Module 3: MEV-Boost Infrastructure

Course: Tool Usage Tutorials
Module: 3 of 6
Duration: 120 minutes
Level: Intermediate-Advanced
Author: Obelisk Core

Table of Contents

Introduction to MEV-Boost

MEV-Boost is a crucial infrastructure component that implements Proposer-Builder Separation (PBS) on Ethereum. It allows validators to maximize their revenue by connecting to a competitive marketplace of block builders who construct optimized blocks containing MEV opportunities.

What is MEV-Boost?

MEV-Boost is middleware software that sits between Ethereum validators and block builders, enabling:

- **Revenue Maximization**: Validators receive higher rewards by outsourcing block construction to specialized builders
- **Competitive Marketplace**: Multiple builders compete to create the most profitable blocks

- **Censorship Resistance**: Validators maintain final control over block acceptance
- **Network Efficiency**: Specialized builders can create more optimized blocks than individual validators

Why MEV-Boost Matters

For Validators:
- Increased staking rewards (typically 10-30% higher)
- Reduced infrastructure requirements for block construction
- Access to MEV opportunities without running specialized software

For MEV Searchers:
- Guaranteed inclusion path for profitable transactions
- Competition among builders for transaction inclusion
- Standardized interface for MEV submission

For the Network:
- More efficient block construction
- Better MEV distribution
- Reduced validator centralization pressure

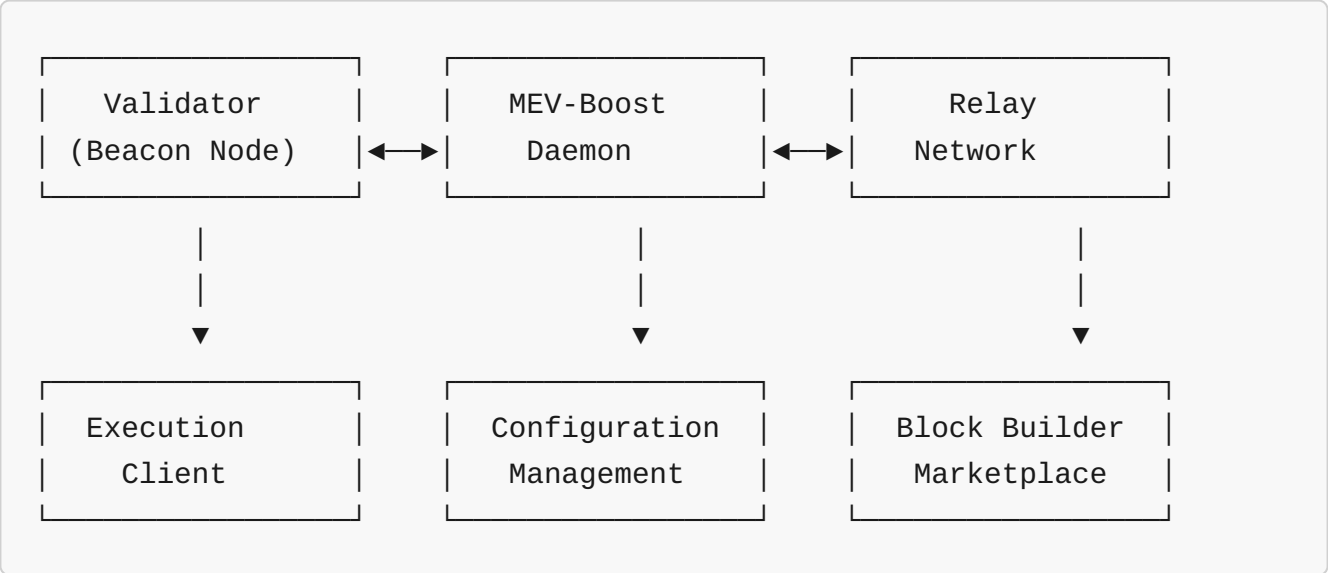
Key Components

1. **MEV-Boost Daemon**: Middleware that handles communication between validators and relays
2. **Relays**: Intermediary services that connect builders and validators
3. **Block Builders**: Specialized services that construct optimized blocks
4. **Consensus Client**: Modified to use MEV-Boost for block production
5. **Execution Client**: Provides blockchain state and transaction pool

---

MEV-Boost Architecture

System Overview



Data Flow

1. **Block Request**: Validator requests optimized block from MEV-Boost
2. **Relay Query**: MEV-Boost queries configured relays for best block
3. **Builder Competition**: Multiple builders submit block proposals to relays
4. **Block Selection**: MEV-Boost selects highest-value block
5. **Block Delivery**: Selected block is delivered to validator
6. **Validation**: Validator validates and optionally accepts block
7. **Block Proposal**: Accepted block is proposed to the network

Communication Protocols

Builder API: Standard interface for block builders to submit blocks
Relay API: Interface for MEV-Boost to communicate with relays
Validator API: Interface for validators to request blocks from MEV-Boost

---

Installation and Setup

System Requirements

Hardware Requirements:
- CPU: 4+ cores (8+ recommended for high-performance setups)
- RAM: 8GB minimum (16GB+ recommended)

- Storage: SSD with 100GB+ free space
- Network: Stable internet connection with low latency

Software Prerequisites:
- Linux (Ubuntu 20.04+ or similar)
- Docker and Docker Compose
- Go 1.19+ (for building from source)
- Ethereum consensus client (Lighthouse, Prysm, Teku, or Nimbus)
- Ethereum execution client (Geth, Besu, Nethermind, or Erigon)

Installation Methods

Method 1: Binary Installation

```
# Download latest MEV-Boost binary
LATEST_VERSION=$(curl -s https://api.github.com/repos/flashbots/mev-
boost/releases/latest | grep tag_name | cut -d '"' -f 4)
wget "https://github.com/flashbots/mev-boost/releases/download/<span
class="math-inline" style="font-family: serif;">{LATEST_VERSION}/mev-
boost_</span>{LATEST_VERSION}_linux_amd64.tar.gz"

# Extract and install
tar -xzf "mev-boost_${LATEST_VERSION}_linux_amd64.tar.gz"
sudo mv mev-boost /usr/local/bin/
sudo chmod +x /usr/local/bin/mev-boost

# Verify installation
mev-boost --version
```

Method 2: Docker Installation

```
# Create MEV-Boost directory structure
mkdir -p ~/mev-boost/{config,logs,data}
cd ~/mev-boost

# Create Docker Compose file
cat > docker-compose.yml << EOF
version: '3.8'

services:
  mev-boost:
    image: flashbots/mev-boost:latest
    container_name: mev-boost
    restart: unless-stopped
    ports:
      - "18550:18550"
    volumes:
      - ./config:/config:ro
      - ./logs:/logs
    command: |
      -mainnet
      -addr 0.0.0.0:18550
      -relay-check
      -relays \${RELAY_URLS}
    environment:
      - RELAY_URLS=https://
0xac6e77dfe25ecd6110b8e780608cce0dab71fdd5ebea22a16c0205200f2f8e2e3ad3b71d3499c54
relay.flashbots.net,https://
0xa15b52576bcbf1072f4a011c0f99f9fb6c66f3e1ff321f11f461d15e31b1cb359caa092c71bbded
0xa7ab7a996c8584251c8f925da3170bdfd6ebc75d50f5ddc4050a6fdc77f2a3b5fce2cc750d0865e
relay.net
    networks:
      - mev-boost-network

  prometheus:
    image: prom/prometheus:latest
    container_name: mev-boost-prometheus
    restart: unless-stopped
    ports:
      - "9090:9090"
    volumes:
      - ./config/prometheus.yml:/etc/prometheus/prometheus.yml:ro
      - prometheus-data:/prometheus
```

```
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--web.enable-lifecycle'
    networks:
      - mev-boost-network

volumes:
  prometheus-data:

networks:
  mev-boost-network:
    driver: bridge
EOF

# Start services
docker-compose up -d
```

Method 3: Build from Source

```
# Install Go (if not already installed)
wget https://go.dev/dl/go1.21.0.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.21.0.linux-amd64.tar.gz
echo 'export PATH=$PATH:/usr/local/go/bin' >> ~/.bashrc
source ~/.bashrc

# Clone and build MEV-Boost
git clone https://github.com/flashbots/mev-boost.git
cd mev-boost
go build -o mev-boost .

# Install binary
sudo mv mev-boost /usr/local/bin/
sudo chmod +x /usr/local/bin/mev-boost
```

Basic Configuration

```
# Create configuration directory
sudo mkdir -p /etc/mev-boost
sudo chown <span class="math-inline" style="display: inline;"><math
xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mi>U</mi><mi>S</mi><mi>E</mi><mi>R</mi><mi>:</
mi></mrow></math></span>USER /etc/mev-boost

# Create basic configuration file
cat > /etc/mev-boost/config.yaml << EOF
# MEV-Boost Configuration
network: mainnet
listen_addr: "0.0.0.0:18550"
relay_check: true
log_level: "info"
log_format: "json"

# Relay Configuration
relays:
  - url: "https://
0xac6e77dfe25ecd6110b8e780608cce0dab71fdd5ebea22a16c0205200f2f8e2e3ad3b71d3499c54
relay.flashbots.net"
    name: "Flashbots"
  - url: "https://
0xa15b52576bcbf1072f4a011c0f99f9fb6c66f3e1ff321f11f461d15e31b1cb359caa092c71bbded
    name: "Aestus"
  - url: "https://
0xa7ab7a996c8584251c8f925da3170bdfd6ebc75d50f5ddc4050a6fdc77f2a3b5fce2cc750d0865e
relay.net"
    name: "Agnostic"

# Performance Settings
request_timeout: "3s"
max_retries: 3
relay_timeout: "950ms"

# Monitoring
metrics_enabled: true
metrics_addr: "0.0.0.0:9545"
EOF
```

Service Setup

```
# Create systemd service file
sudo tee /etc/systemd/system/mev-boost.service > /dev/null << EOF
[Unit]
Description=MEV-Boost
Documentation=https://github.com/flashbots/mev-boost
Wants=network-online.target
After=network-online.target
AssertFileIsExecutable=/usr/local/bin/mev-boost

[Service]
Type=simple
Restart=on-failure
RestartSec=3
TimeoutStopSec=300
ExecStart=/usr/local/bin/mev-boost \\
  -mainnet \\
  -addr 0.0.0.0:18550 \\
  -relay-check \\
  -relays https://
0xac6e77dfe25ecd6110b8e780608cce0dab71fdd5ebea22a16c0205200f2f8e2e3ad3b71d3499c54
relay.flashbots.net,https://
0xa15b52576bcbf1072f4a011c0f99f9fb6c66f3e1ff321f11f461d15e31b1cb359caa092c71bbded
\\
  -log-level info \\
  -log-format json

User=mev-boost
Group=mev-boost
StandardOutput=journal
StandardError=journal
SyslogIdentifier=mev-boost

NoNewPrivileges=true
PrivateTmp=true
ProtectHome=true
ProtectSystem=full
ReadWritePaths=/var/lib/mev-boost
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
AmbientCapabilities=CAP_NET_BIND_SERVICE

[Install]
WantedBy=multi-user.target
```

```
EOF

# Create user and directories
sudo useradd --no-create-home --shell /bin/false mev-boost
sudo mkdir -p /var/lib/mev-boost
sudo chown -R mev-boost:mev-boost /var/lib/mev-boost

# Enable and start service
sudo systemctl enable mev-boost
sudo systemctl start mev-boost
sudo systemctl status mev-boost
```

Relay Configuration

Understanding Relays

Relays are critical infrastructure that:
- Connect block builders to validators
- Validate and cache block proposals
- Provide cryptographic proofs for block commitments
- Ensure builders cannot observe validator choices
- Handle payment verification and enforcement

Major Relay Providers

Flashbots Relay

```
# Flashbots configuration
FLASHBOTS_RELAY="https://
0xac6e77dfe25ecd6110b8e780608cce0dab71fdd5ebea22a16c0205200f2f8e2e3ad3b71d3499c54
relay.flashbots.net"

# Features:
# - Most established and liquid relay
# - High builder participation
# - Transparent policies
# - Strong reputation system
```

Eden Network Relay

```
# Eden configuration
EDEN_RELAY="https://
0xb3ee7afcf27f1f1259ac1787876318c6584ee353097a50ed84f51a1f21a323b3736f271a895c7ce

# Features:
# - Priority for Eden stakers
# - MEV redistribution model
# - Community-focused approach
```

BloXroute Relay

```
# BloXroute configuration
BLOXROUTE_RELAY="https://
0x8b5d2e73e2a3a55c6c87b8b6eb92e0149a125c852751db1422fa951e42a09b82c142c3ea98d0d99
profit.blxrbdn.com"

# Features:
# - Global network optimization
# - Low latency infrastructure
# - Advanced routing algorithms
```

Aestus Relay

```
# Aestus configuration
AESTUS_RELAY="https://
0xa15b52576bcbf1072f4a011c0f99f9fb6c66f3e1ff321f11f461d15e31b1cb359caa092c71bbded

# Features:
# - Non-censoring policy
# - Transparent operations
# - Educational resources
```

Multi-Relay Configuration

```
# Configure multiple relays for redundancy and competition
cat > /etc/mev-boost/relays.conf << EOF
# Primary relays (high reliability)
https://
0xac6e77dfe25ecd6110b8e780608cce0dab71fdd5ebea22a16c0205200f2f8e2e3ad3b71d3499c54
relay.flashbots.net
https://
0xa15b52576bcbf1072f4a011c0f99f9fb6c66f3e1ff321f11f461d15e31b1cb359caa092c71bbded

# Secondary relays (additional opportunities)
https://
0xb3ee7afcf27f1f1259ac1787876318c6584ee353097a50ed84f51a1f21a323b3736f271a895c7ce
https://
0x8b5d2e73e2a3a55c6c87b8b6eb92e0149a125c852751db1422fa951e42a09b82c142c3ea98d0d99
profit.blxrbdn.com

# Specialized relays (specific use cases)
https://
0xa7ab7a996c8584251c8f925da3170bdfd6ebc75d50f5ddc4050a6fdc77f2a3b5fce2cc750d0865e
relay.net
EOF

# Start MEV-Boost with relay list
mev-boost \
  -mainnet \
  -addr 0.0.0.0:18550 \
  -relay-check \
  -relays <span class="math-inline" style="display: inline;"><math
xmlns="http://www.w3.org/1998/Math/MathML" display="inline"><mrow><mo
stretchy="false">&#x00028;</mo><mi>c</mi><mi>a</mi><mi>t</
mi><mo>&#x0002F;</mo><mi>e</mi><mi>t</mi><mi>c</mi><mo>&#x0002F;</
mo><mi>m</mi><mi>e</mi><mi>v</mi><mo>&#x02212;</mo><mi>b</mi><mi>o</
mi><mi>o</mi><mi>s</mi><mi>t</mi><mo>&#x0002F;</mo><mi>r</mi><mi>e</
mi><mi>l</mi><mi>a</mi><mi>y</mi><mi>s</mi><mo>&#x0002E;</mo><mi>c</
mi><mi>o</mi><mi>n</mi><mi>f</mi><mo stretchy="false">&#x0007C;</
mo><mi>t</mi><msup><mi>r</mi><mi>&#x02032;</mi></msup><msup><mi>\n</
mi><mi>&#x02033;</mi></msup><msup><mo>&#x0002C;</mo><mi>&#x02032;</
mi></msup><mo stretchy="false">&#x0007C;</mo><mi>s</mi><mi>e</
mi><msup><mi>d</mi><mi>&#x02032;</mi></msup><mi>s</mi><mo>&#x0002F;</
mo><mo>&#x0002C;</mo></mrow></math></span>//')
```

Relay Health Monitoring

```
#!/usr/bin/env python3
"""
Relay health monitoring script
"""

import requests
import time
import json
from datetime import datetime
from typing import Dict, List, Any

class RelayMonitor:
    def __init__(self, relays: Dict[str, str]):
        self.relays = relays
        self.health_data = {}

    def check_relay_health(self, relay_name: str, relay_url: str) ->
Dict[str, Any]:
        """Check individual relay health"""

        try:
            # Extract base URL (remove validator registration part)
            base_url = relay_url.split('@')[1] if '@' in relay_url else
relay_url
            if not base_url.startswith('http'):
                base_url = f"https://{base_url}"

            # Check relay status endpoint
            status_url = f"{base_url}/eth/v1/builder/status"

            start_time = time.time()
            response = requests.get(status_url, timeout=5)
            response_time = time.time() - start_time

            if response.status_code == 200:
                return {
                    'name': relay_name,
                    'url': base_url,
                    'status': 'healthy',
                    'response_time': response_time,
                    'data': response.json() if response.text else {},
                    'timestamp': datetime.utcnow().isoformat()
```

```
                }
            else:
                return {
                    'name': relay_name,
                    'url': base_url,
                    'status': 'unhealthy',
                    'response_time': response_time,
                    'error': f"HTTP {response.status_code}",
                    'timestamp': datetime.utcnow().isoformat()
                }

        except requests.exceptions.Timeout:
            return {
                'name': relay_name,
                'url': base_url,
                'status': 'timeout',
                'error': 'Request timeout',
                'timestamp': datetime.utcnow().isoformat()
            }
        except Exception as e:
            return {
                'name': relay_name,
                'url': base_url,
                'status': 'error',
                'error': str(e),
                'timestamp': datetime.utcnow().isoformat()
            }

    def monitor_all_relays(self) -> Dict[str, Any]:
        """Monitor all configured relays"""

        results = {}
        healthy_count = 0
        total_count = len(self.relays)

        for name, url in self.relays.items():
            result = self.check_relay_health(name, url)
            results[name] = result

            if result['status'] == 'healthy':
                healthy_count += 1
```

```
        summary = {
            'total_relays': total_count,
            'healthy_relays': healthy_count,
            'health_percentage': (healthy_count / total_count) * 100,
            'timestamp': datetime.utcnow().isoformat(),
            'relays': results
        }

        return summary

    def log_health_report(self, results: Dict[str, Any]):
        """Log health monitoring results"""

        print(f"\n{'='*60}")
        print(f"MEV-Boost Relay Health Report -
{results['timestamp']}")
        print(f"{'='*60}")

        print(f"Overall Health: {results['health_percentage']:.1f}% "
              f"({results['healthy_relays']}/{results['total_relays']}
relays healthy)")

        print(f"\nRelay Details:")
        for name, data in results['relays'].items():
            status_emoji = "✅" if data['status'] == 'healthy' else
"❌"
            response_time = data.get('response_time', 0)

            print(f"{status_emoji} {name}:")
            print(f"   Status: {data['status']}")
            if 'response_time' in data:
                print(f"   Response Time: {response_time:.3f}s")
            if 'error' in data:
                print(f"   Error: {data['error']}")

        # Alert if too many relays are down
        if results['health_percentage'] < 50:
            print(f"\n⚠️  WARNING: Less than 50% of relays are
healthy!")
            print(f"   This may impact MEV-Boost performance and
rewards.")
```

```
    def run_continuous_monitoring(self, interval_seconds: int = 300):
        """Run continuous relay monitoring"""

        print(f"Starting continuous relay monitoring (interval:
{interval_seconds}s)")

        while True:
            try:
                results = self.monitor_all_relays()
                self.log_health_report(results)

                # Save to file for external monitoring
                with open('/var/log/mev-boost-relay-health.json', 'w')
as f:
                    json.dump(results, f, indent=2)

                time.sleep(interval_seconds)

            except KeyboardInterrupt:
                print("\nMonitoring stopped by user")
                break
            except Exception as e:
                print(f"Monitoring error: {e}")
                time.sleep(interval_seconds)

# Configuration
RELAYS = {
    'Flashbots': 'https://boost-relay.flashbots.net',
    'Aestus': 'https://aestus.live',
    'Eden': 'https://relay.edennetwork.io',
    'BloXroute': 'https://bloxroute.max-profit.blxrbdn.com',
    'Agnostic': 'https://agnostic-relay.net'
}

if __name__ == "__main__":
    monitor = RelayMonitor(RELAYS)

    # Run single check
    results = monitor.monitor_all_relays()
    monitor.log_health_report(results)
```

```
    # Uncomment for continuous monitoring
    # monitor.run_continuous_monitoring(300)  # Check every 5 minutes
```

Builder Marketplace Integration

Understanding Block Builders

Block builders are specialized services that:
- Construct optimized blocks with MEV opportunities
- Compete for validator selection through higher bids
- Handle complex transaction ordering and bundling
- Provide guarantees about block execution and payments

Major Block Builders

Flashbots Builder

```
"""
Flashbots Builder Integration
"""

import requests
import json
from typing import Dict, Any, Optional

class FlashbotsBuilderAPI:
    def __init__(self):
        self.base_url = "https://builder-relay-mainnet.flashbots.net"
        self.headers = {
            "Content-Type": "application/json",
            "User-Agent": "MEV-Boost-Monitor/1.0"
        }

    def get_builder_info(self) -> Optional[Dict[str, Any]]:
        """Get builder information and status"""

        try:
            response = requests.get(
                f"{self.base_url}/eth/v1/builder/status",
                headers=self.headers,
                timeout=5
            )

            if response.status_code == 200:
                return response.json()
            else:
                return None

        except Exception as e:
            print(f"Error getting builder info: {e}")
            return None

    def get_validator_registration(self, pubkey: str) ->
Optional[Dict[str, Any]]:
        """Get validator registration status"""

        try:
            response = requests.get(
                f"{self.base_url}/eth/v1/builder/validators/{pubkey}",
```

```
                headers=self.headers,
                timeout=5
            )

            if response.status_code == 200:
                return response.json()
            else:
                return None

        except Exception as e:
            print(f"Error getting validator registration: {e}")
            return None

# Initialize builder API
flashbots_builder = FlashbotsBuilderAPI()
```

Titan Builder

```
"""
Titan Builder Integration
"""

class TitanBuilderAPI:
    def __init__(self):
        self.base_url = "https://rpc.titanbuilder.xyz"
        self.headers = {
            "Content-Type": "application/json"
        }

    def get_builder_stats(self) -> Optional[Dict[str, Any]]:
        """Get Titan builder statistics"""

        try:
            response = requests.get(
                f"{self.base_url}/stats",
                headers=self.headers,
                timeout=5
            )

            if response.status_code == 200:
                return response.json()
            else:
                return None

        except Exception as e:
            print(f"Error getting Titan stats: {e}")
            return None

# Initialize Titan API
titan_builder = TitanBuilderAPI()
```

Builder Performance Monitoring

```
"""
Builder performance monitoring and comparison
"""

import asyncio
import aiohttp
import time
from datetime import datetime, timedelta
from typing import List, Dict, Any

class BuilderPerformanceMonitor:
    def __init__(self):
        self.builders = {
            'flashbots': 'https://boost-relay.flashbots.net',
            'eden': 'https://relay.edennetwork.io',
            'bloxroute': 'https://bloxroute.max-profit.blxrbdn.com'
        }
        self.performance_data = {}

    async def get_builder_bids(self, session: aiohttp.ClientSession,
                               relay_url: str, slot: int) -> Dict[str,
Any]:
        """Get builder bids for a specific slot"""

        try:
            url = f"{relay_url}/eth/v1/builder/header/{slot}/0x00/0x00"

            async with session.get(url, timeout=5) as response:
                if response.status == 200:
                    data = await response.json()
                    return {
                        'relay': relay_url,
                        'slot': slot,
                        'status': 'success',
                        'bid_value': data.get('message',
{}).get('value', '0'),
                        'response_time': time.time()
                    }
                else:
                    return {
                        'relay': relay_url,
                        'slot': slot,
```

```
                        'status': 'error',
                        'error': f"HTTP {response.status}"
                    }

        except Exception as e:
            return {
                'relay': relay_url,
                'slot': slot,
                'status': 'error',
                'error': str(e)
            }

    async def monitor_builder_competition(self, current_slot: int) ->
Dict[str, Any]:
        """Monitor builder competition for current slot"""

        async with aiohttp.ClientSession() as session:
            tasks = []

            for name, url in self.builders.items():
                task = self.get_builder_bids(session, url,
current_slot)
                tasks.append(task)

            results = await asyncio.gather(*tasks,
return_exceptions=True)

            # Process results
            successful_bids = []
            failed_bids = []

            for result in results:
                if isinstance(result, dict) and result.get('status') ==
'success':
                    successful_bids.append(result)
                else:
                    failed_bids.append(result)

            # Find highest bid
            highest_bid = None
            if successful_bids:
                highest_bid = max(successful_bids,
```

```
                                key=lambda x: int(x.get('bid_value',
'0')))

            return {
                'slot': current_slot,
                'timestamp': datetime.utcnow().isoformat(),
                'successful_bids': len(successful_bids),
                'failed_bids': len(failed_bids),
                'highest_bid': highest_bid,
                'all_bids': successful_bids,
                'errors': failed_bids
            }

    def analyze_builder_performance(self, historical_data: List[Dict])
-> Dict[str, Any]:
        """Analyze builder performance over time"""

        relay_stats = {}

        for entry in historical_data:
            for bid in entry.get('all_bids', []):
                relay = bid['relay']

                if relay not in relay_stats:
                    relay_stats[relay] = {
                        'total_bids': 0,
                        'successful_bids': 0,
                        'total_value': 0,
                        'avg_bid_value': 0,
                        'win_rate': 0
                    }

                relay_stats[relay]['total_bids'] += 1

                if bid.get('status') == 'success':
                    relay_stats[relay]['successful_bids'] += 1
                    bid_value = int(bid.get('bid_value', '0'))
                    relay_stats[relay]['total_value'] += bid_value

        # Calculate averages and win rates
        for relay, stats in relay_stats.items():
            if stats['successful_bids'] > 0:
```

```
                stats['avg_bid_value'] = stats['total_value'] /
stats['successful_bids']

            stats['win_rate'] = (stats['successful_bids'] /
stats['total_bids']
                                 if stats['total_bids'] > 0 else 0)

        return relay_stats

# Example usage
async def run_builder_monitoring():
    monitor = BuilderPerformanceMonitor()

    # Get current slot (simplified - would use beacon chain API)
    current_slot = 8000000  # Example slot

    # Monitor current competition
    competition_data = await
monitor.monitor_builder_competition(current_slot)

    print(f"Builder Competition Results for Slot {current_slot}:")
    print(f"Successful bids: {competition_data['successful_bids']}")
    print(f"Failed bids: {competition_data['failed_bids']}")

    if competition_data['highest_bid']:
        highest = competition_data['highest_bid']
        value_eth = int(highest['bid_value']) / 10**18
        print(f"Highest bid: {value_eth:.6f} ETH from
{highest['relay']}")

# Run monitoring
# asyncio.run(run_builder_monitoring())
```

Validator Configuration

Consensus Client Integration

Lighthouse Configuration

```
# Configure Lighthouse to use MEV-Boost
cat > /etc/lighthouse/lighthouse.conf << EOF
# Lighthouse MEV-Boost Configuration
network=mainnet
datadir=/var/lib/lighthouse

# MEV-Boost settings
builder=http://localhost:18550
always-prepare-payload=true
prepare-payload-lookahead=4000

# Performance optimizations
checkpoint-sync-url=https://mainnet-checkpoint-sync.attestant.io
# Additional optimizations
target-peers=100
subscribe-all-attestation-subnets=true
EOF

# Start Lighthouse with MEV-Boost
lighthouse bn \
  --network mainnet \
  --datadir /var/lib/lighthouse \
  --http \
  --http-address 0.0.0.0 \
  --http-port 5052 \
  --builder http://localhost:18550 \
  --always-prepare-payload \
  --prepare-payload-lookahead 4000
```

Prysm Configuration

```
# Configure Prysm beacon node for MEV-Boost
cat > /etc/prysm/beacon.yaml << EOF
# Prysm MEV-Boost Configuration
datadir: "/var/lib/prysm"
p2p-host-ip: "0.0.0.0"
p2p-max-peers: 100
rpc-host: "0.0.0.0"
rpc-port: 4000
grpc-gateway-host: "0.0.0.0"
grpc-gateway-port: 3500

# MEV-Boost integration
http-mev-relay: "http://localhost:18550"
enable-builder: true

# Performance settings
checkpoint-sync-url: "https://mainnet-checkpoint-sync.attestant.io"
genesis-beacon-api-url: "https://mainnet-checkpoint-sync.attestant.io"
EOF

# Start Prysm beacon node
/usr/local/bin/beacon-chain \
  --config-file=/etc/prysm/beacon.yaml \
  --accept-terms-of-use
```

Teku Configuration

```
# Configure Teku for MEV-Boost
cat > /etc/teku/teku.yaml << EOF
# Teku MEV-Boost Configuration
network: "mainnet"
data-path: "/var/lib/teku"
rest-api-enabled: true
rest-api-interface: "0.0.0.0"
rest-api-port: 5051

# MEV-Boost settings
builder-endpoint: "http://localhost:18550"
validators-builder-registration-default-enabled: true

# Performance optimizations
p2p-port: 9000
p2p-discovery-enabled: true
initial-state: "https://mainnet-checkpoint-sync.attestant.io/eth/v2/
debug/beacon/states/finalized"

# Execution engine
ee-endpoint: "http://localhost:8551"
ee-jwt-secret-file: "/var/lib/jwtsecret/jwt.hex"
EOF

# Start Teku
java -jar teku.jar --config-file=/etc/teku/teku.yaml
```

Validator Registration

```
"""
Validator registration for MEV-Boost
"""

import json
import requests
from eth_account import Account
from eth_utils import to_checksum_address
from typing import Dict, Any

class ValidatorRegistration:
    def __init__(self, private_key: str, fee_recipient: str):
        self.account = Account.from_key(private_key)
        self.fee_recipient = to_checksum_address(fee_recipient)

    def create_registration_message(self, validator_pubkey: str,
                                    gas_limit: int = 30000000) ->
Dict[str, Any]:
        """Create validator registration message"""

        message = {
            "fee_recipient": self.fee_recipient,
            "gas_limit": str(gas_limit),
            "timestamp": str(int(time.time())),
            "pubkey": validator_pubkey
        }

        return message

    def sign_registration(self, message: Dict[str, Any]) -> Dict[str,
Any]:
        """Sign validator registration message"""

        # This is a simplified signing process
        # Real implementation would use BLS signatures
        message_hash = self.account.keccak(text=json.dumps(message,
sort_keys=True))
        signature = self.account.sign_message(message_hash)

        return {
            "message": message,
            "signature": signature.signature.hex()
```

```
        }

    def register_validator(self, relay_urls: List[str],
                           validator_pubkey: str) -> Dict[str, Any]:
        """Register validator with relays"""

        # Create registration message
        message = self.create_registration_message(validator_pubkey)
        signed_registration = self.sign_registration(message)

        results = {}

        for relay_url in relay_urls:
            try:
                # Submit registration to relay
                response = requests.post(
                    f"{relay_url}/eth/v1/builder/validators",
                    json=[signed_registration],
                    headers={"Content-Type": "application/json"},
                    timeout=10
                )

                results[relay_url] = {
                    "status": "success" if response.status_code == 200
else "failed",
                    "response_code": response.status_code,
                    "response": response.text
                }

            except Exception as e:
                results[relay_url] = {
                    "status": "error",
                    "error": str(e)
                }

        return results

# Example usage
def register_validators():
    # Configuration (use secure key management in production)
    validator_private_key = "your_validator_private_key"
    fee_recipient = "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c"
```

```
    # Relay URLs for registration
    relay_urls = [
        "https://boost-relay.flashbots.net",
        "https://aestus.live",
        "https://relay.edennetwork.io"
    ]

    # Validator public keys to register
    validator_pubkeys = [
        "0x1234...",  # Your validator public keys
        "0x5678...",
    ]

    registration = ValidatorRegistration(validator_private_key,
fee_recipient)

    for pubkey in validator_pubkeys:
        print(f"Registering validator {pubkey[:10]}...")
        results = registration.register_validator(relay_urls, pubkey)

        for relay, result in results.items():
            status = result["status"]
            print(f"  {relay}: {status}")

            if result["status"] == "failed":
                print(f"    Error: {result.get('response', 'Unknown
error')}")

# Run registration
# register_validators()
```

Monitoring and Metrics

Prometheus Metrics Configuration

```
# prometheus.yml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  - "mev_boost_rules.yml"

scrape_configs:
  - job_name: 'mev-boost'
    static_configs:
      - targets: ['localhost:9545']
    scrape_interval: 5s
    metrics_path: /metrics

  - job_name: 'lighthouse'
    static_configs:
      - targets: ['localhost:5054']
    scrape_interval: 10s

  - job_name: 'node-exporter'
    static_configs:
      - targets: ['localhost:9100']
    scrape_interval: 15s

alerting:
  alertmanagers:
    - static_configs:
        - targets:
          - alertmanager:9093
```

Grafana Dashboard Configuration

```
{
  "dashboard": {
    "id": null,
    "title": "MEV-Boost Monitoring Dashboard",
    "tags": ["mev-boost", "ethereum"],
    "timezone": "browser",
    "panels": [
      {
        "title": "Relay Response Times",
        "type": "stat",
        "targets": [
          {
            "expr": "mev_boost_relay_response_time_seconds",
            "legendFormat": "{{relay_name}}"
          }
        ],
        "fieldConfig": {
          "defaults": {
            "unit": "s",
            "thresholds": {
              "steps": [
                {"color": "green", "value": 0},
                {"color": "yellow", "value": 0.5},
                {"color": "red", "value": 1.0}
              ]
            }
          }
        }
      },
      {
        "title": "Successful Blocks",
        "type": "stat",
        "targets": [
          {
            "expr": "increase(mev_boost_blocks_received_total[1h])",
            "legendFormat": "Blocks per hour"
          }
        ]
      },
      {
        "title": "Block Values",
        "type": "graph",
```

```
        "targets": [
          {
            "expr": "mev_boost_block_value_eth",
            "legendFormat": "Block Value (ETH)"
          }
        ],
        "yAxes": [
          {
            "label": "ETH",
            "min": 0
          }
        ]
      },
      {
        "title": "Relay Health Status",
        "type": "table",
        "targets": [
          {
            "expr": "mev_boost_relay_status",
            "legendFormat": "{{relay_name}}"
          }
        ]
      }
    ]
  }
}
```

Custom Monitoring Script

```
#!/usr/bin/env python3
"""
Advanced MEV-Boost monitoring with alerting
"""

import time
import json
import requests
import smtplib
from datetime import datetime, timedelta
from email.mime.text import MIMEText
from typing import Dict, List, Any, Optional
import logging

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

class MEVBoostMonitor:
    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.mev_boost_url = config.get('mev_boost_url', 'http://
localhost:18550')
        self.metrics_url = config.get('metrics_url', 'http://localhost:
9545/metrics')
        self.alert_thresholds = config.get('alert_thresholds', {})
        self.notification_config = config.get('notifications', {})

        # State tracking
        self.last_successful_block = None
        self.relay_health_history = {}
        self.alert_cooldowns = {}

    def check_mev_boost_status(self) -> Dict[str, Any]:
        """Check MEV-Boost daemon status"""
```

```
        try:
            # Check if MEV-Boost is responding
            response = requests.get(f"{self.mev_boost_url}/",
timeout=5)

            return {
                'status': 'healthy' if response.status_code == 200 else
'unhealthy',
                'response_code': response.status_code,
                'response_time': response.elapsed.total_seconds(),
                'timestamp': datetime.utcnow().isoformat()
            }

        except Exception as e:
            return {
                'status': 'error',
                'error': str(e),
                'timestamp': datetime.utcnow().isoformat()
            }

    def check_relay_connectivity(self) -> Dict[str, Any]:
        """Check connectivity to all configured relays"""

        relay_status = {}

        for relay_name, relay_url in self.config.get('relays',
{}).items():
            try:
                # Remove validator pubkey from URL for health check
                base_url = relay_url.split('@')[1] if '@' in relay_url
else relay_url
                if not base_url.startswith('http'):
                    base_url = f"https://{base_url}"

                start_time = time.time()
                response = requests.get(f"{base_url}/eth/v1/builder/
status", timeout=5)
                response_time = time.time() - start_time

                relay_status[relay_name] = {
                    'status': 'healthy' if response.status_code == 200
else 'unhealthy',
```

```
                    'response_time': response_time,
                    'response_code': response.status_code
                }

            except Exception as e:
                relay_status[relay_name] = {
                    'status': 'error',
                    'error': str(e),
                    'response_time': None
                }

        return relay_status

    def get_recent_blocks_data(self) -> Optional[Dict[str, Any]]:
        """Get recent blocks data from beacon chain"""

        try:
            # This would connect to your beacon node API
            beacon_url = self.config.get('beacon_url', 'http://
localhost:5052')

            # Get recent block data
            response = requests.get(
                f"{beacon_url}/eth/v1/beacon/blocks/head",
                timeout=10
            )

            if response.status_code == 200:
                block_data = response.json()

                # Extract relevant information
                return {
                    'slot': block_data.get('message', {}).get('slot'),
                    'proposer_index': block_data.get('message',
{}).get('proposer_index'),
                    'timestamp': datetime.utcnow().isoformat(),
                    'block_found': True
                }
            else:
                return None

        except Exception as e:
```

```
            logging.error(f"Error getting recent blocks: {e}")
            return None

    def analyze_performance_metrics(self) -> Dict[str, Any]:
        """Analyze MEV-Boost performance metrics"""

        try:
            # Get Prometheus metrics
            response = requests.get(self.metrics_url, timeout=5)

            if response.status_code != 200:
                return {'error': 'Failed to fetch metrics'}

            metrics_text = response.text

            # Parse key metrics (simplified parsing)
            metrics = {}

            for line in metrics_text.split('\n'):
                if line.startswith('mev_boost_'):
                    parts = line.split(' ')
                    if len(parts) >= 2:
                        metric_name = parts[0]
                        metric_value = parts[1]
                        metrics[metric_name] = float(metric_value)

            # Calculate derived metrics
            analysis = {
                'total_requests':
metrics.get('mev_boost_requests_total', 0),
                'successful_requests':
metrics.get('mev_boost_requests_successful_total', 0),
                'avg_response_time':
metrics.get('mev_boost_avg_response_time_seconds', 0),
                'blocks_received':
metrics.get('mev_boost_blocks_received_total', 0),
                'timestamp': datetime.utcnow().isoformat()
            }

            # Calculate success rate
            if analysis['total_requests'] > 0:
                analysis['success_rate'] = (
```

```
                    analysis['successful_requests'] /
analysis['total_requests']
                )
            else:
                analysis['success_rate'] = 0

            return analysis

        except Exception as e:
            logging.error(f"Error analyzing metrics: {e}")
            return {'error': str(e)}

    def check_alert_conditions(self, monitoring_data: Dict[str, Any]) -
> List[Dict[str, Any]]:
        """Check if any alert conditions are met"""

        alerts = []
        current_time = datetime.utcnow()

        # Check MEV-Boost health
        mev_boost_status = monitoring_data.get('mev_boost_status', {})
        if mev_boost_status.get('status') != 'healthy':
            alert_key = 'mev_boost_unhealthy'
            if self.should_send_alert(alert_key):
                alerts.append({
                    'type': 'critical',
                    'title': 'MEV-Boost Unhealthy',
                    'message': f"MEV-Boost status:
{mev_boost_status.get('status')}",
                    'timestamp': current_time.isoformat()
                })
                self.update_alert_cooldown(alert_key)

        # Check relay connectivity
        relay_status = monitoring_data.get('relay_status', {})
        unhealthy_relays = [name for name, status in
relay_status.items()
                            if status.get('status') != 'healthy']

        if len(unhealthy_relays) > len(relay_status) / 2:  # More than
50% unhealthy
            alert_key = 'relays_mostly_unhealthy'
```

```
            if self.should_send_alert(alert_key):
                alerts.append({
                    'type': 'warning',
                    'title': 'Multiple Relays Unhealthy',
                    'message': f"Unhealthy relays: {',
'.join(unhealthy_relays)}",
                    'timestamp': current_time.isoformat()
                })
                self.update_alert_cooldown(alert_key)

        # Check performance metrics
        performance = monitoring_data.get('performance_metrics', {})

        # Check response time
        avg_response_time = performance.get('avg_response_time', 0)
        response_time_threshold =
self.alert_thresholds.get('max_response_time', 2.0)

        if avg_response_time > response_time_threshold:
            alert_key = 'high_response_time'
            if self.should_send_alert(alert_key):
                alerts.append({
                    'type': 'warning',
                    'title': 'High Response Time',
                    'message': f"Average response time:
{avg_response_time:.2f}s",
                    'timestamp': current_time.isoformat()
                })
                self.update_alert_cooldown(alert_key)

        # Check success rate
        success_rate = performance.get('success_rate', 1.0)
        success_rate_threshold =
self.alert_thresholds.get('min_success_rate', 0.95)

        if success_rate < success_rate_threshold:
            alert_key = 'low_success_rate'
            if self.should_send_alert(alert_key):
                alerts.append({
                    'type': 'warning',
                    'title': 'Low Success Rate',
                    'message': f"Success rate: {success_rate:.1%}",
```

```
                    'timestamp': current_time.isoformat()
                })
                self.update_alert_cooldown(alert_key)

        return alerts

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

        cooldown_period = timedelta(minutes=30)  # 30 minute 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()

    def send_notifications(self, alerts: List[Dict[str, Any]]):
        """Send notifications for alerts"""

        if not alerts:
            return

        for alert in alerts:
            logging.warning(f"ALERT: {alert['title']} -
{alert['message']}")

            # Send email if configured
            if self.notification_config.get('email',
{}).get('enabled'):
                self.send_email_alert(alert)

            # Send Discord webhook if configured
            if self.notification_config.get('discord',
{}).get('enabled'):
                self.send_discord_alert(alert)
```

```
    def send_email_alert(self, alert: Dict[str, Any]):
        """Send email alert"""

        try:
            email_config = self.notification_config['email']

            msg = MIMEText(f"""
MEV-Boost Alert: {alert['title']}

Message: {alert['message']}
Severity: {alert['type']}
Time: {alert['timestamp']}

This is an automated alert from your MEV-Boost monitoring system.
            """)

            msg['Subject'] = f"MEV-Boost Alert: {alert['title']}"
            msg['From'] = email_config['from']
            msg['To'] = email_config['to']

            with smtplib.SMTP(email_config['smtp_server'],
email_config['smtp_port']) as server:
                if email_config.get('use_tls'):
                    server.starttls()
                if email_config.get('username'):
                    server.login(email_config['username'],
email_config['password'])
                server.send_message(msg)

            logging.info(f"Email alert sent: {alert['title']}")

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

    def send_discord_alert(self, alert: Dict[str, Any]):
        """Send Discord webhook alert"""

        try:
            discord_config = self.notification_config['discord']

            color = 0xFF0000 if alert['type'] == 'critical' else
0xFFFF00  # Red or Yellow
```

```
            payload = {
                "embeds": [{
                    "title": f"🚨 MEV-Boost Alert: {alert['title']}",
                    "description": alert['message'],
                    "color": color,
                    "fields": [
                        {"name": "Severity", "value": alert['type'],
"inline": True},
                        {"name": "Time", "value": alert['timestamp'],
"inline": True}
                    ]
                }]
            }

            response = requests.post(discord_config['webhook_url'],
json=payload)

            if response.status_code == 204:
                logging.info(f"Discord alert sent: {alert['title']}")
            else:
                logging.error(f"Discord alert failed:
{response.status_code}")

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

    def run_monitoring_cycle(self) -> Dict[str, Any]:
        """Run complete monitoring cycle"""

        logging.info("Starting monitoring cycle")

        # Gather monitoring data
        monitoring_data = {
            'mev_boost_status': self.check_mev_boost_status(),
            'relay_status': self.check_relay_connectivity(),
            'recent_blocks': self.get_recent_blocks_data(),
            'performance_metrics': self.analyze_performance_metrics(),
            'timestamp': datetime.utcnow().isoformat()
        }

        # Check for alerts
```

```
        alerts = self.check_alert_conditions(monitoring_data)

        # Send notifications if needed
        if alerts:
            self.send_notifications(alerts)

        # Log summary
        logging.info(f"Monitoring cycle complete. "
                     f"MEV-Boost: {monitoring_data['mev_boost_status']
['status']}, "
                     f"Alerts: {len(alerts)}")

        return monitoring_data

    def run_continuous_monitoring(self, interval_seconds: int = 60):
        """Run continuous monitoring"""

        logging.info(f"Starting continuous monitoring (interval:
{interval_seconds}s)")

        while True:
            try:
                monitoring_data = self.run_monitoring_cycle()

                # Save monitoring data
                with open('/var/log/mev-boost-monitoring.json', 'w') as
f:
                    json.dump(monitoring_data, f, indent=2)

                time.sleep(interval_seconds)

            except KeyboardInterrupt:
                logging.info("Monitoring stopped by user")
                break
            except Exception as e:
                logging.error(f"Monitoring cycle error: {e}")
                time.sleep(interval_seconds)

# Configuration
config = {
    'mev_boost_url': 'http://localhost:18550',
    'metrics_url': 'http://localhost:9545/metrics',
```

```
    'beacon_url': 'http://localhost:5052',
    'relays': {
        'Flashbots': 'https://boost-relay.flashbots.net',
        'Aestus': 'https://aestus.live',
        'Eden': 'https://relay.edennetwork.io'
    },
    'alert_thresholds': {
        'max_response_time': 2.0,  # seconds
        'min_success_rate': 0.95   # 95%
    },
    'notifications': {
        'email': {
            'enabled': True,
            'smtp_server': 'smtp.gmail.com',
            'smtp_port': 587,
            'username': 'your-email@gmail.com',
            'password': 'your-app-password',
            'from': 'your-email@gmail.com',
            'to': 'alerts@yourdomain.com',
            'use_tls': True
        },
        'discord': {
            'enabled': True,
            'webhook_url': 'https://discord.com/api/webhooks/your/
webhook/url'
        }
    }
}

if __name__ == "__main__":
    monitor = MEVBoostMonitor(config)

    # Run single monitoring cycle
    results = monitor.run_monitoring_cycle()
    print(json.dumps(results, indent=2))

    # Uncomment for continuous monitoring
    # monitor.run_continuous_monitoring(60)  # Check every minute
```

Summary

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

✅ **Set up MEV-Boost infrastructure** with proper installation, configuration, and service management

✅ **Configure relay connections** with multiple providers for redundancy and competitive block sourcing

✅ **Integrate with the builder marketplace** to access optimized blocks from specialized builders

✅ **Configure validator clients** (Lighthouse, Prysm, Teku) to work with MEV-Boost infrastructure

✅ **Implement monitoring and metrics** with Prometheus, Grafana, and custom alerting systems

✅ **Optimize performance** through proper configuration and infrastructure management

✅ **Apply security best practices** for validator registration and relay communication

✅ **Troubleshoot common issues** and maintain healthy MEV-Boost operations

MEV-Boost is critical infrastructure that connects validators to the MEV ecosystem, enabling higher rewards while maintaining decentralization. Proper setup and monitoring ensure optimal performance and reliability for your staking operations.

Next Steps:
- Set up your MEV-Boost infrastructure in a testnet environment
- Register your validators with multiple relays
- Implement comprehensive monitoring and alerting
- Optimize performance based on your specific setup and requirements

Remember: MEV-Boost infrastructure requires careful configuration and ongoing monitoring to ensure optimal performance and maximize validator rewards while maintaining security and reliability.