Module 5: Monitoring & Alerting Systems

Course: Tool Usage Tutorials
Module: 5 of 6
Duration: 95 minutes
Level: Intermediate-Advanced
Author: Obelisk Core

Table of Contents

Introduction to MEV Monitoring

Effective monitoring and alerting systems are crucial for successful MEV operations. They provide real-time visibility into opportunities, system health, and performance metrics while enabling rapid response to issues and market changes.

Why Monitoring Matters for MEV

Opportunity Detection
- Real-time identification of profitable MEV opportunities
- Market condition changes that affect strategy viability
- Competitor activity and strategy adaptation
- Price anomalies and arbitrage windows

System Health
- Infrastructure uptime and performance monitoring
- Connection status to critical services (RPCs, relays, DEXs)

- Resource utilization and capacity planning
- Error rate tracking and anomaly detection

Performance Optimization
- Transaction success rates and failure analysis
- Gas cost optimization and bidding strategy effectiveness
- Latency monitoring and bottleneck identification
- ROI tracking and profitability analysis

Risk Management
- Capital allocation and exposure monitoring
- Slippage and failed transaction tracking
- Security incident detection and response
- Regulatory compliance and audit trails

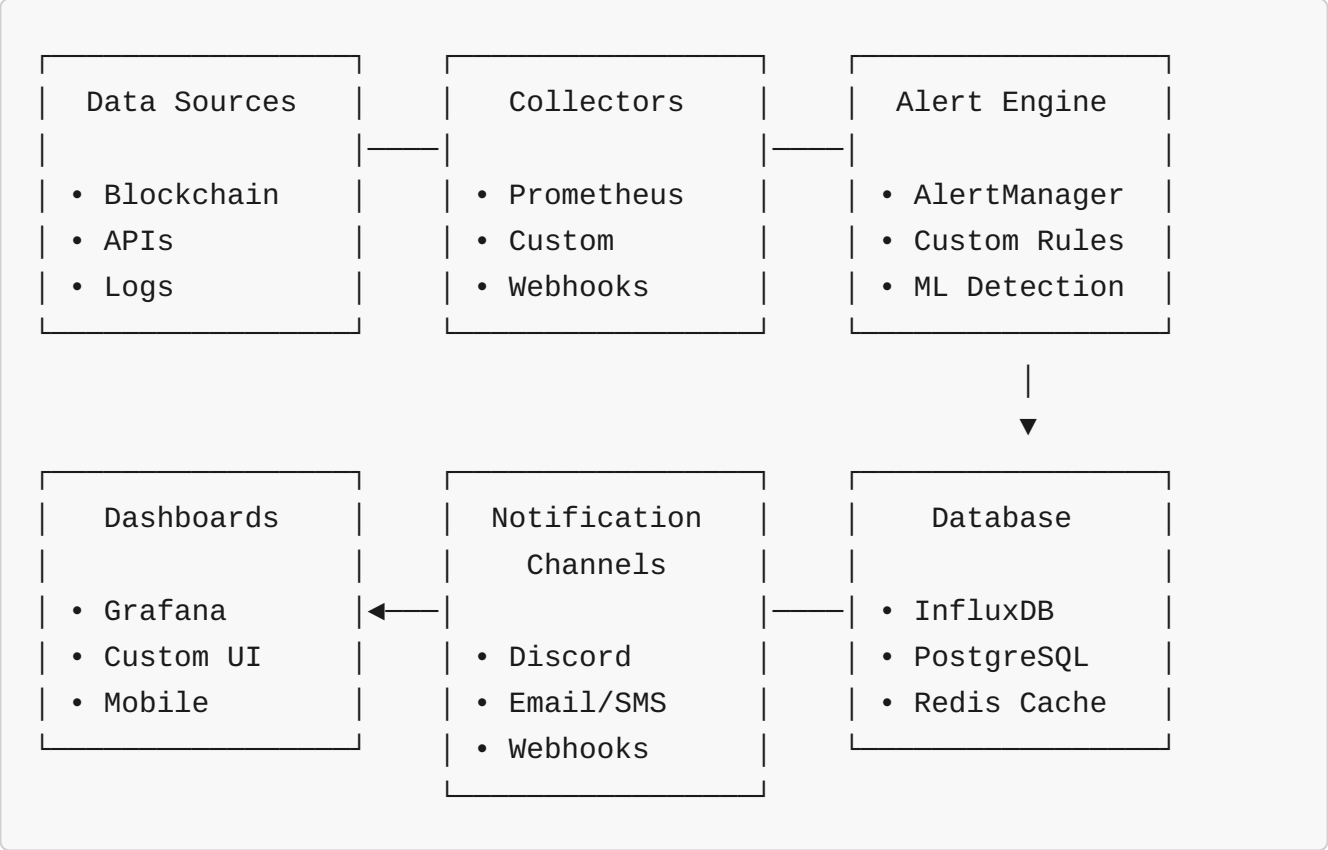
Key Monitoring Components

1. **Data Collectors**: Gather metrics from various sources
2. **Alert Engine**: Evaluate conditions and trigger notifications
3. **Notification System**: Deliver alerts via multiple channels
4. **Dashboard**: Visualize metrics and system status
5. **Log Aggregation**: Centralize and analyze log data
6. **Health Checks**: Monitor system component status
7. **Automated Response**: React to specific conditions automatically

---

System Architecture

Monitoring Infrastructure Overview



Core Monitoring Components

```
import time
import json
import asyncio
import logging
import threading
from datetime import datetime, timedelta
from typing import Dict, List, Any, Optional, Callable
from dataclasses import dataclass
from enum import Enum
import queue
import smtplib
import requests
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('/var/log/mev-monitoring.log'),
        logging.StreamHandler()
    ]
)

class AlertLevel(Enum):
    INFO = "info"
    WARNING = "warning"
    ERROR = "error"
    CRITICAL = "critical"

class AlertChannel(Enum):
    EMAIL = "email"
    DISCORD = "discord"
    TELEGRAM = "telegram"
    WEBHOOK = "webhook"
    SMS = "sms"

@dataclass
class Metric:
    name: str
    value: float
```

```
    timestamp: datetime
    labels: Dict[str, str] = None

    def __post_init__(self):
        if self.labels is None:
            self.labels = {}

@dataclass
class Alert:
    id: str
    title: str
    message: str
    level: AlertLevel
    timestamp: datetime
    source: str
    metrics: List[Metric] = None
    resolved: bool = False

    def __post_init__(self):
        if self.metrics is None:
            self.metrics = []

class MetricCollector:
    """Base class for metric collection"""

    def __init__(self, name: str):
        self.name = name
        self.is_running = False
        self.metrics_queue = queue.Queue()

    def collect_metrics(self) -> List[Metric]:
        """Override this method to implement specific metric
collection"""
        raise NotImplementedError

    def start_collection(self, interval_seconds: int = 60):
        """Start metric collection in background thread"""
        self.is_running = True

        def collection_loop():
            while self.is_running:
                try:
```

```
                    metrics = self.collect_metrics()
                    for metric in metrics:
                        self.metrics_queue.put(metric)
                    time.sleep(interval_seconds)
                except Exception as e:
                    logging.error(f"Error in {self.name} metric
collection: {e}")
                    time.sleep(interval_seconds)

        thread = threading.Thread(target=collection_loop, daemon=True)
        thread.start()

    def stop_collection(self):
        """Stop metric collection"""
        self.is_running = False

    def get_metrics(self) -> List[Metric]:
        """Get collected metrics"""
        metrics = []
        while not self.metrics_queue.empty():
            try:
                metrics.append(self.metrics_queue.get_nowait())
            except queue.Empty:
                break
        return metrics

class AlertEngine:
    """Central alert processing and rule evaluation engine"""

    def __init__(self):
        self.rules = []
        self.alert_history = []
        self.subscribers = {}
        self.is_running = False

    def add_rule(self, rule_func: Callable, name: str):
        """Add alert rule function"""
        self.rules.append({
            'function': rule_func,
            'name': name
        })
```

```
    def subscribe_to_alerts(self, channel: AlertChannel, callback:
Callable):
        """Subscribe to alerts on specific channel"""
        if channel not in self.subscribers:
            self.subscribers[channel] = []
        self.subscribers[channel].append(callback)

    def evaluate_metrics(self, metrics: List[Metric]) -> List[Alert]:
        """Evaluate metrics against all rules"""
        alerts = []

        for rule in self.rules:
            try:
                rule_alerts = rule['function'](metrics)
                if rule_alerts:
                    if isinstance(rule_alerts, list):
                        alerts.extend(rule_alerts)
                    else:
                        alerts.append(rule_alerts)
            except Exception as e:
                logging.error(f"Error evaluating rule {rule['name']}:
{e}")

        return alerts

    def process_alert(self, alert: Alert):
        """Process and distribute alert"""

        # Store in history
        self.alert_history.append(alert)

        # Log alert
        logging.warning(f"ALERT [{alert.level.value.upper()}]:
{alert.title} - {alert.message}")

        # Notify subscribers
        for channel, callbacks in self.subscribers.items():
            for callback in callbacks:
                try:
                    callback(alert)
                except Exception as e:
                    logging.error(f"Error notifying {channel}
```

```
subscriber: {e}")

    def start_processing(self, metric_collectors:
List[MetricCollector],
                         interval_seconds: int = 30):
        """Start alert processing loop"""
        self.is_running = True

        async def processing_loop():
            while self.is_running:
                try:
                    # Collect metrics from all collectors
                    all_metrics = []
                    for collector in metric_collectors:
                        metrics = collector.get_metrics()
                        all_metrics.extend(metrics)

                    # Evaluate rules if we have metrics
                    if all_metrics:
                        alerts = self.evaluate_metrics(all_metrics)

                        # Process alerts
                        for alert in alerts:
                            self.process_alert(alert)

                    await asyncio.sleep(interval_seconds)

                except Exception as e:
                    logging.error(f"Error in alert processing loop:
{e}")
                    await asyncio.sleep(interval_seconds)

        # Run in asyncio event loop
        asyncio.create_task(processing_loop())

    def stop_processing(self):
        """Stop alert processing"""
        self.is_running = False
```

Real-Time Data Collection

Blockchain Data Collector

```
import websockets
from web3 import Web3

class BlockchainDataCollector(MetricCollector):
    """Collect real-time blockchain metrics"""

    def __init__(self, web3_provider_url: str, websocket_url: str =
None):
        super().__init__("blockchain")
        self.w3 = Web3(Web3.HTTPProvider(web3_provider_url))
        self.websocket_url = websocket_url
        self.last_block = None

    def collect_metrics(self) -> List[Metric]:
        """Collect blockchain metrics"""
        metrics = []
        current_time = datetime.utcnow()

        try:
            # Get latest block
            latest_block = self.w3.eth.get_block('latest')
            block_number = latest_block['number']

            # Block metrics
            metrics.append(Metric(
                name="ethereum_latest_block",
                value=block_number,
                timestamp=current_time,
                labels={"network": "mainnet"}
            ))

            # Block time
            if self.last_block:
                block_time = latest_block['timestamp'] -
self.last_block['timestamp']
                metrics.append(Metric(
                    name="ethereum_block_time_seconds",
                    value=block_time,
                    timestamp=current_time
                ))

            # Gas price metrics
```

```
            gas_price = self.w3.eth.gas_price
            metrics.append(Metric(
                name="ethereum_gas_price_gwei",
                value=gas_price / 10**9,
                timestamp=current_time
            ))

            # Transaction count
            tx_count = len(latest_block['transactions'])
            metrics.append(Metric(
                name="ethereum_block_transaction_count",
                value=tx_count,
                timestamp=current_time
            ))

            # Network difficulty (for PoW, historical)
            if 'difficulty' in latest_block:
                metrics.append(Metric(
                    name="ethereum_network_difficulty",
                    value=latest_block['difficulty'],
                    timestamp=current_time
                ))

            self.last_block = latest_block

        except Exception as e:
            logging.error(f"Error collecting blockchain metrics: {e}")

            # Connection error metric
            metrics.append(Metric(
                name="ethereum_connection_error",
                value=1,
                timestamp=current_time,
                labels={"error": str(e)[:100]}
            ))

        return metrics

    async def stream_mempool_data(self, callback: Callable):
        """Stream mempool data via WebSocket"""

        if not self.websocket_url:
```

```
            logging.warning("No WebSocket URL configured for mempool
streaming")
            return

        try:
            async with websockets.connect(self.websocket_url) as
websocket:
                # Subscribe to pending transactions
                subscription = {
                    "jsonrpc": "2.0",
                    "id": 1,
                    "method": "eth_subscribe",
                    "params": ["newPendingTransactions"]
                }

                await websocket.send(json.dumps(subscription))

                while True:
                    try:
                        message = await websocket.recv()
                        data = json.loads(message)

                        if 'params' in data:
                            tx_hash = data['params']['result']

                            # Get transaction details
                            try:
                                tx =
self.w3.eth.get_transaction(tx_hash)
                                callback(tx)
                            except:
                                pass  # Transaction might not be
available yet

                    except websockets.exceptions.ConnectionClosed:
                        logging.warning("WebSocket connection closed,
reconnecting...")
                        break
                    except Exception as e:
                        logging.error(f"Error processing mempool data:
{e}")
```

```
        except Exception as e:
            logging.error(f"Error connecting to WebSocket: {e}")

class MEVDataCollector(MetricCollector):
    """Collect MEV-specific metrics"""

    def __init__(self, etherscan_api_key: str):
        super().__init__("mev")
        self.etherscan_api_key = etherscan_api_key
        self.mev_contracts = [
            '0x...',  # Known MEV bot addresses
            '0x...'
        ]

    def collect_metrics(self) -> List[Metric]:
        """Collect MEV metrics"""
        metrics = []
        current_time = datetime.utcnow()

        try:
            # MEV transaction count
            mev_tx_count = self.count_recent_mev_transactions()
            metrics.append(Metric(
                name="mev_transaction_count_1h",
                value=mev_tx_count,
                timestamp=current_time
            ))

            # Average gas price for MEV transactions
            avg_mev_gas_price = self.get_average_mev_gas_price()
            if avg_mev_gas_price:
                metrics.append(Metric(
                    name="mev_average_gas_price_gwei",
                    value=avg_mev_gas_price / 10**9,
                    timestamp=current_time
                ))

            # MEV opportunity metrics (simulated)
            opportunity_count = self.scan_for_opportunities()
            metrics.append(Metric(
                name="mev_opportunities_detected_1h",
                value=opportunity_count,
```

```
                timestamp=current_time
            ))

        except Exception as e:
            logging.error(f"Error collecting MEV metrics: {e}")

        return metrics

    def count_recent_mev_transactions(self) -> int:
        """Count MEV transactions in the last hour"""
        # Implementation would query Etherscan or local database
        # for transactions involving known MEV contracts
        import random
        return random.randint(50, 200)  # Simulated

    def get_average_mev_gas_price(self) -> Optional[float]:
        """Get average gas price for MEV transactions"""
        # Implementation would analyze recent MEV transactions
        import random
        return random.uniform(50e9, 150e9)  # Simulated (50-150 gwei)

    def scan_for_opportunities(self) -> int:
        """Scan for current MEV opportunities"""
        # Implementation would check:
        # - Price differences across DEXs
        # - Liquidation opportunities
        # - Large pending transactions
        import random
        return random.randint(5, 25)  # Simulated

class SystemMetricsCollector(MetricCollector):
    """Collect system performance metrics"""

    def __init__(self):
        super().__init__("system")

    def collect_metrics(self) -> List[Metric]:
        """Collect system metrics"""
        metrics = []
        current_time = datetime.utcnow()

        try:
```

```
            import psutil

            # CPU usage
            cpu_percent = psutil.cpu_percent(interval=1)
            metrics.append(Metric(
                name="system_cpu_usage_percent",
                value=cpu_percent,
                timestamp=current_time
            ))

            # Memory usage
            memory = psutil.virtual_memory()
            metrics.append(Metric(
                name="system_memory_usage_percent",
                value=memory.percent,
                timestamp=current_time
            ))

            metrics.append(Metric(
                name="system_memory_available_gb",
                value=memory.available / (1024**3),
                timestamp=current_time
            ))

            # Disk usage
            disk = psutil.disk_usage('/')
            metrics.append(Metric(
                name="system_disk_usage_percent",
                value=(disk.used / disk.total) * 100,
                timestamp=current_time
            ))

            # Network stats
            network = psutil.net_io_counters()
            metrics.append(Metric(
                name="system_network_bytes_sent",
                value=network.bytes_sent,
                timestamp=current_time
            ))

            metrics.append(Metric(
                name="system_network_bytes_recv",
```

```
                value=network.bytes_recv,
                timestamp=current_time
            ))

            # Load average (Unix systems)
            try:
                load_avg = psutil.getloadavg()
                metrics.append(Metric(
                    name="system_load_average_1m",
                    value=load_avg[0],
                    timestamp=current_time
                ))
            except AttributeError:
                pass  # Not available on Windows

        except ImportError:
            logging.warning("psutil not available, skipping system
metrics")
        except Exception as e:
            logging.error(f"Error collecting system metrics: {e}")

        return metrics
```

Alert Configuration

Alert Rules and Conditions

```
class AlertRules:
    """Collection of alert rule functions"""

    @staticmethod
    def high_gas_price_rule(metrics: List[Metric]) -> Optional[Alert]:
        """Alert when gas price exceeds threshold"""

        gas_price_threshold = 200  # gwei

        for metric in metrics:
            if (metric.name == "ethereum_gas_price_gwei" and
                metric.value > gas_price_threshold):

                return Alert(
                    id=f"high_gas_{int(time.time())}",
                    title="High Gas Price Alert",
                    message=f"Gas price is {metric.value:.1f} gwei
(threshold: {gas_price_threshold} gwei)",
                    level=AlertLevel.WARNING,
                    timestamp=metric.timestamp,
                    source="gas_monitor",
                    metrics=[metric]
                )

        return None

    @staticmethod
    def system_resource_alert(metrics: List[Metric]) -> List[Alert]:
        """Alert on system resource issues"""

        alerts = []

        for metric in metrics:
            # High CPU usage
            if (metric.name == "system_cpu_usage_percent" and
                metric.value > 90):

                alerts.append(Alert(
                    id=f"high_cpu_{int(time.time())}",
                    title="High CPU Usage",
                    message=f"CPU usage is {metric.value:.1f}%",
                    level=AlertLevel.ERROR,
```

```
                    timestamp=metric.timestamp,
                    source="system_monitor",
                    metrics=[metric]
                ))

            # High memory usage
            elif (metric.name == "system_memory_usage_percent" and
                  metric.value > 85):

                alerts.append(Alert(
                    id=f"high_memory_{int(time.time())}",
                    title="High Memory Usage",
                    message=f"Memory usage is {metric.value:.1f}%",
                    level=AlertLevel.ERROR,
                    timestamp=metric.timestamp,
                    source="system_monitor",
                    metrics=[metric]
                ))

            # Low disk space
            elif (metric.name == "system_disk_usage_percent" and
                  metric.value > 90):

                alerts.append(Alert(
                    id=f"low_disk_{int(time.time())}",
                    title="Low Disk Space",
                    message=f"Disk usage is {metric.value:.1f}%",
                    level=AlertLevel.CRITICAL,
                    timestamp=metric.timestamp,
                    source="system_monitor",
                    metrics=[metric]
                ))

        return alerts

    @staticmethod
    def mev_opportunity_alert(metrics: List[Metric]) ->
Optional[Alert]:
        """Alert on MEV opportunities"""

        opportunity_threshold = 20  # opportunities per hour
```

```
        for metric in metrics:
            if (metric.name == "mev_opportunities_detected_1h" and
                metric.value > opportunity_threshold):

                return Alert(
                    id=f"mev_opportunity_{int(time.time())}",
                    title="High MEV Activity Detected",
                    message=f"{metric.value:.0f} MEV opportunities
detected in the last hour",
                    level=AlertLevel.INFO,
                    timestamp=metric.timestamp,
                    source="mev_monitor",
                    metrics=[metric]
                )

        return None

    @staticmethod
    def blockchain_connection_alert(metrics: List[Metric]) ->
Optional[Alert]:
        """Alert on blockchain connection issues"""

        for metric in metrics:
            if metric.name == "ethereum_connection_error":

                return Alert(
                    id=f"blockchain_error_{int(time.time())}",
                    title="Blockchain Connection Error",
                    message=f"Failed to connect to Ethereum node:
{metric.labels.get('error', 'Unknown error')}",
                    level=AlertLevel.CRITICAL,
                    timestamp=metric.timestamp,
                    source="blockchain_monitor",
                    metrics=[metric]
                )

        return None

    @staticmethod
    def custom_threshold_rule(metric_name: str, threshold: float,
                              comparison: str = 'greater',
                              level: AlertLevel = AlertLevel.WARNING) ->
```

```
Callable:
        """Create custom threshold-based alert rule"""

        def rule_function(metrics: List[Metric]) -> Optional[Alert]:
            for metric in metrics:
                if metric.name == metric_name:

                    triggered = False
                    if comparison == 'greater' and metric.value >
threshold:
                        triggered = True
                    elif comparison == 'less' and metric.value <
threshold:
                        triggered = True
                    elif comparison == 'equal' and metric.value ==
threshold:
                        triggered = True

                    if triggered:
                        return Alert(
                            id=f"custom_{metric_name}
_{int(time.time())}",
                            title=f"Custom Alert: {metric_name}",
                            message=f"{metric_name} is {metric.value}
(threshold: {comparison} {threshold})",
                            level=level,
                            timestamp=metric.timestamp,
                            source="custom_rule",
                            metrics=[metric]
                        )

            return None

        return rule_function

# Advanced Alert Conditions
class AdvancedAlertConditions:
    """More sophisticated alert conditions"""

    def __init__(self):
        self.metric_history = {}
        self.alert_cooldowns = {}
```

```
    def moving_average_rule(self, metric_name: str, window_size: int,
                            threshold: float, comparison: str =
'greater') -> Callable:
        """Alert based on moving average of metrics"""

        def rule_function(metrics: List[Metric]) -> Optional[Alert]:
            # Store metrics in history
            if metric_name not in self.metric_history:
                self.metric_history[metric_name] = []

            # Add new metrics to history
            for metric in metrics:
                if metric.name == metric_name:
                    self.metric_history[metric_name].append(metric)

            # Keep only recent metrics
            cutoff_time = datetime.utcnow() -
timedelta(minutes=window_size)
            self.metric_history[metric_name] = [
                m for m in self.metric_history[metric_name]
                if m.timestamp > cutoff_time
            ]

            # Calculate moving average
            if len(self.metric_history[metric_name]) >= 3:  # Minimum
samples
                values = [m.value for m in
self.metric_history[metric_name]]
                avg_value = sum(values) / len(values)

                triggered = False
                if comparison == 'greater' and avg_value > threshold:
                    triggered = True
                elif comparison == 'less' and avg_value < threshold:
                    triggered = True

                if triggered:
                    # Check cooldown
                    cooldown_key = f"moving_avg_{metric_name}"
                    if self.should_send_alert(cooldown_key,
timedelta(minutes=10)):
```

```
                        self.update_alert_cooldown(cooldown_key)

                        return Alert(
                            id=f"moving_avg_{metric_name}
_{int(time.time())}",
                            title=f"Moving Average Alert:
{metric_name}",
                            message=f"{metric_name} moving average
({window_size}min) is {avg_value:.2f} (threshold: {comparison}
{threshold})",
                            level=AlertLevel.WARNING,
                            timestamp=datetime.utcnow(),
                            source="moving_average_rule"
                        )

            return None

        return rule_function

    def rate_of_change_rule(self, metric_name: str, change_threshold:
float,
                            time_window_minutes: int = 5) -> Callable:
        """Alert on rapid rate of change"""

        def rule_function(metrics: List[Metric]) -> Optional[Alert]:
            # Store metrics
            if metric_name not in self.metric_history:
                self.metric_history[metric_name] = []

            for metric in metrics:
                if metric.name == metric_name:
                    self.metric_history[metric_name].append(metric)

            # Keep only recent metrics
            cutoff_time = datetime.utcnow() -
timedelta(minutes=time_window_minutes)
            recent_metrics = [
                m for m in self.metric_history[metric_name]
                if m.timestamp > cutoff_time
            ]
```

```
            if len(recent_metrics) >= 2:
                # Calculate rate of change
                latest_value = recent_metrics[-1].value
                earliest_value = recent_metrics[0].value
                time_diff = (recent_metrics[-1].timestamp -
recent_metrics[0].timestamp).total_seconds() / 60  # minutes

                if time_diff > 0:
                    rate_of_change = abs(latest_value -
earliest_value) / time_diff

                    if rate_of_change > change_threshold:
                        cooldown_key = f"rate_change_{metric_name}"
                        if self.should_send_alert(cooldown_key,
timedelta(minutes=5)):

                            self.update_alert_cooldown(cooldown_key)

                            return Alert(
                                id=f"rate_change_{metric_name}
_{int(time.time())}",
                                title=f"Rapid Change Alert:
{metric_name}",
                                message=f"{metric_name} changed by
{rate_of_change:.2f}/min (threshold: {change_threshold}/min)",
                                level=AlertLevel.WARNING,
                                timestamp=datetime.utcnow(),
                                source="rate_of_change_rule"
                            )

            return None

        return rule_function

    def should_send_alert(self, alert_key: str, cooldown_period:
timedelta) -> bool:
        """Check if alert should be sent considering cooldown"""

        if alert_key in self.alert_cooldowns:
            last_alert_time = self.alert_cooldowns[alert_key]
            if datetime.utcnow() - last_alert_time < cooldown_period:
                return False
```

```
        return True

    def update_alert_cooldown(self, alert_key: str):
        """Update alert cooldown timestamp"""
        self.alert_cooldowns[alert_key] = datetime.utcnow()
```

Multi-Channel Notifications

Notification Channels Implementation

```
class NotificationChannel:
    """Base notification channel"""

    def send_notification(self, alert: Alert) -> bool:
        """Send notification for alert"""
        raise NotImplementedError

class EmailNotification(NotificationChannel):
    """Email notification channel"""

    def __init__(self, smtp_server: str, smtp_port: int, username: str,
                 password: str, from_email: str, to_emails: List[str],
use_tls: bool = True):
        self.smtp_server = smtp_server
        self.smtp_port = smtp_port
        self.username = username
        self.password = password
        self.from_email = from_email
        self.to_emails = to_emails
        self.use_tls = use_tls

    def send_notification(self, alert: Alert) -> bool:
        """Send email notification"""

        try:
            # Create message
            msg = MIMEMultipart()
            msg['From'] = self.from_email
            msg['To'] = ', '.join(self.to_emails)
            msg['Subject'] = f"[{alert.level.value.upper()}]
{alert.title}"

            # Create email body
            body = self.format_alert_email(alert)
            msg.attach(MIMEText(body, 'html'))

            # Send email
            with smtplib.SMTP(self.smtp_server, self.smtp_port) as
server:
                if self.use_tls:
                    server.starttls()
                server.login(self.username, self.password)
```

```
                server.send_message(msg)

            logging.info(f"Email notification sent for alert:
{alert.id}")
            return True

        except Exception as e:
            logging.error(f"Failed to send email notification: {e}")
            return False

    def format_alert_email(self, alert: Alert) -> str:
        """Format alert as HTML email"""

        # Color coding by alert level
        color_map = {
            AlertLevel.INFO: "#17a2b8",
            AlertLevel.WARNING: "#ffc107",
            AlertLevel.ERROR: "#fd7e14",
            AlertLevel.CRITICAL: "#dc3545"
        }

        color = color_map.get(alert.level, "#6c757d")

        html = f"""
        <html>
        <body style="font-family: Arial, sans-serif; margin: 20px;">
            <div style="border-left: 4px solid {color}; padding-left:
20px; margin-bottom: 20px;">
                <h2 style="color: {color}; margin: 0;">{alert.title}</
h2>
                <p style="color: #666; margin: 5px 0;">
                    <strong>Level:</strong>
{alert.level.value.upper()} |
                    <strong>Time:</strong>
{alert.timestamp.strftime('%Y-%m-%d %H:%M:%S UTC')} |
                    <strong>Source:</strong> {alert.source}
                </p>
            </div>

            <div style="background-color: #f8f9fa; padding: 15px;
border-radius: 5px; margin-bottom: 20px;">
                <h3 style="margin-top: 0;">Message:</h3>
```

```
                <p>{alert.message}</p>
            </div>

            {self.format_metrics_table(alert.metrics) if alert.metrics
else ''}

            <div style="margin-top: 30px; padding-top: 20px; border-
top: 1px solid #dee2e6; color: #6c757d; font-size: 12px;">
                <p>This is an automated alert from the MEV Monitoring
System.</p>
                <p>Alert ID: {alert.id}</p>
            </div>
        </body>
        </html>
        """

        return html

    def format_metrics_table(self, metrics: List[Metric]) -> str:
        """Format metrics as HTML table"""

        if not metrics:
            return ""

        table_rows = ""
        for metric in metrics:
            labels_str = ", ".join([f"{k}={v}" for k, v in
metric.labels.items()]) if metric.labels else ""
            table_rows += f"""
            <tr>
                <td style="padding: 8px; border-bottom: 1px solid
#dee2e6;">{metric.name}</td>
                <td style="padding: 8px; border-bottom: 1px solid
#dee2e6;">{metric.value}</td>
                <td style="padding: 8px; border-bottom: 1px solid
#dee2e6;">{metric.timestamp.strftime('%H:%M:%S')}</td>
                <td style="padding: 8px; border-bottom: 1px solid
#dee2e6;">{labels_str}</td>
            </tr>
            """

        return f"""
```

```
        <div style="margin-bottom: 20px;">
            <h3>Related Metrics:</h3>
            <table style="width: 100%; border-collapse: collapse;
border: 1px solid #dee2e6;">
                <thead>
                    <tr style="background-color: #e9ecef;">
                        <th style="padding: 10px; text-align: left;
border-bottom: 2px solid #dee2e6;">Metric</th>
                        <th style="padding: 10px; text-align: left;
border-bottom: 2px solid #dee2e6;">Value</th>
                        <th style="padding: 10px; text-align: left;
border-bottom: 2px solid #dee2e6;">Time</th>
                        <th style="padding: 10px; text-align: left;
border-bottom: 2px solid #dee2e6;">Labels</th>
                    </tr>
                </thead>
                <tbody>
                    {table_rows}
                </tbody>
            </table>
        </div>
        """

class DiscordNotification(NotificationChannel):
    """Discord webhook notification"""

    def __init__(self, webhook_url: str):
        self.webhook_url = webhook_url

    def send_notification(self, alert: Alert) -> bool:
        """Send Discord notification"""

        try:
            # Color coding by alert level
            color_map = {
                AlertLevel.INFO: 0x17a2b8,
                AlertLevel.WARNING: 0xffc107,
                AlertLevel.ERROR: 0xfd7e14,
                AlertLevel.CRITICAL: 0xdc3545
            }

            embed = {
```

```
                "title": f"🚨 {alert.title}",
                "description": alert.message,
                "color": color_map.get(alert.level, 0x6c757d),
                "timestamp": alert.timestamp.isoformat(),
                "fields": [
                    {
                        "name": "Level",
                        "value": alert.level.value.upper(),
                        "inline": True
                    },
                    {
                        "name": "Source",
                        "value": alert.source,
                        "inline": True
                    },
                    {
                        "name": "Alert ID",
                        "value": alert.id,
                        "inline": True
                    }
                ],
                "footer": {
                    "text": "MEV Monitoring System"
                }
            }

            # Add metrics fields
            if alert.metrics:
                for metric in alert.metrics[:5]:  # Limit to 5 metrics
                    embed["fields"].append({
                        "name": metric.name,
                        "value": f"{metric.value}
({metric.timestamp.strftime('%H:%M:%S')})",
                        "inline": True
                    })

            payload = {
                "embeds": [embed]
            }

            response = requests.post(self.webhook_url, json=payload,
timeout=10)
```

```
            response.raise_for_status()

            logging.info(f"Discord notification sent for alert:
{alert.id}")
            return True

        except Exception as e:
            logging.error(f"Failed to send Discord notification: {e}")
            return False

class TelegramNotification(NotificationChannel):
    """Telegram bot notification"""

    def __init__(self, bot_token: str, chat_ids: List[str]):
        self.bot_token = bot_token
        self.chat_ids = chat_ids
        self.base_url = f"https://api.telegram.org/bot{bot_token}"

    def send_notification(self, alert: Alert) -> bool:
        """Send Telegram notification"""

        try:
            # Format message
            level_emoji = {
                AlertLevel.INFO: "ℹ️",
                AlertLevel.WARNING: "⚠️",
                AlertLevel.ERROR: "❌",
                AlertLevel.CRITICAL: "🚨"
            }

            emoji = level_emoji.get(alert.level, "📋")

            message = f"""
{emoji} *{alert.title}*

{alert.message}

*Level:* {alert.level.value.upper()}
*Source:* {alert.source}
*Time:* {alert.timestamp.strftime('%Y-%m-%d %H:%M:%S UTC')}
*Alert ID:* `{alert.id}`
            """
```

```
            # Add metrics information
            if alert.metrics:
                message += "\n\n*Metrics:*\n"
                for metric in alert.metrics[:3]:  # Limit to 3 metrics
                    message += f"• {metric.name}: {metric.value}\n"

            # Send to all configured chats
            success_count = 0
            for chat_id in self.chat_ids:
                if self.send_message(chat_id, message):
                    success_count += 1

            if success_count > 0:
                logging.info(f"Telegram notification sent to
{success_count} chats for alert: {alert.id}")
                return True
            else:
                return False

        except Exception as e:
            logging.error(f"Failed to send Telegram notification: {e}")
            return False

    def send_message(self, chat_id: str, message: str) -> bool:
        """Send message to specific Telegram chat"""

        try:
            payload = {
                "chat_id": chat_id,
                "text": message,
                "parse_mode": "Markdown"
            }

            response = requests.post(
                f"{self.base_url}/sendMessage",
                json=payload,
                timeout=10
            )

            return response.status_code == 200
```

```
        except Exception as e:
            logging.error(f"Failed to send Telegram message to
{chat_id}: {e}")
            return False

class WebhookNotification(NotificationChannel):
    """Generic webhook notification"""

    def __init__(self, webhook_url: str, headers: Dict[str, str] =
None):
        self.webhook_url = webhook_url
        self.headers = headers or {"Content-Type": "application/json"}

    def send_notification(self, alert: Alert) -> bool:
        """Send webhook notification"""

        try:
            payload = {
                "alert_id": alert.id,
                "title": alert.title,
                "message": alert.message,
                "level": alert.level.value,
                "timestamp": alert.timestamp.isoformat(),
                "source": alert.source,
                "resolved": alert.resolved,
                "metrics": [
                    {
                        "name": metric.name,
                        "value": metric.value,
                        "timestamp": metric.timestamp.isoformat(),
                        "labels": metric.labels
                    }
                    for metric in alert.metrics
                ] if alert.metrics else []
            }

            response = requests.post(
                self.webhook_url,
                json=payload,
                headers=self.headers,
                timeout=10
            )
```

```
            response.raise_for_status()

            logging.info(f"Webhook notification sent for alert:
{alert.id}")
            return True

        except Exception as e:
            logging.error(f"Failed to send webhook notification: {e}")
            return False

# Notification Manager
class NotificationManager:
    """Manage multiple notification channels"""

    def __init__(self):
        self.channels = {}
        self.channel_configs = {}

    def add_channel(self, name: str, channel: NotificationChannel,
                    alert_levels: List[AlertLevel] = None):
        """Add notification channel"""

        self.channels[name] = channel
        self.channel_configs[name] = {
            "channel": channel,
            "alert_levels": alert_levels or list(AlertLevel),
            "enabled": True
        }

    def send_alert(self, alert: Alert):
        """Send alert to appropriate channels"""

        for name, config in self.channel_configs.items():
            if (config["enabled"] and
                alert.level in config["alert_levels"]):

                try:
                    success =
config["channel"].send_notification(alert)
                    if success:
                        logging.info(f"Alert {alert.id} sent via
```

```
{name}")
                    else:
                        logging.warning(f"Failed to send alert
{alert.id} via {name}")
                except Exception as e:
                    logging.error(f"Error sending alert {alert.id} via
{name}: {e}")

    def enable_channel(self, name: str):
        """Enable notification channel"""
        if name in self.channel_configs:
            self.channel_configs[name]["enabled"] = True

    def disable_channel(self, name: str):
        """Disable notification channel"""
        if name in self.channel_configs:
            self.channel_configs[name]["enabled"] = False
```

Summary

In this comprehensive module, you've learned how to:

✅ **Build monitoring infrastructure** with metric collection, alert engines, and notification systems

✅ **Collect real-time data** from blockchain, MEV sources, and system performance metrics

✅ **Configure alert rules** with thresholds, moving averages, and rate-of-change detection

✅ **Implement multi-channel notifications** via email, Discord, Telegram, and webhooks

✅ **Monitor system health** with automated diagnostics and performance tracking

✅ **Manage logs effectively** with centralized collection and analysis

✅ **Create monitoring dashboards** for real-time visibility into MEV operations

✅ **Set up automated responses** to specific alert conditions

Effective monitoring and alerting systems are essential for successful MEV operations, providing the visibility and responsiveness needed to capitalize on opportunities while maintaining system health and security.

Next Steps:
- Implement your monitoring infrastructure with appropriate metric collectors
- Configure alert rules specific to your MEV strategies and risk tolerance
- Set up multiple notification channels for redundancy

- Create comprehensive dashboards for operational visibility
- Test your alerting system with simulated conditions

Remember: Monitoring systems require ongoing maintenance and tuning. Regularly review alert thresholds, add new metrics as your MEV operations evolve, and ensure your notification channels remain reliable and responsive.