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 **Liquidation Engine** for a high-leverage perpetual futures exchange that monitors undercollateralized positions, executes partial liquidations, and manages an insurance fund to protect protocol solvency. This is a critical risk management component that prevents systemic losses from cascading liquidations.

System Context

In a leveraged perpetual futures DEX (supporting up to 1000x leverage):

- Users can become undercollateralized due to adverse price movements
- Positions must be liquidated before losses exceed collateral
- Partial liquidations are preferred to minimize market impact
- Insurance fund absorbs remaining losses if liquidation is insufficient
- Liquidation must happen quickly to prevent protocol losses
- System must handle oracle manipulation attempts
- Real-time monitoring of 1000+ concurrent positions
- Sub-second reaction time to price changes
- Liquidators are rewarded for executing liquidations

Your component is the last line of defense protecting the protocol from insolvency during volatile market conditions.

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 liquidation history and analytics
    - Redis for high-performance position tracking
2. Familiarize yourself with:
    - Liquidation mechanics in derivatives trading
    - Margin maintenance and liquidation thresholds
    - Solana transaction priority and compute budgets
    - Oracle price feed integration

Core Requirements

Part 1: Solana Smart Contract (Anchor Program)

Liquidation Program Instructions:

1. **Liquidate Position (Partial)**

```
pub fn liquidate_partial(
    ctx: Context<LiquidatePartial>,
    liquidation_amount: u64,
) -> Result<()>
```

- Verify position is liquidatable (margin ratio < maintenance)
- Reduce position size by specified amount
- Calculate and distribute liquidation fee
- Pay liquidator reward (e.g., 2.5% of liquidated value)
- Update position account state

2. **Liquidate Position (Full)**

```
pub fn liquidate_full(
    ctx: Context<LiquidateFull>,
) -> Result<()>
```

- Close entire position
- Calculate remaining margin after PnL
- If margin positive: pay liquidator fee, return rest to user
- If margin negative (bad debt): transfer to insurance fund
- Emit liquidation event

3. **Account Structures**

```
#[account]
pub struct LiquidationRecord {
    pub position_owner: Pubkey,
    pub liquidator: Pubkey,
    pub symbol: String,
    pub liquidated_size: u64,
    pub liquidation_price: u64,
    pub margin_before: u64,
    pub margin_after: u64,
    pub liquidator_reward: u64,
    pub bad_debt: u64,
    pub timestamp: i64,
}

#[account]
pub struct InsuranceFund {
    pub authority: Pubkey,
    pub balance: u64,
    pub total_contributions: u64,
    pub total_bad_debt_covered: u64,
    pub utilization_ratio: u64,  // basis points
}
```

4. **Liquidation Logic**

- Margin Ratio = (Margin + Unrealized PnL) / Position Value
- Liquidatable when Margin Ratio < Maintenance Margin Ratio (e.g., 0.5%)
- Partial liquidation reduces position by 50% of size
- Full liquidation occurs if partial liquidation insufficient

5. **Security Requirements**

- Verify oracle price is fresh (< 30 seconds old)
- Prevent liquidation of healthy positions
- Rate limit liquidations per position
- Handle concurrent liquidation attempts

Part 2: Rust Backend - Liquidation Engine Service

Core Components:

1. **Position Monitor**

   Real-time monitoring system for all leveraged positions:

```rust
use tokio::time::{interval, Duration};
use std::collections::HashMap;

pub struct LiquidationEngine {
    check_interval_ms: u64,              // Check every 1 second
    oracle: Arc<PriceOracle>,
    position_manager: Arc<PositionManager>,
    liquidation_executor: Arc<LiquidationExecutor>,
}

impl LiquidationEngine {
    pub fn new() -> Self {
        Self {
            check_interval_ms: 1000,
            oracle: Arc::new(PriceOracle::new()),
            position_manager: Arc::new(PositionManager::new()),
            liquidation_executor: Arc::new(LiquidationExecutor::new()),
        }
    }

    pub async fn start(&self) {
        let mut timer = interval(Duration::from_millis(self.check_interval_ms));

        loop {
            timer.tick().await;
            if let Err(e) = self.check_all_positions().await {
                eprintln!("Error checking positions: {:?}", e);
            }
        }
    }

    async fn check_all_positions(&self) -> Result<(), LiquidationError> {
        let open_positions = self.position_manager.get_open_positions().await?;
        let mut price_cache: HashMap<String, f64> = HashMap::new();

        for position in open_positions {
            // Get or cache mark price (avoid redundant oracle calls)
            let mark_price = if let Some(&cached) = price_cache.get(&position.symbol) {
                cached
            } else {
                let price = self.oracle.get_mark_price(&position.symbol).await?;
                price_cache.insert(position.symbol.clone(), price);
                price
            };

            // Calculate unrealized PnL
            let unrealized_pnl = if position.is_long {
                position.size * (mark_price - position.entry_price)
            } else {
                position.size * (position.entry_price - mark_price)
            };

            // Calculate margin ratio
            let position_value = position.size * mark_price;
            let margin_ratio = (position.collateral + unrealized_pnl) / position_value;

            // Check if position needs liquidation
            let maintenance_margin_ratio =
self.get_maintenance_margin_ratio(position.leverage);

            if margin_ratio < maintenance_margin_ratio {
                // Flag for liquidation
                self.liquidation_executor.liquidate_position(
                    position.id,
                    position.user,
                    mark_price,
                    margin_ratio,
                ).await?;
            }
        }

        Ok(())
    }

    fn get_maintenance_margin_ratio(&self, leverage: u16) -> f64 {
        // Based on leverage tiers
        match leverage {
            1..=20 => 0.025,    // 2.5%
            21..=50 => 0.01,    // 1.0%
            51..=100 => 0.005, // 0.5%
            101..=500 => 0.0025, // 0.25%
            501..=1000 => 0.001, // 0.1%
            _ => 0.025, // Default to safest
            }
        }
    }
```

2. **Liquidation Queue**

```
    pub struct LiquidationQueue {
        // Priority queue of positions to liquidate
        // Order by urgency (lowest margin ratio first)
    }
```

   - Priority queue based on margin ratio
   - Deduplication to prevent double liquidation
   - Cooldown period after liquidation attempt
   - Failed liquidation retry logic

3. **Liquidation Executor**

   - Build liquidation transactions
   - Calculate optimal liquidation size (partial vs full)
   - Sign transactions with liquidator keypair
   - Submit with high priority fee
   - Monitor transaction confirmation
   - Handle transaction failures

4. **Insurance Fund Manager**

```
    pub struct InsuranceFundManager {
        // Track insurance fund balance
        // Cover bad debt from failed liquidations
        // Alert when fund is depleted
    }
```

   - Monitor insurance fund balance
   - Calculate bad debt coverage capacity
   - Emit alerts when fund below threshold
   - Track historical bad debt events

5. **Oracle Price Feed Integration**

   - Connect to Pyth Network for mark prices
   - Fallback to Switchboard oracle
   - Validate price confidence intervals
   - Handle oracle downtime gracefully
   - Cache prices for performance

Liquidation Flow:

```
Position Monitor → Liquidation Queue → Liquidation Executor → Solana Program
       ↓                                         ↓
   Redis Cache                          Transaction Confirmation
                                               ↓
                                      Insurance Fund (if bad debt)
```

Part 3: Database Schema

Design schema for:

- Liquidation history (all liquidation events)
- Bad debt records

- Liquidator performance (rewards earned)
- Insurance fund transactions
- Failed liquidation attempts (for analysis)

Part 4: Integration & APIs

1. **Internal APIs**
    - Position data feed from position manager
    - Oracle price feed subscription
    - Insurance fund state queries
2. **Monitoring Endpoints**

```
GET /liquidations/pending     - Positions at risk
GET /liquidations/history     - Recent liquidations
GET /insurance-fund/status    - Fund balance & health
GET /liquidators/stats        - Liquidator performance
```

3. **WebSocket Streams**
    - Real-time liquidation events
    - At-risk position alerts
    - Insurance fund status updates
4. **Alert System**
    - Critical alerts for liquidation failures
    - Warnings when insurance fund low
    - Notifications for large liquidations

Technical Requirements

1. **Performance**
    - Monitor 10,000+ positions in real-time
    - Detect liquidatable positions within 100ms of price change
    - Execute liquidations within 500ms of detection
    - Handle burst liquidations during market volatility (100+ simultaneous)
2. **Reliability**
    - No missed liquidations (100% coverage)
    - Retry failed liquidations automatically
    - Handle Solana RPC failures gracefully
    - Maintain operation during price oracle outages
3. **Accuracy**
    - Precise margin ratio calculations
    - Correct liquidation price determination
    - Accurate bad debt accounting
    - Proper liquidator reward distribution
4. **Testing**
    - Unit tests for liquidation logic
    - Integration tests with mock positions
    - Stress tests for concurrent liquidations
    - Anchor program tests for all scenarios
    - Simulation of extreme market conditions

5. **Code Quality**
    - Thread-safe position monitoring
    - Efficient data structures for priority queue
    - Robust error handling
    - Comprehensive logging for audit trail

Bonus Section (Recommended)

1. **Advanced Liquidation Strategies**
    - Dynamic partial liquidation sizing
    - Batch liquidations for efficiency
    - Liquidation cost optimization (minimize compute units)
    - MEV protection for liquidators
2. **Risk Management**
    - Pre-liquidation warnings to users
    - Automatic position size reduction before liquidation
    - Dynamic maintenance margin based on volatility
    - Circuit breakers for extreme events
3. **Insurance Fund Optimization**
    - Automatic fund replenishment strategies
    - Yield generation on idle funds
    - Multi-tier insurance fund structure
    - Socialized loss mechanism if fund depleted
4. **Monitoring & Analytics**
    - Real-time dashboard for liquidation activity
    - Historical liquidation analysis
    - Liquidator leaderboard
    - Bad debt trends and forecasting
    - Alert escalation system
5. **Performance Optimization**
    - WebSocket connection pooling for position updates
    - Redis caching for hot positions
    - Parallel transaction submission
    - Optimized compute budget usage

Documentation Requirements

Provide detailed documentation covering:

1. **System Architecture**
    - Component interaction diagram
    - Liquidation decision flow
    - Data flow and state management
    - Threading/async model
2. **Liquidation Mechanics**
    - When positions become liquidatable
    - Partial vs full liquidation criteria
    - Liquidator reward calculation
    - Bad debt handling process

3. **Smart Contract Documentation**
    - Instruction specifications
    - Account structures
    - Security measures
    - Edge case handling
4. **Backend Service Documentation**
    - Module responsibilities
    - Configuration parameters
    - API specifications
    - Monitoring and alerting
5. **Operational Guide**
    - How to run the liquidation engine
    - Monitoring liquidation health
    - Handling stuck liquidations
    - Insurance fund management

Deliverables

1. **Complete Source Code**
    - Anchor program (liquidation logic)
    - Rust backend service (liquidation engine)
    - Database migrations/schema
    - Comprehensive test suite
    - Configuration files
2. **Video Demonstration** (10-15 minutes)
    - System architecture overview
    - Live demo of liquidation process
    - Code walkthrough of monitoring logic
    - Explanation of partial vs full liquidation
    - Discussion of edge cases (bad debt, oracle failures)
3. **Technical Documentation**
    - Architecture documentation with diagrams
    - Liquidation mechanics explanation
    - API documentation
    - Deployment guide
    - Performance analysis (if bonus completed)
4. **Test Results**
    - Unit test coverage report
    - Stress test results
    - Liquidation latency measurements
    - Concurrent liquidation handling

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.