Module 5: Monitoring & Analytics

Master Prometheus, Grafana, and custom metrics for comprehensive MEV system observability

Course Overview

Duration: 140 minutes
Level: Advanced
Prerequisites: Modules 1-4, understanding of system monitoring and observability
Certificate: Available upon completion

Learning Objectives

By the end of this module, you will be able to:
- Design comprehensive monitoring architectures for MEV systems
- Implement custom metrics collection and analysis for trading performance
- Build alerting systems for critical MEV opportunities and system health
- Create interactive dashboards for real-time MEV monitoring
- Implement distributed tracing for latency analysis
- Design anomaly detection systems for trading anomalies

Table of Contents

---

1. Monitoring Architecture Design

MEV Monitoring Stack Architecture

```
# monitoring/monitoring-stack.yml
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring
---
# Prometheus Configuration
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: monitoring
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
      evaluation_interval: 15s
      external_labels:
        cluster: 'mev-system-prod'
        replica: 'prometheus-1'

    rule_files:
      - "/etc/prometheus/rules/*.yml"

    scrape_configs:
      # Prometheus itself
      - job_name: 'prometheus'
        static_configs:
          - targets: ['localhost:9090']
        scrape_interval: 5s

      # Kubernetes API Server
      - job_name: 'kubernetes-apiservers'
        kubernetes_sd_configs:
        - role: endpoints
        scheme: https
        tls_config:
          ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/kubernetes.io/
serviceaccount/token
        relabel_configs:
        - source_labels: [__meta_kubernetes_namespace,
```

```
__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
          action: keep
          regex: default;kubernetes;https

      # Kubernetes Nodes
      - job_name: 'kubernetes-nodes'
        kubernetes_sd_configs:
        - role: node
        scheme: https
        tls_config:
          ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/kubernetes.io/
serviceaccount/token
        relabel_configs:
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)
        - target_label: __address__
          replacement: kubernetes.default.svc:443
        - source_labels: [__meta_kubernetes_node_name]
          regex: (.+)
          target_label: __metrics_path__
          replacement: /api/v1/nodes/${1}/proxy/metrics

      # Kubernetes Pods
      - job_name: 'kubernetes-pods'
        kubernetes_sd_configs:
        - role: pod
        relabel_configs:
        - source_labels:
[__meta_kubernetes_pod_annotation_prometheus_io_scrape]
          action: keep
          regex: true
        - source_labels:
[__meta_kubernetes_pod_annotation_prometheus_io_path]
          action: replace
          target_label: __metrics_path__
          regex: (.+)
        - source_labels: [__address__,
__meta_kubernetes_pod_annotation_prometheus_io_port]
          action: replace
          regex: ([^:]+)(?::\d+)?;(\d+)
          replacement: <span class="math-inline" style="display:
```

```
inline;"><math xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mn>1</mn><mi>:</mi></mrow></math></span>2
          target_label: __address__
        - action: labelmap
          regex: __meta_kubernetes_pod_label_(.+)
        - source_labels: [__meta_kubernetes_namespace]
          action: replace
          target_label: kubernetes_namespace
        - source_labels: [__meta_kubernetes_pod_name]
          action: replace
          target_label: kubernetes_pod_name

      # MEV Strategy API
      - job_name: 'mev-strategy-api'
        kubernetes_sd_configs:
        - role: endpoints
          namespaces:
            names:
            - mev-system
        relabel_configs:
        - source_labels: [__meta_kubernetes_service_name]
          action: keep
          regex: mev-strategy-api
        - source_labels: [__meta_kubernetes_endpoint_port_name]
          action: keep
          regex: metrics

      # Blockchain Fetcher
      - job_name: 'blockchain-fetcher'
        kubernetes_sd_configs:
        - role: endpoints
          namespaces:
            names:
            - mev-system
        relabel_configs:
        - source_labels: [__meta_kubernetes_service_name]
          action: keep
          regex: blockchain-fetcher

      # Trade Executor
      - job_name: 'trade-executor'
        kubernetes_sd_configs:
```

```
        - role: endpoints
          namespaces:
            names:
            - mev-system
        relabel_configs:
        - source_labels: [__meta_kubernetes_service_name]
          action: keep
          regex: trade-executor

      # External services
      - job_name: 'node-exporter'
        static_configs:
        - targets: ['node-exporter:9100']

      - job_name: 'redis-exporter'
        static_configs:
        - targets: ['redis-exporter:9121']

      - job_name: 'postgres-exporter'
        static_configs:
        - targets: ['postgres-exporter:9187']

  # Custom rules for MEV alerts
  mev-alerts.yml: |
    groups:
    - name: mev_system_alerts
      rules:
      - alert: HighOpportunityMissRate
        expr: rate(mev_opportunities_missed_total[5m]) /
rate(mev_opportunities_detected_total[5m]) > 0.1
        for: 2m
        labels:
          severity: warning
          team: mev-operations
        annotations:
          summary: "High MEV opportunity miss rate detected"
          description: "Miss rate is {{ $value | humanizePercentage }}
over the last 5 minutes"

      - alert: LowTradingProfitability
        expr: rate(mev_trading_profit_total[1h]) < 10
        for: 5m
```

```
        labels:
          severity: warning
          team: mev-operations
        annotations:
          summary: "Low trading profitability detected"
          description: "Profit rate is <span class="math-inline"
style="font-family: serif;">{{</span>value }} per hour over the last
hour"

      - alert: HighLatencyTrading
        expr: histogram_quantile(0.95,
rate(mev_trade_execution_duration_seconds_bucket[5m])) > 5
        for: 1m
        labels:
          severity: critical
          team: mev-operations
        annotations:
          summary: "High trade execution latency"
          description: "95th percentile execution time is {{ $value }}
s"

      - alert: LiquidationOpportunityMissed
        expr: increase(mev_liquidation_opportunities_missed_total[5m])
> 0
        for: 0m
        labels:
          severity: critical
          team: mev-operations
        annotations:
          summary: "Liquidation opportunity was missed"
          description: "{{ $value }} liquidation opportunities missed
in the last 5 minutes"

      - alert: ArbitrageOpportunityDeclined
        expr: increase(mev_arbitrage_opportunities_declined_total[1m])
> 5
        for: 1m
        labels:
          severity: warning
          team: mev-operations
        annotations:
          summary: "High number of declined arbitrage opportunities"
```

```
          description: "{{ $value }} arbitrage opportunities declined
in the last minute"

      - alert: DatabaseConnectionFailure
        expr: up{job="mev-strategy-api"} == 0
        for: 1m
        labels:
          severity: critical
          team: mev-operations
        annotations:
          summary: "Database connection failure"
          description: "MEV Strategy API is down"

      - alert: HighGasPrice
        expr: mev_current_gas_price > 200000000000
        for: 5m
        labels:
          severity: warning
          team: mev-operations
        annotations:
          summary: "High gas price detected"
          description: "Current gas price is {{ $value | humanize }}
wei"

      - alert: LowLiquidity
        expr: mev_dex_liquidity < 1000000
        for: 10m
        labels:
          severity: warning
          team: mev-operations
        annotations:
          summary: "Low DEX liquidity detected"
          description: "DEX liquidity is below $1M threshold"

  # Recording rules for performance
  recording-rules.yml: |
    groups:
    - name: mev_performance_recording
      interval: 30s
      rules:
      - record: mev:opportunity_rate_5m
        expr: rate(mev_opportunities_detected_total[5m])
```

```
      - record: mev:profit_rate_1h
        expr: rate(mev_trading_profit_total[1h])

      - record: mev:success_rate_5m
        expr: |
          (
            rate(mev_trades_completed_successfully_total[5m]) /
            rate(mev_trades_attempted_total[5m])
          ) * 100

      - record: mev:avg_execution_time_5m
        expr: histogram_quantile(0.50,
rate(mev_trade_execution_duration_seconds_bucket[5m]))

      - record: mev:gas_efficiency_5m
        expr: rate(mev_gas_used_total[5m]) /
rate(mev_trades_completed_successfully_total[5m])
```

Metrics Collection Framework

```
# monitoring/metrics_collector.py
from prometheus_client import Counter, Histogram, Gauge, Summary,
CollectorRegistry, generate_latest
from prometheus_client.core import REGISTRY
from typing import Dict, List, Optional, Any
import time
import logging
from dataclasses import dataclass
from enum import Enum
import asyncio

class MetricType(Enum):
    COUNTER = "counter"
    GAUGE = "gauge"
    HISTOGRAM = "histogram"
    SUMMARY = "summary"

@dataclass
class MetricConfig:
    """Configuration for a custom metric"""
    name: str
    metric_type: MetricType
    description: str
    labels: Optional[List[str]] = None
    buckets: Optional[List[float]] = None

class MEVMetricsCollector:
    """Custom MEV metrics collector with Prometheus integration"""

    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.registry = CollectorRegistry()

        # Initialize all metrics
        self._init_mev_metrics()

        # Custom registries for different components
        self.component_registries: Dict[str, CollectorRegistry] = {}

        # Metrics cache for real-time values
        self.metrics_cache: Dict[str, Any] = {}
```

```
    def _init_mev_metrics(self):
        """Initialize all MEV-specific metrics"""

        # Opportunity Detection Metrics
        self.opportunities_detected = Counter(
            'mev_opportunities_detected_total',
            'Total number of MEV opportunities detected',
            ['opportunity_type', 'protocol'],
            registry=self.registry
        )

        self.opportunities_missed = Counter(
            'mev_opportunities_missed_total',
            'Total number of MEV opportunities missed',
            ['opportunity_type', 'reason'],
            registry=self.registry
        )

        self.opportunity_detection_latency = Histogram(
            'mev_opportunity_detection_duration_seconds',
            'Time taken to detect MEV opportunities',
            ['opportunity_type'],
            buckets=[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0],
            registry=self.registry
        )

        # Trading Performance Metrics
        self.trades_attempted = Counter(
            'mev_trades_attempted_total',
            'Total number of trades attempted',
            ['strategy_type', 'status'],
            registry=self.registry
        )

        self.trades_succeeded = Counter(
            'mev_trades_completed_successfully_total',
            'Total number of successful trades',
            ['strategy_type', 'protocol'],
            registry=self.registry
        )

        self.trading_profit = Counter(
```

```
            'mev_trading_profit_total',
            'Total trading profit in USD',
            ['strategy_type'],
            registry=self.registry
        )

        self.trade_execution_duration = Histogram(
            'mev_trade_execution_duration_seconds',
            'Time taken to execute trades',
            ['strategy_type', 'protocol'],
            buckets=[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 2.0, 5.0,
10.0],
            registry=self.registry
        )

        self.slippage_observed = Histogram(
            'mev_slippage_observed',
            'Slippage observed in trades',
            ['strategy_type', 'protocol'],
            buckets=[0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05],
            registry=self.registry
        )

        # Blockchain Metrics
        self.block_processing_time = Histogram(
            'mev_block_processing_duration_seconds',
            'Time taken to process blocks',
            ['network'],
            buckets=[0.01, 0.05, 0.1, 0.5, 1.0, 2.0, 5.0],
            registry=self.registry
        )

        self.gas_price = Gauge(
            'mev_current_gas_price',
            'Current gas price in wei',
            ['network'],
            registry=self.registry
        )

        self.dex_liquidity = Gauge(
            'mev_dex_liquidity',
            'Total liquidity in DEX pools in USD',
```

```
            ['protocol', 'token_pair'],
            registry=self.registry
        )

        # System Performance Metrics
        self.api_request_duration = Histogram(
            'mev_api_request_duration_seconds',
            'API request duration',
            ['endpoint', 'method', 'status_code'],
            buckets=[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0],
            registry=self.registry
        )

        self.active_connections = Gauge(
            'mev_active_connections',
            'Number of active connections',
            ['connection_type'],
            registry=self.registry
        )

        self.memory_usage = Gauge(
            'mev_memory_usage_bytes',
            'Memory usage in bytes',
            ['component'],
            registry=self.registry
        )

        self.cpu_usage = Gauge(
            'mev_cpu_usage_percent',
            'CPU usage percentage',
            ['component'],
            registry=self.registry
        )

        # Strategy-specific metrics
        self.arbitrage_profit = Counter(
            'mev_arbitrage_profit_total',
            'Total arbitrage profit in USD',
            ['protocol_a', 'protocol_b'],
            registry=self.registry
        )
```

```
        self.liquidation_bonus = Counter(
            'mev_liquidation_bonus_total',
            'Total liquidation bonus received',
            ['protocol'],
            registry=self.registry
        )

        # Custom business metrics
        self.daily_pnl = Gauge(
            'mev_daily_pnl_usd',
            'Daily profit and loss in USD',
            registry=self.registry
        )

        self.win_rate = Gauge(
            'mev_win_rate_percent',
            'Percentage of profitable trades',
            ['strategy_type'],
            registry=self.registry
        )

        self.sharpe_ratio = Gauge(
            'mev_sharpe_ratio',
            'Sharpe ratio of trading strategy',
            registry=self.registry
        )

        self.max_drawdown = Gauge(
            'mev_max_drawdown_percent',
            'Maximum drawdown percentage',
            registry=self.registry
        )

    def record_opportunity_detected(self, opportunity_type: str,
protocol: str):
        """Record detected MEV opportunity"""
        self.opportunities_detected.labels(
            opportunity_type=opportunity_type,
            protocol=protocol
        ).inc()

    def record_opportunity_missed(self, opportunity_type: str, reason:
```

```
str):
        """Record missed MEV opportunity"""
        self.opportunities_missed.labels(
            opportunity_type=opportunity_type,
            reason=reason
        ).inc()

    def record_opportunity_detection_time(self, opportunity_type: str,
duration: float):
        """Record opportunity detection latency"""
        self.opportunity_detection_latency.labels(
            opportunity_type=opportunity_type
        ).observe(duration)

    def record_trade_attempted(self, strategy_type: str, status: str =
'pending'):
        """Record trade attempt"""
        self.trades_attempted.labels(
            strategy_type=strategy_type,
            status=status
        ).inc()

    def record_trade_succeeded(self, strategy_type: str, protocol:
str):
        """Record successful trade"""
        self.trades_succeeded.labels(
            strategy_type=strategy_type,
            protocol=protocol
        ).inc()

    def record_trading_profit(self, strategy_type: str, profit: float):
        """Record trading profit"""
        self.trading_profit.labels(
            strategy_type=strategy_type
        ).inc(profit)

    def record_trade_execution_time(self, strategy_type: str, protocol:
str, duration: float):
        """Record trade execution time"""
        self.trade_execution_duration.labels(
            strategy_type=strategy_type,
            protocol=protocol
```

```
        ).observe(duration)

    def record_slippage(self, strategy_type: str, protocol: str,
slippage: float):
        """Record observed slippage"""
        self.slippage_observed.labels(
            strategy_type=strategy_type,
            protocol=protocol
        ).observe(slippage)

    def update_gas_price(self, network: str, gas_price: int):
        """Update current gas price"""
        self.gas_price.labels(network=network).set(gas_price)

    def update_dex_liquidity(self, protocol: str, token_pair: str,
liquidity: float):
        """Update DEX liquidity"""
        self.dex_liquidity.labels(
            protocol=protocol,
            token_pair=token_pair
        ).set(liquidity)

    def record_api_request(self, endpoint: str, method: str,
status_code: int, duration: float):
        """Record API request metrics"""
        self.api_request_duration.labels(
            endpoint=endpoint,
            method=method,
            status_code=str(status_code)
        ).observe(duration)

    def update_active_connections(self, connection_type: str, count:
int):
        """Update active connections count"""
        self.active_connections.labels(
            connection_type=connection_type
        ).set(count)

    def update_memory_usage(self, component: str, usage_bytes: int):
        """Update memory usage"""
        self.memory_usage.labels(component=component).set(usage_bytes)
```

```
    def update_cpu_usage(self, component: str, usage_percent: float):
        """Update CPU usage"""
        self.cpu_usage.labels(component=component).set(usage_percent)

    def update_daily_pnl(self, pnl: float):
        """Update daily P&L"""
        self.daily_pnl.set(pnl)

    def update_win_rate(self, strategy_type: str, win_rate: float):
        """Update win rate"""
        self.win_rate.labels(strategy_type=strategy_type).set(win_rate)

    def update_sharpe_ratio(self, sharpe: float):
        """Update Sharpe ratio"""
        self.sharpe_ratio.set(sharpe)

    def update_max_drawdown(self, drawdown: float):
        """Update maximum drawdown"""
        self.max_drawdown.set(drawdown)

    def get_metrics(self, registry: Optional[CollectorRegistry] = None)
-> bytes:
        """Get Prometheus metrics in text format"""
        if registry is None:
            registry = self.registry

        return generate_latest(registry)

    def create_component_registry(self, component_name: str) ->
CollectorRegistry:
        """Create separate registry for component"""
        registry = CollectorRegistry()
        self.component_registries[component_name] = registry
        return registry

    def get_component_metrics(self, component_name: str) ->
Optional[bytes]:
        """Get metrics for specific component"""
        registry = self.component_registries.get(component_name)
        if registry:
            return generate_latest(registry)
        return None
```

```
    def add_custom_metric(self, metric_config: MetricConfig, value:
float, labels: Optional[Dict[str, str]] = None):
        """Add custom metric to collection"""

        if labels is None:
            labels = {}

        metric_key = f"{metric_config.name}_{hash(str(labels))}"

        # Create metric based on type
        if metric_config.metric_type == MetricType.COUNTER:
            # Counter implementation
            if metric_key not in self.metrics_cache:
                self.metrics_cache[metric_key] = Counter(
                    metric_config.name,
                    metric_config.description,
                    list(labels.keys()) + (metric_config.labels or []),
                    registry=self.registry
                )
            self.metrics_cache[metric_key].labels(**labels).inc(value)

        elif metric_config.metric_type == MetricType.GAUGE:
            # Gauge implementation
            if metric_key not in self.metrics_cache:
                self.metrics_cache[metric_key] = Gauge(
                    metric_config.name,
                    metric_config.description,
                    list(labels.keys()) + (metric_config.labels or []),
                    registry=self.registry
                )
            self.metrics_cache[metric_key].labels(**labels).set(value)

        elif metric_config.metric_type == MetricType.HISTOGRAM:
            # Histogram implementation
            if metric_key not in self.metrics_cache:
                self.metrics_cache[metric_key] = Histogram(
                    metric_config.name,
                    metric_config.description,
                    list(labels.keys()) + (metric_config.labels or []),
                    buckets=metric_config.buckets or [0.1, 0.5, 1.0,
2.5, 5.0, 10.0],
```

```
                    registry=self.registry
                )

self.metrics_cache[metric_key].labels(**labels).observe(value)

    def export_metrics_to_json(self) -> Dict[str, Any]:
        """Export current metrics values as JSON"""
        # This would be used for real-time dashboards
        current_time = time.time()

        return {
            'timestamp': current_time,
            'opportunities': {
                'detected_total':
self._get_counter_value(self.opportunities_detected),
                'missed_total':
self._get_counter_value(self.opportunities_missed),
                'detection_rate_5m':
self._calculate_rate(self.opportunities_detected, 300)
            },
            'trading': {
                'attempted_total':
self._get_counter_value(self.trades_attempted),
                'succeeded_total':
self._get_counter_value(self.trades_succeeded),
                'profit_total':
self._get_counter_value(self.trading_profit),
                'success_rate': self._calculate_success_rate()
            },
            'performance': {
                'avg_execution_time':
self._get_histogram_average(self.trade_execution_duration),
                'avg_slippage':
self._get_histogram_average(self.slippage_observed)
            },
            'system': {
                'gas_price_eth':
self._get_gauge_value(self.gas_price) / 1e9,
                'api_request_rate_1m':
self._calculate_rate(self.api_request_duration, 60)
            }
        }
```

```
    def _get_counter_value(self, counter) -> int:
        """Get total value of a counter"""
        try:
            # This is a simplified implementation
            # In practice, you'd iterate through the samples
            return 0
        except Exception:
            return 0

    def _get_gauge_value(self, gauge) -> float:
        """Get current value of a gauge"""
        try:
            # This is a simplified implementation
            return 0.0
        except Exception:
            return 0.0

    def _get_histogram_average(self, histogram) -> float:
        """Get average value from histogram"""
        try:
            # This would calculate the weighted average
            return 0.0
        except Exception:
            return 0.0

    def _calculate_rate(self, metric, window_seconds: int) -> float:
        """Calculate rate of change for a counter"""
        # This would calculate the rate of increase over the window
        return 0.0

    def _calculate_success_rate(self) -> float:
        """Calculate overall trading success rate"""
        try:
            attempted = self._get_counter_value(self.trades_attempted)
            succeeded = self._get_counter_value(self.trades_succeeded)

            if attempted > 0:
                return (succeeded / attempted) * 100
            return 0.0
        except Exception:
            return 0.0
```

```
# Metrics decorator for automatic collection
def mev_metrics_decorator(metrics_collector: MEVMetricsCollector):
    """Decorator to automatically collect metrics for functions"""

    def decorator(func):
        async def async_wrapper(*args, **kwargs):
            start_time = time.time()
            success = False

            try:
                result = await func(*args, **kwargs)
                success = True
                return result
            finally:
                duration = time.time() - start_time
                component_name = func.__name__

                metrics_collector.record_api_request(
                    endpoint=f"/{component_name}",
                    method="POST",
                    status_code=200 if success else 500,
                    duration=duration
                )

        def sync_wrapper(*args, **kwargs):
            start_time = time.time()
            success = False

            try:
                result = func(*args, **kwargs)
                success = True
                return result
            finally:
                duration = time.time() - start_time
                component_name = func.__name__

                metrics_collector.record_api_request(
                    endpoint=f"/{component_name}",
                    method="POST",
                    status_code=200 if success else 500,
                    duration=duration
```

```
                )

        if asyncio.iscoroutinefunction(func):
            return async_wrapper
        else:
            return sync_wrapper

    return decorator

# Context manager for transaction metrics
class TransactionMetrics:
    """Context manager for automatic transaction metric collection"""

    def __init__(self,
                 metrics_collector: MEVMetricsCollector,
                 strategy_type: str,
                 protocol: str):
        self.metrics_collector = metrics_collector
        self.strategy_type = strategy_type
        self.protocol = protocol
        self.start_time = None
        self.success = False

    def __enter__(self):
        self.start_time = time.time()

self.metrics_collector.record_trade_attempted(self.strategy_type,
'in_progress')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.start_time is not None:
            duration = time.time() - self.start_time
            self.metrics_collector.record_trade_execution_time(
                self.strategy_type, self.protocol, duration
            )

            if exc_type is None:
                self.success = True
                self.metrics_collector.record_trade_succeeded(
                    self.strategy_type, self.protocol
                )
```

```
            else:
                self.metrics_collector.record_trade_attempted(
                    self.strategy_type, 'failed'
                )

# Usage example
async def example_usage():
    """Example of using the metrics collector"""

    # Initialize metrics collector
    metrics = MEVMetricsCollector()

    # Record some sample metrics
    metrics.record_opportunity_detected('arbitrage', 'uniswap')
    metrics.record_opportunity_detected('liquidation', 'aave')

    # Use transaction metrics context manager
    with TransactionMetrics(metrics, 'arbitrage', 'uniswap'):
        # Simulate trade execution
        await asyncio.sleep(0.1)
        # Trade would be executed here

    # Update system metrics
    metrics.update_gas_price('ethereum', 50000000000)  # 50 gwei
    metrics.update_daily_pnl(150.75)
    metrics.update_win_rate('arbitrage', 78.5)

    # Get metrics output
    metrics_output = metrics.get_metrics()
    print(metrics_output.decode('utf-8'))

    # Export JSON for real-time dashboards
    json_metrics = metrics.export_metrics_to_json()
    print(json_metrics)
```

---

2. Custom MEV Metrics

Trading Performance Metrics

```python
# monitoring/mev_performance_metrics.py
import numpy as np
import pandas as pd
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass
from datetime import datetime, timedelta
import math

@dataclass
class TradeRecord:
    """Individual trade record for performance analysis"""
    timestamp: datetime
    strategy: str
    symbol: str
    side: str  # 'buy' or 'sell'
    quantity: float
    price: float
    fees: float
    pnl: float
    execution_time_ms: float
    slippage_bps: float  # basis points
    success: bool

@dataclass
class PerformanceMetrics:
    """Comprehensive performance metrics"""
    total_return: float
    annualized_return: float
    volatility: float
    sharpe_ratio: float
    sortino_ratio: float
    max_drawdown: float
    calmar_ratio: float
    win_rate: float
    profit_factor: float
    avg_trade_duration: float
    total_trades: int
    winning_trades: int
    losing_trades: int
    largest_win: float
    largest_loss: float
    avg_win: float
```

```
    avg_loss: float
    expectancy: float
    recovery_factor: float
    risk_adjusted_return: float

class MEVPerformanceAnalyzer:
    """Advanced MEV performance analysis with custom metrics"""

    def __init__(self, risk_free_rate: float = 0.02):
        self.risk_free_rate = risk_free_rate
        self.logger = logging.getLogger(__name__)

        # Rolling window for calculations
        self.rolling_window_days = 30

    def analyze_trade_performance(self, trades: List[TradeRecord]) ->
PerformanceMetrics:
        """Analyze comprehensive trading performance"""

        if not trades:
            return self._empty_metrics()

        df = self._trades_to_dataframe(trades)

        # Basic metrics
        total_trades = len(df)
        winning_trades = len(df[df['pnl'] > 0])
        losing_trades = len(df[df['pnl'] < 0])
        win_rate = (winning_trades / total_trades) * 100 if
total_trades > 0 else 0

        # Financial metrics
        total_pnl = df['pnl'].sum()
        gross_profit = df[df['pnl'] > 0]['pnl'].sum()
        gross_loss = abs(df[df['pnl'] < 0]['pnl'].sum())

        profit_factor = gross_profit / gross_loss if gross_loss > 0
else float('inf')

        # Return calculations
        total_return = total_pnl / self._calculate_initial_capital(df)
if total_trades > 0 else 0
```

```
        annualized_return = self._calculate_annualized_return(df,
total_return)

        # Risk metrics
        daily_returns = self._calculate_daily_returns(df)
        volatility = self._calculate_volatility(daily_returns)

        # Risk-adjusted returns
        sharpe_ratio = self._calculate_sharpe_ratio(daily_returns)
        sortino_ratio = self._calculate_sortino_ratio(daily_returns)
        max_drawdown = self._calculate_max_drawdown(df)
        calmar_ratio = annualized_return / abs(max_drawdown) if
max_drawdown != 0 else 0

        # Trade analysis
        avg_trade_duration = df['execution_time_ms'].mean()
        largest_win = df['pnl'].max() if winning_trades > 0 else 0
        largest_loss = df['pnl'].min() if losing_trades > 0 else 0
        avg_win = df[df['pnl'] > 0]['pnl'].mean() if winning_trades > 0
else 0
        avg_loss = df[df['pnl'] < 0]['pnl'].mean() if losing_trades > 0
else 0

        # Expectancy
        avg_pnl_per_trade = total_pnl / total_trades if total_trades >
0 else 0
        expectancy = (win_rate / 100) * avg_win + ((100 - win_rate) /
100) * avg_loss

        # Recovery factor
        recovery_factor = total_pnl / abs(max_drawdown) if
max_drawdown != 0 else float('inf')

        # Risk-adjusted return
        risk_adjusted_return = total_return / volatility if volatility
> 0 else 0

        return PerformanceMetrics(
            total_return=total_return,
            annualized_return=annualized_return,
            volatility=volatility,
            sharpe_ratio=sharpe_ratio,
```

```
            sortino_ratio=sortino_ratio,
            max_drawdown=max_drawdown,
            calmar_ratio=calmar_ratio,
            win_rate=win_rate,
            profit_factor=profit_factor,
            avg_trade_duration=avg_trade_duration,
            total_trades=total_trades,
            winning_trades=winning_trades,
            losing_trades=losing_trades,
            largest_win=largest_win,
            largest_loss=largest_loss,
            avg_win=avg_win,
            avg_loss=avg_loss,
            expectancy=expectancy,
            recovery_factor=recovery_factor,
            risk_adjusted_return=risk_adjusted_return
        )

    def _trades_to_dataframe(self, trades: List[TradeRecord]) ->
pd.DataFrame:
        """Convert trade records to pandas DataFrame"""

        data = []
        for trade in trades:
            data.append({
                'timestamp': trade.timestamp,
                'strategy': trade.strategy,
                'symbol': trade.symbol,
                'side': trade.side,
                'quantity': trade.quantity,
                'price': trade.price,
                'fees': trade.fees,
                'pnl': trade.pnl,
                'execution_time_ms': trade.execution_time_ms,
                'slippage_bps': trade.slippage_bps,
                'success': trade.success
            })

        df = pd.DataFrame(data)
        df['timestamp'] = pd.to_datetime(df['timestamp'])
        df = df.sort_values('timestamp')
```

```
        return df

    def _calculate_initial_capital(self, df: pd.DataFrame) -> float:
        """Calculate initial capital for return calculation"""
        # This would be based on your actual capital allocation
        # For now, use a fixed amount
        return 100000.0

    def _calculate_annualized_return(self, df: pd.DataFrame,
total_return: float) -> float:
        """Calculate annualized return"""

        if len(df) < 2:
            return 0.0

        # Calculate time period in years
        start_date = df['timestamp'].min()
        end_date = df['timestamp'].max()
        days_diff = (end_date - start_date).days

        if days_diff <= 0:
            return 0.0

        years = days_diff / 365.25
        annualized_return = (1 + total_return) ** (1 / years) - 1

        return annualized_return

    def _calculate_daily_returns(self, df: pd.DataFrame) -> pd.Series:
        """Calculate daily returns from trade data"""

        # Group by date and sum P&L
        df['date'] = df['timestamp'].dt.date
        daily_pnl = df.groupby('date')['pnl'].sum()

        # Convert to daily returns (assuming constant capital)
        initial_capital = self._calculate_initial_capital(df)
        daily_returns = daily_pnl / initial_capital

        return daily_returns

    def _calculate_volatility(self, daily_returns: pd.Series) -> float:
```

```
        """Calculate annualized volatility"""

        if len(daily_returns) < 2:
            return 0.0

        return daily_returns.std() * math.sqrt(252)  # Annualize

    def _calculate_sharpe_ratio(self, daily_returns: pd.Series) ->
float:
        """Calculate Sharpe ratio"""

        if len(daily_returns) < 2:
            return 0.0

        excess_returns = daily_returns - (self.risk_free_rate / 252)
# Daily risk-free rate

        if excess_returns.std() == 0:
            return 0.0

        return (excess_returns.mean() * 252) / (excess_returns.std() *
math.sqrt(252))

    def _calculate_sortino_ratio(self, daily_returns: pd.Series) ->
float:
        """Calculate Sortino ratio"""

        if len(daily_returns) < 2:
            return 0.0

        excess_returns = daily_returns - (self.risk_free_rate / 252)

        # Only consider downside deviation
        downside_returns = excess_returns[excess_returns < 0]

        if len(downside_returns) == 0:
            return float('inf')

        downside_std = downside_returns.std() * math.sqrt(252)

        if downside_std == 0:
            return 0.0
```

```
        return (excess_returns.mean() * 252) / downside_std

    def _calculate_max_drawdown(self, df: pd.DataFrame) -> float:
        """Calculate maximum drawdown"""

        # Calculate cumulative P&L
        df['cumulative_pnl'] = df['pnl'].cumsum()
        df['cumulative_pnl'] += self._calculate_initial_capital(df)  #
Add initial capital

        # Calculate running maximum
        df['running_max'] = df['cumulative_pnl'].expanding().max()

        # Calculate drawdown
        df['drawdown'] = (df['cumulative_pnl'] - df['running_max']) /
df['running_max']

        return df['drawdown'].min()

    def _empty_metrics(self) -> PerformanceMetrics:
        """Return empty metrics object"""
        return PerformanceMetrics(
            total_return=0.0, annualized_return=0.0, volatility=0.0,
            sharpe_ratio=0.0, sortino_ratio=0.0, max_drawdown=0.0,
            calmar_ratio=0.0, win_rate=0.0, profit_factor=0.0,
            avg_trade_duration=0.0, total_trades=0, winning_trades=0,
            losing_trades=0, largest_win=0.0, largest_loss=0.0,
            avg_win=0.0, avg_loss=0.0, expectancy=0.0,
            recovery_factor=0.0, risk_adjusted_return=0.0
        )

    def calculate_strategy_performance(self, trades: List[TradeRecord])
-> Dict[str, PerformanceMetrics]:
        """Calculate performance metrics by strategy"""

        df = self._trades_to_dataframe(trades)
        strategies = df['strategy'].unique()

        performance_by_strategy = {}

        for strategy in strategies:
```

```
            strategy_trades = df[df['strategy'] == strategy]
            strategy_trades_list = [
                TradeRecord(
                    timestamp=row['timestamp'],
                    strategy=row['strategy'],
                    symbol=row['symbol'],
                    side=row['side'],
                    quantity=row['quantity'],
                    price=row['price'],
                    fees=row['fees'],
                    pnl=row['pnl'],
                    execution_time_ms=row['execution_time_ms'],
                    slippage_bps=row['slippage_bps'],
                    success=row['success']
                )
                for _, row in strategy_trades.iterrows()
            ]

            performance_by_strategy[strategy] =
self.analyze_trade_performance(strategy_trades_list)

        return performance_by_strategy

    def calculate_rolling_performance(self,
                                      trades: List[TradeRecord],
                                      window_days: int = 30) ->
pd.DataFrame:
        """Calculate rolling performance metrics over time"""

        df = self._trades_to_dataframe(trades)

        if len(df) < 2:
            return pd.DataFrame()

        # Group by date
        df['date'] = df['timestamp'].dt.date
        daily_pnl = df.groupby('date')['pnl'].agg(['sum',
'count']).reset_index()
        daily_pnl.columns = ['date', 'daily_pnl', 'trade_count']

        # Calculate cumulative metrics
        daily_pnl['cumulative_pnl'] = daily_pnl['daily_pnl'].cumsum()
```

```
        daily_pnl['cumulative_return'] = daily_pnl['cumulative_pnl'] /
100000  # Assuming 100k capital

        # Calculate rolling metrics
        daily_pnl['rolling_sharpe'] = daily_pnl['daily_pnl'].rolling(
            window=window_days
        ).apply(lambda x: self._calculate_rolling_sharpe(x, 100000))

        daily_pnl['rolling_volatility'] =
daily_pnl['daily_pnl'].rolling(
            window=window_days
        ).std() * math.sqrt(252)

        daily_pnl['rolling_win_rate'] = daily_pnl['daily_pnl'].rolling(
            window=window_days
        ).apply(lambda x: self._calculate_rolling_win_rate(x))

        return daily_pnl

    def _calculate_rolling_sharpe(self, daily_pnl: pd.Series, capital:
float) -> float:
        """Calculate rolling Sharpe ratio"""

        if len(daily_pnl) < 2:
            return 0.0

        daily_returns = daily_pnl / capital

        if daily_returns.std() == 0:
            return 0.0

        excess_returns = daily_returns - (self.risk_free_rate / 252)

        return (excess_returns.mean() * 252) / (excess_returns.std() *
math.sqrt(252))

    def _calculate_rolling_win_rate(self, daily_pnl: pd.Series) ->
float:
        """Calculate rolling win rate"""

        if len(daily_pnl) == 0:
            return 0.0
```

```
        winning_days = len(daily_pnl[daily_pnl > 0])
        total_days = len(daily_pnl)

        return (winning_days / total_days) * 100 if total_days > 0 else
0.0

    def generate_performance_report(self, trades: List[TradeRecord]) ->
Dict[str, Any]:
        """Generate comprehensive performance report"""

        overall_performance = self.analyze_trade_performance(trades)
        strategy_performance =
self.calculate_strategy_performance(trades)
        rolling_performance =
self.calculate_rolling_performance(trades)

        report = {
            'summary': {
                'total_trades': overall_performance.total_trades,
                'total_return_pct': overall_performance.total_return *
100,
                'annualized_return_pct':
overall_performance.annualized_return * 100,
                'sharpe_ratio': overall_performance.sharpe_ratio,
                'max_drawdown_pct': overall_performance.max_drawdown *
100,
                'win_rate_pct': overall_performance.win_rate,
                'profit_factor': overall_performance.profit_factor
            },
            'detailed_metrics': overall_performance,
            'strategy_breakdown': strategy_performance,
            'rolling_performance':
rolling_performance.to_dict('records') if not rolling_performance.empty
else [],
            'recommendations':
self._generate_recommendations(overall_performance,
strategy_performance)
        }

        return report
```

```
    def _generate_recommendations(self,
                                  overall: PerformanceMetrics,
                                  strategies: Dict[str,
PerformanceMetrics]) -> List[str]:
        """Generate performance improvement recommendations"""

        recommendations = []

        # Win rate recommendations
        if overall.win_rate < 60:

recommendations.append("Consider improving win rate through better
entry/exit criteria")
        elif overall.win_rate > 80:
            recommendations.append("High win rate detected - consider
if position sizing is appropriate")

        # Sharpe ratio recommendations
        if overall.sharpe_ratio < 1.0:
            recommendations.append("Low risk-adjusted returns -
consider diversifying strategies or improving risk management")
        elif overall.sharpe_ratio > 2.0:
            recommendations.append("Excellent risk-adjusted returns -
consider scaling up position sizes")

        # Drawdown recommendations
        if overall.max_drawdown > -0.2:  # -20%
            recommendations.append("High drawdown detected - implement
stricter risk management")

        # Strategy-specific recommendations
        best_strategy = max(strategies.items(), key=lambda x:
x[1].sharpe_ratio)
        worst_strategy = min(strategies.items(), key=lambda x:
x[1].sharpe_ratio)

        if best_strategy[1].sharpe_ratio > 2 *
worst_strategy[1].sharpe_ratio:
            recommendations.append(
                f"Strategy '{best_strategy[0]}' significantly
outperforms '{worst_strategy[0]}' - "
                f"consider reallocating capital"
```

```
            )

        # Execution time recommendations
        if overall.avg_trade_duration > 1000:  # 1 second

recommendations.append("High trade execution times detected - consider
optimizing for faster execution")

        return recommendations

# Custom metrics for MEV-specific performance
class MEVSpecificMetrics:
    """MEV-specific performance metrics and analysis"""

    def __init__(self):
        self.logger = logging.getLogger(__name__)

    def calculate_arbitrage_efficiency(self, arbitrage_trades:
List[TradeRecord]) -> Dict[str, float]:
        """Calculate arbitrage-specific efficiency metrics"""

        if not arbitrage_trades:
            return {}

        df = pd.DataFrame([{
            'timestamp': trade.timestamp,
            'price_impact': trade.slippage_bps / 10000,  # Convert bps
to decimal
            'execution_time': trade.execution_time_ms,
            'profit': trade.pnl,
            'success': trade.success
        } for trade in arbitrage_trades])

        # Calculate efficiency metrics
        efficiency_metrics = {
            'avg_price_impact': df['price_impact'].mean(),
            'avg_execution_time_ms': df['execution_time'].mean(),
            'profit_rate': df['profit'].sum() / len(df),
            'success_rate': df['success'].mean() * 100,
            'efficiency_score': self._calculate_efficiency_score(df)
        }
```

```
        return efficiency_metrics

    def calculate_liquidation_performance(self, liquidation_trades:
List[TradeRecord]) -> Dict[str, float]:
        """Calculate liquidation-specific performance metrics"""

        if not liquidation_trades:
            return {}

        df = pd.DataFrame([{
            'timestamp': trade.timestamp,
            'bonus_received': trade.pnl,  # In liquidation, P&L is the
bonus
            'execution_time': trade.execution_time_ms,
            'competition_level': trade.slippage_bps,
# Use slippage as proxy for competition
            'success': trade.success
        } for trade in liquidation_trades])

        performance_metrics = {
            'avg_bonus_received': df['bonus_received'].mean(),
            'bonus_success_rate': df['success'].mean() * 100,
            'competition_intensity': df['competition_level'].mean(),
            'avg_execution_time': df['execution_time'].mean(),
            'liquidation_efficiency':
self._calculate_liquidation_efficiency(df)
        }

        return performance_metrics

    def calculate_gas_efficiency(self, all_trades: List[TradeRecord]) -
> Dict[str, float]:
        """Calculate gas usage efficiency metrics"""

        # This would require gas usage data for each trade
        # For now, provide a template

        gas_metrics = {
            'avg_gas_per_trade': 0.0,  # Would be calculated from
actual gas data
            'gas_cost_per_dollar_profit': 0.0,
            'gas_efficiency_score': 0.0,
```

```
            'gas_optimization_opportunities': []
        }

        return gas_metrics

    def _calculate_efficiency_score(self, df: pd.DataFrame) -> float:
        """Calculate overall arbitrage efficiency score"""

        # Composite score based on multiple factors
        price_impact_score = max(0, 1 - df['price_impact'].mean() *
100)  # Lower is better
        execution_score = max(0, 1 - df['execution_time'].mean() /
1000)   # Lower is better
        success_score = df['success'].mean()  # Higher is better

        # Weighted average
        efficiency_score = (
            price_impact_score * 0.4 +
            execution_score * 0.3 +
            success_score * 0.3
        )

        return efficiency_score

    def _calculate_liquidation_efficiency(self, df: pd.DataFrame) ->
float:
        """Calculate liquidation efficiency score"""

        bonus_score = min(1, df['bonus_received'].mean() / 100)  #
Normalize bonus
        speed_score = max(0, 1 - df['execution_time'].mean() / 500)  #
500ms threshold
        success_score = df['success'].mean()

        efficiency_score = (
            bonus_score * 0.5 +
            speed_score * 0.3 +
            success_score * 0.2
        )

        return efficiency_score
```

Summary

This comprehensive module on monitoring and analytics provides essential skills for building sophisticated observability systems for MEV operations. Key achievements:

✅ What You Learned:

- **Comprehensive monitoring architecture** with Prometheus, Grafana, and custom metrics collection
- **Advanced MEV-specific metrics** including opportunity detection, trading performance, and system health
- **Performance analytics framework** with statistical analysis and risk metrics calculation
- **Custom alerting systems** for critical MEV opportunities and operational issues
- **Distributed tracing implementation** for latency analysis and bottleneck identification
- **Anomaly detection systems** for trading anomalies and system abnormalities

🚀 Production Implementation:

- **Real-time dashboards** with interactive visualizations for MEV monitoring
- **Automated alerting pipelines** with multiple notification channels and escalation
- **Performance regression detection** with statistical monitoring and trend analysis
- **Capacity planning and scaling** based on metrics-driven insights
- **Compliance and audit logging** for regulatory requirements

💡 Best Practices:

- **Metric hierarchy design** with clear naming conventions and labeling
- **Alert design principles** with proper thresholds and notification routing
- **Dashboard layout optimization** for operational efficiency and decision-making
- **Data retention policies** balancing performance and storage costs
- **Security considerations** for sensitive trading data and metrics access

The monitoring and analytics skills from this module enable building comprehensive observability systems that provide deep insights into MEV system performance, detect opportunities in real-time, and ensure operational excellence with enterprise-grade monitoring capabilities.

Next: In Module 6, we'll explore security and authentication systems, covering API key management, OAuth implementation, secure communications, and compliance frameworks for MEV operations.