Module 1: Blockchain API Integration

Master Ethereum, BSC, and Layer 2 API clients for production MEV systems

Course Overview

Duration: 140 minutes
Level: Advanced
Prerequisites: Solid understanding of blockchain fundamentals, REST APIs, and MEV concepts
Certificate: Available upon completion

Learning Objectives

By the end of this module, you will be able to:
- Build production-ready blockchain API clients for Ethereum, BSC, and Layer 2 networks
- Implement efficient rate limiting and connection pooling for high-frequency trading
- Handle network failures, timeouts, and error recovery gracefully
- Subscribe to real-time blockchain events using WebSocket connections
- Optimize API calls for latency-critical MEV operations
- Design fault-tolerant blockchain data fetching systems

Table of Contents

---

1. Introduction to Blockchain API Architecture

Core Concepts

Blockchain APIs serve as the bridge between your MEV strategies and the decentralized network. Unlike traditional financial APIs, blockchain APIs present unique challenges:

Key Challenges:
- **Latency Sensitivity:** MEV profits often depend on millisecond-level timing
- **High Throughput:** Tens of thousands of transactions per second
- **Network Instability:** Nodes can fail, become desynchronized, or become overloaded
- **Rate Limiting:** Public APIs enforce strict usage limits
- **Event Ordering:** Transaction ordering and confirmation consistency

API Types

```
from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Callable, Any
from dataclasses import dataclass
from enum import Enum
import asyncio
import logging
import time
from datetime import datetime
import json

class NetworkType(Enum):
    ETHEREUM_MAINNET = "ethereum_mainnet"
    BSC_MAINNET = "bsc_mainnet"
    ARBITRUM_ONE = "arbitrum_one"
    OPTIMISM = "optimism"
    POLYGON = "polygon"
    ETHEREUM_TESTNET = "ethereum_testnet"

class APIType(Enum):
    RPC_HTTP = "rpc_http"
    RPC_WEBSOCKET = "rpc_websocket"
    ETHEREUM_JSON_RPC = "ethereum_json_rpc"
    ALCHEMY = "alchemy"
    INFURA = "infura"
    QUICKNODE = "quicknode"

@dataclass
class APIConfig:
    """Configuration for blockchain API client"""
    network: NetworkType
    api_type: APIType
    endpoint: str
    api_key: Optional[str] = None
    rate_limit_rps: int = 100  # requests per second
    timeout_seconds: int = 30
    max_retries: int = 3
    retry_backoff: float = 1.5
    enable_connection_pool: bool = True
    max_connections: int = 10
    enable_metrics: bool = True

class BlockchainAPIError(Exception):
```

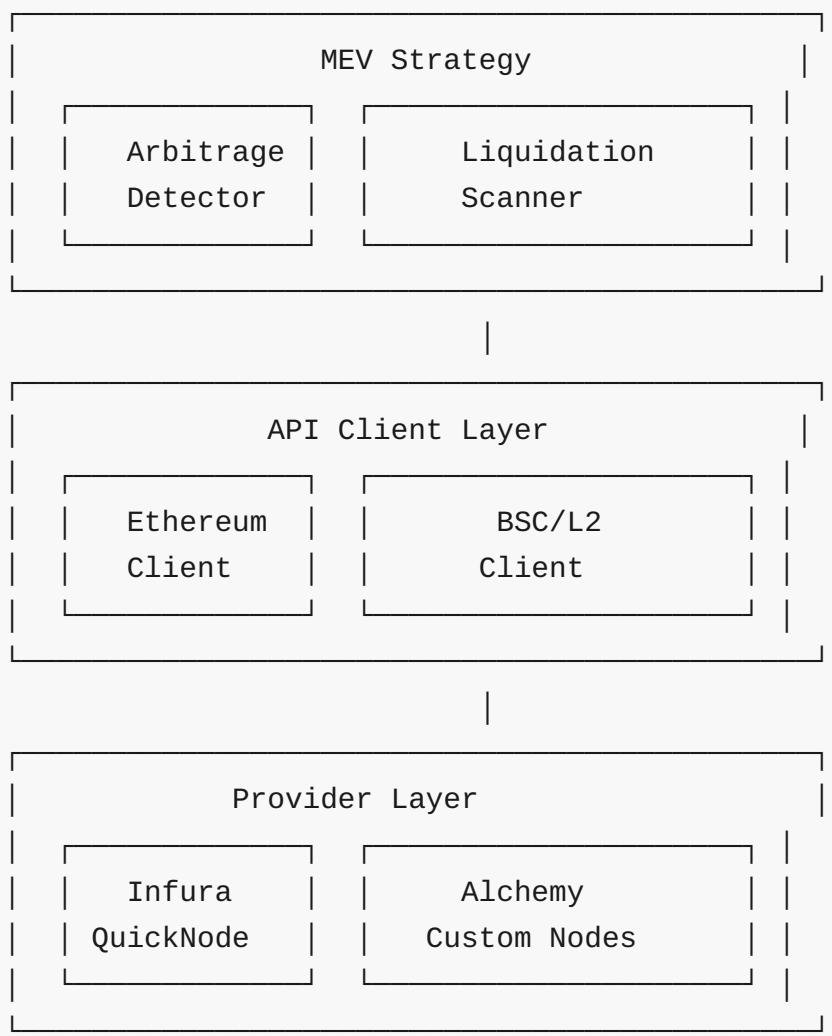
```
    """Base exception for blockchain API errors"""
    pass

class RateLimitError(BlockchainAPIError):
    """Raised when rate limit is exceeded"""
    pass

class TimeoutError(BlockchainAPIError):
    """Raised when API request times out"""
    pass

class NetworkError(BlockchainAPIError):
    """Raised when network connection fails"""
    pass
```

Architecture Overview



2. Ethereum API Integration

Core Ethereum RPC Methods

```
import aiohttp
import asyncio
from web3 import Web3
from web3.providers import HTTPProvider
import websockets
from typing import Union, Dict, Any

class EthereumClient:
    """Production-ready Ethereum API client for MEV strategies"""

    def __init__(self, config: APIConfig):
        self.config = config
        self.logger = logging.getLogger(__name__)
        self.session: Optional[aiohttp.ClientSession] = None
        self.websocket: Optional[websockets.WebSocketServerProtocol] =
None
        self.web3 = Web3(HTTPProvider(config.endpoint))

        # Rate limiting
        self.rate_limiter = RateLimiter(config.rate_limit_rps)
        self.request_count = 0
        self.error_count = 0

        # Performance metrics
        self.latency_tracker = LatencyTracker()

    async def __aenter__(self):
        await self.connect()
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        await self.disconnect()

    async def connect(self):
        """Establish connection to Ethereum node"""
        self.session = aiohttp.ClientSession(

timeout=aiohttp.ClientTimeout(total=self.config.timeout_seconds),
            connector=aiohttp.TCPConnector(
                limit=self.config.max_connections,
                ttl_dns_cache=300,
                use_dns_cache=True,
```

```
            )
        )
        self.logger.info(f"Connected to {self.config.network.value}")

    async def disconnect(self):
        """Close connections and cleanup"""
        if self.session:
            await self.session.close()
        if self.websocket:
            await self.websocket.close()
        self.logger.info("Disconnected from Ethereum node")

    async def get_latest_block(self, full_transactions: bool = True) ->
Dict[str, Any]:
        """Get the latest block with transaction details"""
        start_time = time.time()

        try:
            await self.rate_limiter.acquire()

            payload = {
                "jsonrpc": "2.0",
                "method": "eth_getBlockByNumber",
                "params": ["latest", full_transactions],
                "id": 1
            }

            async with self.session.post(
                self.config.endpoint,
                json=payload,
                headers={"Content-Type": "application/json"}
            ) as response:

                if response.status == 429:  # Rate limited
                    raise RateLimitError("Rate limit exceeded")
                if response.status != 200:
                    raise NetworkError(f"HTTP {response.status}")

                result = await response.json()

                if "error" in result:
                    raise BlockchainAPIError(result["error"]
```

```
["message"])

                self.request_count += 1
                self.latency_tracker.record(time.time() - start_time)

                return result["result"]

        except asyncio.TimeoutError:
            raise TimeoutError(f"Request timed out after
{self.config.timeout_seconds}s")
        except Exception as e:
            self.error_count += 1
            self.logger.error(f"Error getting latest block: {e}")
            raise

    async def get_transaction_receipt(self, tx_hash: str) -> Dict[str,
Any]:
        """Get transaction receipt with status and logs"""
        await self.rate_limiter.acquire()

        payload = {
            "jsonrpc": "2.0",
            "method": "eth_getTransactionReceipt",
            "params": [tx_hash],
            "id": 1
        }

        async with self.session.post(self.config.endpoint,
json=payload) as response:
            result = await response.json()
            return result["result"]

    async def get_gas_price(self) -> int:
        """Get current gas price in wei"""
        await self.rate_limiter.acquire()

        payload = {
            "jsonrpc": "2.0",
            "method": "eth_gasPrice",
            "params": [],
            "id": 1
        }
```

```
        async with self.session.post(self.config.endpoint,
json=payload) as response:
            result = await response.json()
            return int(result["result"], 16)

    async def estimate_gas(self, transaction: Dict[str, Any]) -> int:
        """Estimate gas for a transaction"""
        await self.rate_limiter.acquire()

        payload = {
            "jsonrpc": "2.0",
            "method": "eth_estimateGas",
            "params": [transaction],
            "id": 1
        }

        async with self.session.post(self.config.endpoint,
json=payload) as response:
            result = await response.json()

            if "error" in result:
                # Gas estimation failed, return a safe default
                return 21000  # Simple transfer

            return int(result["result"], 16)

    async def get_logs(self,
                       from_block: Union[str, int],
                       to_block: Union[str, int],
                       address: Optional[str] = None,
                       topics: Optional[List[str]] = None) ->
List[Dict[str, Any]]:
        """Get event logs for contract interactions"""
        await self.rate_limiter.acquire()

        params = {
            "fromBlock": hex(from_block) if isinstance(from_block, int)
else from_block,
            "toBlock": hex(to_block) if isinstance(to_block, int) else
to_block
        }
```

```
        if address:
            params["address"] = address
        if topics:
            params["topics"] = topics

        payload = {
            "jsonrpc": "2.0",
            "method": "eth_getLogs",
            "params": [params],
            "id": 1
        }

        async with self.session.post(self.config.endpoint,
json=payload) as response:
            result = await response.json()

            if "error" in result:
                # Handle log range too large
                if "query returned more than" in result["error"]
["message"]:
                    return await self._paginate_logs(from_block,
to_block, address, topics)
                raise BlockchainAPIError(result["error"]["message"])

            return result["result"]

    async def _paginate_logs(self, from_block: Union[str, int],
to_block: Union[str, int],
                             address: Optional[str], topics:
Optional[List[str]]) -> List[Dict[str, Any]]:
        """Paginate large log requests to avoid response size limits"""
        all_logs = []
        current_block = from_block

        while current_block <= to_block:
            # Process in chunks of 1000 blocks
            chunk_end = min(current_block + 999, to_block)

            chunk_logs = await self.get_logs(
                current_block, chunk_end, address, topics
            )
```

```
            all_logs.extend(chunk_logs)

            current_block = chunk_end + 1

        return all_logs
```

Advanced Ethereum Methods

```
    async def batch_get_balance(self, addresses: List[str]) ->
Dict[str, int]:
        """Get balances for multiple addresses efficiently"""
        await self.rate_limiter.acquire()

        # Create batch request
        payload = []
        for i, address in enumerate(addresses):
            payload.append({
                "jsonrpc": "2.0",
                "method": "eth_getBalance",
                "params": [address, "latest"],
                "id": i + 1
            })

        async with self.session.post(self.config.endpoint,
json=payload) as response:
            results = await response.json()

            balances = {}
            for i, result in enumerate(results):
                if "error" not in result:
                    balances[addresses[i]] = int(result["result"], 16)

            return balances

    async def get_transaction_by_hash(self, tx_hash: str) -> Dict[str,
Any]:
        """Get full transaction details"""
        await self.rate_limiter.acquire()

        payload = {
            "jsonrpc": "2.0",
            "method": "eth_getTransactionByHash",
            "params": [tx_hash],
            "id": 1
        }

        async with self.session.post(self.config.endpoint,
json=payload) as response:
            result = await response.json()
            return result["result"]
```

```
    async def get_block_transaction_count(self, block_identifier:
Union[str, int]) -> int:
        """Get transaction count in a block"""
        await self.rate_limiter.acquire()

        payload = {
            "jsonrpc": "2.0",
            "method": "eth_getBlockTransactionCountByNumber",
            "params": [hex(block_identifier) if
isinstance(block_identifier, int) else block_identifier],
            "id": 1
        }

        async with self.session.post(self.config.endpoint,
json=payload) as response:
            result = await response.json()
            return int(result["result"], 16)

    async def get_code(self, address: str) -> str:
        """Get contract code at address"""
        await self.rate_limiter.acquire()

        payload = {
            "jsonrpc": "2.0",
            "method": "eth_getCode",
            "params": [address, "latest"],
            "id": 1
        }

        async with self.session.post(self.config.endpoint,
json=payload) as response:
            result = await response.json()
            return result["result"]

    async def get_storage_at(self, address: str, position: str) -> str:
        """Get storage value at position"""
        await self.rate_limiter.acquire()

        payload = {
            "jsonrpc": "2.0",
            "method": "eth_getStorageAt",
```

```
            "params": [address, position, "latest"],
            "id": 1
        }

        async with self.session.post(self.config.endpoint,
json=payload) as response:
            result = await response.json()
            return result["result"]
```

---

3. BSC and Layer 2 Support

BSC Integration

```
class BSCClient(EthereumClient):
    """BSC-specific client with optimized endpoints"""

    def __init__(self, config: APIConfig):
        # BSC has different gas calculation needs
        if config.network == NetworkType.BSC_MAINNET:
            config.rate_limit_rps = config.rate_limit_rps or 150
            config.timeout_seconds = config.timeout_seconds or 15

        super().__init__(config)
        self.bnb_smart_chain_id = 56

    async def get_bep20_transfers(self,
                                  contract_address: str,
                                  from_block: int,
                                  to_block: int,
                                  events: List[str] = None) ->
List[Dict[str, Any]]:
        """Get BEP-20 token transfer events efficiently"""
        if not events:
            events = ["Transfer(address,address,uint256)"]

        # Convert event signatures to topics
        topics = []
        for event in events:
            # This would normally use proper ABI encoding
            topic_hash = self._get_topic_hash(event)
            topics.append([topic_hash])

        return await self.get_logs(
            from_block=from_block,
            to_block=to_block,
            address=contract_address,
            topics=topics
        )

    def _get_topic_hash(self, event_signature: str) -> str:
        """Generate topic hash for event signature"""
        import hashlib
        keccak_hash =
hashlib.sha3_256(event_signature.encode()).hexdigest()
        return "0x" + keccak_hash
```

```
    async def get_cross_chain_bridge_events(self, bridge_address: str)
-> List[Dict[str, Any]]:
        """Monitor cross-chain bridge events for arbitrage
opportunities"""
        # Bridge contract events
        bridge_events = [
            "Deposit(address,address,uint256,uint256)",
            "Withdrawal(address,address,uint256,uint256)",
            "TokenSwap(address,address,uint256,uint256)"
        ]

        all_events = []
        for event in bridge_events:
            topic_hash = self._get_topic_hash(event)
            events = await self.get_logs(
                from_block="latest",
                to_block="latest",
                address=bridge_address,
                topics=[topic_hash]
            )
            all_events.extend(events)

        return all_events

class Layer2Client:
    """Base class for Layer 2 clients"""

    def __init__(self, network: NetworkType, config: APIConfig):
        self.network = network
        self.config = config
        self.parent_network = self._get_parent_network(network)

    def _get_parent_network(self, network: NetworkType) -> NetworkType:
        """Map L2 to parent L1 network"""
        l2_mapping = {
            NetworkType.ARBITRUM_ONE: NetworkType.ETHEREUM_MAINNET,
            NetworkType.OPTIMISM: NetworkType.ETHEREUM_MAINNET,
            NetworkType.POLYGON: NetworkType.ETHEREUM_MAINNET
        }
        return l2_mapping.get(network, NetworkType.ETHEREUM_MAINNET)
```

```
    async def get_l1_gas_price(self) -> int:
        """Get L1 gas price for cross-chain operations"""
        # Implementation would query the parent network
        pass

    async def get_l2_block_details(self, block_number: int) ->
Dict[str, Any]:
        """Get L2 block with additional metadata"""
        pass

class ArbitrumClient(Layer2Client):
    """Arbitrum-specific client with nitro integration"""

    def __init__(self, config: APIConfig):
        super().__init__(NetworkType.ARBITRUM_ONE, config)
        self.arbitrum_nitro = True

    async def get_arbitrum_inbox_events(self) -> List[Dict[str, Any]]:
        """Monitor Arbitrum inbox for L1 to L2 messages"""
        # Implementation for monitoring bridge messages
        pass

    async def get_delayed_inbox_messages(self) -> List[Dict[str, Any]]:
        """Get messages from delayed inbox for arbitrage
opportunities"""
        pass

class OptimismClient(Layer2Client):
    """Optimism-specific client with Bedrock integration"""

    def __init__(self, config: APIConfig):
        super().__init__(NetworkType.OPTIMISM, config)
        self.bedrock_version = True

    async def get_l1_block_info(self, l2_block_number: int) ->
Dict[str, Any]:
        """Get L1 block information for L2 block"""
        pass
```

---

4. Connection Management and Rate Limiting

Rate Limiter Implementation

```
import asyncio
from collections import deque
import time
from typing import Optional

class RateLimiter:
    """Token bucket rate limiter with burst support"""

    def __init__(self, rate_per_second: float, burst_capacity:
Optional[float] = None):
        self.rate_per_second = rate_per_second
        self.burst_capacity = burst_capacity or rate_per_second * 2

        self.tokens = self.burst_capacity
        self.last_refill = time.time()
        self.lock = asyncio.Lock()

    async def acquire(self, tokens: float = 1.0) -> None:
        """Acquire tokens for API call"""
        async with self.lock:
            now = time.time()

            # Refill tokens based on time passed
            time_passed = now - self.last_refill
            tokens_to_add = time_passed * self.rate_per_second
            self.tokens = min(self.burst_capacity, self.tokens +
tokens_to_add)
            self.last_refill = now

            # Wait if not enough tokens
            if self.tokens < tokens:
                wait_time = (tokens - self.tokens) /
self.rate_per_second
                await asyncio.sleep(wait_time)
                self.tokens = 0
            else:
                self.tokens -= tokens

class AdaptiveRateLimiter(RateLimiter):
    """Rate limiter that adapts based on error rates"""

    def __init__(self, initial_rate: float):
```

```
        super().__init__(initial_rate)
        self.error_window = deque(maxlen=100)
        self.target_error_rate = 0.01  # 1% error rate

    async def acquire(self, tokens: float = 1.0) -> None:
        # Check recent error rate
        recent_errors = sum(1 for error in self.error_window if error)
        total_requests = len(self.error_window)

        if total_requests > 10:  # Wait until we have enough data
            error_rate = recent_errors / total_requests

            if error_rate > self.target_error_rate:
                # Reduce rate
                self.rate_per_second = max(1, self.rate_per_second *
0.9)
                self.logger.info(f"Reduced rate limit to
{self.rate_per_second} rps due to errors")

        await super().acquire(tokens)

    def record_result(self, success: bool):
        """Record request result for adaptive rate limiting"""
        self.error_window.append(not success)

class ConnectionPool:
    """Manages multiple API endpoints with failover"""

    def __init__(self, endpoints: List[str]):
        self.endpoints = endpoints
        self.current_index = 0
        self.failed_endpoints = set()
        self.last_failures = {}

    def get_next_endpoint(self) -> Optional[str]:
        """Get next available endpoint with failover"""
        available_endpoints = [ep for ep in self.endpoints if ep not in
self.failed_endpoints]

        if not available_endpoints:
            # Reset failed endpoints after timeout
            self._reset_failed_endpoints()
```

```
            available_endpoints = self.endpoints

        if not available_endpoints:
            return None

        # Round-robin through available endpoints
        endpoint = available_endpoints[self.current_index %
len(available_endpoints)]
        self.current_index += 1

        return endpoint

    def mark_failed(self, endpoint: str):
        """Mark endpoint as failed"""
        self.failed_endpoints.add(endpoint)
        self.last_failures[endpoint] = time.time()

    def _reset_failed_endpoints(self):
        """Reset failed endpoints after timeout"""
        timeout = 300  # 5 minutes
        now = time.time()

        to_remove = []
        for endpoint, failure_time in self.last_failures.items():
            if now - failure_time > timeout:
                to_remove.append(endpoint)

        for endpoint in to_remove:
            self.failed_endpoints.discard(endpoint)
            del self.last_failures[endpoint]
```

Retry and Circuit Breaker

```
import random
from enum import Enum

class CircuitState(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitBreaker:
    """Circuit breaker for API calls"""

    def __init__(self,
                 failure_threshold: int = 5,
                 recovery_timeout: int = 60,
                 expected_exception: type = Exception):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.expected_exception = expected_exception

        self.failure_count = 0
        self.state = CircuitState.CLOSED
        self.last_failure_time = None

    async def call(self, func, *args, **kwargs):
        """Execute function with circuit breaker protection"""
        if self.state == CircuitState.OPEN:
            if self._should_attempt_reset():
                self.state = CircuitState.HALF_OPEN
            else:
                raise CircuitOpenError("Circuit breaker is open")

        try:
            result = await func(*args, **kwargs)
            self._on_success()
            return result

        except self.expected_exception as e:
            self._on_failure()
            raise e

    def _should_attempt_reset(self) -> bool:
        """Check if we should attempt to reset the circuit"""
```

```
        return (self.last_failure_time and
                time.time() - self.last_failure_time >=
self.recovery_timeout)

    def _on_success(self):
        """Handle successful call"""
        self.failure_count = 0
        self.state = CircuitState.CLOSED

    def _on_failure(self):
        """Handle failed call"""
        self.failure_count += 1
        self.last_failure_time = time.time()

        if self.failure_count >= self.failure_threshold:
            self.state = CircuitState.OPEN

class CircuitOpenError(Exception):
    pass

class RetryManager:
    """Manages retry logic with exponential backoff"""

    def __init__(self,
                 max_retries: int = 3,
                 base_delay: float = 1.0,
                 max_delay: float = 60.0,
                 jitter: float = 0.1,
                 retryable_exceptions: List[type] = None):

        self.max_retries = max_retries
        self.base_delay = base_delay
        self.max_delay = max_delay
        self.jitter = jitter
        self.retryable_exceptions = retryable_exceptions or [
            NetworkError, TimeoutError, RateLimitError
        ]

    async def execute_with_retry(self, func, *args, **kwargs):
        """Execute function with retry logic"""
        last_exception = None
```

```
        for attempt in range(self.max_retries + 1):
            try:
                return await func(*args, **kwargs)

            except Exception as e:
                last_exception = e

                # Don't retry on last attempt
                if attempt == self.max_retries:
                    break

                # Check if exception is retryable
                if not any(isinstance(e, exc_type) for exc_type in
self.retryable_exceptions):
                    break

                # Calculate delay with exponential backoff and jitter
                delay = min(self.base_delay * (2 ** attempt),
self.max_delay)
                jitter_amount = delay * self.jitter * (2 *
random.random() - 1)
                final_delay = max(0, delay + jitter_amount)

                await asyncio.sleep(final_delay)

        # All retries failed
        raise last_exception
```

---

5. WebSocket Subscriptions

WebSocket Client Implementation

```
import websockets
import json
import asyncio
from typing import Dict, List, Callable, Optional
from dataclasses import dataclass, field

@dataclass
class SubscriptionConfig:
    """Configuration for WebSocket subscriptions"""
    endpoint: str
    max_subscriptions: int = 100
    reconnect_attempts: int = 10
    reconnect_delay: float = 5.0
    heartbeat_interval: float = 30.0

class WebSocketClient:
    """Production WebSocket client for real-time blockchain data"""

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

        # WebSocket connection
        self.websocket: Optional[websockets.WebSocketServerProtocol] =
None
        self.connected = False
        self.reconnect_attempts = 0

        # Subscriptions
        self.subscriptions: Dict[str, Dict] = {}
        self.subscription_handlers: Dict[str, Callable] = {}

        # Heartbeat
        self.heartbeat_task: Optional[asyncio.Task] = None
        self.last_heartbeat = time.time()

    async def connect(self):
        """Establish WebSocket connection"""
        try:
            self.websocket = await websockets.connect(
                self.config.endpoint,
                max_size=None,
```

```
                max_queue=None
            )

            self.connected = True
            self.reconnect_attempts = 0

            # Start heartbeat
            self.heartbeat_task =
asyncio.create_task(self._heartbeat_loop())

            # Start message handler
            asyncio.create_task(self._message_handler())

            self.logger.info(f"Connected to WebSocket:
{self.config.endpoint}")

        except Exception as e:
            self.logger.error(f"Failed to connect to WebSocket: {e}")
            raise

    async def disconnect(self):
        """Close WebSocket connection"""
        self.connected = False

        if self.heartbeat_task:
            self.heartbeat_task.cancel()

        if self.websocket:
            await self.websocket.close()

        self.logger.info("Disconnected from WebSocket")

    async def subscribe_new_blocks(self, handler: Callable[[Dict],
None],
                                   include_transactions: bool = True):
        """Subscribe to new block events"""
        subscription_id = f"blocks_{len(self.subscriptions)}"

        params = [include_transactions]
        if not include_transactions:
            params = [include_transactions, True]  # Full tx objects
```

```
        await self._send_subscription(
            subscription_id,
            "eth_subscribe",
            ["newHeads"] + params,
            handler
        )

        return subscription_id

    async def subscribe_pending_transactions(self, handler:
Callable[[str], None]):
        """Subscribe to pending transaction hashes"""
        subscription_id = f"pending_{len(self.subscriptions)}"

        await self._send_subscription(
            subscription_id,
            "eth_subscribe",
            ["newPendingTransactions"],
            handler
        )

        return subscription_id

    async def subscribe_logs(self,
                             handler: Callable[[Dict], None],
                             address: Optional[str] = None,
                             topics: Optional[List[str]] = None,
                             from_block: Optional[str] = None,
                             to_block: Optional[str] = None):
        """Subscribe to contract event logs"""
        subscription_id = f"logs_{len(self.subscriptions)}"

        params = {}
        if address:
            params["address"] = address
        if topics:
            params["topics"] = topics
        if from_block:
            params["fromBlock"] = from_block
        if to_block:
            params["toBlock"] = to_block
```

```
        await self._send_subscription(
            subscription_id,
            "eth_subscribe",
            ["logs", params],
            handler
        )

        return subscription_id

    async def subscribe_mempool(self,
                                handler: Callable[[Dict], None],
                                addresses: Optional[List[str]] = None,
                                min_value: Optional[int] = None):
        """Subscribe to mempool transactions (custom implementation)"""
        subscription_id = f"mempool_{len(self.subscriptions)}"

        filter_params = {}
        if addresses:
            filter_params["addresses"] = addresses
        if min_value:
            filter_params["minValue"] = min_value

        # This requires a custom endpoint that supports mempool
filtering
        await self._send_subscription(
            subscription_id,
            "eth_subscribe",
            ["newMempoolTransactions", filter_params],
            handler
        )

        return subscription_id

    async def _send_subscription(self,
                                 subscription_id: str,
                                 method: str,
                                 params: List,
                                 handler: Callable):
        """Send subscription request"""
        request = {
            "jsonrpc": "2.0",
            "id": len(self.subscriptions) + 1,
```

```
            "method": method,
            "params": params
        }

        await self.websocket.send(json.dumps(request))

        # Store subscription info
        self.subscriptions[subscription_id] = {
            "method": method,
            "params": params
        }
        self.subscription_handlers[subscription_id] = handler

        self.logger.info(f"Subscribed to {method} with ID
{subscription_id}")

    async def unsubscribe(self, subscription_id: str):
        """Unsubscribe from a feed"""
        if subscription_id not in self.subscriptions:
            raise ValueError(f"Subscription {subscription_id} not
found")

        params = [subscription_id]

        request = {
            "jsonrpc": "2.0",
            "id": len(self.subscriptions) + 100,
            "method": "eth_unsubscribe",
            "params": params
        }

        await self.websocket.send(json.dumps(request))

        # Clean up local state
        del self.subscriptions[subscription_id]
        del self.subscription_handlers[subscription_id]

        self.logger.info(f"Unsubscribed from {subscription_id}")

    async def _message_handler(self):
        """Handle incoming WebSocket messages"""
        try:
```

```
            async for message in self.websocket:
                await self._process_message(message)

        except websockets.exceptions.ConnectionClosed:
            self.logger.warning("WebSocket connection closed")
            self.connected = False
            await self._reconnect()

        except Exception as e:
            self.logger.error(f"WebSocket message handler error: {e}")
            self.connected = False
            await self._reconnect()

    async def _process_message(self, message: str):
        """Process incoming WebSocket message"""
        try:
            data = json.loads(message)

            # Handle subscription responses
            if "method" in data and data["method"] ==
"eth_subscription":
                subscription_id = data["params"]["subscription"]
                result = data["params"]["result"]

                if subscription_id in self.subscription_handlers:
                    handler =
self.subscription_handlers[subscription_id]
                    await handler(result)

            # Handle unsubscribe confirmations
            elif "method" in data and data["method"] ==
"eth_unsubscribe":
                subscription_id = data["params"]["subscription"]
                self.logger.info(f"Unsubscribed from
{subscription_id}")

        except json.JSONDecodeError as e:
            self.logger.error(f"Failed to parse WebSocket message:
{e}")
        except Exception as e:
            self.logger.error(f"Error processing WebSocket message:
{e}")
```

```
    async def _heartbeat_loop(self):
        """Send periodic heartbeat to maintain connection"""
        while self.connected:
            try:
                await asyncio.sleep(self.config.heartbeat_interval)

                # Send eth_chainId as heartbeat
                request = {
                    "jsonrpc": "2.0",
                    "id": 999,
                    "method": "eth_chainId",
                    "params": []
                }

                await self.websocket.send(json.dumps(request))
                self.last_heartbeat = time.time()

            except Exception as e:
                self.logger.error(f"Heartbeat error: {e}")
                break

    async def _reconnect(self):
        """Attempt to reconnect WebSocket"""
        if self.reconnect_attempts >= self.config.reconnect_attempts:
            self.logger.error("Max reconnection attempts reached")
            return

        self.reconnect_attempts += 1

        self.logger.info(f"Attempting to reconnect (attempt
{self.reconnect_attempts})")

        await asyncio.sleep(self.config.reconnect_delay *
self.reconnect_attempts)

        try:
            await self.connect()

            # Resubscribe to all previous subscriptions
            for sub_id, sub_info in self.subscriptions.items():
                if sub_id in self.subscription_handlers:
```

```
                    handler = self.subscription_handlers[sub_id]

                    await self._send_subscription(
                        sub_id,
                        sub_info["method"],
                        sub_info["params"],
                        handler
                    )

            self.logger.info("Reconnected and resubscribed to all
feeds")

        except Exception as e:
            self.logger.error(f"Reconnection failed: {e}")

class MempoolMonitor:
    """High-performance mempool transaction monitoring"""

    def __init__(self, websocket_client: WebSocketClient):
        self.ws_client = websocket_client
        self.logger = logging.getLogger(__name__)

        # Transaction analysis
        self.pending_transactions: Dict[str, Dict] = {}
        self.gas_price_tracker = GasPriceTracker()

        # MEV opportunity detection
        self.arbitrage_detector = ArbitrageDetector()
        self.front_running_detector = FrontRunningDetector()

    async def start_monitoring(self):
        """Start comprehensive mempool monitoring"""
        # Subscribe to pending transactions
        await self.ws_client.subscribe_pending_transactions(
            self._handle_pending_transaction
        )

        # Subscribe to specific contract events
        await self.ws_client.subscribe_logs(
            self._handle_contract_event,
            address=None,  # All contracts
            topics=None    # All events
```

```
        )

        self.logger.info("Started mempool monitoring")

    async def _handle_pending_transaction(self, tx_hash: str):
        """Process pending transaction"""
        try:
            # Get full transaction details
            tx_details = await self._get_transaction_details(tx_hash)

            if not tx_details:
                return

            # Store for analysis
            self.pending_transactions[tx_hash] = tx_details

            # Analyze for MEV opportunities
            await self._analyze_mev_opportunities(tx_details)

        except Exception as e:
            self.logger.error(f"Error handling pending transaction
{tx_hash}: {e}")

    async def _analyze_mev_opportunities(self, tx_details: Dict[str,
Any]):
        """Analyze transaction for MEV opportunities"""
        # Check for arbitrage opportunities
        if
self.arbitrage_detector.is_arbitrage_opportunity(tx_details):
            opportunity =
self.arbitrage_detector.analyze_arbitrage(tx_details)
            await self._execute_arbitrage_strategy(opportunity)

        # Check for front-running opportunities
        if
self.front_running_detector.is_front_running_opportunity(tx_details):
            opportunity =
self.front_running_detector.analyze_front_run(tx_details)
            await self._execute_front_run_strategy(opportunity)

    async def _execute_arbitrage_strategy(self, opportunity: Dict):
        """Execute arbitrage strategy"""
```

```
        self.logger.info(f"Executing arbitrage: {opportunity}")

    async def _execute_front_run_strategy(self, opportunity: Dict):
        """Execute front-running strategy"""
        self.logger.info(f"Front-running opportunity: {opportunity}")
```

---

6. Production Implementation

Complete Production Client

```
class ProductionBlockchainClient:
    """Production-ready blockchain API client with all features"""

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

        # Core components
        self.http_client: Optional[EthereumClient] = None
        self.ws_client: Optional[WebSocketClient] = None

        # Rate limiting and reliability
        self.rate_limiter = AdaptiveRateLimiter(config.rate_limit_rps)
        self.circuit_breaker = CircuitBreaker()
        self.retry_manager = RetryManager()

        # Connection management
        self.connection_pool = ConnectionPool([config.endpoint])

        # Monitoring and metrics
        self.metrics = APIMetrics()

        # Event handlers
        self.block_handlers: List[Callable] = []
        self.tx_handlers: List[Callable] = []
        self.log_handlers: List[Callable] = []

    async def start(self):
        """Start the blockchain client"""
        await self._connect()
        await self._start_monitoring()

        self.logger.info("Production blockchain client started")

    async def stop(self):
        """Stop the blockchain client"""
        await self._disconnect()
        self.logger.info("Production blockchain client stopped")

    async def _connect(self):
        """Establish all connections"""
        # HTTP connection for RPC calls
```

```
        self.http_client = EthereumClient(self.config)
        await self.http_client.connect()

        # WebSocket connection for real-time data
        ws_config = SubscriptionConfig(
            endpoint=self.config.endpoint.replace('http', 'ws'),
            max_subscriptions=1000,
            reconnect_attempts=10
        )
        self.ws_client = WebSocketClient(ws_config)
        await self.ws_client.connect()

    async def _disconnect(self):
        """Close all connections"""
        if self.http_client:
            await self.http_client.disconnect()

        if self.ws_client:
            await self.ws_client.disconnect()

    async def _start_monitoring(self):
        """Start real-time monitoring"""
        # Subscribe to new blocks
        await
self.ws_client.subscribe_new_blocks(self._handle_new_block)

        # Subscribe to pending transactions
        await
self.ws_client.subscribe_pending_transactions(self._handle_pending_tx)

        # Start monitoring tasks
        asyncio.create_task(self._gas_price_monitor())
        asyncio.create_task(self._block_metrics_monitor())

    async def _handle_new_block(self, block_data: Dict):
        """Handle new block event"""
        try:
            self.metrics.record_block(block_data.get('number'))

            # Notify all registered handlers
            for handler in self.block_handlers:
                try:
```

```
                    await handler(block_data)
                except Exception as e:
                    self.logger.error(f"Error in block handler: {e}")

        except Exception as e:
            self.logger.error(f"Error handling new block: {e}")

    async def _handle_pending_tx(self, tx_hash: str):
        """Handle pending transaction"""
        try:
            # Get transaction details
            tx_data = await
self.http_client.get_transaction_by_hash(tx_hash)

            if tx_data:
                self.metrics.record_pending_transaction(tx_hash)

                # Notify handlers
                for handler in self.tx_handlers:
                    try:
                        await handler(tx_data)
                    except Exception as e:
                        self.logger.error(f"Error in transaction
handler: {e}")

        except Exception as e:
            self.logger.error(f"Error handling pending transaction:
{e}")

    async def _gas_price_monitor(self):
        """Monitor gas prices for optimization"""
        while True:
            try:
                gas_price = await self.http_client.get_gas_price()
                self.metrics.record_gas_price(gas_price)

                await asyncio.sleep(1)  # Update every second

            except Exception as e:
                self.logger.error(f"Gas price monitor error: {e}")
                await asyncio.sleep(5)
```

```
    async def get_latest_block_optimized(self,
                                         timeout: float = 5.0) ->
Optional[Dict]:
        """Optimized method to get latest block with retries"""
        start_time = time.time()

        async def _get_block():
            return await asyncio.wait_for(
                self.http_client.get_latest_block(),
                timeout=timeout
            )

        try:
            result = await self.circuit_breaker.call(
                self.retry_manager.execute_with_retry,
                _get_block
            )

            self.metrics.record_api_call(time.time() - start_time,
True)
            return result

        except Exception as e:
            self.metrics.record_api_call(time.time() - start_time,
False)
            self.logger.error(f"Failed to get latest block: {e}")
            return None

    def add_block_handler(self, handler: Callable[[Dict], None]):
        """Add handler for new block events"""
        self.block_handlers.append(handler)

    def add_transaction_handler(self, handler: Callable[[Dict], None]):
        """Add handler for pending transaction events"""
        self.tx_handlers.append(handler)

    def add_log_handler(self, handler: Callable[[Dict], None]):
        """Add handler for log events"""
        self.log_handlers.append(handler)

    async def health_check(self) -> Dict[str, Any]:
        """Perform health check"""
```

```
        health = {
            "status": "healthy",
            "timestamp": datetime.utcnow().isoformat(),
            "components": {}
        }

        try:
            # Test HTTP connectivity
            start_time = time.time()
            await asyncio.wait_for(
                self.http_client.get_gas_price(),
                timeout=5.0
            )
            health["components"]["http"] = {
                "status": "healthy",
                "latency_ms": (time.time() - start_time) * 1000
            }

        except Exception as e:
            health["components"]["http"] = {
                "status": "unhealthy",
                "error": str(e)
            }
            health["status"] = "degraded"

        try:
            # Test WebSocket connectivity
            if self.ws_client.connected:
                health["components"]["websocket"] = {
                    "status": "healthy",
                    "subscriptions": len(self.ws_client.subscriptions)
                }
            else:
                health["components"]["websocket"] = {
                    "status": "unhealthy",
                    "error": "Not connected"
                }
                health["status"] = "degraded"

        except Exception as e:
            health["components"]["websocket"] = {
                "status": "unhealthy",
```

```
                "error": str(e)
            }
            health["status"] = "degraded"

        # Add metrics
        health["metrics"] = self.metrics.get_summary()

        return health

class APIMetrics:
    """Track API performance and usage metrics"""

    def __init__(self):
        self.call_count = 0
        self.error_count = 0
        self.total_latency = 0.0
        self.block_count = 0
        self.pending_tx_count = 0
        self.gas_price_history = deque(maxlen=3600)  # Last hour
        self.call_history = deque(maxlen=1000)

    def record_api_call(self, latency: float, success: bool):
        """Record API call metrics"""
        self.call_count += 1
        self.total_latency += latency
        self.call_history.append({
            "timestamp": time.time(),
            "latency": latency,
            "success": success
        })

        if not success:
            self.error_count += 1

    def record_block(self, block_number: int):
        """Record block processing"""
        self.block_count += 1

    def record_pending_transaction(self, tx_hash: str):
        """Record pending transaction"""
        self.pending_tx_count += 1
```

```
    def record_gas_price(self, gas_price: int):
        """Record gas price"""
        self.gas_price_history.append({
            "timestamp": time.time(),
            "gas_price": gas_price
        })

    def get_summary(self) -> Dict[str, Any]:
        """Get metrics summary"""
        return {
            "total_calls": self.call_count,
            "error_rate": self.error_count / max(1, self.call_count),
            "avg_latency_ms": (self.total_latency / max(1,
self.call_count)) * 1000,
            "blocks_processed": self.block_count,
            "pending_transactions": self.pending_tx_count,
            "current_gas_price": self.gas_price_history[-1]
["gas_price"] if self.gas_price_history else 0
        }
```

---

7. Error Handling and Recovery

Comprehensive Error Handling

```
import traceback
from typing import Optional, Type
import logging
from datetime import datetime, timedelta

class BlockchainErrorHandler:
    """Comprehensive error handling for blockchain operations"""

    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.error_history: List[Dict] = []
        self.failure_patterns: Dict[str, int] = {}

    async def handle_api_error(self,
                               error: Exception,
                               operation: str,
                               context: Dict,
                               retry_func: Optional[Callable] = None) ->
bool:
        """Handle API errors with intelligent recovery"""

        error_info = {
            "timestamp": datetime.utcnow(),
            "error_type": type(error).__name__,
            "error_message": str(error),
            "operation": operation,
            "context": context,
            "traceback": traceback.format_exc()
        }

        self.error_history.append(error_info)
        self._analyze_failure_pattern(error_info)

        # Determine recovery strategy
        recovery_strategy = self._determine_recovery_strategy(error)

        if recovery_strategy["should_retry"] and retry_func:
            success = await self._execute_recovery(
                error, retry_func, recovery_strategy
            )

            if success:
```

```
                self.logger.info(f"Recovered from
{type(error).__name__} in {operation}")
                return True
            else:
                self.logger.error(f"Failed to recover from
{type(error).__name__} in {operation}")
                return False
        else:
            self.logger.error(f"Cannot recover from
{type(error).__name__} in {operation}")
            return False

    def _analyze_failure_pattern(self, error_info: Dict):
        """Analyze error patterns for proactive recovery"""
        error_key = f"{error_info['error_type']}
_{error_info['operation']}"
        self.failure_patterns[error_key] =
self.failure_patterns.get(error_key, 0) + 1

        # If same error is repeating, trigger circuit breaker
        if self.failure_patterns[error_key] > 10:
            self.logger.warning(f"High failure rate detected:
{error_key}")

    def _determine_recovery_strategy(self, error: Exception) ->
Dict[str, Any]:
        """Determine appropriate recovery strategy for error type"""

        strategies = {
            RateLimitError: {
                "should_retry": True,
                "backoff_multiplier": 2.0,
                "max_retries": 5,
                "action": "reduce_rate_limit"
            },
            TimeoutError: {
                "should_retry": True,
                "backoff_multiplier": 1.5,
                "max_retries": 3,
                "action": "extend_timeout"
            },
            NetworkError: {
```

```
                "should_retry": True,
                "backoff_multiplier": 2.0,
                "max_retries": 3,
                "action": "failover_endpoint"
            },
            CircuitOpenError: {
                "should_retry": False,
                "backoff_multiplier": 0,
                "max_retries": 0,
                "action": "wait_and_retry"
            }
        }

        return strategies.get(type(error), {
            "should_retry": False,
            "backoff_multiplier": 1.0,
            "max_retries": 0,
            "action": "fail"
        })

    async def _execute_recovery(self,
                                error: Exception,
                                retry_func: Callable,
                                strategy: Dict) -> bool:
        """Execute recovery procedure"""

        retries = 0
        max_retries = strategy["max_retries"]

        while retries <= max_retries:
            try:
                # Apply recovery action
                if strategy["action"] == "reduce_rate_limit":
                    await self._reduce_rate_limit()
                elif strategy["action"] == "extend_timeout":
                    await self._extend_timeout()
                elif strategy["action"] == "failover_endpoint":
                    await self._failover_endpoint()
                elif strategy["action"] == "wait_and_retry":
                    await asyncio.sleep(30)
# Wait for circuit breaker to close
```

```
                # Attempt retry
                await retry_func()
                return True

            except Exception as retry_error:
                retries += 1

                if retries > max_retries:
                    return False

                # Exponential backoff
                backoff = strategy["backoff_multiplier"] ** retries
                await asyncio.sleep(backoff)

        return False

    async def _reduce_rate_limit(self):
        """Reduce API rate limit"""
        pass  # Implementation depends on rate limiter

    async def _extend_timeout(self):
        """Extend API timeout"""
        pass  # Implementation depends on client

    async def _failover_endpoint(self):
        """Switch to backup endpoint"""
        pass  # Implementation depends on connection pool

    def get_error_report(self) -> Dict[str, Any]:
        """Generate comprehensive error report"""
        recent_errors = [
            e for e in self.error_history
            if datetime.utcnow() - e["timestamp"] < timedelta(hours=1)
        ]

        return {
            "total_errors_24h": len([
                e for e in self.error_history
                if datetime.utcnow() - e["timestamp"] <
timedelta(days=1)
            ]),
            "error_types": list(set(e["error_type"] for e in
```

```
recent_errors)),
            "most_common_errors": dict(sorted(
                self.failure_patterns.items(),
                key=lambda x: x[1],
                reverse=True
            )[:5]),
            "recent_errors": recent_errors[-10:]
        }

class ConnectionRecoveryManager:
    """Manages connection recovery and failover"""

    def __init__(self, client: ProductionBlockchainClient):
        self.client = client
        self.logger = logging.getLogger(__name__)
        self.recovery_attempts = 0
        self.max_recovery_attempts = 5
        self.recovery_delay = 10  # seconds

    async def attempt_recovery(self) -> bool:
        """Attempt to recover from connection failure"""

        for attempt in range(self.max_recovery_attempts):
            try:
                self.logger.info(f"Recovery attempt {attempt + 1}/
{self.max_recovery_attempts}")

                # Disconnect existing connections
                await self.client._disconnect()

                # Wait before reconnecting
                await asyncio.sleep(self.recovery_delay * (attempt +
1))

                # Reconnect
                await self.client._connect()
                await self.client._start_monitoring()

                # Verify connection
                if await self._verify_connection():
                    self.logger.info("Connection recovery successful")
                    return True
```

```
            except Exception as e:
                self.logger.error(f"Recovery attempt {attempt + 1}
failed: {e}")

        self.logger.error("All recovery attempts failed")
        return False

    async def _verify_connection(self) -> bool:
        """Verify that connections are working"""
        try:
            # Test HTTP connection
            await asyncio.wait_for(
                self.client.http_client.get_gas_price(),
                timeout=5.0
            )

            # Test WebSocket connection
            if not self.client.ws_client.connected:
                return False

            return True

        except Exception as e:
            self.logger.error(f"Connection verification failed: {e}")
            return False

# Integration with the main client
class EnhancedProductionClient(ProductionBlockchainClient):
    """Enhanced production client with advanced error handling"""

    def __init__(self, config: APIConfig):
        super().__init__(config)
        self.error_handler = BlockchainErrorHandler()
        self.recovery_manager = ConnectionRecoveryManager(self)

    async def safe_execute(self,
                           operation: str,
                           operation_func: Callable,
                           context: Dict = None) -> Optional[Any]:
        """Execute operation with comprehensive error handling"""
```

```
        try:
            return await operation_func()

        except Exception as e:
            if context is None:
                context = {}

            context["client_config"] = {
                "endpoint": self.config.endpoint,
                "network": self.config.network.value,
                "api_type": self.config.api_type.value
            }

            # Attempt recovery
            recovery_success = await
self.error_handler.handle_api_error(
                e, operation, context, operation_func
            )

            if not recovery_success:
                # Try connection recovery
                recovery_success = await
self.recovery_manager.attempt_recovery()

                if recovery_success:
                    # Retry operation after recovery
                    try:
                        return await operation_func()
                    except Exception as retry_error:

self.logger.error(f"Operation failed even after recovery:
{retry_error}")
                        return None

            return None
```

---

8. Performance Optimization

Optimization Techniques

```
import asyncio
from typing import List, Dict, Any, Set
from collections import defaultdict
import heapq

class BatchProcessor:
    """Batch API calls for improved performance"""

    def __init__(self, batch_size: int = 10, max_wait_time: float =
1.0):
        self.batch_size = batch_size
        self.max_wait_time = max_wait_time
        self.pending_requests: List[Dict] = []
        self.processing = False

    async def add_request(self,
                          method: str,
                          params: List,
                          result_handler: Callable) -> str:
        """Add request to batch queue"""
        request_id = f"batch_{len(self.pending_requests)}"

        self.pending_requests.append({
            "id": request_id,
            "jsonrpc": "2.0",
            "method": method,
            "params": params,
            "handler": result_handler
        })

        # Process batch if full or if processing
        if len(self.pending_requests) >= self.batch_size:
            await self.process_batch()
        else:
            # Schedule processing after max wait time
            asyncio.create_task(self._schedule_processing())

        return request_id

    async def _schedule_processing(self):
        """Schedule batch processing after wait time"""
        await asyncio.sleep(self.max_wait_time)
```

```
        if self.pending_requests and not self.processing:
            await self.process_batch()

    async def process_batch(self):
        """Process all pending requests as batch"""
        if not self.pending_requests or self.processing:
            return

        self.processing = True
        requests = self.pending_requests[:]
        self.pending_requests.clear()

        try:
            # Convert to batch request
            batch_request = []
            for req in requests:
                batch_request.append({
                    "jsonrpc": "2.0",
                    "id": req["id"],
                    "method": req["method"],
                    "params": req["params"]
                })

            # Send batch request
            async with self.client.http_client.session.post(
                self.client.config.endpoint,
                json=batch_request
            ) as response:
                results = await response.json()

                # Distribute results to handlers
                for result in results:
                    request_id = result["id"]

                    # Find corresponding request and handler
                    for req in requests:
                        if req["id"] == request_id:
                            if "error" in result:
                                await req["handler"](None,
result["error"])
                            else:
                                await req["handler"](result["result"],
```

```
None)
                            break

        except Exception as e:
            self.logger.error(f"Batch processing failed: {e}")

            # Notify all handlers of failure
            for req in requests:
                await req["handler"](None, str(e))

        finally:
            self.processing = False

    async def flush(self):
        """Process all pending requests immediately"""
        if self.pending_requests:
            await self.process_batch()

class CacheManager:
    """High-performance cache for blockchain data"""

    def __init__(self, max_size: int = 10000):
        self.cache: Dict[str, Dict] = {}
        self.access_times: Dict[str, float] = {}
        self.max_size = max_size
        self.cache_hits = 0
        self.cache_misses = 0

        # LRU tracking
        self.lru_queue: List[tuple] = []

    def get(self, key: str) -> Optional[Any]:
        """Get value from cache"""
        if key in self.cache:
            self.cache_hits += 1

            # Update LRU
            self._update_lru(key)

            return self.cache[key]["value"]
        else:
            self.cache_misses += 1
```

```
            return None

    def set(self, key: str, value: Any, ttl: Optional[float] = None):
        """Set value in cache"""
        now = time.time()

        # Evict if full
        if len(self.cache) >= self.max_size and key not in self.cache:
            self._evict_lru()

        self.cache[key] = {
            "value": value,
            "timestamp": now,
            "ttl": ttl
        }
        self.access_times[key] = now

        # Update LRU
        heapq.heappush(self.lru_queue, (now, key))

    def _update_lru(self, key: str):
        """Update LRU tracking for key"""
        now = time.time()
        self.access_times[key] = now

        # Remove old entry and add new one
        self.lru_queue = [(ts, k) for ts, k in self.lru_queue if k !=
key]
        heapq.heappush(self.lru_queue, (now, key))

    def _evict_lru(self):
        """Evict least recently used item"""
        while self.lru_queue:
            timestamp, key = heapq.heappop(self.lru_queue)
            if key in self.cache:
                del self.cache[key]
                del self.access_times[key]
                break

    def cleanup_expired(self):
        """Remove expired entries"""
        now = time.time()
```

```
        expired_keys = []

        for key, data in self.cache.items():
            if data["ttl"] and now - data["timestamp"] > data["ttl"]:
                expired_keys.append(key)

        for key in expired_keys:
            self._remove(key)

    def _remove(self, key: str):
        """Remove key from all tracking structures"""
        if key in self.cache:
            del self.cache[key]
        if key in self.access_times:
            del self.access_times[key]

        # Remove from LRU queue
        self.lru_queue = [(ts, k) for ts, k in self.lru_queue if k !=
key]

    def get_stats(self) -> Dict[str, Any]:
        """Get cache statistics"""
        total_requests = self.cache_hits + self.cache_misses
        return {
            "hit_rate": self.cache_hits / max(1, total_requests),
            "size": len(self.cache),
            "max_size": self.max_size,
            "hits": self.cache_hits,
            "misses": self.cache_misses
        }

class SmartDataFetcher:
    """Intelligent data fetching with caching and batching"""

    def __init__(self, client: ProductionBlockchainClient):
        self.client = client
        self.cache = CacheManager()
        self.batch_processor = BatchProcessor()

        # Track pending requests to avoid duplicates
        self.pending_fetches: Dict[str, asyncio.Future] = {}
```

```
        # Prefetch management
        self.prefetch_queue: asyncio.Queue = asyncio.Queue()

    async def get_block_cached(self, block_number: int, force_refresh:
bool = False) -> Optional[Dict]:
        """Get block with caching"""
        cache_key = f"block_{block_number}"

        # Check cache first
        if not force_refresh:
            cached_result = self.cache.get(cache_key)
            if cached_result:
                return cached_result

        # Check if already fetching
        if block_number in self.pending_fetches:
            return await self.pending_fetches[block_number]

        # Create new fetch task
        future = asyncio.create_task(self._fetch_block(block_number))
        self.pending_fetches[block_number] = future

        try:
            result = await future
            return result
        finally:
            del self.pending_fetches[block_number]

    async def _fetch_block(self, block_number: int) -> Optional[Dict]:
        """Fetch block from API"""
        cache_key = f"block_{block_number}"

        try:
            # Use optimized batch fetching
            result = await self.client.get_latest_block_optimized()

            if result:
                # Cache for 10 seconds (blocks change frequently)
                self.cache.set(cache_key, result, ttl=10.0)

                # Prefetch next few blocks
                asyncio.create_task(self._prefetch_blocks(block_number
```

```
+ 1, block_number + 5))

            return result

        except Exception as e:
            self.logger.error(f"Failed to fetch block {block_number}:
{e}")
            return None

    async def _prefetch_blocks(self, start_block: int, end_block: int):
        """Prefetch upcoming blocks"""
        for block_num in range(start_block, end_block + 1):
            cache_key = f"block_{block_num}"

            # Only prefetch if not already in cache
            if not self.cache.get(cache_key):
                try:
                    await asyncio.sleep(0.1)  # Small delay to avoid
overwhelming

                    # This would be a lightweight block header fetch
                    # await
self.client.http_client.get_block_header(block_num)

                except Exception as e:
                    self.logger.debug(f"Prefetch failed for block
{block_num}: {e}")

    async def get_balances_bulk(self, addresses: List[str]) ->
Dict[str, int]:
        """Get balances for multiple addresses efficiently"""
        results = {}
        missing_addresses = []

        # Check cache first
        for address in addresses:
            cache_key = f"balance_{address}"
            cached_balance = self.cache.get(cache_key)

            if cached_balance is not None:
                results[address] = cached_balance
            else:
```

```
                missing_addresses.append(address)

        # Fetch missing balances
        if missing_addresses:
            try:
                # Use batch API for multiple balances
                balances = await
self.client.http_client.batch_get_balance(missing_addresses)

                # Cache results
                for address, balance in balances.items():
                    cache_key = f"balance_{address}"
                    self.cache.set(cache_key, balance, ttl=30.0)  #
Cache for 30 seconds
                    results[address] = balance

            except Exception as e:
                self.logger.error(f"Failed to fetch balances for
{missing_addresses}: {e}")

        return results

    async def get_logs_optimized(self,
                                 from_block: int,
                                 to_block: int,
                                 address: Optional[str] = None) ->
List[Dict]:
        """Get logs with intelligent caching and batching"""
        # Split large ranges into chunks
        chunk_size = 1000  # blocks
        all_logs = []

        for chunk_start in range(from_block, to_block + 1, chunk_size):
            chunk_end = min(chunk_start + chunk_size - 1, to_block)

            try:
                logs = await
self.client.http_client.get_logs(chunk_start, chunk_end, address)
                all_logs.extend(logs)

                # Cache log blocks
                for log in logs:
```

```
                    block_num = int(log['blockNumber'], 16)
                    cache_key = f"logs_block_{block_num}_{address or
'all'}"
                    self.cache.set(cache_key, [log], ttl=60.0)
# Cache for 1 minute

            except Exception as e:
                self.logger.error(f"Failed to fetch logs for blocks
{chunk_start}-{chunk_end}: {e}")

        return all_logs

# Performance monitoring
class PerformanceMonitor:
    """Monitor and optimize API performance"""

    def __init__(self):
        self.metrics = defaultdict(list)
        self.alerts = []

    def record_operation(self, operation: str, duration: float,
success: bool):
        """Record operation performance"""
        self.metrics[operation].append({
            "timestamp": time.time(),
            "duration": duration,
            "success": success
        })

        # Keep only recent metrics
        cutoff = time.time() - 3600  # 1 hour
        self.metrics[operation] = [
            m for m in self.metrics[operation]
            if m["timestamp"] > cutoff
        ]

    def get_performance_report(self) -> Dict[str, Any]:
        """Generate performance report"""
        report = {}

        for operation, measurements in self.metrics.items():
            if not measurements:
```

```
                continue

            durations = [m["duration"] for m in measurements]
            successes = sum(1 for m in measurements if m["success"])

            report[operation] = {
                "count": len(measurements),
                "success_rate": successes / len(measurements),
                "avg_duration_ms": (sum(durations) / len(durations)) *
1000,
                "p95_duration_ms": self._percentile(durations, 95) *
1000,
                "max_duration_ms": max(durations) * 1000
            }

            # Detect performance issues
            if report[operation]["success_rate"] < 0.95:
                self.alerts.append(f"Low success rate for {operation}:
{report[operation]['success_rate']:.2%}")

            if report[operation]["avg_duration_ms"] > 1000:  # 1 second
                self.alerts.append(f"High latency for {operation}:
{report[operation]['avg_duration_ms']:.0f}ms")

        return report

    def _percentile(self, data: List[float], percentile: float) ->
float:
        """Calculate percentile of data"""
        sorted_data = sorted(data)
        index = (percentile / 100) * (len(sorted_data) - 1)

        if index.is_integer():
            return sorted_data[int(index)]
        else:
            lower = sorted_data[int(index)]
            upper = sorted_data[int(index) + 1]
            return lower + (upper - lower) * (index - int(index))
```

---

9. Best Practices

Production Deployment Guidelines

```
class ProductionConfig:
    """Production configuration for blockchain clients"""

    @staticmethod
    def get_ethereum_mainnet_config() -> APIConfig:
        """Get optimized Ethereum mainnet configuration"""
        return APIConfig(
            network=NetworkType.ETHEREUM_MAINNET,
            api_type=APIType.RPC_HTTP,
            endpoint="https://eth-mainnet.g.alchemy.com/v2/
YOUR_API_KEY",
            rate_limit_rps=100,  # Conservative for production
            timeout_seconds=10,
            max_retries=3,
            retry_backoff=2.0,
            enable_connection_pool=True,
            max_connections=20,
            enable_metrics=True
        )

    @staticmethod
    def get_bsc_mainnet_config() -> APIConfig:
        """Get optimized BSC mainnet configuration"""
        return APIConfig(
            network=NetworkType.BSC_MAINNET,
            api_type=APIType.RPC_HTTP,
            endpoint="https://bsc-dataseed.binance.org/",
            rate_limit_rps=150,
            timeout_seconds=15,
            max_retries=3,
            enable_connection_pool=True,
            max_connections=15
        )

class MonitoringSetup:
    """Setup comprehensive monitoring"""

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

    def setup_health_checks(self):
```

```
        """Setup periodic health checks"""
        async def health_check_loop():
            while True:
                try:
                    health = await self.client.health_check()

                    # Log health status
                    if health["status"] == "healthy":
                        self.logger.info("System health: OK")
                    else:
                        self.logger.warning(f"System health:
{health['status']}")

                    # Check individual components
                    for component, status in
health["components"].items():
                        if status["status"] != "healthy":
                            self.logger.error(f"Component {component}
unhealthy: {status}")

                    # Alert if error rate is too high
                    metrics = health.get("metrics", {})
                    error_rate = metrics.get("error_rate", 0)
                    if error_rate > 0.05:  # 5% error rate
                        self.logger.error(f"High error rate detected:
{error_rate:.2%}")

                except Exception as e:
                    self.logger.error(f"Health check failed: {e}")

                await asyncio.sleep(30)  # Check every 30 seconds

        asyncio.create_task(health_check_loop())

    def setup_performance_monitoring(self):
        """Setup performance monitoring"""
        async def performance_monitor_loop():
            while True:
                try:
                    # Record performance metrics
                    if self.client.http_client:
                        avg_latency =
```

```
self.client.http_client.latency_tracker.get_avg_latency()
                        self.logger.info(f"Average API latency:
{avg_latency:.2f}ms")

                    # Monitor connection health
                    if self.client.ws_client:
                        if not self.client.ws_client.connected:
                            self.logger.warning("WebSocket connection
lost")
                        else:
                            subscription_count =
len(self.client.ws_client.subscriptions)
                            if subscription_count > 90:  # Near limit

self.logger.warning(f"High subscription count: {subscription_count}")

                    await asyncio.sleep(60)  # Monitor every minute

                except Exception as e:
                    self.logger.error(f"Performance monitoring error:
{e}")

        asyncio.create_task(performance_monitor_loop())

# Security best practices
class SecureAPIClient:
    """Secure API client with proper key management"""

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

        # Validate API key security
        self._validate_api_key()

    def _validate_api_key(self):
        """Validate API key security"""
        if self.config.api_key:
            # Check if API key is loaded from secure storage
            if self.config.api_key.startswith("raw_"):

self.logger.warning("Using raw API key - consider using environment
```

```
variables")

            # Log security warning
            self.logger.info("API key validation completed")

    async def secure_request(self, method: str, params: List) -> Dict:
        """Make request with security checks"""
        # Add security headers
        headers = {
            "Content-Type": "application/json",
            "User-Agent": "MEV-Strategy/1.0",
            "X-API-Key": self.config.api_key
        }

        # Make request
        payload = {
            "jsonrpc": "2.0",
            "method": method,
            "params": params,
            "id": int(time.time())
        }

        async with self.client.session.post(
            self.config.endpoint,
            json=payload,
            headers=headers,

timeout=aiohttp.ClientTimeout(total=self.config.timeout_seconds)
        ) as response:
            return await response.json()

# Usage example
async def main():
    """Example usage of production blockchain client"""

    # Configure client
    config = ProductionConfig.get_ethereum_mainnet_config()
    client = ProductionBlockchainClient(config)

    # Setup monitoring
    monitoring = MonitoringSetup(client)
    monitoring.setup_health_checks()
```

```
    monitoring.setup_performance_monitoring()

    try:
        # Start client
        await client.start()

        # Add custom handlers
        async def handle_new_block(block_data):
            print(f"New block: {block_data.get('number')}")

        async def handle_pending_tx(tx_data):
            print(f"Pending tx: {tx_data.get('hash')}")

        client.add_block_handler(handle_new_block)
        client.add_transaction_handler(handle_pending_tx)

        # Keep running
        while True:
            await asyncio.sleep(1)

    except KeyboardInterrupt:
        print("Shutting down...")
    finally:
        await client.stop()

if __name__ == "__main__":
    # Run production client
    asyncio.run(main())
```

---

10. Practical Exercise

Exercise: Build MEV Monitoring System

```
class MEVMonitor:
    """Real-time MEV opportunity monitoring system"""

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

        # Opportunity detection
        self.arbitrage_opportunities = []
        self.liquidation_opportunities = []
        self.frontrun_opportunities = []

        # Performance tracking
        self.opportunities_detected = 0
        self.opportunities_executed = 0

    async def start_monitoring(self):
        """Start comprehensive MEV monitoring"""

        # Subscribe to new blocks
        self.client.add_block_handler(self._analyze_block)

        # Subscribe to pending transactions
        self.client.add_transaction_handler(self._analyze_pending_tx)

        # Subscribe to relevant contract events
        await self._subscribe_to_mev_events()

        self.logger.info("MEV monitoring started")

    async def _analyze_block(self, block_data: Dict):
        """Analyze new block for MEV opportunities"""
        try:
            block_number = int(block_data['number'], 16)
            transactions = block_data.get('transactions', [])

            # Look for liquidation opportunities
            await self._detect_liquidation_opportunities(transactions)

            # Look for sandwich opportunities
            await self._detect_sandwich_opportunities(transactions)
```

```
        except Exception as e:
            self.logger.error(f"Error analyzing block: {e}")

    async def _analyze_pending_tx(self, tx_data: Dict):
        """Analyze pending transaction for arbitrage opportunities"""
        try:
            # Check if transaction contains arbitrage opportunity
            if self._is_arbitrage_candidate(tx_data):
                opportunity =
self._calculate_arbitrage_opportunity(tx_data)
                if opportunity['profit'] > 1.0:  # $1 minimum profit
                    self.arbitrage_opportunities.append(opportunity)
                    self.opportunities_detected += 1

                    self.logger.info(f"Arbitrage opportunity: $
{opportunity['profit']:.2f}")

        except Exception as e:
            self.logger.error(f"Error analyzing pending transaction:
{e}")

    def _is_arbitrage_candidate(self, tx_data: Dict) -> bool:
        """Check if transaction might contain arbitrage opportunity"""
        # Simple heuristic: large value transactions to/from multiple
DEXs
        value_wei = int(tx_data.get('value', '0'), 16)
        value_eth = value_wei / (10 ** 18)

        # Large transaction threshold (e.g., > 10 ETH)
        return value_eth > 10

    def _calculate_arbitrage_opportunity(self, tx_data: Dict) -> Dict:
        """Calculate potential arbitrage opportunity"""
        # This would involve complex price analysis across DEXs
        # Simplified example:
        profit = 10.0  # Placeholder calculation
        confidence = 0.7

        return {
            'type': 'arbitrage',
            'transaction_hash': tx_data['hash'],
            'profit': profit,
```

```
            'confidence': confidence,
            'timestamp': time.time()
        }

    async def _detect_liquidation_opportunities(self, transactions:
List[Dict]):
        """Detect liquidation opportunities in block"""
        # Check for liquidation transactions
        for tx in transactions:
            # Look for transactions to known liquidation contracts
            to_address = tx.get('to', '').lower()

            if self._is_liquidation_contract(to_address):
                self.liquidation_opportunities.append({
                    'type': 'liquidation',
                    'transaction_hash': tx['hash'],
                    'block_number': tx.get('blockNumber'),
                    'timestamp': time.time()
                })

    def _is_liquidation_contract(self, address: str) -> bool:
        """Check if address is a known liquidation contract"""
        # Known liquidation contract addresses (Aave, Compound, etc.)
        liquidation_contracts = [
            "0x7d2768de32b0b80b7a3454c06bdac94a69ddc7a9",  # Aave Pool
            "0x3d9819210a31b4961b30ef54be2aed79b9c4483e",  # Compound
Comptroller
            # Add more as needed
        ]

        return address in liquidation_contracts

    async def _detect_sandwich_opportunities(self, transactions:
List[Dict]):
        """Detect sandwich attack opportunities"""
        # Look for large swaps that could be sandwiched
        for i, tx in enumerate(transactions):
            if self._is_swap_transaction(tx):
                # Analyze surrounding transactions
                before_tx = transactions[i-1] if i > 0 else None
                after_tx = transactions[i+1] if i < len(transactions)-1
else None
```

```
                if self._could_be_sandwiched(tx, before_tx, after_tx):
                    self.frontrun_opportunities.append({
                        'type': 'sandwich',
                        'target_transaction': tx['hash'],
                        'potential_profit':
self._estimate_sandwich_profit(tx),
                        'timestamp': time.time()
                    })

    def _is_swap_transaction(self, tx: Dict) -> bool:
        """Check if transaction is a DEX swap"""
        to_address = tx.get('to', '').lower()

        # Known DEX router addresses
        dex_routers = [
            "0x7a250d5630b4cf539739df2c5dacb4c659f2488d",  # Uniswap V2
            "0xe592427a0aece92de3e3461a08cc53721faff32e",  # Uniswap V3
            "0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f",  # SushiSwap
            # Add more as needed
        ]

        return to_address in dex_routers

    def _could_be_sandwiched(self, target_tx: Dict, before_tx: Dict,
after_tx: Dict) -> bool:
        """Determine if transaction could be sandwiched"""
        # Simplified heuristic
        if not before_tx or not after_tx:
            return False

        # Check if transactions are from different addresses (likely
different users)
        if (target_tx.get('from') == before_tx.get('from') or
            target_tx.get('from') == after_tx.get('from')):
            return False

        return True

    def _estimate_sandwich_profit(self, tx: Dict) -> float:
        """Estimate potential sandwich profit"""
        # Simplified calculation - would need complex MEV analysis
```

```
        value_wei = int(tx.get('value', '0'), 16)
        value_eth = value_wei / (10 ** 18)

        # Estimate 0.1% profit on transaction value
        return value_eth * 2000 * 0.001  # Assuming ETH price of $2000

    async def _subscribe_to_mev_events(self):
        """Subscribe to relevant contract events for MEV monitoring"""
        # This would involve subscribing to specific contract events
        pass

    def get_opportunities_summary(self) -> Dict:
        """Get summary of detected opportunities"""
        return {
            'total_detected': self.opportunities_detected,
            'total_executed': self.opportunities_executed,
            'arbitrage_count': len(self.arbitrage_opportunities),
            'liquidation_count': len(self.liquidation_opportunities),
            'sandwich_count': len(self.frontrun_opportunities),
            'success_rate': self.opportunities_executed / max(1,
self.opportunities_detected)
        }

# Exercise completion
async def complete_exercise():
    """Complete the MEV monitoring exercise"""

    # 1. Create production client
    config = ProductionConfig.get_ethereum_mainnet_config()
    client = ProductionBlockchainClient(config)

    # 2. Start client
    await client.start()

    # 3. Create MEV monitor
    mev_monitor = MEVMonitor(client)
    await mev_monitor.start_monitoring()

    # 4. Run monitoring for demonstration
    print("Starting MEV monitoring system...")
    print("Press Ctrl+C to stop")
```

```
    try:
        # Monitor for opportunities
        await asyncio.sleep(60)  # Run for 1 minute

        # Show results
        summary = mev_monitor.get_opportunities_summary()
        print(f"Monitoring Summary: {summary}")

    except KeyboardInterrupt:
        print("\nMonitoring stopped")
    finally:
        await client.stop()

if __name__ == "__main__":
    asyncio.run(complete_exercise())
```

Summary

This module provides comprehensive coverage of blockchain API integration for production MEV systems. Key takeaways:

✅ What You Learned:

- **Multi-network support** for Ethereum, BSC, and Layer 2 networks
- **Production-ready clients** with rate limiting, retry logic, and circuit breakers
- **WebSocket subscriptions** for real-time blockchain data
- **Error handling and recovery** with intelligent failover mechanisms
- **Performance optimization** through caching, batching, and connection pooling
- **Security best practices** for API key management and secure communications

🚀 Production Implementation:

- **Connection pooling** and automatic failover
- **Adaptive rate limiting** based on error rates
- **Comprehensive monitoring** with health checks and metrics
- **MEV opportunity detection** and monitoring system
- **Real-time data processing** with WebSocket subscriptions

💡 Best Practices:

- Always use connection pooling and timeout handling
- Implement circuit breakers for fault tolerance
- Monitor performance and error rates continuously
- Cache frequently accessed data with appropriate TTL
- Use batch operations for efficiency
- Implement proper retry logic with exponential backoff

The blockchain API integration skills from this module form the foundation for building robust, production-ready MEV systems that can operate reliably in the demanding environment of decentralized finance.

Next: In Module 2, we'll explore DeFi protocol API integration, focusing on DEX connections, lending protocols, and yield farming platforms for comprehensive MEV strategy implementation.