Module 2: Tenderly Transaction Simulation

Course: Tool Usage Tutorials
Module: 2 of 6
Duration: 75 minutes
Level: Beginner-Intermediate
Author: Obelisk Core

Table of Contents

Introduction to Tenderly

Tenderly is a comprehensive blockchain development platform that provides powerful transaction simulation, debugging, and monitoring capabilities. For MEV practitioners, Tenderly is invaluable for testing strategies safely before committing real funds, analyzing transaction failures, and optimizing gas usage.

Why Tenderly for MEV?

Safe Strategy Testing
- Simulate complex MEV transactions without risk
- Test multiple scenarios and edge cases
- Validate transaction logic before execution

Gas Optimization
- Precise gas estimation for complex transactions

- Identify gas-inefficient operations
- Optimize transaction ordering for better profitability

Debugging Capabilities
- Deep transaction analysis and stack traces
- State change visualization
- Error diagnosis and resolution

Performance Analysis
- Transaction execution profiling
- Bottleneck identification
- Performance optimization insights

Key Features for MEV

1. **Transaction Simulation**: Test transactions against live or historical blockchain state
2. **Gas Profiler**: Detailed gas usage analysis and optimization recommendations
3. **Debugger**: Step-through transaction execution with full state visibility
4. **Fork Networks**: Create private blockchain forks for testing
5. **State Overrides**: Modify blockchain state for testing scenarios
6. **Batch Simulations**: Test multiple transactions simultaneously
7. **API Integration**: Programmatic access to all features

Setting Up Tenderly Environment

Account Setup

1. **Create Tenderly Account**
   - Visit tenderly.co
   - Sign up with GitHub or email
   - Verify your account
2. **Create Organization and Project**
   ```bash
   # Install Tenderly CLI
   npm install -g @tenderly/cli

# Login to Tenderly
tenderly login
```

Initialize project
tenderly init
```

1. **Generate API Keys**
   - Go to Settings → Authorization
   - Create new API key
   - Store securely in environment variables

## Python SDK Installation

```
Install required packages
pip install requests aiohttp web3 python-dotenv

Create requirements.txt
cat > requirements.txt << EOF
requests>=2.31.0
aiohttp>=3.8.0
web3>=6.0.0
python-dotenv>=1.0.0
eth-abi>=4.0.0
eth-utils>=2.0.0
EOF

pip install -r requirements.txt
```
```

Environment Configuration

```
# .env file
TENDERLY_API_KEY=your_api_key_here
TENDERLY_PROJECT=your_project_name
TENDERLY_USERNAME=your_username
WEB3_PROVIDER_URL=https://eth-mainnet.alchemyapi.io/v2/your_key
PRIVATE_KEY=your_private_key_here

# Load environment variables
from dotenv import load_dotenv
import os

load_dotenv()

TENDERLY_API_KEY = os.getenv('TENDERLY_API_KEY')
TENDERLY_PROJECT = os.getenv('TENDERLY_PROJECT')
TENDERLY_USERNAME = os.getenv('TENDERLY_USERNAME')
WEB3_PROVIDER_URL = os.getenv('WEB3_PROVIDER_URL')
```

Basic Tenderly Client Setup

```
import requests
import json
from web3 import Web3
from typing import Dict, Any, Optional

class TenderlyClient:
    def __init__(self, api_key: str, username: str, project: str):
        self.api_key = api_key
        self.username = username
        self.project = project
        self.base_url = "https://api.tenderly.co/api/v1"

        self.headers = {
            "Content-Type": "application/json",
            "X-Access-Key": self.api_key
        }

    def simulate_transaction(self, transaction_data: Dict[str, Any]) ->
Dict[str, Any]:
        """Simulate a transaction using Tenderly API"""
        url = f"{self.base_url}/account/{self.username}/project/
{self.project}/simulate"

        payload = {
            "network_id": "1",  # Mainnet
            "from": transaction_data["from"],
            "to": transaction_data["to"],
            "input": transaction_data.get("data", "0x"),
            "gas": transaction_data.get("gas", 8000000),
            "gas_price": transaction_data.get("gasPrice", "0"),
            "value": transaction_data.get("value", "0"),
            "save": True,
            "save_if_fails": True
        }

        response = requests.post(url, headers=self.headers,
json=payload)
        return response.json()

    def get_simulation(self, simulation_id: str) -> Dict[str, Any]:
        """Get simulation results by ID"""
        url = f"{self.base_url}/account/{self.username}/project/
```

```
{self.project}/simulations/{simulation_id}"
        response = requests.get(url, headers=self.headers)
        return response.json()

# Initialize client
tenderly = TenderlyClient(
    api_key=TENDERLY_API_KEY,
    username=TENDERLY_USERNAME,
    project=TENDERLY_PROJECT
)
```

Basic Transaction Simulation

Simple Transaction Simulation

```
from web3 import Web3
import json

# Connect to Web3
w3 = Web3(Web3.HTTPProvider(WEB3_PROVIDER_URL))

def simulate_simple_transfer():
    """Simulate a simple ETH transfer"""

    transaction = {
        "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",  #
Example address
        "to": "0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045",    #
Vitalik's address
        "value": w3.to_wei(0.1, 'ether'),
        "gas": 21000,
        "gasPrice": w3.to_wei(20, 'gwei')
    }

    # Simulate transaction
    result = tenderly.simulate_transaction(transaction)

    if result.get("simulation"):
        simulation = result["simulation"]
        print(f"Simulation ID: {simulation['id']}")
        print(f"Status: {simulation['status']}")
        print(f"Gas Used: {simulation['gas_used']}")
        print(f"Success: {not simulation['error_message']}")

        return simulation
    else:
        print("Simulation failed:", result.get("error"))
        return None

# Run simulation
simulation_result = simulate_simple_transfer()
```

Contract Interaction Simulation

```
def simulate_uniswap_swap():
    """Simulate a Uniswap V2 token swap"""

    # Uniswap V2 Router contract
    router_address = "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D"

    # Example: Swap ETH for USDC
    swap_data =
w3.keccak(text="swapExactETHForTokens(uint256,address[],address,uint256)")
[:4]

    # Encode function call
    from eth_abi import encode

    amount_out_min = 1000 * 10**6  # 1000 USDC minimum
    path = [
        "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2",  # WETH
        "0xA0b86a33E6417c8f4CF46d764e1c1a0D8e34b4E"   # USDC
    ]
    to_address = "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c"
    deadline = int(w3.eth.get_block('latest')['timestamp']) + 300  # 5
minutes

    encoded_params = encode(
        ['uint256', 'address[]', 'address', 'uint256'],
        [amount_out_min, path, to_address, deadline]
    )

    transaction = {
        "from": to_address,
        "to": router_address,
        "value": w3.to_wei(1, 'ether'),  # 1 ETH to swap
        "data": "0x" + (swap_data + encoded_params).hex(),
        "gas": 200000,
        "gasPrice": w3.to_wei(30, 'gwei')
    }

    # Simulate the swap
    result = tenderly.simulate_transaction(transaction)

    if result.get("simulation"):
        simulation = result["simulation"]
```

```
        print(f"Swap Simulation Results:")
        print(f"- Status: {simulation['status']}")
        print(f"- Gas Used: {simulation['gas_used']}")
        print(f"- Gas Cost: {simulation['gas_used'] *
transaction['gasPrice'] / 10**18:.6f} ETH")

        # Analyze state changes
        if "state_changes" in simulation:
            print("\nState Changes:")
            for change in simulation["state_changes"]:
                print(f"- {change['address']}:
{change['solidity_diff']}")

        return simulation
    else:
        print("Swap simulation failed:", result.get("error"))
        return None

# Run Uniswap simulation
swap_result = simulate_uniswap_swap()
```

Advanced Simulation Features

State Overrides

```
def simulate_with_state_overrides():
    """Simulate transaction with modified blockchain state"""

    # Example: Override account balance to test large transactions
    transaction = {
        "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
        "to": "0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045",
        "value": w3.to_wei(1000, 'ether'),  # Large amount
        "gas": 21000,
        "gasPrice": w3.to_wei(20, 'gwei')
    }

    # State overrides
    state_overrides = {
        "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c": {
            "balance": hex(w3.to_wei(2000, 'ether'))  # Override
balance
        }
    }

    payload = {
        "network_id": "1",
        "from": transaction["from"],
        "to": transaction["to"],
        "value": str(transaction["value"]),
        "gas": transaction["gas"],
        "gas_price": str(transaction["gasPrice"]),
        "state_overrides": state_overrides,
        "save": True
    }

    url = f"{tenderly.base_url}/account/{tenderly.username}/project/
{tenderly.project}/simulate"
    response = requests.post(url, headers=tenderly.headers,
json=payload)

    return response.json()

# Test with state overrides
override_result = simulate_with_state_overrides()
```

Fork Network Creation

```
def create_fork_network():
    """Create a private fork for testing"""

    payload = {
        "network_id": "1",  # Fork from mainnet
        "block_number": w3.eth.block_number,  # Current block
        "chain_config": {
            "chain_id": 1337
        }
    }

    url = f"{tenderly.base_url}/account/{tenderly.username}/project/
{tenderly.project}/fork"
    response = requests.post(url, headers=tenderly.headers,
json=payload)

    if response.status_code == 200:
        fork_data = response.json()
        fork_id = fork_data["simulation_fork"]["id"]
        fork_rpc = f"https://rpc.tenderly.co/fork/{fork_id}"

        print(f"Fork created successfully!")
        print(f"Fork ID: {fork_id}")
        print(f"RPC URL: {fork_rpc}")

        return fork_id, fork_rpc
    else:
        print("Failed to create fork:", response.text)
        return None, None

# Create fork
fork_id, fork_rpc = create_fork_network()
```

Batch Simulations

```
async def simulate_batch_transactions():
    """Simulate multiple transactions in batch"""
    import aiohttp
    import asyncio

    # Prepare multiple transactions
    transactions = [
        {
            "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
            "to": "0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045",
            "value": w3.to_wei(0.1, 'ether'),
            "gas": 21000,
            "gasPrice": w3.to_wei(20, 'gwei')
        },
        {
            "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
            "to": "0xA0b86a33E6417c8f4CF46d764e1c1a0D8e34b4E",
            "value": w3.to_wei(0.05, 'ether'),
            "gas": 21000,
            "gasPrice": w3.to_wei(25, 'gwei')
        }
    ]

    async def simulate_single(session, tx_data):
        """Simulate single transaction async"""
        payload = {
            "network_id": "1",
            "from": tx_data["from"],
            "to": tx_data["to"],
            "value": str(tx_data["value"]),
            "gas": tx_data["gas"],
            "gas_price": str(tx_data["gasPrice"]),
            "save": True
        }

        url = f"{tenderly.base_url}/account/{tenderly.username}/
project/{tenderly.project}/simulate"

        async with session.post(url, headers=tenderly.headers,
json=payload) as response:
            return await response.json()
```

```
    # Run batch simulations
    async with aiohttp.ClientSession() as session:
        tasks = [simulate_single(session, tx) for tx in transactions]
        results = await asyncio.gather(*tasks)

    # Process results
    for i, result in enumerate(results):
        if result.get("simulation"):
            sim = result["simulation"]
            print(f"Transaction {i+1}: {sim['status']}, Gas:
{sim['gas_used']}")
        else:
            print(f"Transaction {i+1} failed: {result.get('error')}")

    return results

# Run batch simulation
batch_results = asyncio.run(simulate_batch_transactions())
```

MEV Strategy Testing

Arbitrage Simulation

```
def simulate_arbitrage_opportunity():
    """Simulate an arbitrage transaction"""

    # Example: Arbitrage between Uniswap and SushiSwap
    # This is a simplified example - real arbitrage is more complex

    # Step 1: Borrow flash loan (simulate)
    flash_loan_amount = w3.to_wei(100, 'ether')

    # Step 2: Buy on DEX 1 (lower price)
    buy_transaction = {
        "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
        "to": "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D",  # Uniswap
Router
        "value": flash_loan_amount,
        "data": "0x...",  # Encoded swap function call
        "gas": 150000,
        "gasPrice": w3.to_wei(50, 'gwei')
    }

    # Simulate buy transaction
    buy_result = tenderly.simulate_transaction(buy_transaction)

    if buy_result.get("simulation"):
        buy_sim = buy_result["simulation"]
        print(f"Buy Simulation:")
        print(f"- Status: {buy_sim['status']}")
        print(f"- Gas Used: {buy_sim['gas_used']}")

        # Step 3: Sell on DEX 2 (higher price)
        sell_transaction = {
            "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
            "to": "0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F",  #
SushiSwap Router
            "value": "0",
            "data": "0x...",  # Encoded swap function call
            "gas": 150000,
            "gasPrice": w3.to_wei(50, 'gwei')
        }

        # Simulate sell transaction
        sell_result = tenderly.simulate_transaction(sell_transaction)
```

```
        if sell_result.get("simulation"):
            sell_sim = sell_result["simulation"]
            print(f"Sell Simulation:")
            print(f"- Status: {sell_sim['status']}")
            print(f"- Gas Used: {sell_sim['gas_used']}")

            # Calculate profitability
            total_gas = buy_sim['gas_used'] + sell_sim['gas_used']
            gas_cost = total_gas * int(buy_transaction['gasPrice'])

            print(f"\nArbitrage Analysis:")
            print(f"- Total Gas Used: {total_gas}")
            print(f"- Gas Cost: {gas_cost / 10**18:.6f} ETH")

            return True
        else:
            print("Sell simulation failed")
            return False
    else:
        print("Buy simulation failed")
        return False

# Test arbitrage
arbitrage_result = simulate_arbitrage_opportunity()
```

Liquidation Testing

```
def simulate_liquidation():
    """Simulate a liquidation transaction"""

    # Example: Aave liquidation
    aave_lending_pool = "0x7d2768dE32b0b80b7a3454c06BdAc94A69DDc7A9"

    # Liquidation parameters
    collateral_asset = "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2"  #
WETH
    debt_asset = "0xA0b86a33E6417c8f4CF46d764e1c1a0D8e34b4E"       #
USDC
    user_to_liquidate = "0x..."  # Address with liquidatable position
    debt_to_cover = 10000 * 10**6  # 10,000 USDC

    # Encode liquidation call
    liquidation_data =
w3.keccak(text="liquidationCall(address,address,address,uint256,bool)")
[:4]

    from eth_abi import encode
    encoded_params = encode(
        ['address', 'address', 'address', 'uint256', 'bool'],
        [collateral_asset, debt_asset, user_to_liquidate,
debt_to_cover, True]
    )

    liquidation_tx = {
        "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
        "to": aave_lending_pool,
        "value": "0",
        "data": "0x" + (liquidation_data + encoded_params).hex(),
        "gas": 300000,
        "gasPrice": w3.to_wei(100, 'gwei')  # High gas for competitive
liquidation
    }

    # Simulate liquidation
    result = tenderly.simulate_transaction(liquidation_tx)

    if result.get("simulation"):
        sim = result["simulation"]
        print(f"Liquidation Simulation:")
```

```
        print(f"- Status: {sim['status']}")
        print(f"- Gas Used: {sim['gas_used']}")
        print(f"- Profitable: {analyze_liquidation_profit(sim)}")

        return sim
    else:
        print("Liquidation simulation failed:", result.get("error"))
        return None

def analyze_liquidation_profit(simulation):
    """Analyze liquidation profitability from simulation results"""

    # This would analyze state changes to calculate profit
    # In practice, you'd look at token balance changes

    gas_cost = simulation['gas_used'] * 100 * 10**9  # 100 gwei gas
price

    # Simplified profit calculation
    # Real implementation would parse state changes for exact amounts
    estimated_profit = 0.1 * 10**18  # Assume 0.1 ETH profit

    net_profit = estimated_profit - gas_cost

    return net_profit > 0

# Test liquidation
liquidation_result = simulate_liquidation()
```

Gas Optimization

Gas Profiling

```
def profile_gas_usage():
    """Profile gas usage for optimization"""

    # Create multiple versions of the same transaction with different
optimizations
    transactions = []

    # Version 1: Basic transaction
    basic_tx = {
        "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
        "to": "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D",
        "value": w3.to_wei(1, 'ether'),
        "data": "0x...",  # Basic swap call
        "gas": 200000,
        "gasPrice": w3.to_wei(30, 'gwei')
    }

    # Version 2: Optimized transaction (packed parameters, etc.)
    optimized_tx = {
        "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
        "to": "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D",
        "value": w3.to_wei(1, 'ether'),
        "data": "0x...",  # Optimized swap call
        "gas": 180000,
        "gasPrice": w3.to_wei(30, 'gwei')
    }

    results = []

    for i, tx in enumerate([basic_tx, optimized_tx]):
        result = tenderly.simulate_transaction(tx)

        if result.get("simulation"):
            sim = result["simulation"]
            results.append({
                "version": f"Version {i+1}",
                "gas_used": sim['gas_used'],
                "status": sim['status'],
                "cost_eth": sim['gas_used'] * tx['gasPrice'] / 10**18
            })

    # Compare results
```

```
    print("Gas Optimization Comparison:")
    for result in results:
        print(f"{result['version']}: {result['gas_used']} gas,
{result['cost_eth']:.6f} ETH")

    if len(results) == 2:
        gas_saved = results[0]['gas_used'] - results[1]['gas_used']
        cost_saved = results[0]['cost_eth'] - results[1]['cost_eth']
        print(f"Optimization saved: {gas_saved} gas ({cost_saved:.6f}
ETH)")

    return results

# Profile gas usage
gas_profile = profile_gas_usage()
```

Dynamic Gas Estimation

```
def estimate_optimal_gas():
    """Estimate optimal gas parameters for current network
conditions"""

    # Get current network conditions
    latest_block = w3.eth.get_block('latest')
    current_gas_price = w3.eth.gas_price

    # Simulate transaction with different gas prices
    base_transaction = {
        "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
        "to": "0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045",
        "value": w3.to_wei(0.1, 'ether'),
        "gas": 21000
    }

    # Test different gas price levels
    gas_price_multipliers = [0.5, 0.8, 1.0, 1.2, 1.5, 2.0]
    results = []

    for multiplier in gas_price_multipliers:
        test_gas_price = int(current_gas_price * multiplier)

        tx = base_transaction.copy()
        tx["gasPrice"] = test_gas_price

        result = tenderly.simulate_transaction(tx)

        if result.get("simulation"):
            sim = result["simulation"]
            results.append({
                "multiplier": multiplier,
                "gas_price": test_gas_price,
                "gas_price_gwei": test_gas_price / 10**9,
                "success": sim['status'],
                "gas_used": sim['gas_used']
            })

    # Analyze results
    print("Gas Price Analysis:")
    for result in results:
        print(f"Multiplier {result['multiplier']}x: "
```

```
                f"{result['gas_price_gwei']:.1f} gwei, "
                f"Success: {result['success']}")

    # Recommend optimal gas price
    successful_results = [r for r in results if r['success']]
    if successful_results:
        optimal = min(successful_results, key=lambda x: x['gas_price'])
        print(f"\nRecommended gas price: {optimal['gas_price_gwei']:.
1f} gwei")
        return optimal['gas_price']

    return current_gas_price

# Estimate optimal gas
optimal_gas = estimate_optimal_gas()
```

Debugging and Analysis

Transaction Debugging

```
def debug_failed_transaction():
    """Debug a failed transaction to identify issues"""

    # Example: Failed transaction data
    failed_tx = {
        "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
        "to": "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D",
        "value": w3.to_wei(10, 'ether'),  # Might be too much
        "data": "0x...",  # Swap function call
        "gas": 150000,
        "gasPrice": w3.to_wei(20, 'gwei')
    }

    # Simulate with debugging enabled
    payload = {
        "network_id": "1",
        "from": failed_tx["from"],
        "to": failed_tx["to"],
        "value": str(failed_tx["value"]),
        "gas": failed_tx["gas"],
        "gas_price": str(failed_tx["gasPrice"]),
        "input": failed_tx.get("data", "0x"),
        "save": True,
        "save_if_fails": True,
        "simulation_type": "full"  # Full debugging
    }

    url = f"{tenderly.base_url}/account/{tenderly.username}/project/
{tenderly.project}/simulate"
    response = requests.post(url, headers=tenderly.headers,
json=payload)

    result = response.json()

    if result.get("simulation"):
        sim = result["simulation"]

        print(f"Debug Results:")
        print(f"- Status: {sim['status']}")
        print(f"- Error: {sim.get('error_message', 'None')}")
        print(f"- Gas Used: {sim['gas_used']} / {failed_tx['gas']}")
```

```
        # Analyze error details
        if sim.get('error_message'):
            analyze_error(sim['error_message'])

        # Analyze state changes
        if sim.get('state_changes'):
            print("\nState Changes:")
            for change in sim['state_changes']:
                print(f"- {change['address']}:
{change.get('solidity_diff', 'Unknown change')}")

        return sim
    else:
        print("Debug simulation failed:", result.get("error"))
        return None

def analyze_error(error_message):
    """Analyze common error patterns"""

    common_errors = {
        "insufficient funds": "Account balance too low",
        "execution reverted": "Contract execution failed",
        "out of gas": "Gas limit too low",
        "nonce too low": "Transaction nonce already used",
        "replacement transaction underpriced": "Gas price too low for
replacement",
        "intrinsic gas too low": "Basic gas requirement not met"
    }

    print(f"\nError Analysis:")
    print(f"Raw error: {error_message}")

    for pattern, explanation in common_errors.items():
        if pattern.lower() in error_message.lower():
            print(f"Likely cause: {explanation}")
            break
    else:
        print("Error pattern not recognized - requires manual
analysis")

# Debug failed transaction
debug_result = debug_failed_transaction()
```

State Change Analysis

```
def analyze_state_changes(simulation_id):
    """Analyze detailed state changes from simulation"""

    # Get full simulation data
    simulation_data = tenderly.get_simulation(simulation_id)

    if not simulation_data.get("simulation"):
        print("Simulation not found")
        return

    sim = simulation_data["simulation"]

    # Analyze balance changes
    balance_changes = {}
    token_transfers = []

    if sim.get("state_changes"):
        for change in sim["state_changes"]:
            address = change["address"]

            # Check for balance changes
            if "balance" in change.get("storage_changes", {}):
                old_balance = change["storage_changes"]["balance"]
["before"]
                new_balance = change["storage_changes"]["balance"]
["after"]
                balance_changes[address] = {
                    "before": old_balance,
                    "after": new_balance,
                    "diff": int(new_balance, 16) - int(old_balance, 16)
                }

            # Analyze ERC20 transfers (simplified)
            if change.get("solidity_diff"):
                if "Transfer" in change["solidity_diff"]:
                    token_transfers.append(change)

    # Print analysis
    print("State Change Analysis:")

    if balance_changes:
        print("\nETH Balance Changes:")
```

```
        for address, change in balance_changes.items():
            diff_eth = change["diff"] / 10**18
            print(f"- {address}: {diff_eth:+.6f} ETH")

    if token_transfers:
        print("\nToken Transfers:")
        for transfer in token_transfers:
            print(f"- {transfer['address']}:
{transfer['solidity_diff']}")

    # Gas analysis
    gas_breakdown = sim.get("gas_breakdown", {})
    if gas_breakdown:
        print(f"\nGas Breakdown:")
        for operation, gas_used in gas_breakdown.items():
            print(f"- {operation}: {gas_used} gas")

    return {
        "balance_changes": balance_changes,
        "token_transfers": token_transfers,
        "gas_breakdown": gas_breakdown
    }

# Example usage (would need actual simulation ID)
# state_analysis = analyze_state_changes("simulation_id_here")
```

Integration with MEV Workflows

Pre-execution Validation

```
class MEVTransactionValidator:
    """Validate MEV transactions before execution"""

    def __init__(self, tenderly_client):
        self.tenderly = tenderly_client
        self.min_profit_threshold = w3.to_wei(0.01, 'ether')  # 0.01
ETH minimum

    def validate_arbitrage(self, buy_tx, sell_tx):
        """Validate arbitrage opportunity"""

        # Simulate buy transaction
        buy_result = self.tenderly.simulate_transaction(buy_tx)
        if not buy_result.get("simulation") or buy_result["simulation"]
["status"] != True:
            return False, "Buy transaction would fail"

        # Simulate sell transaction
        sell_result = self.tenderly.simulate_transaction(sell_tx)
        if not sell_result.get("simulation") or
sell_result["simulation"]["status"] != True:
            return False, "Sell transaction would fail"

        # Calculate profitability
        buy_gas = buy_result["simulation"]["gas_used"]
        sell_gas = sell_result["simulation"]["gas_used"]
        total_gas_cost = (buy_gas + sell_gas) * int(buy_tx["gasPrice"])

        # This would need to analyze state changes for actual profit
        # Simplified for example
        estimated_profit = self.estimate_arbitrage_profit(buy_result,
sell_result)

        net_profit = estimated_profit - total_gas_cost

        if net_profit < self.min_profit_threshold:
            return False, f"Insufficient profit: {net_profit / 10**18:.
6f} ETH"

        return True, f"Valid arbitrage: {net_profit / 10**18:.6f} ETH
profit"
```

```
    def estimate_arbitrage_profit(self, buy_result, sell_result):
        """Estimate profit from arbitrage simulations"""
        # This would analyze token balance changes
        # Simplified implementation
        return w3.to_wei(0.05, 'ether')  # Example profit

    def validate_liquidation(self, liquidation_tx):
        """Validate liquidation opportunity"""

        result = self.tenderly.simulate_transaction(liquidation_tx)

        if not result.get("simulation"):
            return False, "Simulation failed"

        sim = result["simulation"]

        if sim["status"] != True:
            return False, f"Liquidation would fail:
{sim.get('error_message')}"

        # Check for liquidation bonus (would analyze state changes)
        profit = self.estimate_liquidation_profit(sim)
        gas_cost = sim["gas_used"] * int(liquidation_tx["gasPrice"])

        if profit < gas_cost:
            return False, "Liquidation not profitable"

        return True, f"Valid liquidation: {(profit - gas_cost) /
10**18:.6f} ETH profit"

    def estimate_liquidation_profit(self, simulation):
        """Estimate liquidation profit"""
        # Would analyze state changes for liquidation bonus
        return w3.to_wei(0.02, 'ether')  # Example

# Initialize validator
validator = MEVTransactionValidator(tenderly)

# Example validation
buy_tx = {...}  # Arbitrage buy transaction
sell_tx = {...}  # Arbitrage sell transaction
```

```
is_valid, message = validator.validate_arbitrage(buy_tx, sell_tx)
print(f"Arbitrage validation: {message}")
```

Automated Testing Pipeline

```
async def run_mev_testing_pipeline():
    """Automated pipeline for testing MEV strategies"""

    # Test scenarios
    test_scenarios = [
        {
            "name": "Uniswap-SushiSwap Arbitrage",
            "type": "arbitrage",
            "transactions": [...]  # Transaction data
        },
        {
            "name": "Aave Liquidation",
            "type": "liquidation",
            "transactions": [...]
        },
        {
            "name": "Compound Liquidation",
            "type": "liquidation",
            "transactions": [...]
        }
    ]

    results = []

    for scenario in test_scenarios:
        print(f"\nTesting: {scenario['name']}")

        # Run simulations for scenario
        scenario_results = []

        for tx in scenario["transactions"]:
            result = tenderly.simulate_transaction(tx)
            scenario_results.append(result)

        # Analyze scenario
        success_rate = sum(1 for r in scenario_results
                           if r.get("simulation", {}).get("status") ==
True) / len(scenario_results)

        avg_gas = sum(r.get("simulation", {}).get("gas_used", 0)
                      for r in scenario_results) / len(scenario_results)
```

```
        results.append({
            "scenario": scenario["name"],
            "success_rate": success_rate,
            "avg_gas": avg_gas,
            "results": scenario_results
        })

        print(f"- Success rate: {success_rate:.1%}")
        print(f"- Average gas: {avg_gas:.0f}")

    # Generate report
    generate_testing_report(results)

    return results

def generate_testing_report(results):
    """Generate testing report"""

    print("\n" + "="*50)
    print("MEV STRATEGY TESTING REPORT")
    print("="*50)

    for result in results:
        print(f"\n{result['scenario']}:")
        print(f"  Success Rate: {result['success_rate']:.1%}")
        print(f"  Average Gas: {result['avg_gas']:.0f}")

        if result['success_rate'] < 0.8:
            print(f"  ⚠️  Low success rate - requires optimization")
        else:
            print(f"  ✅ Good success rate")

    print(f"\nOverall Statistics:")
    overall_success = sum(r['success_rate'] for r in results) /
len(results)
    print(f"  Average Success Rate: {overall_success:.1%}")

# Run testing pipeline
# testing_results = await run_mev_testing_pipeline()
```

Best Practices

Security Considerations

```
def secure_simulation_practices():
    """Implement security best practices for Tenderly usage"""

    # 1. API Key Management
    def rotate_api_keys():
        """Rotate API keys regularly"""
        # Implementation would involve:
        # - Generating new API key
        # - Updating environment variables
        # - Revoking old key
        pass

    # 2. Data Privacy
    def sanitize_simulation_data(transaction):
        """Remove sensitive data before simulation"""
        sanitized = transaction.copy()

        # Don't log sensitive addresses in production
        if "from" in sanitized:
            sanitized["from"] = "0x" + "x" * 38  # Mask address

        return sanitized

    # 3. Rate Limiting
    class RateLimitedTenderlyClient:
        def __init__(self, base_client, max_requests_per_minute=100):
            self.client = base_client
            self.max_requests = max_requests_per_minute
            self.request_times = []

        def simulate_transaction(self, transaction):
            # Check rate limit
            now = time.time()
            self.request_times = [t for t in self.request_times if now
- t < 60]

            if len(self.request_times) >= self.max_requests:
                time.sleep(60 - (now - self.request_times[0]))

            self.request_times.append(now)
            return self.client.simulate_transaction(transaction)
```

```
    # 4. Error Handling
    def robust_simulation(transaction, max_retries=3):
        """Simulate with retry logic"""

        for attempt in range(max_retries):
            try:
                result = tenderly.simulate_transaction(transaction)

                if result.get("error"):
                    if attempt == max_retries - 1:
                        raise Exception(f"Simulation failed:
{result['error']}")
                    time.sleep(2 ** attempt)  # Exponential backoff
                    continue

                return result

            except Exception as e:
                if attempt == max_retries - 1:
                    raise e
                time.sleep(2 ** attempt)

        raise Exception("Max retries exceeded")

    return {
        "rate_limited_client": RateLimitedTenderlyClient(tenderly),
        "sanitize_data": sanitize_simulation_data,
        "robust_simulation": robust_simulation
    }

# Implement security practices
security_tools = secure_simulation_practices()
```

Performance Optimization

```
def optimize_tenderly_performance():
    """Optimize Tenderly usage for better performance"""

    # 1. Connection Pooling
    import aiohttp
    import asyncio

    class OptimizedTenderlyClient:
        def __init__(self, api_key, username, project):
            self.api_key = api_key
            self.username = username
            self.project = project
            self.base_url = "https://api.tenderly.co/api/v1"

            # Connection pool settings
            connector = aiohttp.TCPConnector(
                limit=100,  # Total connection pool size
                limit_per_host=30,  # Connections per host
                ttl_dns_cache=300,  # DNS cache TTL
                use_dns_cache=True,
            )

            timeout = aiohttp.ClientTimeout(total=30, connect=10)

            self.session = aiohttp.ClientSession(
                connector=connector,
                timeout=timeout,
                headers={
                    "Content-Type": "application/json",
                    "X-Access-Key": self.api_key
                }
            )

        async def simulate_async(self, transaction):
            """Async simulation with connection pooling"""

            payload = {
                "network_id": "1",
                "from": transaction["from"],
                "to": transaction["to"],
                "value": str(transaction.get("value", "0")),
                "gas": transaction.get("gas", 8000000),
```

```
                "gas_price": str(transaction.get("gasPrice", "0")),
                "input": transaction.get("data", "0x"),
                "save": False  # Don't save for performance testing
            }

            url = f"{self.base_url}/account/{self.username}/project/
{self.project}/simulate"

            async with self.session.post(url, json=payload) as
response:
                return await response.json()

        async def close(self):
            """Close session"""
            await self.session.close()

    # 2. Caching Strategy
    import hashlib

    class CachedSimulations:
        def __init__(self):
            self.cache = {}

        def get_cache_key(self, transaction):
            """Generate cache key for transaction"""
            # Create deterministic hash from transaction data
            tx_string = json.dumps(transaction, sort_keys=True)
            return hashlib.md5(tx_string.encode()).hexdigest()

        def get_cached_result(self, transaction):
            """Get cached simulation result"""
            key = self.get_cache_key(transaction)
            return self.cache.get(key)

        def cache_result(self, transaction, result):
            """Cache simulation result"""
            key = self.get_cache_key(transaction)
            self.cache[key] = result

        def simulate_with_cache(self, transaction):
            """Simulate with caching"""
            # Check cache first
```

```
            cached = self.get_cached_result(transaction)
            if cached:
                return cached

            # Simulate and cache
            result = tenderly.simulate_transaction(transaction)
            self.cache_result(transaction, result)

            return result

    # 3. Batch Processing
    async def batch_simulate_optimized(transactions, batch_size=10):
        """Optimized batch simulation"""

        client = OptimizedTenderlyClient(
            TENDERLY_API_KEY, TENDERLY_USERNAME, TENDERLY_PROJECT
        )

        try:
            results = []

            # Process in batches
            for i in range(0, len(transactions), batch_size):
                batch = transactions[i:i + batch_size]

                # Run batch concurrently
                tasks = [client.simulate_async(tx) for tx in batch]
                batch_results = await asyncio.gather(*tasks,
return_exceptions=True)

                results.extend(batch_results)

                # Small delay between batches to avoid rate limiting
                if i + batch_size < len(transactions):
                    await asyncio.sleep(0.1)

            return results

        finally:
            await client.close()

    return {
```

```
        "optimized_client": OptimizedTenderlyClient,
        "cached_simulations": CachedSimulations(),
        "batch_simulate": batch_simulate_optimized
    }

# Initialize performance tools
perf_tools = optimize_tenderly_performance()
```

Troubleshooting

Common Issues and Solutions

```
def troubleshoot_tenderly_issues():
    """Common Tenderly issues and their solutions"""

    issues_and_solutions = {
        "API_KEY_INVALID": {
            "symptoms": ["401 Unauthorized", "Invalid API key"],
            "solutions": [
                "Check API key is correctly set in environment",
                "Verify API key hasn't expired",
                "Regenerate API key in Tenderly dashboard"
            ]
        },

        "SIMULATION_TIMEOUT": {
            "symptoms": ["Request timeout", "No response"],
            "solutions": [
                "Reduce transaction complexity",
                "Lower gas limit",
                "Check network connectivity",
                "Implement retry logic with exponential backoff"
            ]
        },

        "RATE_LIMIT_EXCEEDED": {
            "symptoms": ["429 Too Many Requests", "Rate limit error"],
            "solutions": [
                "Implement request rate limiting",
                "Use batch simulation where possible",
                "Upgrade Tenderly plan for higher limits",
                "Add delays between requests"
            ]
        },

        "SIMULATION_FAILED": {
            "symptoms": ["Transaction reverts", "Execution failed"],
            "solutions": [
                "Check account balances",
                "Verify contract addresses",
                "Review transaction data encoding",
                "Test with state overrides"
            ]
        },
```

```
        "NETWORK_ISSUES": {
            "symptoms": ["Connection errors", "DNS resolution failed"],
            "solutions": [
                "Check internet connectivity",
                "Try different DNS servers",
                "Use VPN if geographical restrictions",
                "Implement connection pooling"
            ]
        }
    }

    return issues_and_solutions

def diagnose_simulation_error(error_response):
    """Diagnose simulation errors"""

    error_msg = str(error_response).lower()

    if "401" in error_msg or "unauthorized" in error_msg:
        return "API_KEY_INVALID"
    elif "429" in error_msg or "rate limit" in error_msg:
        return "RATE_LIMIT_EXCEEDED"
    elif "timeout" in error_msg:
        return "SIMULATION_TIMEOUT"
    elif "revert" in error_msg or "execution failed" in error_msg:
        return "SIMULATION_FAILED"
    elif "connection" in error_msg or "dns" in error_msg:
        return "NETWORK_ISSUES"
    else:
        return "UNKNOWN_ERROR"

def get_solution(error_type):
    """Get solutions for specific error type"""

    solutions = troubleshoot_tenderly_issues()

    if error_type in solutions:
        issue = solutions[error_type]
        print(f"Issue: {error_type}")
        print(f"Symptoms: {', '.join(issue['symptoms'])}")
        print(f"Solutions:")
```

```
        for i, solution in enumerate(issue['solutions'], 1):
            print(f"  {i}. {solution}")
    else:
        print("Unknown error type. Check Tenderly documentation or
support.")

# Example error diagnosis
# error_type = diagnose_simulation_error("401 Unauthorized")
# get_solution(error_type)
```

Health Monitoring

```
def monitor_tenderly_health():
    """Monitor Tenderly service health"""

    def check_api_status():
        """Check if Tenderly API is responding"""
        try:
            # Simple test simulation
            test_tx = {
                "from": "0x742d35Cc6634C0532925a3b8D9C244769b5C4c4c",
                "to": "0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045",
                "value": "1000000000000000000",  # 1 ETH
                "gas": 21000,
                "gasPrice": "20000000000"  # 20 gwei
            }

            result = tenderly.simulate_transaction(test_tx)

            if result.get("simulation"):
                return True, "API is responding"
            else:
                return False, f"API error: {result.get('error')}"

        except Exception as e:
            return False, f"Connection error: {str(e)}"

    def measure_response_time():
        """Measure API response time"""
        import time

        start_time = time.time()

        try:
            is_healthy, message = check_api_status()
            response_time = time.time() - start_time

            return response_time, is_healthy, message

        except Exception as e:
            response_time = time.time() - start_time
            return response_time, False, str(e)

    def health_check_report():
```

```
        """Generate health check report"""

        print("Tenderly Health Check Report")
        print("=" * 40)

        # Check API status
        response_time, is_healthy, message = measure_response_time()

        print(f"API Status: {'✅ Healthy' if is_healthy else '❌
Unhealthy'}")
        print(f"Response Time: {response_time:.2f} seconds")
        print(f"Message: {message}")

        # Performance thresholds
        if response_time > 5.0:
            print("⚠️  High response time - may impact MEV execution
speed")
        elif response_time > 2.0:
            print("⚠️  Moderate response time - monitor performance")

        return {
            "healthy": is_healthy,
            "response_time": response_time,
            "message": message,
            "timestamp": time.time()
        }

    return health_check_report

# Monitor health
health_monitor = monitor_tenderly_health()
health_report = health_monitor()
```

Quick Reference

Essential Commands

```
# Basic simulation
result = tenderly.simulate_transaction({
    "from": "0x...",
    "to": "0x...",
    "value": "1000000000000000000",
    "gas": 21000,
    "gasPrice": "20000000000"
})

# Gas estimation
optimal_gas = estimate_optimal_gas()

# Debug failed transaction
debug_result = debug_failed_transaction()

# Validate MEV opportunity
is_valid, message = validator.validate_arbitrage(buy_tx, sell_tx)

# Batch simulation
results = await batch_simulate_optimized(transactions)

# Health check
health_report = health_monitor()
```

Configuration Checklist

- [ ] Tenderly account created and verified
- [ ] API key generated and stored securely
- [ ] Project initialized with correct settings
- [ ] Environment variables configured
- [ ] Rate limiting implemented
- [ ] Error handling and retry logic in place
- [ ] Caching strategy implemented for repeated simulations
- [ ] Health monitoring setup
- [ ] Security practices followed

Summary

In this module, you've learned how to:

✅ **Set up Tenderly environment** with proper API configuration and security practices

✅ **Perform basic transaction simulations** to test MEV strategies safely before execution

✅ **Use advanced features** like state overrides, fork networks, and batch simulations

✅ **Test MEV strategies** including arbitrage and liquidation opportunities with profitability analysis

✅ **Optimize gas usage** through profiling and dynamic estimation techniques

✅ **Debug failed transactions** with detailed error analysis and state change examination

✅ **Integrate Tenderly** into automated MEV workflows for pre-execution validation

✅ **Implement best practices** for security, performance, and reliability

✅ **Troubleshoot common issues** and monitor service health

Tenderly is an essential tool for MEV practitioners, providing the safety and insights needed to develop profitable strategies while minimizing risks. The simulation capabilities allow you to test complex scenarios without financial exposure, while the debugging features help optimize performance and resolve issues quickly.

Next Steps:

- Practice simulating different MEV scenarios
- Integrate Tenderly validation into your trading workflows
- Explore advanced features like custom networks and webhooks
- Set up automated monitoring for your simulation infrastructure

Remember: Always validate strategies thoroughly in simulation before deploying with real funds. Tenderly simulations are invaluable but should be combined with other risk management practices for comprehensive MEV strategy development.