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 **Collateral Vault Management System** for a decentralized perpetual futures exchange that securely holds user collateral (USDT) in program-controlled vaults, manages deposits and withdrawals, and enables cross-program invocations for trading operations. This system is the foundation of non-custodial trading on the platform.

System Context

In a high-performance perpetual futures DEX architecture:

- Users deposit USDT collateral into program-controlled vaults (PDAs)
- Vaults are PDA-based accounts owned by the smart contract
- Collateral can be used for opening leveraged positions
- System must track locked vs available collateral in real-time
- Withdrawals require verification of no open positions or sufficient free collateral
- Cross-program invocations (CPIs) enable position management and settlement
- Vault security is paramount (handles all user funds)
- Support for 1000+ concurrent users with isolated vault accounts
- Atomic operations to prevent double-spending or race conditions

Your component is the custody layer that ensures secure, non-custodial management of all user funds in the protocol.

Initial Setup

1. Set up a Rust development environment with:
   - Rust 1.75+ with async/await support
   - Anchor framework 0.29+
   - Solana CLI tools
   - SPL Token program knowledge
   - PostgreSQL for transaction history
2. Familiarize yourself with:
   - Program Derived Addresses (PDAs)
   - SPL Token program operations
   - Cross-program invocations (CPIs)
   - Rent-exempt accounts
   - Token account ownership

Core Requirements

Part 1: Solana Smart Contract (Anchor Program)

Collateral Vault Program Instructions:

1. **Initialize User Vault**

```rust
pub fn initialize_vault(
    ctx: Context<InitializeVault>,
) -> Result<()>
```

- Create PDA-based vault for user
- Create associated token account for USDT
- Set user as authority
- Initialize balance tracking
- Make account rent-exempt

2. **Deposit Collateral**

```rust
pub fn deposit(
    ctx: Context<Deposit>,
    amount: u64,
) -> Result<()>
```

- Transfer USDT from user wallet to vault
- Use SPL Token transfer instruction
- Update vault balance record
- Emit deposit event
- Validate minimum deposit (if applicable)

Detailed Implementation with SPL Token CPI:

```
use anchor_spl::token::{self, Transfer};
use anchor_lang::prelude::*;

pub fn deposit(
    ctx: Context<Deposit>,
    amount: u64,
) -> Result<()> {
    require!(amount > 0, ErrorCode::InvalidAmount);

    // Transfer USDT from user to vault using Cross-Program Invocation (CPI)
    token::transfer(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            Transfer {
                from: ctx.accounts.user_token_account.to_account_info(),
                to: ctx.accounts.vault_token_account.to_account_info(),
                authority: ctx.accounts.user.to_account_info(),
            }
        ),
        amount
    )?;

    // Update vault state
    let vault = &mut ctx.accounts.vault;
    vault.total_balance += amount;
    vault.available_balance += amount;
    vault.total_deposited += amount;

    // Emit event for off-chain indexing
    emit!(DepositEvent {
        user: ctx.accounts.user.key(),
        amount,
        new_balance: vault.total_balance,
        timestamp: Clock::get()?.unix_timestamp,
    });

    Ok(())
}

#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(mut)]
    pub user: Signer<'info>,

    #[account(
        mut,
        seeds = [b"vault", user.key().as_ref()],
        bump = vault.bump,
    )]
    pub vault: Account<'info, CollateralVault>,

    #[account(mut)]
    pub user_token_account: Account<'info, TokenAccount>,

    #[account(mut)]
    pub vault_token_account: Account<'info, TokenAccount>,

    pub token_program: Program<'info, Token>,
}
```

3. **Withdraw Collateral**

```
pub fn withdraw(
    ctx: Context<Withdraw>,
    amount: u64,
) -> Result<()>
```

- Verify user has no open positions
- Check available (unlocked) balance
- Transfer USDT from vault to user wallet
- Use CPI to SPL Token program
- Update balance record
- Emit withdrawal event

4. **Lock Collateral**

```
pub fn lock_collateral(
    ctx: Context<LockCollateral>,
    amount: u64,
) -> Result<()>
```

- Called by position management program (CPI)
- Lock collateral for margin requirement
- Update locked_balance
- Verify sufficient available balance
- Prevent withdrawal of locked funds

5. **Unlock Collateral**

```
pub fn unlock_collateral(
    ctx: Context<UnlockCollateral>,
    amount: u64,
) -> Result<()>
```

- Called when position is closed
- Release locked collateral
- Make funds available for withdrawal
- Update balance tracking

6. **Transfer Collateral (Internal)**

```
pub fn transfer_collateral(
    ctx: Context<TransferCollateral>,
    from_vault: Pubkey,
    to_vault: Pubkey,
    amount: u64,
) -> Result<()>
```

- Transfer between vaults (for settlements/liquidations)
- Only callable by authorized programs
- Atomic balance updates
- Emit transfer event

Account Structures:

```
#[account]
pub struct CollateralVault {
    pub owner: Pubkey,
    pub token_account: Pubkey,
    pub total_balance: u64,
    pub locked_balance: u64,
    pub available_balance: u64,  // total - locked
    pub total_deposited: u64,
    pub total_withdrawn: u64,
    pub created_at: i64,
    pub bump: u8,
}

#[account]
pub struct VaultAuthority {
    pub authorized_programs: Vec<Pubkey>,  // Programs that can lock/unlock
    pub bump: u8,
}

#[derive(AnchorSerialize, AnchorDeserialize)]
pub struct TransactionRecord {
    pub vault: Pubkey,
    pub transaction_type: TransactionType,
    pub amount: u64,
    pub timestamp: i64,
}

#[derive(AnchorSerialize, AnchorDeserialize)]
pub enum TransactionType {
    Deposit,
    Withdrawal,
    Lock,
    Unlock,
    Transfer,
}
```

Security Requirements:

- Only vault owner can withdraw
- Only authorized programs can lock/unlock
- Validate sufficient balance before operations
- Prevent integer overflow/underflow
- Ensure atomic state updates

Part 2: Rust Backend - Vault Management Service

Core Components:

1. **Vault Manager**

   ```
   pub struct VaultManager {
       // Manage vault lifecycle
       // Handle deposits and withdrawals
       // Query vault state
   }
   ```

   - Initialize vaults for new users
   - Process deposit requests
   - Handle withdrawal requests
   - Query vault balances
   - Track transaction history

2. **Balance Tracker**

   - Monitor vault balances in real-time
   - Calculate available balance
   - Alert on low balances
   - Reconcile on-chain vs off-chain state
   - Detect discrepancies

3. **Transaction Builder**
    - Build deposit transactions
    - Build withdrawal transactions
    - Handle SPL Token accounts
    - Set transaction fees appropriately
    - Include compute budget instructions
4. **Cross-Program Integration**

    ```
    pub struct CPIManager {
        // Handle CPIs to vault program
        // Called by position manager
        // Lock/unlock collateral
    }
    ```

    - Interface for position management to lock collateral
    - Safe CPI invocations
    - Handle CPI errors gracefully
    - Maintain consistency across programs
5. **Vault Monitor**
    - Continuously monitor all vaults
    - Detect unauthorized access attempts
    - Alert on unusual activity
    - Track total value locked (TVL)
    - Generate analytics

Vault Lifecycle:

```
Initialize → Deposit → [Lock ↔ Unlock] → Withdraw
                    ↓
          Open Position (CPI)
```

Part 3: Database Schema

Design schema for:

- Vault accounts (owner, balances, status)
- Transaction history (deposits, withdrawals, locks)
- Balance snapshots (hourly/daily)
- Reconciliation logs
- Audit trail

Part 4: Integration & APIs

1. **REST API Endpoints**

    ```
    POST   /vault/initialize         - Create new vault
    POST   /vault/deposit            - Deposit collateral
    POST   /vault/withdraw           - Withdraw collateral
    GET    /vault/balance/:user      - Get vault balance
    GET    /vault/transactions/:user - Transaction history
    GET    /vault/tvl                - Total value locked
    ```

2. **WebSocket Streams**
    - Real-time balance updates
    - Deposit/withdrawal notifications

- Lock/unlock events
- TVL updates

3. **Internal Interfaces**

   - Position manager (lock/unlock calls)
   - Liquidation engine (transfer collateral)
   - Settlement relayer (settle trades)

Technical Requirements

1. **Security**

   - Secure PDA derivation
   - Proper authority checks
   - No fund loss scenarios
   - Prevent unauthorized access
   - Atomic state updates

2. **Performance**

   - Support 10,000+ vaults
   - Deposit/withdrawal < 2 seconds
   - Balance queries < 50ms
   - Handle 100+ operations per second

3. **Reliability**

   - Consistent state between on-chain and off-chain
   - Handle transaction failures gracefully
   - Automatic retry for failed operations
   - Balance reconciliation mechanisms

4. **Testing**

   - Unit tests for all vault operations
   - Integration tests for SPL Token transfers
   - CPI tests with mock programs
   - Anchor program tests
   - Security tests (unauthorized access attempts)

5. **Code Quality**

   - Safe handling of token operations
   - Clear error messages
   - Comprehensive logging
   - Well-documented CPIs

Bonus Section (Recommended)

1. **Advanced Features**

   - Multi-signature vaults
   - Vault delegation (authorized users)
   - Partial withdrawals with time locks
   - Emergency withdrawal mechanism

2. **Yield Integration**

   - Integrate with Solana lending protocols
   - Generate yield on idle collateral
   - Auto-compound strategies

- Risk-adjusted yield optimization

3. **Security Enhancements**
   - Withdraw delay for security
   - Withdrawal whitelist
   - Rate limiting on withdrawals
   - Multi-factor authentication integration
4. **Analytics & Reporting**
   - TVL tracking and charts
   - Deposit/withdrawal flows
   - User balance distributions
   - Collateral utilization metrics
5. **Optimization**
   - Batch deposit/withdrawal processing
   - Compressed account storage
   - Efficient PDA derivation
   - Transaction fee optimization

Documentation Requirements

Provide detailed documentation covering:

1. **System Architecture**
   - Vault structure diagram
   - PDA derivation scheme
   - CPI flow diagram
   - Security model
2. **Smart Contract Documentation**
   - Account structures
   - Instruction specifications
   - PDA seeds and bumps
   - Authority validation
3. **SPL Token Integration**
   - Token transfer flows
   - Associated token accounts
   - CPI to Token program
   - Error handling
4. **Backend Service Documentation**
   - Module architecture
   - API specifications
   - Transaction building
   - Database schema
5. **Security Analysis**
   - Threat model
   - Access control mechanisms
   - Attack surface analysis
   - Best practices

Deliverables

1. **Complete Source Code**
   - Anchor program (vault management)
   - Rust backend service
   - Database migrations
   - Test suite
   - Configuration files
2. **Video Demonstration** (10-15 minutes)
   - System architecture
   - Live demo (deposit, withdrawal, lock/unlock)
   - Code walkthrough
   - CPI explanation
   - Security measures
3. **Technical Documentation**
   - Architecture documentation
   - SPL Token integration guide
   - API documentation
   - Deployment guide
   - Security analysis
4. **Test Results**
   - Test coverage report
   - Security test results
   - Performance metrics
   - CPI test logs

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.