Hello,

Thank you for applying to join our team at GoQuant. After a thorough review of your application, we are pleased to inform you that you have been selected to move forward in our recruitment process.

As the next step, we ask you to complete the following assignment. This will provide us with a deeper understanding of your skills and how well they align with the role you have applied for. Please ensure that you submit your completed work within 7 days from today.

To summarise, the assignment is described below:

Objective

Build a **Settlement Relayer & Batch Processing System** for a high-performance decentralized perpetual futures exchange that bridges off-chain trade execution with on-chain settlement on Solana. This system is a critical component that enables high-frequency trading by batching multiple trades into efficient on-chain transactions while maintaining cryptographic proof of execution.

System Context

In a hybrid dark pool perpetual futures DEX architecture:

- Trades are matched off-chain in a dark pool matching engine (millisecond latency)
- The settlement relayer accumulates matched trades in 1-second batches
- Batches are cryptographically verified using Merkle trees
- Net positions are calculated to minimize on-chain transactions
- Settlement transactions are submitted to Solana for finality
- The system must handle 100+ trades per second throughput
- Settlement latency target: < 2 seconds from trade execution to on-chain confirmation

Your component sits between the matching engine and the blockchain, handling the critical path from trade execution to on-chain settlement.

Initial Setup

1. Set up a Rust development environment with:
   - Rust 1.75+ with async/await support
   - Anchor framework 0.29+
   - Solana CLI tools
   - PostgreSQL or SQLite for trade persistence
2. Familiarize yourself with:
   - Solana's transaction model and compute units
   - Anchor program development
   - Merkle tree construction for verification
   - Net position calculation algorithms

Core Requirements

Part 1: Solana Smart Contract (Anchor Program)

Settlement Program Instructions:

1. **Initialize Settlement Vault**
   - Create PDA-based vault for each trading pair
   - Store vault authority and configuration
   - Handle rent-exempt account requirements

2. **Process Settlement Batch**
    - Accept Merkle root of trade batch
    - Verify batch signature from authorized relayer
    - Update net positions for affected accounts
    - Emit settlement events for off-chain indexing
    - Handle partial settlement failures gracefully
3. **Account Structures**

```rust
#[account]
pub struct SettlementVault {
    pub authority: Pubkey,
    pub symbol: String,
    pub total_settled: u64,
    pub last_settlement: i64,
    pub bump: u8,
}

#[account]
pub struct PositionAccount {
    pub owner: Pubkey,
    pub symbol: String,
    pub size: i64,  // Positive = long, negative = short
    pub entry_price: u64,
    pub margin: u64,
    pub last_update: i64,
}
```

4. **Security Requirements**
    - Only authorized relayer can submit settlements
    - Validate Merkle proofs for batch integrity
    - Prevent replay attacks (nonce/timestamp)
    - Handle concurrent settlement attempts

Part 2: Rust Backend - Settlement Relayer Service

Core Components:

1. **Batch Accumulator**
    - Collect trades from matching engine (via channel/queue)
    - Accumulate trades in 1-second windows using tokio timers
    - Thread-safe batch management (Arc<Mutex<>> or similar)
    - Handle high throughput (100+ trades/second)

   Reference Implementation:

```rust
use tokio::sync::Mutex;
use std::sync::Arc;
use std::time::Duration;

pub struct SettlementRelayer {
    pending_trades: Arc<Mutex<Vec<Trade>>>,
    batch_window_ms: u64,           // 1 second
    max_batch_size: usize,          // Trades per batch
}

impl SettlementRelayer {
    pub fn new() -> Self {
        Self {
            pending_trades: Arc::new(Mutex::new(Vec::new())),
            batch_window_ms: 1000,
            max_batch_size: 50,
        }
    }

    pub async fn start(&self) {
        // Start batch timer (1 second intervals)
        let mut interval = tokio::time::interval(Duration::from_secs(1));

        loop {
            interval.tick().await;
            if let Err(e) = self.settle_batch().await {
                eprintln!("Settlement error: {}", e);
            }
        }
    }

    pub async fn add_trade(&self, trade: Trade) -> Result<(), SettlementError> {
        let mut pending = self.pending_trades.lock().await;
        pending.push(trade);

        // Trigger immediate settlement if batch is full
        if pending.len() >= self.max_batch_size {
            drop(pending); // Release lock before settling
            self.settle_batch().await?;
        }

        Ok(())
    }

    async fn settle_batch(&self) -> Result<(), SettlementError> {
        let mut pending = self.pending_trades.lock().await;
        if pending.is_empty() {
            return Ok(());
        }

        let trades_to_settle = pending.drain(..).collect::<Vec<_>>();
        drop(pending); // Release lock

        // Build and submit settlement transaction
        let tx = self.build_settlement_transaction(&trades_to_settle).await?;
        let signature = self.send_and_confirm(tx).await?;

        self.mark_trades_settled(&trades_to_settle, signature).await?;
        Ok(())
    }
}
```

2. **Net Position Calculator**

```rust
pub struct NetPositionCalculator {
    // Calculate net position changes per user per symbol
    // Example: User trades +10, -3, +5 = net +12 position change
    // Minimize on-chain transactions by netting internally
}
```

3. **Merkle Tree Builder**

   - Construct Merkle tree from trade batch
   - Generate proofs for individual trades
   - Compute Merkle root for on-chain verification
   - Efficient hashing (use sha256 or keccak256)

4. **Transaction Builder & Submitter**
    - Build Solana transactions with proper compute budget
    - Sign transactions with relayer keypair
    - Submit to Solana RPC with retry logic
    - Handle transaction confirmation
    - Monitor for failed/dropped transactions
5. **Settlement Monitor**
    - Track settlement success/failure rates
    - Implement retry logic for failed settlements
    - Alert on settlement delays or failures
    - Store settlement history in database

Data Flow:

```
Matching Engine → [Batch Accumulator] → [Net Position Calc] → [Merkle Tree] →
[Transaction Builder] → Solana RPC → [Settlement Monitor]
```

Part 3: Database Schema

Design schema for:

- Pending trades (awaiting settlement)
- Settlement batches (Merkle root, timestamp, status)
- Settlement history (on-chain tx signatures)
- Failed settlements (for retry)

Part 4: Integration & APIs

1. **Input Interface**
    - Accept trades from matching engine (design the interface)
    - Trade format: user_id, symbol, side, quantity, price, timestamp
2. **Output Interface**
    - Expose settlement status endpoint
    - Provide settlement confirmation callbacks
    - Real-time settlement monitoring (WebSocket/streaming)

Technical Requirements

1. **Performance**
    - Process 100+ trades/second sustained throughput
    - 1-second batch window strictly enforced
    - Settlement latency < 2 seconds (from batch close to on-chain confirmation)
    - Efficient memory usage for high-frequency operation
2. **Reliability**
    - Persistent storage of pending trades
    - Automatic retry for failed settlements (exponential backoff)
    - Graceful degradation if Solana RPC is slow
    - No trade loss under any failure scenario
3. **Testing**
    - Unit tests for net position calculation

- Integration tests for full settlement flow
- Anchor program tests (using `anchor test`)
- Load testing to verify throughput targets

4. **Code Quality**
    - Clean, idiomatic Rust code
    - Proper error handling (Result types, custom errors)
    - Comprehensive logging for debugging
    - Documentation for all public interfaces

Bonus Section (Recommended)

1. **Advanced Optimization**
    - Parallel transaction submission for multiple symbols
    - Transaction packing optimization (max trades per tx)
    - Priority fee calculation based on network congestion
    - State compression for position accounts
2. **Monitoring & Observability**
    - Prometheus metrics for settlement rates
    - Detailed latency breakdowns (batch, merkle, tx build, submission)
    - Alert system for settlement failures
    - Real-time dashboard for settlement health
3. **Advanced Features**
    - Support for settlement disputes/reversals
    - Multi-relayer coordination (leader election)
    - Cross-program invocation (CPI) to liquidation engine
    - Optimistic settlement with fraud proofs
4. **Performance Benchmarking**
    - Measure end-to-end settlement latency
    - Analyze bottlenecks in the pipeline
    - Compare different Merkle tree implementations
    - Optimize Solana compute unit usage

Documentation Requirements

Provide detailed documentation covering:

1. **System Architecture**
    - Component diagram showing data flow
    - Explanation of design decisions
    - Trade-offs considered (consistency vs latency, etc.)
2. **Smart Contract Documentation**
    - Account structures and their purposes
    - Instruction specifications
    - Security considerations and assumptions
    - Upgrade path and versioning strategy
3. **Backend Service Documentation**
    - Module structure and responsibilities
    - Threading/async model explanation
    - Database schema and rationale

   - Configuration parameters and tuning

4. **Integration Guide**
   - How to connect to the matching engine
   - How to query settlement status
   - Error handling and retry logic
   - Monitoring and debugging tips

Deliverables

1. **Complete Source Code**
   - Anchor program (smart contracts)
   - Rust backend service (relayer)
   - Database migrations/schema
   - Configuration files
   - Comprehensive test suite

2. **Video Demonstration** (10-15 minutes)
   - System overview and architecture
   - Live demo of settlement flow
   - Code walkthrough of critical components
   - Explanation of design choices
   - Discussion of challenges and solutions

3. **Technical Documentation**
   - Architecture documentation
   - API specifications
   - Deployment instructions
   - Performance analysis (if bonus completed)

4. **Test Results**
   - Unit test coverage report
   - Integration test logs
   - Load test results showing throughput
   - Settlement latency measurements

INSTRUCTIONS FOR SUBMISSION - MANDATORY FOR ACCEPTANCE

SUBMIT THE ASSIGNMENT VIA EMAIL TO: careers@goquant.io **AND CC:** himanshu.vairagade@goquant.io

REQUIRED ATTACHMENTS:

- Your resume
- Source code (GitHub repository link or zip file)
- Video demonstration (YouTube unlisted link or file)
- Technical documentation (PDF or Markdown)
- Test results and performance data

Upon receiving your submission, our team will carefully review your work. Should your assignment meet our expectations, we will move forward with the final steps, which could include extending a formal offer to join our team.

We look forward to seeing your work and wish you the best of luck in this important stage of the application.

Best Regards,
GoQuant Team

Confidentiality Notice (PLEASE READ)

The contents of this assignment and any work produced in response to it are strictly confidential. This document and the developed solution are intended solely for the GoQuant recruitment process and should not be posted publicly or shared with anyone outside the recruitment team. For example: do not publicly post your assignment on GitHub or Youtube. Everything must remain private and only accessible to GoQuant's team.