Senior Software Engineer Coding Challenge: CEX-DEX Arbitrage Bot

Overview

Build a real-time arbitrage detection system in Go that monitors price discrepancies between a centralized exchange (CEX) orderbook and Uniswap V3 DEX pricing for the ETH-USDC trading pair.

Challenge Requirements

Core Functionality

1. **CEX Orderbook Integration**

   - Integrate with Binance API to fetch ETH-USDC orderbook snapshots
   - Snapshot the orderbook on every Ethereum block (~12 seconds)
   - Calculate the effective execution price for configurable trade sizes (e.g., 1 ETH, 10 ETH, 100 ETH)

2. **Ethereum Block Streaming**

   - **WebSocket Connection**: Establish a WebSocket connection to an Ethereum node (Infura, Alchemy, or local node)
   - Subscribe to new block headers using `eth_subscribe` with `newHeads`
   - **Connection Management**: Implement robust reconnection logic
     - Detect connection drops (ping/pong, timeout detection)
     - Automatic reconnection with exponential backoff
     - Resume from last known block to avoid gaps
     - Handle graceful degradation if WebSocket unavailable (fallback to polling)
   - Process each block atomically to trigger orderbook snapshots

3. **DEX Price Integration**

   - Connect to an Ethereum node (can use Infura/Alchemy)
   - Query Uniswap V3 ETH-USDC pool for equivalent pricing
   - Use the QuoterV2 contract or simulate swap calculations
   - Fetch data synchronized with each new Ethereum block

Understanding Uniswap V3 Pools

Uniswap V3 uses a concentrated liquidity model where:

- Liquidity providers allocate capital within specific price ranges (ticks)
- The pool's current price is determined by the ratio of reserves and the current tick
- A swap moves through sequential ticks, consuming liquidity at each price level
- Pool state (liquidity, sqrtPriceX96, tick) changes with every swap transaction

Why Requote on Every Block:

- While you *could* use `eth_subscribe` with `logs` to watch for `Swap` events on the pool contract, requoting on every block is preferred because:
  1. **Atomic Consistency**: You get a consistent snapshot of both CEX and DEX state at a specific block height
  2. **No Event Gaps**: Log subscriptions can miss events during reconnections or node issues
  3. **State Changes Beyond Swaps**: Pool state can change through liquidity additions/removals, not just swaps
  4. **Arbitrage Window**: Knowing the price at each block boundary helps you understand the exact window of opportunity
  5. **Simpler Architecture**: Block-driven polling is more predictable than event-driven for this use case

For this challenge, implement the block-driven approach with requoting, but be prepared to discuss trade-offs with event-driven approaches.

3. **Arbitrage Detection**
   - Compare CEX vs DEX pricing accounting for:
     - Slippage on both sides
     - Gas costs (estimate for DEX operations)
     - Trading fees
   - Detect profitable arbitrage opportunities
   - Print detailed arbitrage information including:
     - Direction (CEX→DEX or DEX→CEX)
     - Expected profit in USD and percentage
     - Execution steps required
     - Risk factors

Senior Engineer Requirements

Implement the following to demonstrate senior-level engineering skills:

1. Caching Strategy

- **Requirement**: Implement a multi-layered caching system
  - L1: In-memory cache for frequently accessed data (recent blocks, pool states)
  - L2: Consider cache invalidation strategies
  - Cache Uniswap pool state, gas price estimates, and historical orderbook data
- **Considerations**:
  - Thread-safe cache access
  - TTL-based expiration

    - Cache warming strategies
    - Memory bounds

2. WebSocket Management & Connection Resiliency

- **Requirement**: Implement production-grade WebSocket handling
    - Maintain persistent WebSocket connection to Ethereum node
    - Implement heartbeat/ping mechanism to detect stale connections
    - Automatic reconnection with exponential backoff and jitter
    - Track last processed block to resume without gaps after reconnection
    - Handle edge cases: late blocks, missed blocks, connection during sync
    - Proper cleanup of connections and subscriptions

3. Concurrency & Performance

- **Requirement**: Design a concurrent architecture using Go patterns
    - Use goroutines and channels effectively
    - Implement worker pools for API calls
    - Handle backpressure when data arrives faster than processing
    - Avoid race conditions
- **Bonus**: Implement graceful shutdown with context cancellation

4. Rate Limiting & Resiliency

- **Requirement**: Implement production-ready error handling
    - Rate limiting for external API calls (CEX API, RPC endpoints)
    - Exponential backoff with jitter for retries
    - Circuit breaker pattern for failing services
    - Metrics/observability hooks (even if just structured logging)

5. Configuration & Extensibility

- **Requirement**: Make the system configurable and extensible
    - Support multiple trading pairs (design for ETH-USDC, but make it extensible)
    - Configurable trade sizes to check
    - Pluggable exchange adapters (interface-based design)
    - Configuration via file (YAML/JSON) or environment variables

6. Data Modeling & Architecture

- **Requirement**: Demonstrate clean architecture principles
    - Clear separation of concerns (data layer, business logic, external integrations)
    - Well-defined interfaces between components
    - Proper error types and handling
    - Consider how you'd test this system (testability)

Expected Output

When an arbitrage opportunity is detected, print output similar to:

```
 === ARBITRAGE OPPORTUNITY DETECTED ===
Block Number: 18234567
Timestamp: 2024-01-15 14:23:45 UTC
Direction: CEX → DEX (Buy on Binance, Sell on Uniswap)

Trade Size: 10.0 ETH
CEX Price: $2,245.30 (effective with slippage)
DEX Price: $2,267.80 (effective with slippage)
Price Difference: $22.50 per ETH (1.00%)

Estimated Profit: $225.00 (before gas and fees)

Execution Steps:
1. Buy 10.0 ETH on Binance at average price $2,245.30
   - Place market order or limit orders consuming top bid/ask levels
   - Required capital: ~$22,453.00 USDC
2. Transfer ETH to trading wallet
3. Execute Uniswap V3 swap: 10.0 ETH → USDC
   - Pool: 0x88e6A0c2dDD26FEEb64F039a2c41296FcB3f5640
   - Expected output: ~22,678 USDC (after 0.3% pool fee)
```

Deliverables

1. **Source Code**
    - Well-structured Go modules
    - Clear package organization
    - Idiomatic Go code
2. **Documentation**
    - README with setup instructions
    - Architecture diagram or description
    - API documentation for key interfaces
    - Configuration examples
3. **Testing** (Bonus)
    - Unit tests for critical business logic
    - Mock implementations for external services
    - Integration test examples
4. **Discussion Points** (Be prepared to discuss)
    - How would you deploy this in production?
    - How would you monitor this system?
    - What are the scaling bottlenecks?
    - How would you handle chain reorgs?
    - Security considerations for handling private keys (if extending to actual trading)
    - How would you extend this to support multiple DEXes or CEXes?

Technical Constraints

- **Language**: Go 1.21+
- **External Dependencies**: Allowed (but be judicious)
    - Ethereum clients: `go-ethereum` (geth)
    - HTTP clients: standard library or popular choices
    - Configuration: `viper`, `yaml`, etc.
- **RPC Access**: Use public endpoints (Infura, Alchemy, or similar)
- **No Trading Required**: Detection only, no actual transaction submission

Evaluation Criteria

We evaluate candidates across four key dimensions:

1. **Code Quality**
    - Idiomatic Go patterns and conventions
    - Clean architecture and code organization
    - Comprehensive error handling
    - Code readability and maintainability
2. **System Design**
    - Concurrency design and patterns
    - WebSocket connection management and reconnection logic
    - Caching strategy and implementation
    - Extensibility and separation of concerns
3. **Production Readiness**
    - Error handling & resiliency patterns
    - Connection failure recovery
    - Configuration management
    - Logging/observability
    - Resource management and cleanup
4. **DeFi & Domain Understanding**
    - Accurate pricing calculations
    - Understanding of Uniswap V3 mechanics
    - Gas cost modeling
    - Understanding of arbitrage mechanics and execution
    - Consideration of real-world constraints and edge cases

Time Expectation

This challenge is designed to take **4-6 hours** for a senior engineer. Focus on:

- Core functionality working correctly
- Demonstrating 2-3 senior engineering patterns well (rather than all of them superficially)
- Clean, readable code with good architecture

We value quality over completeness. It's better to implement fewer features with production-quality code than to rush through all requirements. Document any shortcuts or TODOs for what you'd improve with more time.

Hints & Resources

WebSocket Ethereum Connection

- **Libraries**:
  - `gorilla/websocket` - Most popular WebSocket client: `go get github.com/gorilla/websocket`
  - `nhooyr.io/websocket` - Modern alternative with better context support
- **Connection Setup**:

  ```
   // Example WebSocket URL for Infura
  wsURL := "wss://mainnet.infura.io/ws/v3/YOUR_API_KEY"

  // Subscribe to new block headers
  subscribeMsg := `{"jsonrpc":"2.0","id":1,"method":"eth_subscribe","params":["newHeads"]}`
  ```

- **Message Handling**:
  - Subscription confirmation: `{"jsonrpc":"2.0","id":1,"result":"0x..."}`
  - Block notifications: `{"jsonrpc":"2.0","method":"eth_subscription","params":{"subscription":"0x...","result":{...}}}`
  - Implement timeout detection (if no message in 30s, reconnect)

Getting Uniswap V3 Pool Quotes

There are two main approaches to get price quotes from Uniswap V3:

Option 1: Using QuoterV2 Contract (Recommended for Beginners)

The QuoterV2 contract simulates swaps and returns expected output amounts without executing trades.

Contract Details:

- QuoterV2 Address: `0xb27308f9F90D607463bb33eA1BeBb41C27CE5AB6`
- Method: `quoteExactInputSingle`
- Etherscan

Go Implementation:

```
 // 1. Install go-ethereum
// go get github.com/ethereum/go-ethereum

// 2. Generate Go bindings from ABI (or use pre-generated)
// abigen --abi quoterv2.abi --pkg uniswap --type QuoterV2 --out quoterv2.go

// 3. Call the contract
import (
    "github.com/ethereum/go-ethereum/ethclient"
    "github.com/ethereum/go-ethereum/common"
    "math/big"
)

client, _ := ethclient.Dial("https://mainnet.infura.io/v3/YOUR_KEY")
quoterAddr := common.HexToAddress("0xb27308f9F90D607463bb33eA1BeBb41C27CE5AB6")

// QuoteExactInputSingle parameters:
// - tokenIn: WETH address (0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2)
// - tokenOut: USDC address (0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48)
// - fee: 3000 (0.3%)
// - amountIn: amount in wei (e.g., 1 ETH = 1e18)
// - sqrtPriceLimitX96: 0 (no limit)

params := QuoteExactInputSingleParams{
    TokenIn:           common.HexToAddress("0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2"),
    TokenOut:          common.HexToAddress("0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48"),
    Fee:               big.NewInt(3000),
    AmountIn:          big.NewInt(1e18), // 1 ETH
    SqrtPriceLimitX96: big.NewInt(0),
}

// Use eth_call to simulate (no gas cost)
amountOut, sqrtPriceX96After, _, gasEstimate, _ := quoter.QuoteExactInputSingle(params)
// amountOut is in USDC units (6 decimals), e.g., 2250000000 = 2250 USDC
```

Without Code Generation (using ABI directly):

```
import (
    "github.com/ethereum/go-ethereum/accounts/abi"
    "github.com/ethereum/go-ethereum/common"
    "strings"
)

// Load ABI
quoterABI := `[{"inputs":[{"components":[{"internalType":"address","name":"tokenIn","type":"address"},...]}],"name":"quoteExactInput

parsedABI, _ := abi.JSON(strings.NewReader(quoterABI))

// Pack the call data
data, _ := parsedABI.Pack("quoteExactInputSingle",
    common.HexToAddress("0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2"), // WETH
    common.HexToAddress("0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48"), // USDC
    big.NewInt(3000),  // fee
    big.NewInt(1e18),  // amountIn
    big.NewInt(0),     // sqrtPriceLimitX96
)

// Make eth_call
msg := ethereum.CallMsg{
    To:   &quoterAddr,
    Data: data,
}
result, _ := client.CallContract(context.Background(), msg, nil)

// Unpack result
var out []interface{}
parsedABI.UnpackIntoInterface(&out, "quoteExactInputSingle", result)
amountOut := out[0].(*big.Int)
```

Option 2: Direct Pool State Calculation (Advanced)

Read pool state directly and calculate swap amounts using Uniswap's math.

Pool Contract:

- ETH-USDC 0.3% Pool: `0x88e6A0c2dDD26FEEb64F039a2c41296FcB3f5640`
- Method: `slot0()` returns current price and tick
- Also need: `liquidity()`, `tickBitmap()`, `ticks()`

Key Functions:

```
// Call slot0() to get current state
// Returns: sqrtPriceX96, tick, observationIndex, observationCardinality, ...

// Then implement Uniswap V3 math:
// - Convert sqrtPriceX96 to actual price
// - Walk through ticks to calculate swap output
// - Account for liquidity at each tick
```

Libraries:

- `github.com/ethereum/go-ethereum` - Core Ethereum client
- `github.com/daoleno/uniswap-sdk-core` - Uniswap SDK port (partial)
- `github.com/daoleno/uniswapv3-sdk` - V3 SDK port (useful for math)

Note: Option 2 requires implementing or porting Uniswap's tick math. For this challenge, **Option 1 (QuoterV2) is recommended** unless you want to demonstrate deep DeFi knowledge.

Important Addresses & Token Decimals

- **WETH**: `0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2` (18 decimals)
- **USDC**: `0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48` (6 decimals)
- **ETH-USDC Pool (0.3%)**: `0x88e6A0c2dDD26FEEb64F039a2c41296FcB3f5640`

- **QuoterV2**: `0xb27308f9F90D607463bb33eA1BeBb41C27CE5AB6`

Understanding Token Decimals

ERC-20 tokens store amounts as integers without decimal points. The `decimals` property tells you how to interpret these integers.

How Decimals Work:

- **ETH/WETH (18 decimals)**: The smallest unit is 1 wei = 10^-18 ETH
  - 1 ETH = 1,000,000,000,000,000,000 wei = 1 × 10^18 wei
  - 0.5 ETH = 500,000,000,000,000,000 wei
  - When you see `big.NewInt(1e18)` in Go, that's 1 ETH
- **USDC (6 decimals)**: The smallest unit is 1 micro-USDC = 10^-6 USDC
  - 1 USDC = 1,000,000 micro-USDC = 1 × 10^6
  - 2,250 USDC = 2,250,000,000
  - 0.50 USDC = 500,000

Example Conversions in Go:

```
import (
    "math/big"
    "fmt"
)

// ETH (18 decimals)
oneETH := new(big.Int).Exp(big.NewInt(10), big.NewInt(18), nil)
fmt.Println(oneETH) // 1000000000000000000

// 10.5 ETH = 10.5 × 10^18
tenPointFiveETH := new(big.Int).Mul(big.NewInt(105), new(big.Int).Exp(big.NewInt(10), big.NewInt(17), nil))
// or: 10500000000000000000

// USDC (6 decimals)
oneUSDC := new(big.Int).Exp(big.NewInt(10), big.NewInt(6), nil)
fmt.Println(oneUSDC) // 1000000

// 2,250.50 USDC = 2250.50 × 10^6 = 2,250,500,000
usdcAmount := big.NewInt(2250500000)
```

Converting from Smart Contract Response:

```
// QuoterV2 returns amountOut in raw token units
amountOut := big.NewInt(2250500000) // USDC has 6 decimals

// Convert to human-readable format
divisor := new(big.Int).Exp(big.NewInt(10), big.NewInt(6), nil)
usdcFloat := new(big.Float).Quo(
    new(big.Float).SetInt(amountOut),
    new(big.Float).SetInt(divisor),
)
fmt.Printf("%.2f USDC\n", usdcFloat) // 2250.50 USDC
```

Why This Matters for Arbitrage:

1. When calling QuoterV2 with 1 ETH input, pass `1000000000000000000` (1 × 10^18)
2. The output will be in USDC units (6 decimals), e.g., `2250000000` = 2,250 USDC
3. Binance prices are already in decimal format (strings like "2245.30")
4. You need to normalize both to compare: divide smart contract values by 10^decimals

Common Mistakes to Avoid:

- ▯ Comparing `2250000000` (raw USDC) directly with `2245.30` (Binance price)
- ▯ Convert both to same format: `2250000000 / 10^6 = 2250.00` vs `2245.30`
- ▯ Using `float64` for large token amounts (precision loss)
- ▯ Use `big.Int` for calculations, convert to `big.Float` only for display

Ethereum RPC

- **Block subscription**: `eth_subscribe` with `newHeads` parameter
- **Contract calls**: `eth_call` (doesn't cost gas, read-only)

- **Gas estimates**: `eth_gasPrice` or `eth_feeHistory`
- **Rate limits**: Public endpoints (Infura/Alchemy) typically allow 100k requests/day on free tier

Binance API

- **Binance Spot API Documentation** : https://binance-docs.github.io/apidocs/spot/en/#order-book
  - **Orderbook Endpoint**: `GET https://api.binance.com/api/v3/depth`
    - Parameter: `symbol=ETHUSDC`
    - Parameter: `limit=100` (or 500, 1000 for deeper orderbook)
  - **Example**: `https://api.binance.com/api/v3/depth?symbol=ETHUSDC&limit=100`
  - **Response Format**:

    ```
    {
      "lastUpdateId": 1234567890,
      "bids": [["2245.30", "1.5"], ["2245.20", "2.3"], ...],
      "asks": [["2245.50", "1.2"], ["2245.60", "0.8"], ...]
    }
    ```

  - **Rate Limit**: 1200 requests per minute (no API key needed for public endpoints)
  - **Note**: Prices are strings, amounts are strings (parse carefully)

Go Libraries Summary

```
# Essential
go get github.com/ethereum/go-ethereum
go get github.com/gorilla/websocket

# Optional but helpful
go get github.com/daoleno/uniswapv3-sdk  # For Uniswap math utilities
go get github.com/shopspring/decimal      # For precise decimal arithmetic
```

ABI Files

You'll need the ABI for QuoterV2. You can:

1. Download from Etherscan: https://etherscan.io/address/0xb27308f9F90D607463bb33eA1BeBb41C27CE5AB6#code
2. Use `abigen` tool (comes with go-ethereum) to generate Go bindings
3. Or manually parse and call using `accounts/abi` package (shown above)

Questions?

Think about edge cases and trade-offs. We value engineers who:

- Ask clarifying questions
- Consider production implications
- Write maintainable code
- Think about observability and debugging

Good luck! We're excited to see your approach to this problem.