time alerts for monitoring

EXECUTION\_ALERTS = {

'ORDER\_REJECTED': 'Order rejected by venue',

'PARTIAL\_FILL\_TIMEOUT': 'Order partially filled, timeout reached',

'CONNECTION\_LOST': 'Lost connection to execution venue',

'POSITION\_LIMIT\_EXCEEDED': 'Position limit would be exceeded',

'RISK\_CHECK\_FAILED': 'Pre-trade risk check failed'

}

## **7. Error Handling & Recovery**

### **7.1 Connection Management**

async def \_handle\_disconnection(self) -> None:

"""Handle loss of connection to execution venue"""

logger.error("Connection lost to execution venue")

self.state = 'DISCONNECTED'

# Attempt reconnection

for attempt in range(self.config['reconnect\_attempts']):

logger.info(f"Reconnection attempt {attempt + 1}")

if await self.connect():

# Reconcile state

await self.\_reconcile\_positions()

await self.\_reconcile\_orders()

self.state = 'CONNECTED'

logger.info("Successfully reconnected")

return

await asyncio.sleep(2 \*\* attempt) # Exponential backoff

# Failed to reconnect

logger.critical("Failed to reconnect - entering safe mode")

await self.\_enter\_safe\_mode()

### **7.2 State Reconciliation**

async def \_reconcile\_positions(self) -> None:

"""Reconcile local position state with venue"""

if self.mode == 'live':

# Query actual positions from Rithmic

venue\_positions = await self.rithmic\_client.get\_positions()

for symbol, venue\_position in venue\_positions.items():

local\_position = self.positions.get(symbol)

if not local\_position:

# Unknown position at venue

logger.error(f"Unknown position at venue: {symbol}")

self.positions[symbol] = Position.from\_venue(venue\_position)

elif local\_position.quantity != venue\_position['quantity']:

# Mismatch

logger.error(f"Position mismatch for {symbol}: "

f"local={local\_position.quantity}, "

f"venue={venue\_position['quantity']}")

# Venue is source of truth

local\_position.quantity = venue\_position['quantity']

local\_position.average\_price = venue\_position['avg\_price']

## **8. Operational Requirements**

### **8.1 Performance Requirements**

* **Order Latency:** <50ms from command to venue submission
* **Fill Processing:** <10ms from notification to position update
* **State Queries:** <1ms for position/order lookups

### **8.2 Reliability Requirements**

* **Order Tracking:** 100% of orders tracked to completion
* **Position Accuracy:** Zero tolerance for position mismatches
* **Audit Trail:** Every action logged with timestamp

### **8.3 Monitoring Requirements**

# Real-time metrics tracked

{

'orders\_per\_minute': 12,

'fill\_rate': 0.98,

'average\_slippage': 0.25, # Ticks

'rejection\_rate': 0.01,

'active\_positions': 2,

'connection\_uptime': 0.999,

'last\_heartbeat': '2025-06-20 10:30:45'

}

## **9. Testing Strategy**

### **9.1 Unit Tests**

* Order state machine transitions
* Position calculation scenarios
* Risk check validations
* Bracket order creation

### **9.2 Integration Tests**

* Full trade lifecycle simulation
* Connection failure and recovery
* State reconciliation scenarios
* Emergency procedures

### **9.3 Market Simulation Tests**

* Partial fill scenarios
* Rapid market movements
* Gap scenarios
* Limit up/down handling

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

* Does NOT make trading decisions
* Does NOT calculate position sizes
* Does NOT determine entry/exit prices
* Does NOT analyze market data
* Does NOT modify risk parameters
* Does NOT generate trading signals
* Does NOT optimize execution algorithms

This completes the ExecutionHandler PRD. It transforms intelligent trading decisions into reliable market actions while maintaining absolute position integrity and providing comprehensive execution quality metrics.

The dual implementation ensures identical behavior in backtest and live environments, while the comprehensive error handling and state reconciliation mechanisms ensure reliability in production.

Ready for the final component - a refined version of the System Kernel?