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 **Position Management System** for a decentralized perpetual futures exchange that handles leveraged positions with precise margin calculations, PnL tracking, and risk management. This system is the core of the trading platform, managing user positions with leverage up to 1000x while ensuring protocol solvency.

System Context

In a high-performance perpetual futures DEX architecture:

- Users open leveraged positions (long/short) on perpetual futures contracts
- Positions are tracked on-chain using Solana Program Derived Addresses (PDAs)
- The system calculates margin requirements, unrealized PnL, and liquidation prices in real-time
- Backend services monitor positions and interact with the liquidation engine
- Position data must be queryable for UI display and risk monitoring
- The platform supports isolated margin with up to 1000x leverage
- Real-time risk management prevents protocol insolvency

Your component manages the lifecycle of trading positions from opening to closing, including all margin and PnL calculations.

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 for position history and analytics
2. Familiarize yourself with:
    - Perpetual futures mechanics (funding rates, mark price, liquidation)
    - Margin calculations (initial margin, maintenance margin)
    - PnL calculation formulas (realized vs unrealized)
    - Solana account rent and PDA derivation

Core Requirements

Part 1: Solana Smart Contract (Anchor Program)

Position Program Instructions:

1. **Open Position**

```
pub fn open_position(
    ctx: Context<OpenPosition>,
    symbol: String,
    side: Side,  // Long or Short
    size: u64,
    leverage: u8,  // 1-1000x
    entry_price: u64,
) -> Result<()>
```

- Create PDA-based position account
- Validate margin requirements
- Lock collateral in vault
- Emit position opened event

2. **Modify Position**

- Increase position size
- Decrease position size (partial close)
- Add margin (reduce leverage)
- Remove margin (if safe)

3. **Close Position**

- Calculate final PnL
- Return remaining margin to user
- Handle funding rate payments
- Update user's realized PnL

4. **Account Structures**

```
#[account]
pub struct Position {
    pub owner: Pubkey,
    pub symbol: String,
    pub side: Side,  // Long(1) or Short(-1)
    pub size: u64,
    pub entry_price: u64,
    pub margin: u64,
    pub leverage: u8,
    pub unrealized_pnl: i64,
    pub realized_pnl: i64,
    pub funding_accrued: i64,
    pub liquidation_price: u64,
    pub last_update: i64,
    pub bump: u8,
}

#[account]
pub struct UserAccount {
    pub owner: Pubkey,
    pub total_collateral: u64,
    pub locked_collateral: u64,
    pub total_pnl: i64,
    pub position_count: u32,
}
```

5. **Margin Calculations**

- Initial Margin = (Position Size × Entry Price) / Leverage
- Maintenance Margin = Initial Margin × Maintenance Margin Ratio (e.g., 50%)
- Margin Ratio = (Margin + Unrealized PnL) / (Position Value)
- Liquidation when Margin Ratio < Maintenance Margin Ratio

6. **Leverage Tiers & Risk Limits**

The system must implement dynamic leverage limits based on position size:

```rust
#[derive(Debug, Clone)]
pub struct LeverageTier {
    pub max_leverage: u16,
    pub initial_margin_rate: f64,    // e.g., 0.05 for 5%
    pub maintenance_margin_rate: f64, // e.g., 0.025 for 2.5%
    pub max_position_size: u64,      // in USDT
}

pub const LEVERAGE_TIERS: [LeverageTier; 5] = [
    LeverageTier {
        max_leverage: 20,
        initial_margin_rate: 0.05,      // 5.0%
        maintenance_margin_rate: 0.025, // 2.5%
        max_position_size: u64::MAX,    // Unlimited
    },
    LeverageTier {
        max_leverage: 50,
        initial_margin_rate: 0.02,      // 2.0%
        maintenance_margin_rate: 0.01,  // 1.0%
        max_position_size: 100_000,
    },
    LeverageTier {
        max_leverage: 100,
        initial_margin_rate: 0.01,      // 1.0%
        maintenance_margin_rate: 0.005, // 0.5%
        max_position_size: 50_000,
    },
    LeverageTier {
        max_leverage: 500,
        initial_margin_rate: 0.005,      // 0.5%
        maintenance_margin_rate: 0.0025, // 0.25%
        max_position_size: 20_000,
    },
    LeverageTier {
        max_leverage: 1000,
        initial_margin_rate: 0.002,     // 0.2%
        maintenance_margin_rate: 0.001, // 0.1%
        max_position_size: 5_000,
    },
];

fn get_leverage_tier(leverage: u16, position_size: u64) -> Result<LeverageTier,
ValidationError> {
    for tier in &LEVERAGE_TIERS {
        if leverage <= tier.max_leverage && position_size <= tier.max_position_size {
            return Ok(tier.clone());
        }
    }
    Err(ValidationError::LeverageExceeded)
}
```

7. **Security Requirements**

- Validate leverage limits using tier system (1-1000x)
- Ensure sufficient margin before opening
- Prevent position size exceeding tier limits
- Handle integer overflow in calculations
- Atomic state updates to prevent race conditions

Part 2: Rust Backend - Position Management Service

Core Components:

1. **Position Manager**

```rust
pub struct PositionManager {
    // Manages position lifecycle
    // Tracks open positions per user
    // Calculates real-time metrics
}
```

- Open new positions via Solana transactions
- Modify existing positions

- Close positions and settle PnL
- Query position data from on-chain accounts

2. **Margin Calculator**

```
pub struct MarginCalculator {
    // Calculate required margin for positions
    // Compute margin ratios
    // Determine liquidation prices
}
```

Key Formulas (Implemented in Rust):

```
// Entry Price (for averaged positions)
fn calculate_average_entry_price(trades: &[(f64, f64)]) -> f64 {
    let (total_value, total_size) = trades.iter()
        .fold((0.0, 0.0), |(val, size), (price, qty)| {
            (val + price * qty, size + qty)
        });
    total_value / total_size
}

// Unrealized PnL
fn calculate_unrealized_pnl(is_long: bool, size: f64, mark_price: f64, entry_price: f64) -> f64
{
    if is_long {
        size * (mark_price - entry_price)
    } else {
        size * (entry_price - mark_price)
    }
}

// Margin Ratio
fn calculate_margin_ratio(collateral: f64, unrealized_pnl: f64, size: f64, mark_price: f64) ->
f64 {
    let position_value = size * mark_price;
    (collateral + unrealized_pnl) / position_value
}

// Liquidation Price (Long)
fn calculate_liquidation_price_long(entry_price: f64, leverage: f64, maintenance_margin_ratio:
f64) -> f64 {
    entry_price * (1.0 - 1.0/leverage + maintenance_margin_ratio)
}

// Liquidation Price (Short)
fn calculate_liquidation_price_short(entry_price: f64, leverage: f64, maintenance_margin_ratio:
f64) -> f64 {
    entry_price * (1.0 + 1.0/leverage - maintenance_margin_ratio)
}
```

3. **PnL Tracker**

- Real-time unrealized PnL calculation
- Track realized PnL history
- Account for funding rate payments
- Handle position averaging (multiple entries)

4. **Position Monitor**

- Continuously monitor all open positions
- Update mark prices from oracle
- Flag positions approaching liquidation
- Emit alerts for risk management
- Publish position data to WebSocket clients

5. **Database Integration**

- Store position history for analytics
- Track position state changes

- Enable historical PnL queries
- Support position snapshots

Position State Machine:

```
OPENING → OPEN → MODIFYING → OPEN → CLOSING → CLOSED
                    ↓
                LIQUIDATING
```

Part 3: Database Schema

Design schema for:

- Positions (current and historical)
- Position modifications (audit trail)
- PnL snapshots (hourly/daily)
- User trading statistics
- Risk metrics per user

Part 4: Integration & APIs

1. **REST API Endpoints**

```
POST   /positions/open       - Open new position
PUT    /positions/:id/modify - Modify position
DELETE /positions/:id/close  - Close position
GET    /positions/:id        - Get position details
GET    /users/:id/positions  - Get user's positions
```

2. **WebSocket Streams**
   - Real-time position updates
   - PnL changes on price movements
   - Margin ratio alerts
   - Position events (opened, modified, closed)
3. **Internal Interfaces**
   - Liquidation engine integration
   - Settlement relayer integration
   - Funding rate system integration

Technical Requirements

1. **Performance**
   - Support 10,000+ concurrent open positions
   - Real-time PnL updates (< 100ms latency)
   - Position queries < 50ms
   - Handle 100+ position operations per second
2. **Accuracy**
   - Precise fixed-point arithmetic (no floating point errors)
   - Correct PnL calculations under all market conditions
   - Accurate liquidation price determination
   - Proper handling of funding rate adjustments
3. **Reliability**

- Atomic position updates (no partial states)
- Consistent state between on-chain and off-chain
- Handle Solana transaction failures gracefully
- Automatic reconciliation of position state

4. **Testing**
   - Unit tests for all margin/PnL calculations
   - Property-based testing for edge cases
   - Integration tests with mock oracle prices
   - Anchor program tests for all instructions
   - Fuzz testing for overflow scenarios
5. **Code Quality**
   - Type-safe position state management
   - Clear separation of concerns
   - Comprehensive error handling
   - Well-documented calculation formulas

Bonus Section (Recommended)

1. **Advanced Risk Management**
   - Dynamic leverage limits based on volatility
   - Position size limits per user tier
   - Maximum open interest per symbol
   - Cross-position margin utilization
2. **Performance Optimization**
   - Batch position updates for efficiency
   - Caching strategies for frequently accessed data
   - Optimized database queries with indexes
   - Parallel processing for position monitoring
3. **Advanced Features**
   - Stop-loss and take-profit orders
   - Trailing stops
   - Position hedging detection
   - Portfolio-level risk metrics (total exposure, VAR)
4. **Analytics & Reporting**
   - Position performance analytics
   - Win rate and profit factor calculations
   - Historical PnL charts
   - Risk-adjusted returns (Sharpe ratio)
5. **State Management**
   - Position versioning for upgrades
   - Historical position reconstruction
   - Snapshot and restore functionality
   - Data migration tools

Documentation Requirements

Provide detailed documentation covering:

1. **System Architecture**

- Component interaction diagram
- Position lifecycle state diagram
- Data flow for position operations
- Design rationale for key decisions

2. **Mathematical Formulas**
   - All margin calculation formulas with examples
   - PnL calculation methodology
   - Liquidation price derivation
   - Funding rate integration
3. **Smart Contract Documentation**
   - Account structures and their purposes
   - Instruction specifications with examples
   - Security considerations
   - Testing strategy
4. **Backend Service Documentation**
   - Module architecture
   - API specifications
   - Database schema documentation
   - Configuration and deployment
5. **Risk Management Guide**
   - How positions are monitored
   - Margin call process
   - Liquidation triggers
   - Protocol safety mechanisms

Deliverables

1. **Complete Source Code**
   - Anchor program (position management)
   - Rust backend service
   - Database migrations/schema
   - Comprehensive test suite
   - Configuration examples
2. **Video Demonstration** (10-15 minutes)
   - System architecture overview
   - Live demo of position lifecycle (open, modify, close)
   - Code walkthrough of margin calculations
   - Explanation of PnL tracking
   - Discussion of edge cases and handling
3. **Technical Documentation**
   - Architecture documentation with diagrams
   - Formula reference guide
   - API documentation
   - Deployment guide
   - Performance analysis (if bonus completed)
4. **Test Results**
   - Unit test coverage report
   - Calculation accuracy verification

- Load test results
- Edge case test scenarios

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.