Risk Management Systems

Duration: 80 minutes
Level: Intermediate-Advanced
Author: Obelisk Core

Learning Objectives

By the end of this module, you will be able to:

- Build comprehensive automated risk monitoring systems
- Create integrated risk dashboards and alerting infrastructure
- Implement machine learning-based risk prediction models
- Design risk management APIs and microservices
- Build automated compliance and reporting systems

Introduction to Risk Management Systems

Risk management systems in MEV trading must be comprehensive, automated, and real-time. Unlike traditional trading, MEV systems face unique challenges: microsecond-level decision making, gas cost uncertainties, network delays, and smart contract execution risks. A robust risk management system acts as the nervous system of your MEV operation.

Why Automated Risk Systems Are Essential

- **Speed requirements**: MEV opportunities disappear in seconds
- **Complexity**: Multiple strategies, protocols, and networks
- **Risk accumulation**: Small risks can compound quickly
- **24/7 operation**: Human monitoring is impractical
- **Compliance**: Automated logging and reporting requirements
- **Scale**: Handling multiple concurrent strategies and positions

System Architecture Principles

1. **Modularity**: Components should be independent and replaceable
2. **Fault tolerance**: System should continue operating despite component failures
3. **Real-time processing**: Millisecond-level risk assessments
4. **Data integrity**: Accurate and consistent risk data

5. **Auditability**: Complete audit trails for compliance
6. **Scalability**: Handle growing strategy complexity and volume

---

Core Risk Management Infrastructure

1. Risk Data Pipeline

```
import asyncio
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
from dataclasses import dataclass
from typing import Dict, List, Optional, Any
import json

@dataclass
class RiskDataPoint:
    timestamp: datetime
    source: str
    data_type: str
    value: Any
    confidence: float
    metadata: Dict[str, Any]

class RiskDataPipeline:
    """
    Centralized data pipeline for risk management
    """
    def __init__(self, config):
        self.config = config
        self.data_sources = {}
        self.data_cache = {}
        self.processors = {}
        self.subscribers = []

    def register_data_source(self, name: str, source_config: Dict):
        """
        Register a new data source
        """
        self.data_sources[name] = {
            'config': source_config,
            'last_update': None,
            'status': 'INACTIVE',
            'error_count': 0
        }

    def register_processor(self, name: str, processor_func):
        """
        Register a data processor
```

```
        """
        self.processors[name] = processor_func

    async def collect_market_data(self) -> List[RiskDataPoint]:
        """
        Collect real-time market data
        """
        data_points = []

        # Simulate data collection from various sources
        sources = [
            {'name': 'price_oracle', 'type': 'price', 'symbols': ['ETH/
USD', 'BTC/USD']},
            {'name': 'gas_tracker', 'type': 'gas_price', 'networks':
['ethereum', 'polygon']},
            {'name': 'liquidity_monitor', 'type': 'liquidity',
'protocols': ['uniswap', 'aave']},
            {'name': 'volatility_index', 'type': 'volatility',
'assets': ['ETH', 'BTC']}
        ]

        for source in sources:
            try:
                # Simulate data collection
                if source['type'] == 'price':
                    for symbol in source['symbols']:
                        data_point = RiskDataPoint(
                            timestamp=datetime.now(),
                            source=source['name'],
                            data_type='price',
                            value=np.random.normal(2000, 100),  # ETH
price simulation
                            confidence=0.95,
                            metadata={'symbol': symbol, 'network':
'ethereum'}
                        )
                        data_points.append(data_point)

                elif source['type'] == 'gas_price':
                    for network in source['networks']:
                        data_point = RiskDataPoint(
                            timestamp=datetime.now(),
```

```
                            source=source['name'],
                            data_type='gas_price',
                            value=np.random.normal(30, 10),  # Gas
price in gwei
                            confidence=0.90,
                            metadata={'network': network}
                        )
                        data_points.append(data_point)

                # Update source status
                if source['name'] in self.data_sources:
                    self.data_sources[source['name']]['status'] =
'ACTIVE'
                    self.data_sources[source['name']]['last_update'] =
datetime.now()

            except Exception as e:
                # Handle source errors
                if source['name'] in self.data_sources:
                    self.data_sources[source['name']]['error_count'] +=
1
                    if self.data_sources[source['name']]['error_count']
> 5:
                        self.data_sources[source['name']]['status'] =
'ERROR'

        return data_points

    async def collect_strategy_data(self) -> List[RiskDataPoint]:
        """
        Collect strategy-specific data
        """
        data_points = []

        strategies = ['arbitrage', 'liquidation', 'sandwich',
'frontrun']

        for strategy in strategies:
            try:
                # Simulate strategy performance data
                data_point = RiskDataPoint(
                    timestamp=datetime.now(),
```

```
                    source='strategy_monitor',
                    data_type='performance',
                    value=np.random.normal(0.001, 0.01),  # Daily
return
                    confidence=1.0,
                    metadata={'strategy': strategy, 'period': 'daily'}
                )
                data_points.append(data_point)

                # Position data
                position_data = RiskDataPoint(
                    timestamp=datetime.now(),
                    source='position_tracker',
                    data_type='position',
                    value=np.random.randint(1, 10),
# Number of active positions
                    confidence=1.0,
                    metadata={'strategy': strategy}
                )
                data_points.append(position_data)

            except Exception as e:
                print(f"Error collecting strategy data for {strategy}:
{e}")

        return data_points

    async def process_data(self, data_points: List[RiskDataPoint]) ->
Dict[str, Any]:
        """
        Process raw data points through registered processors
        """
        processed_data = {}

        for data_point in data_points:
            try:
                # Route data to appropriate processor
                for processor_name, processor_func in
self.processors.items():
                    if self._should_process(processor_name,
data_point):
                        result = processor_func(data_point)
```

```
                        if result:
                            processed_data[f"{processor_name}
_{data_point.source}"] = result

                # Cache recent data
                cache_key = f"{data_point.source}
_{data_point.data_type}"
                self.data_cache[cache_key] = {
                    'data': data_point,
                    'timestamp': datetime.now(),
                    'processed': True
                }

            except Exception as e:
                print(f"Error processing data point: {e}")

        return processed_data

    def _should_process(self, processor_name: str, data_point:
RiskDataPoint) -> bool:
        """
        Determine if processor should handle this data point
        """
        processor_rules = {
            'risk_calculator': ['price', 'gas_price', 'volatility'],
            'correlation_tracker': ['price', 'performance'],
            'volatility_monitor': ['price', 'volatility'],
            'liquidity_tracker': ['liquidity', 'position']
        }

        return data_point.data_type in
processor_rules.get(processor_name, [])

    async def run_pipeline(self):
        """
        Main pipeline execution loop
        """
        while True:
            try:
                # Collect data
                market_data = await self.collect_market_data()
                strategy_data = await self.collect_strategy_data()
```

```
                all_data = market_data + strategy_data

                # Process data
                processed_data = await self.process_data(all_data)

                # Notify subscribers
                await self._notify_subscribers(processed_data)

                # Wait before next cycle
                await asyncio.sleep(1)  # 1 second cycle

            except Exception as e:
                print(f"Pipeline error: {e}")
                await asyncio.sleep(5)  # Longer wait on error

# Example usage and setup
async def main():
    # Initialize pipeline
    pipeline = RiskDataPipeline({})

    # Register data sources
    pipeline.register_data_source('price_oracle', {'endpoint':
'https://api.example.com/prices'})
    pipeline.register_data_source('gas_tracker', {'endpoint': 'https://
api.example.com/gas'})

    # Register processors
    def risk_calculator(data_point):
        if data_point.data_type == 'price':
            # Simple volatility calculation
            return {
                'volatility': abs(data_point.value - 2000) / 2000,
                'risk_score': min(1.0, abs(data_point.value - 2000) /
500)
            }
        return None

    def correlation_tracker(data_point):
        if data_point.data_type == 'price':
            return {
                'price_change': data_point.value,
                'timestamp': data_point.timestamp.isoformat()
```

```
            }
        return None

    pipeline.register_processor('risk_calculator', risk_calculator)
    pipeline.register_processor('correlation_tracker',
correlation_tracker)

    # Start pipeline (commented out to avoid infinite loop in example)
    # await pipeline.run_pipeline()

print("Risk Data Pipeline Example:")
print("=" * 40)
print("Pipeline initialized with:")
print("- 2 data sources registered")
print("- 2 processors registered")
print("- Ready for real-time data collection")
```

2. Risk Calculation Engine

```
class RiskCalculationEngine:
    """
    Centralized engine for risk calculations
    """
    def __init__(self):
        self.calculators = {}
        self.cache = {}
        self.calculation_history = []

    def register_calculator(self, name: str, calculator_func):
        """
        Register a risk calculator
        """
        self.calculators[name] = calculator_func

    def calculate_portfolio_risk(self, positions: Dict, market_data:
Dict, config: Dict) -> Dict:
        """
        Calculate comprehensive portfolio risk
        """
        risk_metrics = {}

        # Position-level risk
        position_risks = {}
        total_exposure = 0

        for position_id, position in positions.items():
            pos_risk = self._calculate_position_risk(position,
market_data, config)
            position_risks[position_id] = pos_risk
            total_exposure += abs(position.get('value', 0))

        risk_metrics['positions'] = position_risks
        risk_metrics['total_exposure'] = total_exposure

        # Portfolio-level risk
        risk_metrics['portfolio'] =
self._calculate_portfolio_level_risk(
            position_risks, market_data, config
        )

        # VaR calculations
```

```
        risk_metrics['var'] = self._calculate_var(positions,
market_data, config)

        # Stress testing
        risk_metrics['stress_tests'] =
self._run_stress_tests(positions, market_data)

        return risk_metrics

    def _calculate_position_risk(self, position: Dict, market_data:
Dict, config: Dict) -> Dict:
        """
        Calculate risk for individual position
        """
        position_type = position.get('type', 'unknown')
        exposure = abs(position.get('value', 0))

        # Base risk calculation
        base_risk = exposure * 0.02  # 2% base risk

        # Adjust for position type
        type_multipliers = {
            'arbitrage': 0.5,      # Lower risk
            'liquidation': 1.2,    # Higher risk
            'sandwich': 1.5,       # Highest risk
            'frontrun': 0.8       # Moderate risk
        }

        risk_multiplier = type_multipliers.get(position_type, 1.0)
        adjusted_risk = base_risk * risk_multiplier

        # Adjust for market conditions
        market_conditions = self._assess_market_conditions(market_data)
        market_adjustment = 1 +
market_conditions['volatility_multiplier'] * 0.5

        final_risk = adjusted_risk * market_adjustment

        return {
            'base_risk': base_risk,
            'adjusted_risk': adjusted_risk,
            'final_risk': final_risk,
```

```
            'risk_percentage': final_risk / exposure if exposure > 0
else 0,
            'type': position_type,
            'confidence': 0.85
        }

    def _assess_market_conditions(self, market_data: Dict) -> Dict:
        """
        Assess current market conditions
        """
        conditions = {
            'volatility_multiplier': 1.0,
            'correlation_stress': 1.0,
            'liquidity_stress': 1.0,
            'gas_stress': 1.0
        }

        # Simulate market condition assessment
        if market_data.get('volatility', 0) > 0.05:  # High volatility
            conditions['volatility_multiplier'] = 2.0

        if market_data.get('correlation', 0) > 0.8:  # High correlation
            conditions['correlation_stress'] = 1.5

        if market_data.get('liquidity_ratio', 1) < 0.5:  # Low
liquidity
            conditions['liquidity_stress'] = 2.0

        if market_data.get('gas_price', 0) > 100:  # High gas
            conditions['gas_stress'] = 1.5

        return conditions

    def _calculate_portfolio_level_risk(self, position_risks: Dict,
market_data: Dict, config: Dict) -> Dict:
        """
        Calculate portfolio-level risk metrics
        """
        total_risk = sum(pos['final_risk'] for pos in
position_risks.values())
        total_exposure = sum(abs(pos.get('value', 0)) for pos in
position_risks.values())
```

```
        # Calculate concentration risk
        exposures = [abs(pos.get('value', 0)) for pos in
position_risks.values()]
        max_concentration = max(exposures) / total_exposure if
total_exposure > 0 else 0

        # Calculate correlation-adjusted risk
        correlation_risk =
self._calculate_correlation_risk(position_risks, market_data)

        return {
            'total_risk': total_risk,
            'risk_per_exposure': total_risk / total_exposure if
total_exposure > 0 else 0,
            'max_concentration': max_concentration,
            'correlation_risk': correlation_risk,
            'risk_rating': self._calculate_risk_rating(total_risk,
max_concentration, correlation_risk)
        }

    def _calculate_correlation_risk(self, position_risks: Dict,
market_data: Dict) -> float:
        """
        Calculate correlation-based risk adjustment
        """
        # Simplified correlation risk calculation
        base_correlation = market_data.get('average_correlation', 0.3)

        # Risk increases with correlation
        correlation_risk = base_correlation * 0.5  # Cap correlation
impact at 50%

        return min(1.0, correlation_risk)

    def _calculate_risk_rating(self, total_risk: float, concentration:
float, correlation_risk: float) -> str:
        """
        Calculate overall risk rating
        """
        # Risk scoring algorithm
        risk_score = 0
```

```
        # Total risk component (40% weight)
        if total_risk > 10000:  # High absolute risk
            risk_score += 0.4
        elif total_risk > 5000:  # Medium absolute risk
            risk_score += 0.2

        # Concentration component (30% weight)
        if concentration > 0.5:  # High concentration
            risk_score += 0.3
        elif concentration > 0.3:  # Medium concentration
            risk_score += 0.15

        # Correlation component (30% weight)
        if correlation_risk > 0.7:  # High correlation
            risk_score += 0.3
        elif correlation_risk > 0.5:  # Medium correlation
            risk_score += 0.15

        # Determine rating
        if risk_score >= 0.7:
            return 'HIGH'
        elif risk_score >= 0.4:
            return 'MEDIUM'
        else:
            return 'LOW'

    def _calculate_var(self, positions: Dict, market_data: Dict,
config: Dict) -> Dict:
        """
        Calculate Value at Risk (VaR)
        """
        # Simplified VaR calculation
        total_exposure = sum(abs(pos.get('value', 0)) for pos in
positions.values())

        # Historical simulation approach
        volatility = market_data.get('portfolio_volatility', 0.02)

        var_95 = total_exposure * volatility * 1.645  # 95% confidence
        var_99 = total_exposure * volatility * 2.326  # 99% confidence
```

```
        return {
            'var_95': var_95,
            'var_99': var_99,
            'expected_shortfall_95': var_95 * 1.2,  # Approximation
            'expected_shortfall_99': var_99 * 1.3,
            'confidence_level': 'Historical simulation'
        }

    def _run_stress_tests(self, positions: Dict, market_data: Dict) ->
Dict:
        """
        Run stress tests on portfolio
        """
        stress_scenarios = {
            'market_crash': {'price_impact': -0.2, 'volatility_spike':
3.0},
            'liquidity_crisis': {'liquidity_impact': -0.5,
'spread_widening': 2.0},
            'correlation_spike': {'correlation_increase': 0.5},
            'gas_price_spike': {'gas_multiplier': 5.0}
        }

        stress_results = {}

        for scenario_name, scenario_params in stress_scenarios.items():
            stress_result = self._calculate_stress_impact(positions,
scenario_params)
            stress_results[scenario_name] = stress_result

        return stress_results

    def _calculate_stress_impact(self, positions: Dict,
scenario_params: Dict) -> Dict:
        """
        Calculate impact of stress scenario
        """
        total_impact = 0
        impacts = {}

        for position_id, position in positions.items():
            position_value = position.get('value', 0)
            position_type = position.get('type', 'unknown')
```

```
            # Calculate scenario-specific impact
            impact = 0

            if 'price_impact' in scenario_params:
                impact += position_value *
scenario_params['price_impact']

            if 'volatility_spike' in scenario_params:
                # Increased volatility increases risk
                vol_impact = position_value * 0.1 *
scenario_params['volatility_spike']
                impact += vol_impact

            if 'gas_multiplier' in scenario_params:
                # Higher gas affects MEV profitability
                if position_type in ['liquidation', 'sandwich']:
                    gas_impact = position_value * 0.05 *
scenario_params['gas_multiplier']
                    impact -= gas_impact

            impacts[position_id] = impact
            total_impact += impact

        return {
            'total_impact': total_impact,
            'impact_percentage': total_impact /
sum(abs(pos.get('value', 0)) for pos in positions.values()) if
positions else 0,
            'position_impacts': impacts,
            'worst_position': max(impacts.items(), key=lambda x:
abs(x[1])) if impacts else None
        }

# Example usage
risk_engine = RiskCalculationEngine()

# Register calculators
def custom_var_calculator(data):
    # Custom VaR calculation
    return {'custom_var': data['exposure'] * 0.1}
```

```
risk_engine.register_calculator('custom_var', custom_var_calculator)

# Test with sample data
sample_positions = {
    'pos_1': {'type': 'arbitrage', 'value': 50000, 'strategy': 'ETH-
USDC'},
    'pos_2': {'type': 'liquidation', 'value': 30000, 'protocol':
'Aave'},
    'pos_3': {'type': 'sandwich', 'value': 20000, 'network':
'ethereum'}
}

sample_market_data = {
    'volatility': 0.03,
    'correlation': 0.4,
    'liquidity_ratio': 0.8,
    'gas_price': 25,
    'portfolio_volatility': 0.025
}

config = {'max_position_risk': 0.02}

risk_metrics = risk_engine.calculate_portfolio_risk(sample_positions,
sample_market_data, config)

print("Risk Calculation Engine Results:")
print("=" * 50)
print(f"Total Exposure: ${risk_metrics['total_exposure']:,.0f}")
print(f"Portfolio Risk Rating: {risk_metrics['portfolio']
['risk_rating']}")
print(f"VaR (95%): ${risk_metrics['var']['var_95']:,.0f}")
print(f"VaR (99%): ${risk_metrics['var']['var_99']:,.0f}")
print(f"Max Concentration: {risk_metrics['portfolio']
['max_concentration']:.1%}")

print("\nPosition Risk Details:")
for pos_id, pos_risk in risk_metrics['positions'].items():
    print(f"{pos_id}: {pos_risk['type']} - Risk: $
{pos_risk['final_risk']:,.0f} ({pos_risk['risk_percentage']:.1%})")

print("\nStress Test Results:")
for scenario, result in risk_metrics['stress_tests'].items():
```

```
    print(f"{scenario}: ${result['total_impact']:,.0f}
({result['impact_percentage']:.1%})")
```

Real-Time Risk Monitoring

3. Risk Monitoring Dashboard

```
from flask import Flask, render_template, jsonify
import json

class RiskMonitoringDashboard:
    """
    Web-based risk monitoring dashboard
    """
    def __init__(self, risk_engine, data_pipeline):
        self.app = Flask(__name__)
        self.risk_engine = risk_engine
        self.data_pipeline = data_pipeline
        self.current_risk_data = {}
        self.alerts = []
        self.setup_routes()

    def setup_routes(self):
        """
        Setup Flask routes for dashboard
        """
        @self.app.route('/')
        def dashboard():
            return render_template('risk_dashboard.html')

        @self.app.route('/api/risk-overview')
        def risk_overview():
            return jsonify(self.get_risk_overview())

        @self.app.route('/api/positions')
        def positions():
            return jsonify(self.get_position_data())

        @self.app.route('/api/alerts')
        def alerts():
            return jsonify(self.get_recent_alerts())

        @self.app.route('/api/var-data')
        def var_data():
            return jsonify(self.get_var_data())

        @self.app.route('/api/stress-tests')
        def stress_tests():
            return jsonify(self.get_stress_test_results())
```

```
    def get_risk_overview(self):
        """
        Get high-level risk overview
        """
        if not self.current_risk_data:
            return {'status': 'no_data'}

        portfolio_risk = self.current_risk_data.get('portfolio', {})

        return {
            'risk_rating': portfolio_risk.get('risk_rating',
'UNKNOWN'),
            'total_exposure':
self.current_risk_data.get('total_exposure', 0),
            'var_95': self.current_risk_data.get('var',
{}).get('var_95', 0),
            'var_99': self.current_risk_data.get('var',
{}).get('var_99', 0),
            'max_concentration':
portfolio_risk.get('max_concentration', 0),
            'correlation_risk': portfolio_risk.get('correlation_risk',
0),
            'last_update': datetime.now().isoformat(),
            'active_alerts': len([a for a in self.alerts if not
a.get('resolved', False)]),
            'risk_trend': self._calculate_risk_trend()
        }

    def get_position_data(self):
        """
        Get position-level risk data
        """
        if not self.current_risk_data:
            return {'positions': []}

        positions = self.current_risk_data.get('positions', {})
        position_list = []

        for pos_id, pos_data in positions.items():
            position_list.append({
                'id': pos_id,
```

```
                'type': pos_data.get('type', 'unknown'),
                'risk_amount': pos_data.get('final_risk', 0),
                'risk_percentage': pos_data.get('risk_percentage', 0),
                'exposure': pos_data.get('value', 0),
                'risk_level':
self._categorize_risk_level(pos_data.get('risk_percentage', 0))
            })

        return {
            'positions': position_list,
            'total_positions': len(position_list),
            'high_risk_positions': len([p for p in position_list if
p['risk_level'] == 'HIGH']),
            'last_update': datetime.now().isoformat()
        }

    def get_recent_alerts(self, limit=10):
        """
        Get recent risk alerts
        """
        recent_alerts = sorted(self.alerts, key=lambda x:
x['timestamp'], reverse=True)[:limit]
        return {
            'alerts': recent_alerts,
            'total_alerts': len(self.alerts),
            'unresolved_alerts': len([a for a in self.alerts if not
a.get('resolved', False)])
        }

    def get_var_data(self):
        """
        Get VaR analysis data
        """
        var_data = self.current_risk_data.get('var', {})

        return {
            'var_95': var_data.get('var_95', 0),
            'var_99': var_data.get('var_99', 0),
            'expected_shortfall_95':
var_data.get('expected_shortfall_95', 0),
            'expected_shortfall_99':
var_data.get('expected_shortfall_99', 0),
```

```
            'confidence_level': var_data.get('confidence_level',
'Unknown'),
            'var_trend': self._calculate_var_trend(),
            'last_update': datetime.now().isoformat()
        }

    def get_stress_test_results(self):
        """
        Get stress test results
        """
        stress_tests = self.current_risk_data.get('stress_tests', {})

        formatted_results = {}
        for scenario, result in stress_tests.items():
            formatted_results[scenario] = {
                'total_impact': result.get('total_impact', 0),
                'impact_percentage': result.get('impact_percentage',
0),
                'worst_position': result.get('worst_position'),
                'scenario_description':
self._get_scenario_description(scenario)
            }

        return {
            'scenarios': formatted_results,
            'most_dangerous_scenario':
self._find_most_dangerous_scenario(formatted_results),
            'last_update': datetime.now().isoformat()
        }

    def _categorize_risk_level(self, risk_percentage):
        """
        Categorize risk level
        """
        if risk_percentage > 0.05:  # >5%
            return 'HIGH'
        elif risk_percentage > 0.02:  # >2%
            return 'MEDIUM'
        else:
            return 'LOW'

    def _calculate_risk_trend(self):
```

```
        """
        Calculate risk trend over time
        """
        # Simplified trend calculation
        return 'STABLE'  # Would implement with historical data

    def _calculate_var_trend(self):
        """
        Calculate VaR trend
        """
        # Simplified trend calculation
        return 'INCREASING'  # Would implement with historical data

    def _get_scenario_description(self, scenario):
        """
        Get description of stress test scenario
        """
        descriptions = {
            'market_crash': '20% price decline with 3x volatility
spike',
            'liquidity_crisis': '50%
liquidity reduction with 2x spread widening',
            'correlation_spike': '50% increase in portfolio
correlation',
            'gas_price_spike': '5x increase in gas prices'
        }
        return descriptions.get(scenario, 'Unknown scenario')

    def _find_most_dangerous_scenario(self, scenarios):
        """
        Find the most dangerous stress test scenario
        """
        if not scenarios:
            return None

        most_dangerous = max(scenarios.items(), key=lambda x: abs(x[1]
['total_impact']))
        return {
            'scenario': most_dangerous[0],
            'impact': most_dangerous[1]['total_impact']
        }
```

```
    def update_risk_data(self, risk_data):
        """
        Update current risk data and check for alerts
        """
        self.current_risk_data = risk_data

        # Generate alerts based on risk data
        self._check_risk_alerts(risk_data)

        # Update historical data for trends
        self._update_risk_history(risk_data)

    def _check_risk_alerts(self, risk_data):
        """
        Check for risk-based alerts
        """
        new_alerts = []

        # VaR alerts
        var_95 = risk_data.get('var', {}).get('var_95', 0)
        total_exposure = risk_data.get('total_exposure', 0)

        if total_exposure > 0:
            var_percentage = var_95 / total_exposure
            if var_percentage > 0.10:  # VaR > 10% of exposure
                new_alerts.append({
                    'id': f"var_high_{datetime.now().timestamp()}",
                    'type': 'VAR_LIMIT',
                    'severity': 'HIGH',
                    'message': f'VaR (95%) of {var_percentage:.1%}
exceeds 10% threshold',
                    'timestamp': datetime.now(),
                    'data': {'var_95': var_95, 'percentage':
var_percentage}
                })

        # Concentration alerts
        max_concentration = risk_data.get('portfolio',
{}).get('max_concentration', 0)
        if max_concentration > 0.50:  # >50% concentration
            new_alerts.append({
                'id':
```

```
f"concentration_high_{datetime.now().timestamp()}",
                'type': 'CONCENTRATION',
                'severity': 'MEDIUM',
                'message': f'Position concentration of
{max_concentration:.1%} exceeds 50%',
                'timestamp': datetime.now(),
                'data': {'concentration': max_concentration}
            })

        # Risk rating alerts
        risk_rating = risk_data.get('portfolio', {}).get('risk_rating',
'UNKNOWN')
        if risk_rating == 'HIGH':
            new_alerts.append({
                'id': f"risk_rating_high_{datetime.now().timestamp()}",
                'type': 'RISK_RATING',
                'severity': 'HIGH',
                'message': 'Portfolio risk rating is HIGH',
                'timestamp': datetime.now(),
                'data': {'risk_rating': risk_rating}
            })

        self.alerts.extend(new_alerts)

    def _update_risk_history(self, risk_data):
        """
        Update risk history for trend analysis
        """
        # Store current risk data with timestamp
        if not hasattr(self, 'risk_history'):
            self.risk_history = []

        self.risk_history.append({
            'timestamp': datetime.now(),
            'risk_data': risk_data.copy()
        })

        # Keep only last 1000 records
        if len(self.risk_history) > 1000:
            self.risk_history.pop(0)

# HTML template for dashboard (simplified)
```

```
dashboard_html = """
<!DOCTYPE html>
<html>
<head>
    <title>Risk Management Dashboard</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 20px; }
        .metric-card {
            border: 1px solid #ddd;
            padding: 15px;
            margin: 10px;
            border-radius: 5px;
            display: inline-block;
            width: 200px;
        }
        .risk-low { background-color: #d4edda; }
        .risk-medium { background-color: #fff3cd; }
        .risk-high { background-color: #f8d7da; }
        .alert { padding: 10px; margin: 5px; border-radius: 3px; }
        .alert-high { background-color: #f8d7da; color: #721c24; }
        .alert-medium { background-color: #fff3cd; color: #856404; }
        .alert-low { background-color: #d4edda; color: #155724; }
    </style>
</head>
<body>
    <h1>Risk Management Dashboard</h1>

    <div id="risk-overview">
        <h2>Risk Overview</h2>
        <div id="overview-metrics"></div>
    </div>

    <div id="positions">
        <h2>Position Risk</h2>
        <div id="position-list"></div>
    </div>

    <div id="alerts">
        <h2>Active Alerts</h2>
        <div id="alert-list"></div>
    </div>
```

```
    <script>
        async function updateDashboard() {
            try {
                const [overview, positions, alerts] = await
Promise.all([
                    fetch('/api/risk-overview').then(r => r.json()),
                    fetch('/api/positions').then(r => r.json()),
                    fetch('/api/alerts').then(r => r.json())
                ]);

                updateOverview(overview);
                updatePositions(positions);
                updateAlerts(alerts);
            } catch (error) {
                console.error('Error updating dashboard:', error);
            }
        }

        function updateOverview(data) {
            const container = document.getElementById('overview-
metrics');
            container.innerHTML = `
                <div class="metric-card risk-$
{data.risk_rating?.toLowerCase()}">
                    <h3>Risk Rating</h3>
                    <div style="font-size: 2em;">${data.risk_rating ||
'Unknown'}</div>
                </div>
                <div class="metric-card">
                    <h3>Total Exposure</h3>
                    <div style="font-size: 1.5em;"><div class="math-
display" style="text-align: center; margin: 1em 0;"><math
xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mrow><mo stretchy="false">&#x00028;</mo><mi>d</
mi><mi>a</mi><mi>t</mi><mi>a</mi><mo>&#x0002E;</mo><mi>t</mi><mi>o</
mi><mi>t</mi><mi>a</mi><msub><mi>l</mi><mi>e</mi></msub><mi>x</
mi><mi>p</mi><mi>o</mi><mi>s</mi><mi>u</mi><mi>r</mi><mi>e</mi><mo
stretchy="false">&#x0007C;</mo><mo stretchy="false">&#x0007C;</
mo><mn>0</mn><mo stretchy="false">&#x00029;</mo><mo>&#x0002E;</
mo><mi>t</mi><mi>o</mi><mi>L</mi><mi>o</mi><mi>c</mi><mi>a</mi><mi>l</
mi><mi>e</mi><mi>S</mi><mi>t</mi><mi>r</mi><mi>i</mi><mi>n</mi><mi>g</
mi><mo stretchy="false">&#x00028;</mo><mo stretchy="false">&#x00029;</
```

```
mo></mrow><mo>&#x0003C;</mo><mo>&#x0002F;</mo><mi>d</mi><mi>i</
mi><mi>v</mi><mo>&#x0003E;</mo><mo>&#x0003C;</mo><mo>&#x0002F;</
mo><mi>d</mi><mi>i</mi><mi>v</mi><mo>&#x0003E;</mo><mo>&#x0003C;</
mo><mi>d</mi><mi>i</mi><mi>v</mi><mi>c</mi><mi>l</mi><mi>a</mi><mi>s</
mi><mi>s</mi><mo>&#x0003D;</mo><mi>"</mi><mi>m</mi><mi>e</mi><mi>t</
mi><mi>r</mi><mi>i</mi><mi>c</mi><mo>&#x02212;</mo><mi>c</mi><mi>a</
mi><mi>r</mi><mi>d</mi><mi>"</mi><mo>&#x0003E;</mo><mo>&#x0003C;</
mo><mi>h</mi><mn>3</mn><mo>&#x0003E;</mo><mi>V</mi><mi>a</mi><mi>R</
mi><mo stretchy="false">&#x00028;</mo><mn>95</mn><mo>&#x0003C;</
mo><mi>d</mi><mi>i</mi><mi>v</mi><mi>s</mi><mi>t</mi><mi>y</mi><mi>l</
mi><mi>e</mi><mo>&#x0003D;</mo><mi>"</mi><mi>f</mi><mi>o</mi><mi>n</
mi><mi>t</mi><mo>&#x02212;</mo><mi>s</mi><mi>i</mi><mi>z</mi><mi>e</
mi><mi>:</mi><mn>1.5em</mn><mi>;</mi><mi>"</mi><mo>&#x0003E;</mo></
mrow></math></div>{(data.var_95 || 0).toLocaleString()}</div>
                </div>
            `;
        }

        function updatePositions(data) {
            const container = document.getElementById('position-list');
            container.innerHTML = data.positions.map(pos => `
                <div class="metric-card risk-$
{pos.risk_level.toLowerCase()}">
                    <h4>${pos.id}</h4>
                    <div>Type: ${pos.type}</div>
                    <div>Risk: ${(pos.risk_percentage *
100).toFixed(1)}%</div>
                    <div>Amount: $${pos.risk_amount.toLocaleString()}</
div>
                </div>
            `).join('');
        }

        function updateAlerts(data) {
            const container = document.getElementById('alert-list');
            container.innerHTML = data.alerts.map(alert => `
                <div class="alert alert-$
{alert.severity.toLowerCase()}">
                    <strong><span class="math-inline" style="display:
inline;"><math xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mrow><mi>a</mi><mi>l</mi><mi>e</mi><mi>r</
mi><mi>t</mi><mo>&#x0002E;</mo><mi>t</mi><mi>y</mi><mi>p</mi><mi>e</
```

```
mi></mrow><mo>&#x0003C;</mo><mo>&#x0002F;</mo><mi>s</mi><mi>t</
mi><mi>r</mi><mi>o</mi><mi>n</mi><mi>g</mi><mo>&#x0003E;</mo><mi>:</
mi></mrow></math></span>{alert.message}
                    <small>${new
Date(alert.timestamp).toLocaleString()}</small>
                </div>
            `).join('');
        }

        // Update dashboard every 5 seconds
        updateDashboard();
        setInterval(updateDashboard, 5000);
    </script>
</body>
</html>
"""

print("Risk Monitoring Dashboard Created")
print("=" * 40)
print("Dashboard features:")
print("- Real-time risk overview")
print("- Position-level risk monitoring")
print("- VaR analysis")
print("- Stress test results")
print("- Alert management")
print("- Historical trend analysis")
```

4. Alert Management System

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import json

class AlertManager:
    """
    Comprehensive alert management system
    """
    def __init__(self, config):
        self.config = config
        self.alert_rules = []
        self.alert_history = []
        self.notification_channels = {}
        self.escalation_rules = []

    def register_alert_rule(self, rule):
        """
        Register a new alert rule
        rule = {
            'name': 'high_var',
            'condition': lambda data: data.get('var_95', 0) > 10000,
            'severity': 'HIGH',
            'cooldown_minutes': 60,
            'channels': ['email', 'slack', 'sms']
        }
        """
        self.alert_rules.append(rule)

    def register_notification_channel(self, name: str, config: Dict):
        """
        Register a notification channel
        """
        self.notification_channels[name] = config

    def register_escalation_rule(self, rule):
        """
        Register escalation rule
        rule = {
            'alert_type': 'VAR_LIMIT',
            'condition': lambda alert: alert['severity'] == 'HIGH',
            'after_minutes': 30,
```

```
            'escalate_to': ['manager', 'compliance'],
            'new_severity': 'CRITICAL'
        }
        """
        self.escalation_rules.append(rule)

    async def process_risk_data(self, risk_data: Dict):
        """
        Process risk data and generate alerts
        """
        triggered_alerts = []

        for rule in self.alert_rules:
            try:
                if rule['condition'](risk_data):
                    # Check cooldown period
                    if self._is_in_cooldown(rule, risk_data):
                        continue

                    alert = self._create_alert(rule, risk_data)
                    triggered_alerts.append(alert)

                    # Send notifications
                    await self._send_notifications(alert,
rule.get('channels', []))

            except Exception as e:
                print(f"Error processing alert rule {rule['name']}:
{e}")

        # Process escalations
        await self._process_escalations(triggered_alerts)

        return triggered_alerts

    def _is_in_cooldown(self, rule: Dict, risk_data: Dict) -> bool:
        """
        Check if alert is in cooldown period
        """
        cooldown_minutes = rule.get('cooldown_minutes', 60)
        cutoff_time = datetime.now() -
timedelta(minutes=cooldown_minutes)
```

```
        # Check recent alerts of same type
        for alert in self.alert_history:
            if (alert.get('rule_name') == rule['name'] and
                alert['timestamp'] > cutoff_time):
                return True

        return False

    def _create_alert(self, rule: Dict, risk_data: Dict) -> Dict:
        """
        Create alert object
        """
        alert = {
            'id': f"{rule['name']}_{datetime.now().timestamp()}",
            'rule_name': rule['name'],
            'severity': rule['severity'],
            'timestamp': datetime.now(),
            'message': rule.get('message', f"Alert triggered by
{rule['name']}"),
            'data': risk_data.copy(),
            'status': 'ACTIVE',
            'notifications_sent': [],
            'acknowledged': False,
            'resolved': False
        }

        self.alert_history.append(alert)
        return alert

    async def _send_notifications(self, alert: Dict, channels:
List[str]):
        """
        Send notifications via specified channels
        """
        for channel in channels:
            try:
                if channel in self.notification_channels:
                    success = await self._send_via_channel(alert,
channel)
                    if success:
                        alert['notifications_sent'].append(channel)
```

```
                else:
                    print(f"Unknown notification channel: {channel}")
            except Exception as e:
                print(f"Error sending notification via {channel}: {e}")

    async def _send_via_channel(self, alert: Dict, channel: str) ->
bool:
        """
        Send notification via specific channel
        """
        channel_config = self.notification_channels[channel]

        if channel == 'email':
            return await self._send_email_notification(alert,
channel_config)
        elif channel == 'slack':
            return await self._send_slack_notification(alert,
channel_config)
        elif channel == 'sms':
            return await self._send_sms_notification(alert,
channel_config)
        elif channel == 'webhook':
            return await self._send_webhook_notification(alert,
channel_config)
        else:
            print(f"Unsupported notification channel: {channel}")
            return False

    async def _send_email_notification(self, alert: Dict, config: Dict)
-> bool:
        """
        Send email notification
        """
        try:
            msg = MIMEMultipart()
            msg['From'] = config['sender_email']
            msg['To'] = ', '.join(config['recipients'])
            msg['Subject'] = f"[{alert['severity']}] Risk Alert:
{alert['rule_name']}"

            body = f"""
            Risk Alert Details:
```

```
            Rule: {alert['rule_name']}
            Severity: {alert['severity']}
            Timestamp: {alert['timestamp']}

            Message: {alert['message']}

            Risk Data:
            {json.dumps(alert['data'], indent=2)}

            Please review and take appropriate action.
            """

            msg.attach(MIMEText(body, 'plain'))

            # In real implementation, use actual SMTP server
            print(f"Email alert sent: {alert['rule_name']}")
            return True

        except Exception as e:
            print(f"Error sending email: {e}")
            return False

    async def _send_slack_notification(self, alert: Dict, config: Dict)
-> bool:
        """
        Send Slack notification
        """
        try:
            # In real implementation, use Slack API
            slack_message = {
                'channel': config.get('channel', '#risk-alerts'),
                'text': f"🚨 Risk Alert: {alert['severity']} -
{alert['message']}",
                'attachments': [{
                    'color': 'danger' if alert['severity'] == 'HIGH'
else 'warning',
                    'fields': [
                        {'title': 'Rule', 'value': alert['rule_name'],
'short': True},
                        {'title': 'Timestamp', 'value':
alert['timestamp'].isoformat(), 'short': True}
```

```
                    ]
                }]
            }

            print(f"Slack alert sent: {alert['rule_name']}")
            return True

        except Exception as e:
            print(f"Error sending Slack notification: {e}")
            return False

    async def _send_sms_notification(self, alert: Dict, config: Dict) -
> bool:
        """
        Send SMS notification
        """
        try:
            # In real implementation, use SMS service API
            message = f"RISK ALERT: {alert['severity']} -
{alert['message']}"

            print(f"SMS alert sent: {alert['rule_name']}")
            return True

        except Exception as e:
            print(f"Error sending SMS: {e}")
            return False

    async def _send_webhook_notification(self, alert: Dict, config:
Dict) -> bool:
        """
        Send webhook notification
        """
        try:
            # In real implementation, make HTTP POST request
            webhook_payload = {
                'alert_id': alert['id'],
                'severity': alert['severity'],
                'message': alert['message'],
                'timestamp': alert['timestamp'].isoformat(),
                'data': alert['data']
            }
```

```
            print(f"Webhook alert sent: {alert['rule_name']}")
            return True

        except Exception as e:
            print(f"Error sending webhook: {e}")
            return False

    async def _process_escalations(self, alerts: List[Dict]):
        """
        Process escalation rules
        """
        current_time = datetime.now()

        for alert in alerts:
            for escalation_rule in self.escalation_rules:
                if (escalation_rule['alert_type'] ==
alert.get('rule_name') and
                    escalation_rule['condition'](alert)):

                    escalation_delay =
timedelta(minutes=escalation_rule['after_minutes'])
                    escalation_time = alert['timestamp'] +
escalation_delay

                    if current_time >= escalation_time:
                        await self._escalate_alert(alert,
escalation_rule)

    async def _escalate_alert(self, alert: Dict, escalation_rule:
Dict):
        """
        Escalate alert to higher priority
        """
        # Create escalation alert
        escalation_alert = alert.copy()
        escalation_alert['id'] = f"{alert['id']}_escalated"
        escalation_alert['severity'] = escalation_rule['new_severity']
        escalation_alert['message'] = f"ESCALATED: {alert['message']}"
        escalation_alert['escalated_from'] = alert['id']
        escalation_alert['escalated_at'] = datetime.now()
```

```
        # Send escalation notifications
        escalation_channels = escalation_rule.get('escalate_to',
['manager'])
        await self._send_notifications(escalation_alert,
escalation_channels)

        self.alert_history.append(escalation_alert)
        print(f"Alert escalated: {alert['rule_name']} ->
{escalation_rule['new_severity']}")

    def get_alert_summary(self, time_period_hours=24):
        """
        Get summary of alerts in specified time period
        """
        cutoff_time = datetime.now() -
timedelta(hours=time_period_hours)

        recent_alerts = [a for a in self.alert_history if
a['timestamp'] > cutoff_time]

        summary = {
            'total_alerts': len(recent_alerts),
            'by_severity': {},
            'by_type': {},
            'active_alerts': len([a for a in recent_alerts if not
a.get('resolved', False)]),
            'acknowledged_alerts': len([a for a in recent_alerts if
a.get('acknowledged', False)]),
            'average_response_time':
self._calculate_average_response_time(recent_alerts)
        }

        # Count by severity
        for alert in recent_alerts:
            severity = alert['severity']
            summary['by_severity'][severity] =
summary['by_severity'].get(severity, 0) + 1

        # Count by type
        for alert in recent_alerts:
            alert_type = alert['rule_name']
            summary['by_type'][alert_type] =
```

```
summary['by_type'].get(alert_type, 0) + 1

        return summary

    def _calculate_average_response_time(self, alerts: List[Dict]) ->
Optional[float]:
        """
        Calculate average response time for alerts
        """
        response_times = []

        for alert in alerts:
            if alert.get('acknowledged') and
alert.get('acknowledged_at'):
                response_time = (alert['acknowledged_at'] -
alert['timestamp']).total_seconds() / 60
                response_times.append(response_time)

        return np.mean(response_times) if response_times else None

# Example usage
alert_manager = AlertManager({})

# Register notification channels
alert_manager.register_notification_channel('email', {
    'sender_email': 'risk@company.com',
    'recipients': ['trader@company.com', 'manager@company.com'],
    'smtp_server': 'smtp.company.com'
})

alert_manager.register_notification_channel('slack', {
    'webhook_url': 'https://hooks.slack.com/...',
    'channel': '#risk-alerts'
})

# Register alert rules
alert_manager.register_alert_rule({
    'name': 'high_var',
    'condition': lambda data: data.get('var', {}).get('var_95', 0) >
10000,
    'severity': 'HIGH',
    'cooldown_minutes': 60,
```

```
    'channels': ['email', 'slack'],
    'message': 'VaR (95%) exceeds $10,000 threshold'
})

alert_manager.register_alert_rule({
    'name': 'high_concentration',
    'condition': lambda data: data.get('portfolio',
{}).get('max_concentration', 0) > 0.5,
    'severity': 'MEDIUM',
    'cooldown_minutes': 30,
    'channels': ['slack'],
    'message': 'Position concentration exceeds 50%'
})

alert_manager.register_alert_rule({
    'name': 'risk_rating_high',
    'condition': lambda data: data.get('portfolio',
{}).get('risk_rating', '') == 'HIGH',
    'severity': 'HIGH',
    'cooldown_minutes': 120,
    'channels': ['email', 'slack', 'sms'],
    'message': 'Portfolio risk rating is HIGH'
})

# Register escalation rules
alert_manager.register_escalation_rule({
    'alert_type': 'high_var',
    'condition': lambda alert: alert['severity'] == 'HIGH',
    'after_minutes': 30,
    'escalate_to': ['compliance'],
    'new_severity': 'CRITICAL'
})

# Test alert processing
sample_risk_data = {
    'var': {'var_95': 15000},
    'portfolio': {
        'max_concentration': 0.6,
        'risk_rating': 'HIGH'
    }
}
```

```
print("Alert Management System:")
print("=" * 40)
print("Registered alert rules:")
for rule in alert_manager.alert_rules:
    print(f"- {rule['name']} ({rule['severity']})")

print("\nNotification channels:")
for channel in alert_manager.notification_channels.keys():
    print(f"- {channel}")

print("\nEscalation rules:")
for rule in alert_manager.escalation_rules:
    print(f"- {rule['alert_type']} -> {rule['new_severity']} after
{rule['after_minutes']} minutes")
```

Machine Learning Risk Prediction

5. ML-Based Risk Models

```
from sklearn.ensemble import IsolationForest, RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import joblib

class MLRiskPredictor:
    """
    Machine Learning-based risk prediction system
    """
    def __init__(self):
        self.models = {}
        self.scalers = {}
        self.feature_importance = {}
        self.training_data = []
        self.prediction_history = []

    def prepare_features(self, risk_data: Dict, market_data: Dict) ->
np.ndarray:
        """
        Prepare features for ML models
        """
        features = []

        # Risk features
        risk_portfolio = risk_data.get('portfolio', {})
        features.extend([
            risk_portfolio.get('total_risk', 0),
            risk_portfolio.get('max_concentration', 0),
            risk_portfolio.get('correlation_risk', 0),
            risk_data.get('total_exposure', 0),
            risk_data.get('var', {}).get('var_95', 0),
            risk_data.get('var', {}).get('var_99', 0)
        ])

        # Market features
        features.extend([
            market_data.get('volatility', 0),
            market_data.get('correlation', 0),
            market_data.get('liquidity_ratio', 1),
            market_data.get('gas_price', 0),
            market_data.get('volume_ratio', 1)
        ])
```

```
        # Time-based features
        current_hour = datetime.now().hour
        features.extend([
            np.sin(2 * np.pi * current_hour / 24),  # Hourly pattern
            np.cos(2 * np.pi * current_hour / 24),
            current_hour / 24  # Normalized hour
        ])

        return np.array(features).reshape(1, -1)

    def train_anomaly_detector(self, historical_data: List[Dict]):
        """
        Train anomaly detection model
        """
        # Prepare training data
        X = []
        for data_point in historical_data:
            features = self.prepare_features(data_point['risk_data'],
data_point['market_data'])
            X.append(features.flatten())

        X = np.array(X)

        # Train Isolation Forest for anomaly detection
        self.models['anomaly_detector'] = IsolationForest(
            contamination=0.1,  # Expect 10% anomalies
            random_state=42
        )

        # Train model
        self.models['anomaly_detector'].fit(X)

        # Calculate feature importance (simplified)
        self.feature_importance['anomaly_detector'] = {
            'var_95': 0.15,
            'max_concentration': 0.20,
            'volatility': 0.18,
            'correlation': 0.12,
            'gas_price': 0.10,
            'liquidity_ratio': 0.08,
            'total_exposure': 0.17
```

```
        }

        print("Anomaly detection model trained successfully")

    def train_risk_classifier(self, historical_data: List[Dict],
labels: List[str]):
        """
        Train risk level classifier
        """
        # Prepare features and labels
        X = []
        for data_point in historical_data:
            features = self.prepare_features(data_point['risk_data'],
data_point['market_data'])
            X.append(features.flatten())

        X = np.array(X)

        # Encode labels
        label_encoder = {label: i for i, label in
enumerate(set(labels))}
        y = [label_encoder[label] for label in labels]

        # Scale features
        self.scalers['risk_classifier'] = StandardScaler()
        X_scaled = self.scalers['risk_classifier'].fit_transform(X)

        # Train Random Forest classifier
        self.models['risk_classifier'] = RandomForestClassifier(
            n_estimators=100,
            random_state=42,
            max_depth=10
        )

        X_train, X_test, y_train, y_test = train_test_split(
            X_scaled, y, test_size=0.2, random_state=42
        )

        self.models['risk_classifier'].fit(X_train, y_train)

        # Calculate accuracy
        accuracy = self.models['risk_classifier'].score(X_test, y_test)
```

```
        print(f"Risk classifier trained with {accuracy:.2%} accuracy")

        # Feature importance
        feature_names = ['var_95', 'max_concentration',
'correlation_risk', 'total_exposure',
                         'var_99', 'volatility', 'correlation',
'liquidity_ratio',
                         'gas_price', 'volume_ratio', 'hour_sin',
'hour_cos', 'hour_norm']

        importance =
self.models['risk_classifier'].feature_importances_
        self.feature_importance['risk_classifier'] =
dict(zip(feature_names, importance))

    def predict_anomaly(self, risk_data: Dict, market_data: Dict) ->
Dict:
        """
        Predict if current state is anomalous
        """
        if 'anomaly_detector' not in self.models:
            return {'error': 'Anomaly detector not trained'}

        # Prepare features
        features = self.prepare_features(risk_data, market_data)

        # Predict anomaly
        anomaly_score =
self.models['anomaly_detector'].decision_function(features)[0]
        is_anomaly = self.models['anomaly_detector'].predict(features)
[0] == -1

        # Calculate confidence
        confidence = abs(anomaly_score)

        prediction = {
            'is_anomaly': bool(is_anomaly),
            'anomaly_score': float(anomaly_score),
            'confidence': float(confidence),
            'risk_level': 'HIGH' if is_anomaly else 'NORMAL',
            'timestamp': datetime.now()
        }
```

```
        # Store prediction
        self.prediction_history.append(prediction)

        return prediction

    def predict_risk_level(self, risk_data: Dict, market_data: Dict) ->
Dict:
        """
        Predict risk level classification
        """
        if 'risk_classifier' not in self.models:
            return {'error': 'Risk classifier not trained'}

        # Prepare features
        features = self.prepare_features(risk_data, market_data)

        # Scale features
        features_scaled =
self.scalers['risk_classifier'].transform(features)

        # Predict
        prediction =
self.models['risk_classifier'].predict(features_scaled)[0]
        probabilities =
self.models['risk_classifier'].predict_proba(features_scaled)[0]

        # Get class names (this would be configured based on training)
        class_names = ['LOW', 'MEDIUM', 'HIGH', 'CRITICAL']

        prediction_dict = {
            'predicted_risk_level': class_names[prediction],
            'probability': float(probabilities[prediction]),
            'confidence': float(max(probabilities)),
            'all_probabilities': dict(zip(class_names,
probabilities.astype(float))),
            'timestamp': datetime.now()
        }

        # Store prediction
        self.prediction_history.append(prediction_dict)
```

```
        return prediction_dict

    def predict_var_forecast(self, historical_var_data: List[float],
periods: int = 7) -> Dict:
        """
        Forecast VaR using time series analysis (simplified)
        """
        if len(historical_var_data) < 30:
            return {'error': 'Insufficient historical data'}

        # Simple moving average forecast
        window = min(30, len(historical_var_data) // 4)
        recent_data = historical_var_data[-window:]

        # Calculate trends
        trend = np.mean(np.diff(recent_data))
        volatility = np.std(recent_data)

        # Generate forecasts
        forecasts = []
        last_value = recent_data[-1]

        for i in range(periods):
            # Add trend with random walk
            forecast = last_value + trend + np.random.normal(0,
volatility * 0.1)
            forecast = max(0, forecast)  # VaR cannot be negative
            forecasts.append(forecast)
            last_value = forecast

        return {
            'forecasts': forecasts,
            'forecast_horizon': periods,
            'trend': float(trend),
            'volatility': float(volatility),
            'confidence_interval': {
                'lower': [f - 1.96 * volatility for f in forecasts],
                'upper': [f + 1.96 * volatility for f in forecasts]
            }
        }

    def get_model_performance(self) -> Dict:
```

```
        """
        Get model performance metrics
        """
        if not self.models:
            return {'error': 'No models trained'}

        performance = {}

        # Recent prediction statistics
        recent_predictions = self.prediction_history[-100:] if
len(self.prediction_history) > 100 else self.prediction_history

        if recent_predictions:
            anomaly_predictions = [p for p in recent_predictions if
'is_anomaly' in p]
            if anomaly_predictions:
                anomaly_rate = sum(1 for p in anomaly_predictions if
p['is_anomaly']) / len(anomaly_predictions)
                performance['recent_anomaly_rate'] = anomaly_rate

            risk_predictions = [p for p in recent_predictions if
'predicted_risk_level' in p]
            if risk_predictions:
                risk_distribution = {}
                for pred in risk_predictions:
                    level = pred['predicted_risk_level']
                    risk_distribution[level] =
risk_distribution.get(level, 0) + 1

                performance['risk_level_distribution'] = {
                    level: count / len(risk_predictions)
                    for level, count in risk_distribution.items()
                }

        return {
            'models_trained': list(self.models.keys()),
            'total_predictions': len(self.prediction_history),
            'performance_metrics': performance,
            'feature_importance': self.feature_importance
        }

    def save_models(self, filepath: str):
```

```
        """
        Save trained models to disk
        """
        model_data = {
            'models': self.models,
            'scalers': self.scalers,
            'feature_importance': self.feature_importance
        }

        joblib.dump(model_data, filepath)
        print(f"Models saved to {filepath}")

    def load_models(self, filepath: str):
        """
        Load trained models from disk
        """
        model_data = joblib.load(filepath)

        self.models = model_data['models']
        self.scalers = model_data['scalers']
        self.feature_importance = model_data['feature_importance']

        print(f"Models loaded from {filepath}")

# Example usage and demonstration
ml_predictor = MLRiskPredictor()

# Generate sample training data
print("Training ML Risk Predictor:")
print("=" * 40)

# Simulate historical data
historical_data = []
historical_labels = []

for i in range(500):  # 500 training samples
    # Simulate risk and market data
    risk_data = {
        'portfolio': {
            'total_risk': np.random.uniform(5000, 20000),
            'max_concentration': np.random.uniform(0.2, 0.8),
            'correlation_risk': np.random.uniform(0.1, 0.9)
```

```
        },
        'total_exposure': np.random.uniform(50000, 200000),
        'var': {
            'var_95': np.random.uniform(5000, 25000),
            'var_99': np.random.uniform(10000, 40000)
        }
    }

    market_data = {
        'volatility': np.random.uniform(0.01, 0.08),
        'correlation': np.random.uniform(0.2, 0.9),
        'liquidity_ratio': np.random.uniform(0.5, 2.0),
        'gas_price': np.random.uniform(10, 200),
        'volume_ratio': np.random.uniform(0.5, 3.0)
    }

    # Generate labels based on risk levels
    total_risk = risk_data['portfolio']['total_risk']
    var_95 = risk_data['var']['var_95']
    volatility = market_data['volatility']

    risk_score = (total_risk / 20000) * 0.4 + (var_95 / 25000) * 0.4 +
(volatility / 0.08) * 0.2

    if risk_score > 0.7:
        label = 'CRITICAL'
    elif risk_score > 0.5:
        label = 'HIGH'
    elif risk_score > 0.3:
        label = 'MEDIUM'
    else:
        label = 'LOW'

    historical_data.append({'risk_data': risk_data, 'market_data':
market_data})
    historical_labels.append(label)

# Train models
ml_predictor.train_anomaly_detector(historical_data)
ml_predictor.train_risk_classifier(historical_data, historical_labels)

# Test predictions
```

```
print("\nTesting ML Predictions:")
print("-" * 30)

# Test anomaly detection
test_risk_data = {
    'portfolio': {'total_risk': 18000, 'max_concentration': 0.75,
'correlation_risk': 0.8},
    'total_exposure': 150000,
    'var': {'var_95': 22000, 'var_99': 35000}
}

test_market_data = {
    'volatility': 0.06, 'correlation': 0.85, 'liquidity_ratio': 0.6,
    'gas_price': 150, 'volume_ratio': 2.5
}

anomaly_pred = ml_predictor.predict_anomaly(test_risk_data,
test_market_data)
risk_pred = ml_predictor.predict_risk_level(test_risk_data,
test_market_data)

print(f"Anomaly Detection:")
print(f"  Is Anomaly: {anomaly_pred['is_anomaly']}")
print(f"  Anomaly Score: {anomaly_pred['anomaly_score']:.3f}")
print(f"  Confidence: {anomaly_pred['confidence']:.3f}")

print(f"\nRisk Level Prediction:")
print(f"  Predicted Level: {risk_pred['predicted_risk_level']}")
print(f"  Probability: {risk_pred['probability']:.3f}")
print(f"  Confidence: {risk_pred['confidence']:.3f}")

# Test VaR forecasting
historical_var = [np.random.uniform(8000, 15000) for _ in range(60)]
var_forecast = ml_predictor.predict_var_forecast(historical_var,
periods=7)

print(f"\nVaR Forecast (7 days):")
for i, forecast in enumerate(var_forecast['forecasts']):
    print(f"  Day {i+1}: ${forecast:,.0f}")

# Model performance
performance = ml_predictor.get_model_performance()
```

```
print(f"\nModel Performance:")
print(f"  Models Trained: {performance['models_trained']}")
print(f"  Total Predictions: {performance['total_predictions']}")

if 'risk_level_distribution' in performance['performance_metrics']:
    print(f"  Risk Level Distribution:")
    for level, percentage in performance['performance_metrics']
['risk_level_distribution'].items():
        print(f"    {level}: {percentage:.1%}")
```

API and Microservices Architecture

6. Risk Management API

```
from flask import Flask, request, jsonify
from flask_restful import Api, Resource
import asyncio
from threading import Thread

class RiskManagementAPI:
    """
    RESTful API for risk management system
    """
    def __init__(self, risk_engine, alert_manager, ml_predictor):
        self.app = Flask(__name__)
        self.api = Api(self.app)
        self.risk_engine = risk_engine
        self.alert_manager = alert_manager
        self.ml_predictor = ml_predictor

        self.setup_routes()

    def setup_routes(self):
        """
        Setup API routes
        """

        class RiskOverview(Resource):
            def get(self):
                """Get current risk overview"""
                try:
                    # Get latest risk data
                    risk_data = getattr(self, 'latest_risk_data', {})

                    if not risk_data:
                        return {'error': 'No risk data available'}, 404

                    overview = {
                        'timestamp': datetime.now().isoformat(),
                        'portfolio_risk_rating':
risk_data.get('portfolio', {}).get('risk_rating', 'UNKNOWN'),
                        'total_exposure':
risk_data.get('total_exposure', 0),
                        'var_95': risk_data.get('var',
{}).get('var_95', 0),
                        'var_99': risk_data.get('var',
```

```
{}).get('var_99', 0),
                        'max_concentration': risk_data.get('portfolio',
{}).get('max_concentration', 0),
                        'correlation_risk': risk_data.get('portfolio',
{}).get('correlation_risk', 0),
                        'active_positions':
len(risk_data.get('positions', {})),
                        'alerts': len([a for a in
self.alert_manager.alert_history
                                       if not a.get('resolved', False)])
                    }

                    return overview, 200

                except Exception as e:
                    return {'error': str(e)}, 500

        class PositionRisk(Resource):
            def get(self):
                """Get position-level risk data"""
                try:
                    risk_data = getattr(self, 'latest_risk_data', {})

                    if not risk_data:
                        return {'error': 'No risk data available'}, 404

                    positions = risk_data.get('positions', {})
                    position_list = []

                    for pos_id, pos_data in positions.items():
                        position_list.append({
                            'id': pos_id,
                            'type': pos_data.get('type', 'unknown'),
                            'risk_amount': pos_data.get('final_risk',
0),
                            'risk_percentage':
pos_data.get('risk_percentage', 0),
                            'exposure': pos_data.get('value', 0),
                            'confidence': pos_data.get('confidence', 0)
                        })

                    return {
```

```
                        'positions': position_list,
                        'total_positions': len(position_list),
                        'total_exposure': sum(p['exposure'] for p in
position_list),
                        'timestamp': datetime.now().isoformat()
                    }, 200

                except Exception as e:
                    return {'error': str(e)}, 500

        class VarAnalysis(Resource):
            def get(self):
                """Get VaR analysis"""
                try:
                    risk_data = getattr(self, 'latest_risk_data', {})

                    if not risk_data:
                        return {'error': 'No risk data available'}, 404

                    var_data = risk_data.get('var', {})

                    return {
                        'var_95': var_data.get('var_95', 0),
                        'var_99': var_data.get('var_99', 0),
                        'expected_shortfall_95':
var_data.get('expected_shortfall_95', 0),
                        'expected_shortfall_99':
var_data.get('expected_shortfall_99', 0),
                        'confidence_level':
var_data.get('confidence_level', 'Unknown'),
                        'timestamp': datetime.now().isoformat()
                    }, 200

                except Exception as e:
                    return {'error': str(e)}, 500

        class Alerts(Resource):
            def get(self):
                """Get alerts"""
                try:
                    limit = request.args.get('limit', 50, type=int)
                    severity = request.args.get('severity')
```

```
                    unresolved_only =
request.args.get('unresolved_only', False, type=bool)

                    alerts = self.alert_manager.alert_history.copy()

                    # Filter by severity if specified
                    if severity:
                        alerts = [a for a in alerts if a['severity'] ==
severity]

                    # Filter unresolved if specified
                    if unresolved_only:
                        alerts = [a for a in alerts if not
a.get('resolved', False)]

                    # Sort by timestamp (most recent first)
                    alerts.sort(key=lambda x: x['timestamp'],
reverse=True)

                    # Limit results
                    alerts = alerts[:limit]

                    return {
                        'alerts': alerts,
                        'total_alerts': len(alerts),
                        'filters': {
                            'severity': severity,
                            'unresolved_only': unresolved_only,
                            'limit': limit
                        }
                    }, 200

                except Exception as e:
                    return {'error': str(e)}, 500

            def post(self):
                """Acknowledge or resolve alert"""
                try:
                    data = request.get_json()
                    alert_id = data.get('alert_id')
                    action = data.get('action')  # 'acknowledge' or
'resolve'
```

```
                    if not alert_id or action not in ['acknowledge',
'resolve']:
                        return {'error': 'Invalid request'}, 400

                    # Find and update alert
                    for alert in self.alert_manager.alert_history:
                        if alert['id'] == alert_id:
                            if action == 'acknowledge':
                                alert['acknowledged'] = True
                                alert['acknowledged_at'] =
datetime.now()
                            elif action == 'resolve':
                                alert['resolved'] = True
                                alert['resolved_at'] = datetime.now()

                            return {'message': f'Alert {action}d
successfully'}, 200

                    return {'error': 'Alert not found'}, 404

                except Exception as e:
                    return {'error': str(e)}, 500

        class MLPredictions(Resource):
            def get(self):
                """Get ML-based risk predictions"""
                try:
                    risk_data = getattr(self, 'latest_risk_data', {})
                    market_data = getattr(self, 'latest_market_data',
{})

                    if not risk_data or not market_data:
                        return {'error': 'No data available for
prediction'}, 404

                    predictions = {}

                    # Anomaly detection
                    if 'anomaly_detector' in self.ml_predictor.models:
                        anomaly_pred =
self.ml_predictor.predict_anomaly(risk_data, market_data)
```

```
                        predictions['anomaly_detection'] = anomaly_pred

                    # Risk level classification
                    if 'risk_classifier' in self.ml_predictor.models:
                        risk_pred =
self.ml_predictor.predict_risk_level(risk_data, market_data)
                        predictions['risk_classification'] = risk_pred

                    return predictions, 200

                except Exception as e:
                    return {'error': str(e)}, 500

        class StressTests(Resource):
            def get(self):
                """Get stress test results"""
                try:
                    risk_data = getattr(self, 'latest_risk_data', {})

                    if not risk_data:
                        return {'error': 'No risk data available'}, 404

                    stress_tests = risk_data.get('stress_tests', {})

                    return {
                        'stress_tests': stress_tests,
                        'timestamp': datetime.now().isoformat()
                    }, 200

                except Exception as e:
                    return {'error': str(e)}, 500

        class SystemHealth(Resource):
            def get(self):
                """Get system health status"""
                try:
                    health_status = {
                        'timestamp': datetime.now().isoformat(),
                        'services': {
                            'risk_engine': 'healthy',
                            'alert_manager': 'healthy',
                            'ml_predictor': 'healthy' if
```

```
self.ml_predictor.models else 'unavailable',
                            'data_pipeline': 'healthy'
                        },
                        'metrics': {
                            'total_predictions':
len(self.ml_predictor.prediction_history),
                            'active_alerts': len([a for a in
self.alert_manager.alert_history
                                                  if not
a.get('resolved', False)]),
                            'models_loaded':
len(self.ml_predictor.models),
                            'alert_rules':
len(self.alert_manager.alert_rules)
                        }
                    }

                    return health_status, 200

                except Exception as e:
                    return {'error': str(e)}, 500

        # Register resources
        self.api.add_resource(RiskOverview, '/api/v1/risk/overview')
        self.api.add_resource(PositionRisk, '/api/v1/risk/positions')
        self.api.add_resource(VarAnalysis, '/api/v1/risk/var')
        self.api.add_resource(Alerts, '/api/v1/alerts')
        self.api.add_resource(MLPredictions, '/api/v1/predictions')
        self.api.add_resource(StressTests, '/api/v1/stress-tests')
        self.api.add_resource(SystemHealth, '/api/v1/health')

    def update_data(self, risk_data, market_data):
        """
        Update latest data for API responses
        """
        # Store latest data (in production, this would be from a real
data pipeline)
        setattr(RiskOverview, 'latest_risk_data', risk_data)
        setattr(PositionRisk, 'latest_risk_data', risk_data)
        setattr(VarAnalysis, 'latest_risk_data', risk_data)
        setattr(MLPredictions, 'latest_risk_data', risk_data)
        setattr(MLPredictions, 'latest_market_data', market_data)
```

```
        setattr(StressTests, 'latest_risk_data', risk_data)

    def run(self, host='0.0.0.0', port=5000, debug=False):
        """
        Run the API server
        """
        self.app.run(host=host, port=port, debug=debug)

# Example API server setup
def create_risk_api_server(risk_engine, alert_manager, ml_predictor):
    """
    Create and configure risk management API server
    """
    api_server = RiskManagementAPI(risk_engine, alert_manager,
ml_predictor)

    # Add error handlers
    @api_server.app.errorhandler(404)
    def not_found(error):
        return {'error': 'Endpoint not found'}, 404

    @api_server.app.errorhandler(500)
    def internal_error(error):
        return {'error': 'Internal server error'}, 500

    @api_server.app.before_request
    def before_request():
        # Add authentication/authorization if needed
        pass

    return api_server

# Example usage
print("Risk Management API Server:")
print("=" * 40)
print("API Endpoints:")
print("GET  /api/v1/risk/overview    - Get risk overview")
print("GET  /api/v1/risk/positions   - Get position risk data")
print("GET  /api/v1/risk/var         - Get VaR analysis")
print("GET  /api/v1/alerts          - Get alerts")
print("POST /api/v1/alerts          - Acknowledge/resolve alerts")
print("GET  /api/v1/predictions     - Get ML predictions")
```

```
print("GET  /api/v1/stress-tests    - Get stress test results")
print("GET  /api/v1/health          - Get system health")

print("\nAPI Features:")
print("- Real-time risk monitoring")
print("- Position-level risk analysis")
print("- VaR calculations and forecasting")
print("- Alert management system")
print("- Machine learning predictions")
print("- Stress testing")
print("- System health monitoring")

# Note: In a real implementation, you would:
# 1. Set up proper database connections
# 2. Add authentication and authorization
# 3. Implement rate limiting
# 4. Add comprehensive logging
# 5. Set up monitoring and alerting for the API itself
# 6. Use production WSGI server like Gunicorn
```

Key Takeaways

1. **Modular architecture enables scalability** - Components should be independent and replaceable.
2. **Real-time processing is crucial** - MEV risks change in microseconds, systems must keep up.
3. **Multiple data sources improve accuracy** - Corroborate risk signals across different data feeds.
4. **Machine learning enhances prediction** - ML models can identify complex risk patterns.
5. **Comprehensive APIs enable integration** - Systems must communicate with other components.
6. **Alert fatigue is a real problem** - Smart filtering and escalation prevent alert overload.
7. **Audit trails are essential** - Complete logging for compliance and debugging.
8. **System health monitoring matters** - Monitor the monitors to ensure reliability.

Summary

Building a comprehensive risk management system for MEV trading requires integrating multiple components: data pipelines, calculation engines, monitoring dashboards, alert systems, machine learning models, and APIs. The key is to design a system that is both powerful enough to handle the complexity of MEV risks and fast enough to respond to rapidly changing conditions.

Remember that the best risk management system is one that operates seamlessly in the background, providing protection without interfering with your trading operations. Focus on building robust foundations that can scale with your business and adapt to changing market conditions.

The ultimate goal is to create a system that not only monitors and reports risk but actively helps you make better trading decisions by providing timely, accurate, and actionable risk insights.