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 **Program Upgrade & Migration System** for a decentralized perpetual futures exchange that enables safe, controlled upgrades of Solana programs with state migration capabilities. This system ensures the protocol can evolve without disrupting user funds or operations, implementing governance, timelock, and verification mechanisms.

System Context

In a live perpetual futures DEX managing significant user funds:

- Smart contracts need periodic upgrades for features and security fixes
- Upgrades must maintain user funds and position data integrity
- Governance through multisig prevents unilateral changes
- Timelock delays (48 hours) give users time to exit if needed
- State migration transfers data from old to new program versions
- Rollback capability handles failed upgrades
- Verification ensures new program meets security standards
- Must support zero-downtime upgrades when possible
- Audit trail for all upgrade proposals and executions

Your component enables protocol evolution while maintaining the highest security standards for a system holding user funds.

Initial Setup

1. Set up a Rust development environment with:
    - Rust 1.75+ with async/await support
    - Anchor framework 0.29+
    - Solana CLI tools
    - Squads Protocol (Solana multisig)
    - PostgreSQL for upgrade history
2. Familiarize yourself with:
    - Solana program upgrades (BPF upgradeable loader)
    - Program buffer accounts
    - Multisig transaction patterns
    - Account data migration strategies
    - Anchor version management

Core Requirements

Part 1: Solana Smart Contract (Anchor Program)

Upgrade Management Program:

1. **Propose Upgrade**

```
pub fn propose_upgrade(
    ctx: Context<ProposeUpgrade>,
    new_program_buffer: Pubkey,
    description: String,
) -> Result<()>
```

- Create upgrade proposal
- Link to new program buffer account
- Set timelock period (48 hours)
- Requires multisig approval
- Store proposal metadata
- Emit proposal event

2. **Approve Upgrade**

```
pub fn approve_upgrade(
    ctx: Context<ApproveUpgrade>,
    proposal_id: Pubkey,
) -> Result<()>
```

- Multisig member approves proposal
- Track approval count
- Check threshold (e.g., 3 of 5)
- Once threshold met, activate timelock
- Cannot execute until timelock expires

3. **Execute Upgrade**

```
pub fn execute_upgrade(
    ctx: Context<ExecuteUpgrade>,
    proposal_id: Pubkey,
) -> Result<()>
```

- Verify timelock period elapsed
- Verify sufficient approvals
- Upgrade program using BPF loader
- Mark proposal as executed
- Emit upgrade event

4. **Cancel Upgrade**

```
pub fn cancel_upgrade(
    ctx: Context<CancelUpgrade>,
    proposal_id: Pubkey,
) -> Result<()>
```

- Emergency cancellation by multisig
- Only before execution
- Refund buffer account rent
- Mark proposal as cancelled

5. **Migrate Account State**

```
pub fn migrate_account(
    ctx: Context<MigrateAccount>,
    old_account: Pubkey,
) -> Result<()>
```

- Read data from old account

- Transform to new format
- Write to new account
- Mark migration complete
- Handle version compatibility

Account Structures:

```rust
#[account]
pub struct UpgradeProposal {
    pub id: u64,
    pub proposer: Pubkey,
    pub program: Pubkey,
    pub new_buffer: Pubkey,
    pub description: String,
    pub proposed_at: i64,
    pub timelock_until: i64,
    pub approvals: Vec<Pubkey>,
    pub approval_threshold: u8,
    pub status: UpgradeStatus,
    pub executed_at: Option<i64>,
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone, PartialEq)]
pub enum UpgradeStatus {
    Proposed,
    Approved,
    TimelockActive,
    Executed,
    Cancelled,
}

#[account]
pub struct MultisigConfig {
    pub members: Vec<Pubkey>,
    pub threshold: u8,
    pub upgrade_authority: Pubkey,
}

#[account]
pub struct ProgramUpgradeState {
    pub authority: Pubkey,              // Multisig only
    pub upgrade_buffer: Pubkey,         // Staged upgrade
    pub timelock_duration: i64,         // 48 hours minimum
    pub pending_upgrade: Option<PendingUpgrade>,
}

pub struct PendingUpgrade {
    pub new_program_hash: [u8; 32],
    pub scheduled_time: i64,
    pub proposal_time: i64,
    pub approved_by: Vec<Pubkey>,       // Multisig approvals
}

#[account]
pub struct AccountVersion {
    pub version: u32,
    pub migrated: bool,
    pub migrated_at: Option<i64>,
}
```

Part 2: Rust Backend - Upgrade Management Service

Core Components:

1. **Proposal Manager**

   Comprehensive upgrade proposal management system:

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

pub struct ProgramUpgrade {
    multisig: Arc<MultisigManager>,
    notification_service: Arc<NotificationService>,
    program_client: Arc<ProgramClient>,
}

impl ProgramUpgrade {
    pub fn new(
        multisig: Arc<MultisigManager>,
        notification_service: Arc<NotificationService>,
        program_client: Arc<ProgramClient>,
    ) -> Self {
        Self {
            multisig,
            notification_service,
            program_client,
        }
    }

    pub async fn propose_upgrade(&self, new_program_buffer: Pubkey) -> Result<String,
UpgradeError> {
        // 1. Multisig proposes upgrade (requires 3/5 approval)
        let proposal_id = self.multisig.propose_transaction(ProposalParams {
            instruction: self.build_upgrade_instruction(&new_program_buffer)?,
            description: "Upgrade to v2.0.0".to_string(),
            timelock: 48 * 60 * 60, // 48 hours
        }).await?;

        // 2. Notify community via multiple channels
        self.notification_service.notify_community(Notification {
            notification_type: NotificationType::ProgramUpgrade,
            proposal_id: proposal_id.clone(),
            changes: "See: github.com/dex/contracts/releases/v2.0.0".to_string(),
        }).await?;

        Ok(proposal_id)
    }

    pub async fn execute_upgrade(&self, proposal_id: &str) -> Result<(), UpgradeError> {
        // Wait for timelock to expire
        self.wait_for_timelock(proposal_id).await?;

        // Execute with multisig approval
        self.multisig.execute_transaction(proposal_id).await?;

        // Verify new program is functioning correctly
        self.verify_upgrade().await?;

        // Announce completion
        self.announce_upgrade().await?;

        Ok(())
    }

    async fn wait_for_timelock(&self, proposal_id: &str) -> Result<(), UpgradeError> {
        let proposal = self.multisig.get_proposal(proposal_id).await?;
        let now = chrono::Utc::now().timestamp();

        if now < proposal.timelock_end {
            let remaining = proposal.timelock_end - now;
            return Err(UpgradeError::TimelockActive {
                remaining_seconds: remaining
            });
        }

        Ok(())
    }
}

// Rollback functionality for failed upgrades
pub async fn rollback_program(
    old_program_id: Pubkey,
    system: &SystemManager
) -> Result<(), RollbackError> {
    // 1. Pause new operations
    system.pause_system().await?;

    // 2. Close all positions at current mark price
    system.emergency_close_all_positions().await?;
```

```
    // 3. Return funds to users
    system.return_all_funds().await?;

    // 4. Deploy old program version
    system.deploy_program(&old_program_id).await?;

    // 5. Resume operations
    system.resume_system().await?;

    Ok(())
}
```

2. **Multisig Coordinator**

```
pub struct MultisigCoordinator {
    // Coordinate with Squads Protocol
    // Collect signatures
    // Execute multisig transactions
}
```

   - Integration with Squads multisig
   - Request approvals from members
   - Track approval status
   - Execute once threshold met
   - Handle rejection scenarios

3. **Timelock Manager**

   - Monitor active timelocks
   - Alert when timelock expires
   - Prevent early execution
   - Countdown notifications
   - Cancellation window tracking

4. **Program Builder**

```
pub struct ProgramBuilder {
    // Build Solana programs
    // Generate program buffers
    // Verify build artifacts
}
```

   - Compile Anchor programs
   - Create program buffer accounts
   - Upload program binary to buffer
   - Verify buffer integrity
   - Calculate program hash for verification

5. **State Migration Manager**

```
pub struct MigrationManager {
    // Handle account migrations
    // Track migration progress
    // Verify data integrity
}
```

   - Identify accounts needing migration
   - Build migration transactions
   - Batch migrate accounts
   - Verify migrated data
   - Handle migration failures
   - Track completion percentage

6. **Rollback Handler**
    - Detect upgrade failures
    - Revert to previous program version
    - Restore account states if needed
    - Emergency pause mechanism
    - Post-mortem analysis

Upgrade Flow:

```
Build Program → Upload Buffer → Propose Upgrade → Collect Approvals →
Timelock (48h) → Execute Upgrade → Migrate Accounts → Verify Success
                        ↓ (if needed)
                   Cancel/Rollback
```

Part 3: Database Schema

Design schema for:

- Upgrade proposals (all proposals with status)
- Approval history (who approved what and when)
- Timelock tracking
- Migration progress (accounts migrated)
- Upgrade history (successful/failed)
- Rollback events

Part 4: Integration & APIs

1. **REST API Endpoints**

```
POST   /upgrade/propose          - Create upgrade proposal
POST   /upgrade/:id/approve      - Approve proposal
POST   /upgrade/:id/execute      - Execute upgrade
POST   /upgrade/:id/cancel       - Cancel proposal
GET    /upgrade/proposals        - List all proposals
GET    /upgrade/:id/status       - Proposal status
POST   /migration/start          - Start account migration
GET    /migration/progress       - Migration status
```

2. **WebSocket Notifications**
    - Proposal created alerts
    - Approval received notifications
    - Timelock countdowns
    - Upgrade execution events
    - Migration progress updates

3. **Internal Interfaces**
    - Integration with all protocol programs
    - Notification system for community
    - Alert system for critical events

Technical Requirements

1. **Security**
    - Secure multisig implementation
    - Proper timelock enforcement
    - Authority validation at each step

- Prevent replay attacks
- Verify program hash before upgrade

2. **Governance**
   - Transparent proposal process
   - Adequate timelock period
   - Clear approval tracking
   - Community notification system
   - Emergency pause capability
3. **Reliability**
   - Handle partial migration failures
   - Atomic upgrade execution
   - Rollback on failure
   - Data integrity verification
   - No fund loss scenarios
4. **Testing**
   - Unit tests for all upgrade logic
   - Integration tests with multisig
   - Migration tests with mock data
   - Rollback scenario testing
   - Timelock enforcement tests
5. **Code Quality**
   - Clear state machine for proposals
   - Robust error handling
   - Comprehensive audit logging
   - Well-documented migration logic

Bonus Section (Recommended)

1. **Advanced Governance**
   - Token-weighted voting (DARK token holders)
   - Proposal discussion period
   - Veto mechanism
   - Governance parameter updates
   - Delegation of voting power
2. **Migration Optimization**
   - Parallel account migration
   - Incremental migration (batches)
   - Zero-downtime migration strategy
   - Data compression for efficiency
   - Lazy migration (migrate on access)
3. **Verification & Security**
   - Automated security scans of new code
   - Formal verification integration
   - Audit report requirements
   - Bug bounty program integration
   - Emergency response procedures
4. **Monitoring & Analytics**
   - Upgrade success metrics

- Migration progress dashboard
- Proposal voting statistics
- Historical upgrade timeline
- Gas cost analysis

5. **Developer Tools**
   - Migration testing framework
   - Upgrade simulation environment
   - State comparison tools
   - Rollback playbooks
   - Documentation generator

Documentation Requirements

Provide detailed documentation covering:

1. **System Architecture**
   - Upgrade process flow diagram
   - Multisig integration
   - Timelock mechanism
   - Migration strategy
2. **Governance Model**
   - Proposal requirements
   - Approval process
   - Timelock rationale
   - Cancellation procedures
3. **Smart Contract Documentation**
   - Account structures
   - Instruction specifications
   - Security measures
   - Upgrade authority management
4. **Migration Guide**
   - How to plan migrations
   - Account versioning strategy
   - Data transformation patterns
   - Testing migrations
5. **Operational Procedures**
   - How to propose upgrade
   - Approval workflow
   - Emergency procedures
   - Rollback process
   - Post-upgrade verification

Deliverables

1. **Complete Source Code**
   - Anchor program (upgrade management)
   - Rust backend service
   - Migration scripts
   - Test suite
   - Configuration files

2. **Video Demonstration** (10-15 minutes)
    - System architecture
    - Live upgrade demo (testnet)
    - Multisig approval process
    - Migration demonstration
    - Rollback scenario
3. **Technical Documentation**
    - Architecture documentation
    - Governance documentation
    - Migration guide
    - API documentation
    - Operational runbook
4. **Test Results**
    - Test coverage report
    - Migration test results
    - Security test results
    - Rollback verification

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.