

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 **Funding Rate Calculation System** for perpetual futures contracts on a decentralized exchange. This system calculates funding rates every second and distributes hourly payments between long and short position holders, ensuring the perpetual contract price converges with the spot price.

System Context

In a perpetual futures DEX architecture:

- Perpetual futures don't have expiration dates (unlike traditional futures)
- Funding rates create economic incentive for price convergence with spot
- System calculates funding rate every 1 second based on mark price vs index price
- Payments are accumulated and distributed hourly
- Positive funding rate: longs pay shorts
- Negative funding rate: shorts pay longs
- System must handle high frequency calculation (86,400 calculations per day per symbol)
- Must support 50+ trading symbols simultaneously

Your component is crucial for maintaining market efficiency and fair pricing in the perpetual futures market.

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 funding rate history
 - Redis for high-frequency calculations
 2. Familiarize yourself with:
 - Perpetual futures funding rate mechanics
 - Premium/discount index calculations
 - Oracle price feed integration (Pyth/Switchboard)
 - High-frequency calculation patterns
-

Core Requirements

Part 1: Solana Smart Contract (Anchor Program)

Funding Rate Program Instructions:

1. **Update Funding Rate**

```
pub fn update_funding_rate(
    ctx: Context<UpdateFundingRate>,
    symbol: String,
    mark_price: u64,
    index_price: u64,
) -> Result<()>
```

- Calculate premium index: (Mark Price - Index Price) / Index Price
- Add interest rate component (e.g., 0.01% daily / 86400 seconds)
- Clamp funding rate to prevent manipulation (e.g., $\pm 0.05\%$)
- Store funding rate in account
- Update calculation timestamp

2. Apply Hourly Funding

```
pub fn apply_hourly_funding(
    ctx: Context<ApplyFunding>,
    position_account: Pubkey,
) -> Result<()>
```

- Calculate accrued funding for the hour
- Funding Payment = Position Size \times Average Funding Rate \times Hours
- Positive payment: deduct from position margin
- Negative payment: add to position margin
- Update position's last funding timestamp
- Emit funding payment event

Account Structures:

```
#[account]
pub struct FundingRateState {
    pub symbol: String,
    pub current_rate: i64, // Signed, basis points (10000 = 1%)
    pub mark_price: u64,
    pub index_price: u64,
    pub premium_index: i64,
    pub interest_rate: i64,
    pub last_update: i64,
    pub hourly_samples: [i64; 3600], // Store 1 hour of 1-second samples
    pub sample_index: u16,
}

#[account]
pub struct FundingHistory {
    pub symbol: String,
    pub timestamp: i64,
    pub funding_rate: i64,
    pub mark_price: u64,
    pub index_price: u64,
    pub total_long_oi: u64, // Open interest
    pub total_short_oi: u64,
}
```

Funding Rate Formula:

```
Premium Index = (Mark Price - Index Price) / Index Price
Interest Rate = 0.01% / 86400 // Daily rate per second
Funding Rate = Premium Index + Interest Rate
Clamped Rate = max(-0.05%, min(0.05%, Funding Rate))

Hourly Payment = Position Size  $\times$  Avg(Funding Rates over hour)  $\times$  1 hour
```

Part 2: Rust Backend - Funding Rate Service

Core Components:**1. Funding Rate Calculator**

Implements high-frequency funding rate calculation for all markets:

```

use std::time::Instant;
use tokio::task::JoinSet;
use std::sync::Arc;

pub struct FundingRatePerformance {
    calculation_interval_ms: u64,          // 1000ms (1 second)
    target_calculation_time_ms: u64,      // Target: < 100ms
    oracle: Arc<ConsolidatedOracle>,
    redis: Arc<RedisPool>,
    alert_service: Arc<AlertService>,
}

#[derive(Debug, Serialize)]
pub struct FundingRate {
    pub symbol: String,
    pub funding_rate: f64,
    pub premium_index: f64,
    pub mark_price: f64,
    pub index_price: f64,
    pub timestamp: i64,
}

impl FundingRatePerformance {
    pub async fn calculate_all_funding_rates(&self, symbols: &[String]) ->
    Result<PerformanceMetrics, FundingError> {
        let start = Instant::now();
        let mut tasks = JoinSet::new();

        // Parallel calculation for all symbols
        for symbol in symbols {
            let symbol = symbol.clone();
            let oracle = self.oracle.clone();

            tasks.spawn(async move {
                let mark_price = oracle.get_mark_price(&symbol).await?;
                let index_price = oracle.get_index_price(&symbol).await?;

                // Calculate premium index
                let premium_index = (mark_price - index_price) / index_price;

                // Interest rate: 0.01% / 24 hours / 3600 seconds
                let interest_rate = 0.0001 / 86400.0;

                let funding_rate = premium_index + interest_rate;

                // Clamp to reasonable bounds (-0.05% to +0.05%)
                let clamped_rate = funding_rate.max(-0.0005).min(0.0005);

                Ok(FundingRate {
                    symbol,
                    funding_rate: clamped_rate,
                    premium_index,
                    mark_price,
                    index_price,
                    timestamp: chrono::Utc::now().timestamp(),
                })
            });
        }

        // Collect results
        let mut funding_rates = Vec::new();
        while let Some(result) = tasks.join_next().await {
            match result {
                Ok(Ok(rate)) => funding_rates.push(rate),
                Ok(Err(e)) => eprintln!("Funding rate calculation error: {}", e),
                Err(e) => eprintln!("Task join error: {}", e),
            }
        }

        let duration = start.elapsed();
        let duration_ms = duration.as_secs_f64() * 1000.0;

        // Store in Redis for fast access
        self.cache_funding_rates(&funding_rates).await?;

        // Alert if performance target not met
        if duration_ms > self.target_calculation_time_ms as f64 {
            self.alert_service.send_alert(&Format!(
                "Funding rate calculation exceeded target: {}ms > {}ms",
                duration_ms, self.target_calculation_time_ms
            )).await?;
        }
    }
}

```

```

        Ok(PerformanceMetrics {
            symbol_count: symbols.len(),
            duration_ms,
            average_per_symbol: duration_ms / symbols.len() as f64,
            updates_per_second: symbols.len() as f64 / duration.as_secs_f64(),
            target_met: duration_ms <= self.target_calculation_time_ms as f64,
        })
    }

    async fn cache_funding_rates(&self, rates: &[FundingRate]) -> Result<(), RedisError> {
        for rate in rates {
            let key = format!("funding_rate:{}", rate.symbol);
            let value = serde_json::to_string(rate)?;
            self.redis.set_ex(&key, &value, 60).await?; // 60 second TTL
        }
        Ok(())
    }
}

```

- Run calculation loop every 1 second (tokio interval)
- Fetch mark price from internal oracle
- Fetch index price from external oracle (Pyth)
- Calculate premium index
- Apply clamping rules
- Store in Redis for fast access

2. Oracle Integration

```

pub struct OracleManager {
    // Connect to Pyth Network
    // Fetch Switchboard as fallback
    // Validate price freshness
}

```

- Subscribe to Pyth price feeds
- Validate price confidence intervals
- Handle stale price data
- Fallback to Switchboard if Pyth unavailable
- Cache prices in memory for performance

3. Funding Rate Aggregator

- Collect 3600 samples (1 per second for 1 hour)
- Calculate hourly average
- Weighted average if needed
- Store aggregated rates in database

4. Payment Distributor

```

pub struct FundingPaymentDistributor {
    // Apply funding payments hourly
    // Process all open positions
    // Handle edge cases
}

```

- Trigger every hour
- Query all open positions for each symbol
- Calculate individual funding payments
- Build batch transaction for on-chain updates
- Handle positions that close mid-hour
- Retry failed payment distributions

5. Historical Data Manager

- Store funding rate time series
- Calculate funding rate statistics

- Provide historical queries for UI
- Data compression for old records

Calculation Pipeline:

```
[1-second Timer] → Oracle Prices → Calculate Rate → Store Redis →  
[Hourly Timer] → Aggregate Samples → Apply Payments → Store History
```

Part 3: Database Schema

Design schema for:

- Funding rate history (per symbol, per second/hour)
- Funding payment records (per position)
- Oracle price history
- Calculation metrics (latency, errors)
- Aggregate statistics (daily avg, max, min)

Part 4: Integration & APIs

1. REST API Endpoints

GET /funding/current/:symbol	- Current funding rate
GET /funding/history/:symbol	- Historical rates
GET /funding/payments/:position	- Payment history
GET /funding/stats/:symbol	- Rate statistics
GET /funding/next-payment	- Next payment time

2. WebSocket Streams

- Real-time funding rate updates (every second)
- Hourly payment notifications
- Funding rate changes
- Premium/discount alerts

3. Internal Interfaces

- Position manager integration (get open positions)
- Oracle service integration
- Settlement system callbacks

Technical Requirements

1. Performance

- 1-second calculation interval (precise timing)
- Support 50+ trading symbols simultaneously
- Calculate funding for 10,000+ positions within 1 minute (hourly)
- Oracle price fetch < 100ms
- Calculation latency < 50ms

2. Accuracy

- Precise fixed-point arithmetic
- No accumulation of rounding errors
- Correct handling of negative funding rates
- Accurate payment calculations

3. Reliability

- Never miss a 1-second calculation
- Handle oracle outages gracefully
- Retry failed payment distributions
- Maintain calculation even during network issues
- Persistent storage of all rates

4. Testing

- Unit tests for funding rate formulas
- Integration tests with mock oracle
- Stress tests for 3600 calculations
- Anchor program tests
- Edge case testing (extreme rates, oracle failures)

5. Code Quality

- Efficient interval-based processing
 - Robust error handling
 - Comprehensive logging
 - Clear separation of concerns
-

Bonus Section (Recommended)

1. Advanced Features

- Dynamic funding rate caps based on volatility
- Multi-leg funding (if position spread across symbols)
- Funding rate prediction model
- Real-time funding rate arbitrage detection

2. Performance Optimization

- Parallel calculation for multiple symbols
- Efficient data structures for 3600 samples
- Batch oracle price fetching
- Compressed storage for historical data

3. Analytics & Insights

- Funding rate charts and visualizations
- Statistical analysis (mean, median, volatility)
- Correlation with market conditions
- Predictive indicators

4. Risk Management

- Alert on extreme funding rates
- Cap large funding payments
- Funding rate manipulation detection
- Circuit breakers for anomalies

5. Oracle Enhancements

- Multi-source price aggregation (more than 2)
 - Outlier detection and filtering
 - Confidence-weighted averaging
 - Oracle downtime fallback strategies
-

Documentation Requirements

Provide detailed documentation covering:

1. System Architecture

- Calculation pipeline diagram
- Oracle integration flow
- Payment distribution process
- Data flow and timing

2. Funding Rate Mechanics

- Mathematical formulas with examples
- Premium index calculation
- Interest rate component
- Payment calculation methodology

3. Smart Contract Documentation

- Account structures
- Instruction specifications
- State management
- Error handling

4. Backend Service Documentation

- Module architecture
- Timing and scheduling
- API specifications
- Configuration parameters

5. Operational Guide

- How to run the service
 - Monitoring funding calculations
 - Handling missed calculations
 - Oracle failover procedures
-

Deliverables

1. Complete Source Code

- Anchor program
- Rust backend service
- Database migrations
- Test suite
- Configuration files

2. Video Demonstration (10-15 minutes)

- System architecture
- Live calculation demo
- Code walkthrough
- Oracle integration
- Payment distribution process

3. Technical Documentation

- Architecture documentation
- Formula reference
- API documentation
- Deployment guide
- Performance analysis

4. Test Results

- Test coverage report

- Timing accuracy verification
 - Load test results
 - Oracle failover testing
-

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.