GoDark DEX Technical Architecture

Version: 1.0
Date: October 27, 2025
Status: Draft

Table of Contents

1. Executive Summary

Overview

GoDark is a decentralized exchange (DEX) for perpetual futures trading that operates on the Solana blockchain. It combines the transparency and non-custodial nature of blockchain technology with the privacy and execution quality of traditional dark pools. The platform is specifically designed to minimize market impact for trades of all sizes while maintaining absolute privacy for order flow.

Key Differentiators

Dark Pool Mechanics

- Hidden order book prevents information leakage
- Orders invisible until execution
- Reduces front-running and sandwich attacks
- Institutional-grade privacy for all participants

High Leverage Trading

- Up to 1,000x leverage on perpetual futures
- Isolated margin model for risk containment
- Real-time liquidation engine
- Insurance fund for bad debt coverage

Hybrid Architecture

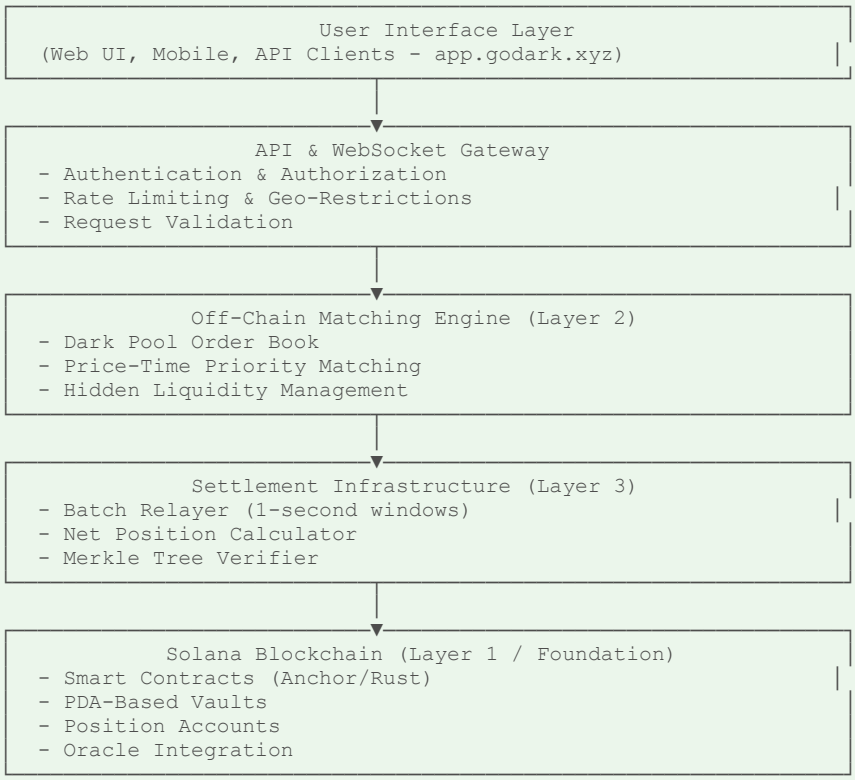
- Off-chain matching for millisecond latency
- On-chain settlement for finality and transparency
- Best of both centralized (speed) and decentralized (security) worlds

Institutional-Grade Execution

- 100+ trades/second throughput
- Sub-second settlement latency
- Advanced order types (Market, Limit, Peg)

- Comprehensive order attributes (AON, MinQty, NBBO Protection)

System Architecture Overview



Target Market

- Professional traders seeking privacy
- Market makers requiring minimal market impact
- Institutions executing large block trades
- Retail traders wanting access to dark pool liquidity
- DeFi-native users comfortable with high leverage

---

2. System Architecture Layers

Layer 1: Blockchain Foundation (Solana L1)

GoDark is built as an application-layer protocol on Solana. While referred to as "Layer 2" in the product specification, this terminology indicates that GoDark operates as a decentralized application on top of Solana's Layer 1 blockchain, rather than being a true L2 scaling solution.

Smart Contract Architecture

The core smart contracts are developed using the Anchor framework in Rust, providing type safety and reducing boilerplate code. The program architecture consists of modular components:

Core Program Modules:

- Market Manager: Creates and manages perpetual markets
- Position Manager: Handles user positions and PnL
- Vault Manager: Controls USDT deposits and withdrawals
- Settlement Processor: Executes batch settlements
- Liquidation Engine: Monitors and executes liquidations

- Funding Rate Calculator: Computes and applies funding rates

Account Structures

Market Account (PDA)

```
pub struct PerpMarket {
    pub authority: Pubkey,              // Program authority
    pub market_id: Pubkey,              // Unique market identifier
    pub symbol: [u8; 32],               // "BTC-USDT-PERP"
    pub base_asset: [u8; 16],           // "BTC"
    pub quote_asset: [u8; 16],          // "USDT"

    // Vault and oracle
    pub usdt_vault: Pubkey,             // USDT token account (PDA)
    pub price_oracle: Pubkey,           // Pyth/Switchboard feed

    // Market parameters
    pub max_leverage: u16,              // 1000 (represents 1000x)
    pub maintenance_margin_ratio: u16,  // In basis points (e.g., 50 = 0.5%)
    pub initial_margin_ratio: u16,      // In basis points
    pub maker_fee: i16,                 // In basis points (can be negative for rebates)
    pub taker_fee: u16,                 // In basis points

    // Market state
    pub funding_rate: i64,              // Current funding rate (scaled by 1e9)
    pub last_funding_update: i64,       // Unix timestamp
    pub total_open_interest: u64,       // Notional value in USDT
    pub total_long_interest: u64,       // Long positions value
    pub total_short_interest: u64,      // Short positions value

    // Insurance and fees
    pub insurance_fund: Pubkey,         // Insurance fund PDA
    pub fee_recipient: Pubkey,          // GoDark fee collection wallet

    // Status
    pub is_active: bool,
    pub bump: u8,                       // PDA bump seed
}
```

User Position Account (PDA)

```
pub struct UserPosition {
    pub owner: Pubkey,                  // User's wallet (can be ephemeral)
    pub parent_wallet: Pubkey,          // Main wallet (if using ephemeral)
    pub market: Pubkey,                 // Reference to PerpMarket

    // Position details
    pub size: i64,                      // Signed: positive=long, negative=short
    pub entry_price: u64,               // Average entry price (scaled by 1e6)
    pub collateral: u64,                // USDT collateral amount
    pub leverage: u16,                  // Actual leverage used

    // Risk metrics
    pub liquidation_price: u64,         // Price at which position liquidates
    pub maintenance_margin: u64,        // Current maintenance requirement

    // PnL tracking
    pub realized_pnl: i64,              // Cumulative realized PnL
    pub unrealized_pnl: i64,            // Current unrealized PnL

    // Funding tracking
    pub funding_index: i64,             // Last funding index applied
    pub accumulated_funding: i64,       // Total funding paid/received
    pub last_funding_update: i64,       // Last funding timestamp

    // Timestamps
    pub open_timestamp: i64,
    pub last_update_timestamp: i64,

    pub bump: u8,
}
```

Ephemeral Vault Account (PDA)

```
pub struct EphemeralVault {
    pub user_wallet: Pubkey,            // Main user wallet
    pub vault_pda: Pubkey,              // PDA holding USDT
    pub created_at: i64,
    pub last_activity: i64,

    // Delegate approval
    pub approved_amount: u64,           // Max USDT delegated
    pub used_amount: u64,               // Amount currently in use
    pub available_amount: u64,          // Free to withdraw

    // Status
    pub is_active: bool,
    pub bump: u8,
}
```

Settlement Batch Account (PDA)

```
pub struct SettlementBatch {
    pub batch_id: [u8; 32],             // Unique batch identifier
    pub relayer: Pubkey,                // Relayer that submitted
    pub timestamp: i64,
    pub trade_count: u16,
    pub merkle_root: [u8; 32],          // Root hash of trades
    pub status: SettlementStatus,        // Pending/Confirmed/Failed
    pub bump: u8,
}

pub enum SettlementStatus {
    Pending,
    Confirmed,
    Failed,
}
```

Token Program Integration

GoDark uses USDT (SPL Token) as the sole quote asset. All collateral deposits, settlements, and fee payments are denominated in USDT.

USDT Flow:

1. User approves GoDark program as delegate for their USDT token account
2. Program transfers approved USDT to program-controlled vault (PDA)
3. Settlements move USDT between position collateral allocations
4. Withdrawals return USDT from vault to user's token account
5. Fees transferred from vault to GoDark fee wallet

Layer 2: Off-Chain Matching Engine

The matching engine operates off-chain to provide millisecond-latency order execution while maintaining dark pool privacy characteristics.

Dark Pool Order Book Mechanics

Key Characteristics:

- Orders are completely hidden from public view
- No pre-trade transparency (no visible order book depth)
- Post-trade transparency only (executed trades published)
- Price discovery occurs through matching, not display
- Prevents information leakage and front-running

Order Storage:

```rust
use serde::{Deserialize, Serialize};
use solana_sdk::pubkey::Pubkey;

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct DarkPoolOrder {
    pub order_id: String,               // UUID
    pub user_id: String,                // Account identifier
    pub wallet_address: Pubkey,         // Solana wallet pubkey
    pub is_ephemeral: bool,             // Using ephemeral wallet?

    // Instrument
    pub symbol: String,                 // "BTC-USDT-PERP"

    // Order details
    pub side: OrderSide,
    pub order_type: OrderType,
    pub size: f64,                      // Base asset quantity
    pub limit_price: Option<f64>,       // For LIMIT orders

    // Time in Force
    pub time_in_force: TimeInForce,
    pub expiry_time: Option<i64>,       // For GTD

    // Order Attributes
    pub all_or_none: bool,              // Must fill entirely or not at all
    pub min_quantity: Option<f64>,      // Minimum fill quantity
    pub nbbo_protection: bool,          // Reject if worse than NBBO

    // Metadata
    pub timestamp: i64,                 // Order submission time
    pub status: OrderStatus,
    pub filled_size: f64,
    pub avg_fill_price: f64,

    // Internal
    pub priority: i64,                  // Price-time priority score
}

#[derive(Debug, Clone, Serialize, Deserialize, PartialEq)]
pub enum OrderSide {
    Buy,
    Sell,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum OrderType {
    Market,
    Limit,
    PegMid,
    PegBid,
    PegAsk,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum TimeInForce {
    IOC,  // Immediate or Cancel
    FOK,  // Fill or Kill
    GTD,  // Good Till Date
    GTC,  // Good Till Cancel
}

#[derive(Debug, Clone, Serialize, Deserialize, PartialEq)]
pub enum OrderStatus {
    Pending,
    PartiallyFilled,
    Filled,
    Cancelled,
    Rejected,
    Expired,
}
```

Price-Time Priority Matching Algorithm

The matching engine uses strict price-time priority to ensure fair execution:

Priority Calculation:

```
fn calculate_priority(order: &DarkPoolOrder) -> i64 {
    // Earlier timestamp = higher priority (lower number)
    // Price is secondary for dark pools (no displayed prices)
    order.timestamp
}
```

Matching Logic:

```rust
use std::collections::HashMap;

pub struct DarkPoolMatcher {
    buy_orders: HashMap<String, Vec<DarkPoolOrder>>,   // By symbol
    sell_orders: HashMap<String, Vec<DarkPoolOrder>>,  // By symbol
}

impl DarkPoolMatcher {
    pub async fn match_order(&mut self, new_order: DarkPoolOrder) -> Result<Vec<Trade>, MatchError>
{
        let mut trades = Vec::new();
        let opposite_orders = self.get_opposite_orders(&new_order.symbol, &new_order.side);

        // Sort by priority (time)
        let mut opposite_orders = opposite_orders.clone();
        opposite_orders.sort_by_key(|order| order.priority);

        let mut new_order = new_order;

        for resting_order in opposite_orders.iter_mut() {
            // Check if orders can match
            if !self.can_match(&new_order, resting_order) {
                continue;
            }

            // Determine execution price (midpoint for dark pool)
            let execution_price = self.determine_execution_price(&new_order, resting_order)?;

            // Check NBBO protection
            if new_order.nbbo_protection && !self.meets_nbbo(&new_order, execution_price)? {
                new_order.status = OrderStatus::Rejected;
                break;
            }

            // Determine fill quantity
            let fill_qty = f64::min(
                new_order.size - new_order.filled_size,
                resting_order.size - resting_order.filled_size
            );

            // Check Min Quantity constraint
            if let Some(min_qty) = new_order.min_quantity {
                if fill_qty < min_qty {
                    continue; // Skip this match
                }
            }

            // Execute trade
            let trade = self.execute_trade(&mut new_order, resting_order, fill_qty,
execution_price)?;
            trades.push(trade);

            // Update fill statuses
            new_order.filled_size += fill_qty;
            resting_order.filled_size += fill_qty;
            new_order.avg_fill_price = self.calculate_avg_price(&new_order, &trades);
            resting_order.avg_fill_price = self.calculate_avg_price(resting_order, &trades);

            if (new_order.filled_size - new_order.size).abs() < f64::EPSILON {
                new_order.status = OrderStatus::Filled;
                break;
            }

            if (resting_order.filled_size - resting_order.size).abs() < f64::EPSILON {
                resting_order.status = OrderStatus::Filled;
            }
        }

        // Handle Time in Force
        self.apply_time_in_force(&mut new_order);

        Ok(trades)
    }

    fn determine_execution_price(&self, order1: &DarkPoolOrder, order2: &DarkPoolOrder) ->
Result<f64, MatchError> {
        // Dark pools typically use midpoint pricing
        // Or reference price from oracle
        let oracle_price = self.get_oracle_price(&order1.symbol)?;

        let price = match (&order1.order_type, &order2.order_type) {
            (OrderType::Market, OrderType::Market) => {
                // Both market orders = oracle price
```

```
                oracle_price
            },
            (OrderType::Limit, OrderType::Limit) => {
                // Midpoint of two limit prices
                let price1 = order1.limit_price.ok_or(MatchError::MissingLimitPrice)?;
                let price2 = order2.limit_price.ok_or(MatchError::MissingLimitPrice)?;
                (price1 + price2) / 2.0
            },
            _ => {
                // One market, one limit = limit price
                if matches!(order1.order_type, OrderType::Limit) {
                    order1.limit_price.ok_or(MatchError::MissingLimitPrice)?
                } else {
                    order2.limit_price.ok_or(MatchError::MissingLimitPrice)?
                }
            }
        };

        Ok(price)
    }
}
```

Peg Orders

Peg orders dynamically adjust their price based on market conditions:

```
pub struct PegOrderManager {
    orders: HashMap<String, Vec<DarkPoolOrder>>,
}

impl PegOrderManager {
    pub async fn update_peg_orders(&mut self, symbol: &str) -> Result<(), OrderError> {
        let peg_orders = self.get_peg_orders(symbol);
        let market_data = self.get_market_data(symbol).await?;

        for order in peg_orders.iter_mut() {
            let new_price = match order.order_type {
                OrderType::PegMid => market_data.mid_price,
                OrderType::PegBid => market_data.best_bid,
                OrderType::PegAsk => market_data.best_ask,
                _ => continue,
            };

            order.limit_price = Some(new_price);

            // Update priority based on new price
            order.priority = self.calculate_priority(order);
        }

        Ok(())
    }
}
```

Order Type Handling

Market Orders:

- Execute immediately at best available price
- Use oracle price as reference
- Always IOC or FOK

Limit Orders:

- Execute only at specified price or better
- Can be GTC (remain until filled)
- Provide price protection

Peg Orders:

- Continuously adjust to market
- Updated every 100ms
- Provide dynamic liquidity

Layer 3: Settlement Infrastructure

The settlement layer bridges the off-chain matching engine with on-chain finality through batched transactions.

Per-Trade Settlement Model

Each matched trade results in a settlement transaction that updates user positions on-chain. Trades are batched into 1-second windows for efficiency.

Settlement Flow:

```
Off-Chain Match → Settlement Queue → Batch Creation → On-Chain Execution → Confirmation
```

Batch Relayer Architecture

Relayer Service:

```rust
use tokio::time::{interval, Duration};
use std::sync::Arc;
use tokio::sync::Mutex;

pub struct SettlementRelayer {
    pending_trades: Arc<Mutex<Vec<Trade>>>,
    batch_window_ms: u64,          // 1 second
    max_batch_size: usize,         // Trades per batch
}

impl SettlementRelayer {
    pub fn new() -> Self {
        Self {
            pending_trades: Arc::new(Mutex::new(Vec::new())),
            batch_window_ms: 1000,
            max_batch_size: 50,
        }
    }

    pub async fn start(&self) {
        self.start_batch_timer().await;
    }

    pub async fn add_trade(&self, trade: Trade) -> Result<(), SettlementError> {
        let mut pending = self.pending_trades.lock().await;
        pending.push(trade);

        // Trigger immediate settlement if batch is full
        if pending.len() >= self.max_batch_size {
            drop(pending); // Release lock before settling
            self.settle_batch().await?;
        }

        Ok(())
    }

    async fn start_batch_timer(&self) {
        let mut timer = interval(Duration::from_millis(self.batch_window_ms));
        let pending_trades = Arc::clone(&self.pending_trades);

        tokio::spawn(async move {
            loop {
                timer.tick().await;

                let has_pending = {
                    let pending = pending_trades.lock().await;
                    !pending.is_empty()
                };

                if has_pending {
                    if let Err(e) = self.settle_batch().await {
                        eprintln!("Batch settlement error: {:?}", e);
                    }
                }
            }
        });
    }

    async fn settle_batch(&self) -> Result<(), SettlementError> {
        // Extract and clear pending trades
        let batch = {
            let mut pending = self.pending_trades.lock().await;
            let trades = pending.drain(..).collect::<Vec<_>>();
            trades
        };

        if batch.is_empty() {
            return Ok(());
        }

        let batch_id = self.generate_batch_id();

        match self.process_settlement(&batch, &batch_id).await {
            Ok(signature) => {
                // Update trade statuses
                self.update_trade_statuses(&batch, &signature).await?;
                Ok(())
            }
            Err(error) => {
                // Re-queue trades for retry
                let mut pending = self.pending_trades.lock().await;
                for trade in batch.iter().rev() {
                    pending.insert(0, trade.clone());
```

```
                }

                self.handle_settlement_error(&batch_id, error).await?;
                Err(SettlementError::BatchFailed)
            }
        }
    }

    async fn process_settlement(&self, batch: &[Trade], batch_id: &str) -> Result<String,
SettlementError> {
        // 1. Calculate net positions
        let net_positions = self.calculate_net_positions(batch)?;

        // 2. Build Merkle tree for verification
        let merkle_tree = self.build_merkle_tree(batch)?;

        // 3. Create settlement transaction
        let tx = self.build_settlement_transaction(
            batch_id,
            &net_positions,
            &merkle_tree.root
        ).await?;

        // 4. Sign and send
        let signature = self.send_transaction(tx).await?;

        // 5. Wait for confirmation
        self.confirm_transaction(&signature).await?;

        Ok(signature)
    }
}
```

3. Core Components

A. Instrument Support

GoDark supports a comprehensive range of perpetual futures contracts across crypto assets and FX pairs.

Symbol Format

All instruments follow the standard format: **Base-Quote-Type**

Examples:

- `BTC-USDT-PERP`
- `ETH-USDT-PERP`
- `USD-USDT-PERP` (FX pair)

Supported Assets

Quote Asset:

- USDT (only)

Base Assets - Crypto (Top 50):

Symbol	Name	Symbol	Name
BTC	Bitcoin	LINK	Chainlink
ETH	Ethereum	ESDe	Ethena Staked
BNB	BNB	XLM	Stellar
XRP	Ripple	BCH	Bitcoin Cash
SOL	Solana	SUI	Sui

Symbol	Name	Symbol	Name
USDC	USD Coin	LEO	LEO Token
TRX	Tron	AVAX	Avalanche
DOGE	Dogecoin	LTC	Litecoin
ADA	Cardano	HBAR	Hedera
HYPE	Hyperliquid	SHIB	Shiba Inu
XMR	Monero	ZEC	Zcash
DAI	Dai	TAO	Bittensor
TON	Toncoin	UNI	Uniswap
MNT	Mantle	OKB	OKB
CRO	Cronos	AAVE	Aave
DOT	Polkadot	BGB	Bitget Token
ENA	Ethena	PYUSD	PayPal USD
WLFI	World Liberty Financial	NEAR	Near
PEPE	Pepe	ETC	Ethereum Classic
USD1	USD1	POL	Polygon
APT	Aptos	ASTER	Aster
M	M	IP	IP
ONDO	Ondo	ARB	Arbitrum
WLD	Worldcoin	PI	Pi Network
KCS	KuCoin Token	ICP	Internet Computer

Base Assets - FX Pairs:

Symbol	Currency
USD	US Dollar
GBP	British Pound
EUR	Euro
JPY	Japanese Yen
CHF	Swiss Franc
AUD	Australian Dollar
CAD	Canadian Dollar
CNY	Chinese Yuan

Symbol	Currency
HKD	Hong Kong Dollar
SGD	Singapore Dollar

Market Configuration

Each market is initialized with specific parameters:

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct MarketConfig {
    pub symbol: String,
    pub base_asset: String,
    pub quote_asset: String,              // Always "USDT"
    pub max_leverage: u16,                // 1000
    pub maintenance_margin_ratio: f64,    // 0.005 (0.5% at high leverage)
    pub initial_margin_ratio: f64,        // 0.01 (1.0% at high leverage)
    pub maker_fee: f64,                   // -0.0002 (-0.02% rebate)
    pub taker_fee: f64,                   // 0.0005 (0.05%)
    pub tick_size: f64,                   // Minimum price increment
    pub step_size: f64,                   // Minimum size increment
    pub min_order_size: f64,
    pub max_order_size: f64,
    pub is_active: bool,
}
```

B. Perpetual Futures Mechanics

Position Management

Position States:

- **OPEN**: Active position with collateral locked
- **CLOSING**: Partial close in progress
- **CLOSED**: Position fully exited
- **LIQUIDATING**: Under liquidation process
- **LIQUIDATED**: Liquidation completed

Position Calculations:

```rust
// Entry Price (Weighted Average)
fn calculate_entry_price(old_size: f64, old_entry: f64, new_size: f64, new_price: f64) -> f64 {
    (old_size * old_entry + new_size * new_price) / (old_size + new_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)
}
```

Leverage System

Isolated Margin Only:

- Each position has dedicated collateral
- Losses limited to position collateral
- No cross-margin between positions
- Maximum 1000x leverage

Leverage Tiers:

Leverage	Initial Margin	Maintenance Margin	Max Position Size
1x - 20x	5.0%	2.5%	Unlimited
21x - 50x	2.0%	1.0%	100,000 USDT
51x - 100x	1.0%	0.5%	50,000 USDT
101x - 500x	0.5%	0.25%	20,000 USDT
501x - 1000x	0.2%	0.1%	5,000 USDT

C. Funding Rate System

The funding rate mechanism keeps the perpetual contract price anchored to the underlying spot price.

Calculation Methodology

Frequency:

- **Calculated:** Every 1 second
- **Paid:** Every 1 hour (3,600 calculations averaged)

Formula:

```
fundingRate = premiumIndex + interestRate

premiumIndex = (markPrice - indexPrice) / indexPrice

interestRate = 0.01% / 24 hours = 0.00000417 per calculation

Where:
- markPrice = VWAP from GoDark perp trades
- indexPrice = VWAP from consolidated CEX oracle
```

Oracle Integration

Consolidated Price Feed:

```rust
use std::collections::HashMap;

struct ExchangeSource {
    exchange: String,
    weight: f64,
}

pub struct ConsolidatedOracle {
    sources: Vec<ExchangeSource>,
}

impl ConsolidatedOracle {
    pub fn new() -> Self {
        Self {
            sources: vec![
                ExchangeSource { exchange: "BINANCE".to_string(), weight: 0.30 },
                ExchangeSource { exchange: "COINBASE".to_string(), weight: 0.25 },
                ExchangeSource { exchange: "KRAKEN".to_string(), weight: 0.15 },
                ExchangeSource { exchange: "BYBIT".to_string(), weight: 0.15 },
                ExchangeSource { exchange: "OKX".to_string(), weight: 0.15 },
            ],
        }
    }

    pub async fn get_vwap(&self, symbol: &str, window_seconds: i64) -> Result<f64, OracleError> {
        let now = chrono::Utc::now().timestamp_millis();
        let start_time = now - (window_seconds * 1000);

        let mut total_value = 0.0;
        let mut total_volume = 0.0;

        for source in &self.sources {
            let trades = self.fetch_trades(&source.exchange, symbol, start_time, now).await?;

            for trade in trades {
                let weighted_volume = trade.volume * source.weight;
                total_value += trade.price * weighted_volume;
                total_volume += weighted_volume;
            }
        }

        if total_volume == 0.0 {
            return Err(OracleError::NoData);
        }

        Ok(total_value / total_volume)
    }

    pub async fn get_fallback_price(&self, symbol: &str) -> Result<f64, OracleError> {
        // Use Pyth and Switchboard as backup
        let pyth_price = self.get_pyth_price(symbol).await?;
        let switchboard_price = self.get_switchboard_price(symbol).await?;
        Ok((pyth_price + switchboard_price) / 2.0)
    }
}
```

Payment Distribution

```rust
pub async fn apply_funding_payment(positions: &mut [UserPosition]) -> Result<(), FundingError> {
    let now = chrono::Utc::now().timestamp_millis();
    let one_hour_ago = now - 3600000;

    // Get average funding rate over last hour
    let funding_rates = get_funding_rates_in_range(one_hour_ago, now).await?;
    let avg_funding_rate = funding_rates.iter().sum::<f64>() / funding_rates.len() as f64;

    for position in positions.iter_mut() {
        let mark_price = get_mark_price(&position.symbol).await?;
        let position_value = position.size.abs() as f64 * mark_price;
        let funding_payment = position_value * avg_funding_rate;

        if position.size > 0 {
            // Long position pays funding (if positive rate)
            position.collateral -= funding_payment as u64;
        } else {
            // Short position receives funding (if positive rate)
            position.collateral += funding_payment as u64;
        }

        position.accumulated_funding += funding_payment as i64;
        position.last_funding_update = now;

        // Check if position needs liquidation after funding
        let margin_ratio = calculate_margin_ratio_from_position(position, mark_price)?;
        if margin_ratio <= position.maintenance_margin as f64 / 10000.0 {
            trigger_liquidation(position).await?;
        }
    }

    Ok(())
}
```

Funding Rate History

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct FundingRateRecord {
    pub symbol: String,
    pub timestamp: i64,
    pub funding_rate: f64,
    pub premium_index: f64,
    pub mark_price: f64,
    pub index_price: f64,
}

// Store every calculation for audit
async fn store_funding_rate(db: &DatabasePool, record: FundingRateRecord) -> Result<(), DbError> {
    sqlx::query(
        "INSERT INTO funding_rates (symbol, timestamp, funding_rate, premium_index, mark_price,
index_price)
         VALUES ($1, $2, $3, $4, $5, $6)"
    )
    .bind(&record.symbol)
    .bind(record.timestamp)
    .bind(record.funding_rate)
    .bind(record.premium_index)
    .bind(record.mark_price)
    .bind(record.index_price)
    .execute(db)
    .await?;

    Ok(())
}
```

D. Liquidation Engine

The liquidation engine monitors all open positions in real-time and triggers liquidations when margin requirements are not met.

Real-Time Monitoring

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

pub struct LiquidationEngine {
    check_interval_ms: u64, // Check every 1 second
}

impl LiquidationEngine {
    pub fn new() -> Self {
        Self {
            check_interval_ms: 1000,
        }
    }

    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.get_open_positions().await?;
        let mut price_cache: HashMap<String, f64> = HashMap::new();

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

            let margin_ratio = self.calculate_margin_ratio(&position, mark_price)?;

            // Liquidation threshold
            if margin_ratio <= position.maintenance_margin as f64 / 10000.0 {
                self.liquidate_position(&position, mark_price).await?;
            }
            // Warning threshold (150% of maintenance)
            else if margin_ratio <= (position.maintenance_margin as f64 / 10000.0) * 1.5 {
                self.send_margin_warning(&position).await?;
            }
        }

        Ok(())
    }

    fn calculate_margin_ratio(&self, position: &UserPosition, mark_price: f64) -> Result<f64,
LiquidationError> {
        let unrealized_pnl = self.calculate_unrealized_pnl(position, mark_price);
        let equity = position.collateral as f64 + unrealized_pnl;
        let position_value = position.size.abs() as f64 * mark_price;

        Ok(equity / position_value)
    }

    fn calculate_unrealized_pnl(&self, position: &UserPosition, mark_price: f64) -> f64 {
        let entry_price = position.entry_price as f64 / 1e6;

        if position.size > 0 {
            // Long position
            position.size as f64 * (mark_price - entry_price)
        } else {
            // Short position
            position.size.abs() as f64 * (entry_price - mark_price)
        }
    }
}
```

Liquidation Process

Partial Liquidation (Preferred):

```rust
async fn liquidate_position(&self, position: &UserPosition, mark_price: f64) -> Result<(),
LiquidationError> {
    const LIQUIDATION_FEE: f64 = 0.005; // 0.5%
    const INSURANCE_FUND_FEE: f64 = 0.0025; // 0.25%

    // 1. Try partial liquidation first (reduce by 50%)
    let liquidation_size = position.size.abs() as f64 * 0.5;

    // 2. Execute liquidation trade at mark price
    let side = if position.size > 0 { OrderSide::Sell } else { OrderSide::Buy };
    let trade = self.execute_liquidation_trade(LiquidationTrade {
        symbol: position.symbol.clone(),
        side,
        size: liquidation_size,
        price: mark_price,
        is_liquidation: true,
    }).await?;

    // 3. Calculate fees
    let notional_value = liquidation_size * mark_price;
    let total_fee = notional_value * LIQUIDATION_FEE;
    let insurance_fee = notional_value * INSURANCE_FUND_FEE;
    let liquidator_reward = total_fee - insurance_fee;

    // 4. Update position
    let mut updated_position = position.clone();
    updated_position.size = if position.size > 0 {
        position.size - liquidation_size as i64
    } else {
        position.size + liquidation_size as i64
    };
    updated_position.collateral -= total_fee as u64;

    // 5. Transfer fees
    self.transfer_to_insurance_fund(insurance_fee).await?;
    self.transfer_to_liquidator(liquidator_reward, &trade.liquidator_wallet).await?;

    // 6. Check if remaining position is healthy
    let new_margin_ratio = self.calculate_margin_ratio(&updated_position, mark_price)?;

    if new_margin_ratio <= updated_position.maintenance_margin as f64 / 10000.0 {
        // Full liquidation required
        self.full_liquidation(&updated_position, mark_price).await?;
    }

    // 7. Emit liquidation event
    self.emit_liquidation_event(LiquidationEvent {
        position_id: position.id.clone(),
        event_type: LiquidationType::Partial,
        liquidated_size: liquidation_size,
        liquidation_price: mark_price,
        collateral_remaining: updated_position.collateral,
    }).await?;

    Ok(())
}
```

Full Liquidation:

```rust
async fn full_liquidation(&self, position: &UserPosition, mark_price: f64) -> Result<(),
LiquidationError> {
    // Close entire position
    let liquidation_size = position.size.abs() as f64;

    let side = if position.size > 0 { OrderSide::Sell } else { OrderSide::Buy };
    let trade = self.execute_liquidation_trade(LiquidationTrade {
        symbol: position.symbol.clone(),
        side,
        size: liquidation_size,
        price: mark_price,
        is_liquidation: true,
    }).await?;

    // Any remaining collateral goes to insurance fund
    let collateral_i64 = position.collateral as i64;
    if collateral_i64 > 0 {
        self.transfer_to_insurance_fund(position.collateral as f64).await?;
    } else if collateral_i64 < 0 {
        // Bad debt - covered by insurance fund
        self.cover_bad_debt(collateral_i64.abs() as f64).await?;
    }

    // Close position
    let mut updated_position = position.clone();
    updated_position.size = 0;
    updated_position.collateral = 0;
    updated_position.status = PositionStatus::Liquidated;

    self.emit_liquidation_event(LiquidationEvent {
        position_id: position.id.clone(),
        event_type: LiquidationType::Full,
        liquidated_size: liquidation_size,
        liquidation_price: mark_price,
        bad_debt: if collateral_i64 < 0 { collateral_i64.abs() as f64 } else { 0.0 },
    }).await?;

    Ok(())
}
```

Insurance Fund

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct InsuranceFund {
    pub total_balance: f64,
    pub contributions: f64,
    pub bad_debt_covered: f64,
    pub utilization_ratio: f64,
}

pub struct InsuranceFundManager {
    db: DatabasePool,
}

impl InsuranceFundManager {
    pub async fn add_contribution(&self, amount: f64, source: &str) -> Result<(), DbError> {
        // Increment totalBalance
        sqlx::query("UPDATE insurance_fund SET total_balance = total_balance + $1, contributions =
contributions + $2")
            .bind(amount)
            .bind(amount)
            .execute(&self.db)
            .await?;

        // Log transaction
        self.log_transaction(FundTransaction {
            transaction_type: "CONTRIBUTION".to_string(),
            amount,
            source: source.to_string(),
            timestamp: chrono::Utc::now().timestamp_millis(),
            position_id: None,
        }).await?;

        Ok(())
    }

    pub async fn cover_bad_debt(&self, amount: f64, position_id: &str) -> Result<(), FundError> {
        // Get fund balance
        let fund: InsuranceFund = sqlx::query_as("SELECT * FROM insurance_fund LIMIT 1")
            .fetch_one(&self.db)
            .await?;

        if fund.total_balance < amount {
            self.alert_critical("Insurance fund insufficient for bad debt").await?;
            // Emergency procedures
            return Err(FundError::InsufficientFunds);
        }

        // Decrement balance, increment bad debt covered
        sqlx::query("UPDATE insurance_fund SET total_balance = total_balance - $1, bad_debt_covered
= bad_debt_covered + $2")
            .bind(amount)
            .bind(amount)
            .execute(&self.db)
            .await?;

        // Log transaction
        self.log_transaction(FundTransaction {
            transaction_type: "BAD_DEBT_COVERAGE".to_string(),
            amount,
            source: String::new(),
            timestamp: chrono::Utc::now().timestamp_millis(),
            position_id: Some(position_id.to_string()),
        }).await?;

        Ok(())
    }

    pub fn get_utilization_ratio(&self, bad_debt_covered: f64, contributions: f64) -> f64 {
        if contributions == 0.0 {
            0.0
        } else {
            bad_debt_covered / contributions
        }
    }
}
```

E. Ephemeral Vault System

Wallet Connection Flow

Step-by-Step Process:

```
1. User Navigation
   ↓
2. Click "Connect Wallet"
   ↓
3. Select Wallet Provider
   - Phantom
   - Trust Wallet
   - Solflare
   - OR Create New Wallet (Sign in with Google/Apple/X/Discord)
   ↓
4. Wallet Connection
   - Browser extension opens
   - User approves connection
   ↓
5. Authorization Screen
   "Authorize X USDT for trading"
   - User inputs amount (e.g., 10,000 USDT)
   - Explains delegate approval
   ↓
6. Sign Approval Transaction
   - Creates token delegate approval
   - User signs with wallet
   ↓
7. Vault Creation (Automatic)
   - Backend detects approval
   - Creates ephemeral vault PDA
   - Links to user account
   ↓
8. Ready to Trade
   - Vault active
   - USDT available for trading
   - One-click withdrawals enabled
```

Delegate Approval Mechanism

```rust
// Token delegate approval allows program to spend USDT
pub fn approve_delegate(
    ctx: Context<ApproveDelegate>,
    amount: u64
) -> Result<()> {
    // SPL Token approve instruction
    token::approve(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            token::Approve {
                to: ctx.accounts.user_usdt_account.to_account_info(),
                delegate: ctx.accounts.godark_program.to_account_info(),
                authority: ctx.accounts.user.to_account_info(),
            }
        ),
        amount
    )?;

    emit!(DelegateApproved {
        user: ctx.accounts.user.key(),
        amount,
        timestamp: Clock::get()?.unix_timestamp
    });

    Ok(())
}
```

Automatic Deposits

When user places first trade, program automatically transfers required collateral:

```
pub fn auto_deposit_for_trade(
    ctx: Context<AutoDeposit>,
    required_collateral: u64
) -> Result<()> {
    let vault = &mut ctx.accounts.vault;

    // Check delegate approval limit
    require!(
        vault.used_amount + required_collateral <= vault.approved_amount,
        ErrorCode::ExceedsApprovedAmount
    );

    // Transfer from user to vault using delegate authority
    let seeds = &[
        b"authority",
        &[ctx.bumps.authority]
    ];
    let signer = &[&seeds[..]];

    token::transfer(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            token::Transfer {
                from: ctx.accounts.user_usdt_account.to_account_info(),
                to: ctx.accounts.vault_usdt_account.to_account_info(),
                authority: ctx.accounts.program_authority.to_account_info(),
            },
            signer
        ),
        required_collateral
    )?;

    vault.used_amount += required_collateral;
    vault.last_activity = Clock::get()?.unix_timestamp;

    Ok(())
}
```

Vault Cleanup

Vaults are automatically cleaned up when:

- All positions closed
- User requests withdrawal
- Inactivity timeout (optional)

```
pub fn cleanup_vault(ctx: Context<CleanupVault>) -> Result<()> {
    let vault = &ctx.accounts.vault;

    // Verify no active positions
    require!(vault.used_amount == 0, ErrorCode::VaultHasActivePositions);

    // Return any remaining balance
    if vault.available_amount > 0 {
        let seeds = &[
            b"vault",
            vault.user_wallet.as_ref(),
            &[vault.bump]
        ];
        let signer = &[&seeds[..]];

        token::transfer(
            CpiContext::new_with_signer(
                ctx.accounts.token_program.to_account_info(),
                token::Transfer {
                    from: ctx.accounts.vault_usdt_account.to_account_info(),
                    to: ctx.accounts.user_usdt_account.to_account_info(),
                    authority: ctx.accounts.vault.to_account_info(),
                },
                signer
            ),
            vault.available_amount
        )?;
    }

    // Close vault account and return rent
    let vault_lamports = ctx.accounts.vault.to_account_info().lamports();
    **ctx.accounts.vault.to_account_info().try_borrow_mut_lamports()? = 0;
    **ctx.accounts.user.try_borrow_mut_lamports()? += vault_lamports;

    Ok(())
}
```

Withdrawal Process

One-Click Withdrawal (No Signature Required):

```
async function withdrawUnlockedBalance(): Promise<void> {
    // Backend initiates withdrawal using program authority
    const vault = await getVault(userWallet);

    if (vault.availableAmount === 0) {
        throw new Error('No unlocked balance to withdraw');
    }

    // Program executes withdrawal automatically
    const tx = await program.methods
        .withdrawFromVault(new BN(vault.availableAmount))
        .accounts({
            vault: vaultPDA,
            vaultUsdtAccount: vaultTokenAccount,
            userUsdtAccount: userTokenAccount,
            user: userWallet,
            tokenProgram: TOKEN_PROGRAM_ID
        })
        .rpc();

    // User receives USDT immediately
    await confirmTransaction(tx);
}
```

Revocation

User can revoke delegate approval at any time:

```
async function revokeWalletAccess(): Promise<void> {
    // 1. Check for active positions
    const activePositions = await getActivePositions(userWallet);

    if (activePositions.length > 0) {
        throw new Error('Close all positions before revoking access');
    }

    // 2. Withdraw all available funds
    await withdrawUnlockedBalance();

    // 3. Revoke token delegate approval
    const tx = new Transaction().add(
        Token.createRevokeInstruction(
            TOKEN_PROGRAM_ID,
            userTokenAccount,
            userWallet,
            []
        )
    );

    await sendAndConfirmTransaction(connection, tx, [userKeypair]);

    // 4. Deactivate vault
    await program.methods
        .revokeVault()
        .accounts({
            vault: vaultPDA,
            user: userWallet
        })
        .rpc();
}
```

---

4. Settlement Flow (Detailed)

This section provides a comprehensive step-by-step breakdown of the complete settlement process from wallet connection to trade finality.

Step 1: Wallet Connection and Authorization

User Action:

```
User clicks "Connect Wallet" → Selects provider → Approves connection
```

Backend Process:

```rust
use ed25519_dalek::{PublicKey, Signature, Verifier};
use jsonwebtoken::{encode, Header, EncodingKey};

async fn handle_wallet_connect(wallet_public_key: &Pubkey) -> Result<ConnectResponse, WalletError>
{
    // 1. Verify wallet signature
    let timestamp = chrono::Utc::now().timestamp_millis();
    let message = format!("Connect to GoDark\nTimestamp: {}", timestamp);
    let signature = wallet.sign_message(&message).await?;

    let public_key_bytes = wallet_public_key.to_bytes();
    let public_key = PublicKey::from_bytes(&public_key_bytes)?;
    let sig = Signature::from_bytes(&signature)?;

    if public_key.verify(message.as_bytes(), &sig).is_err() {
        return Err(WalletError::InvalidSignature);
    }

    // 2. Check if wallet already has account
    let existing_account = sqlx::query_as::<_, Account>(
        "SELECT * FROM accounts WHERE wallet = $1"
    )
    .bind(wallet_public_key.to_string())
    .fetch_optional(&db)
    .await?;

    if existing_account.is_none() {
        // Prompt for account creation
        return Ok(ConnectResponse {
            status: "NEEDS_ACCOUNT".to_string(),
            wallet: wallet_public_key.to_string(),
            token: None,
        });
    }

    let account = existing_account.unwrap();

    // 3. Generate session token
    let claims = SessionClaims {
        wallet: wallet_public_key.to_string(),
        account_id: account.id,
        exp: (chrono::Utc::now() + chrono::Duration::hours(24)).timestamp() as usize,
    };

    let secret = std::env::var("JWT_SECRET")?;
    let session_token = encode(
        &Header::default(),
        &claims,
        &EncodingKey::from_secret(secret.as_ref())
    )?;

    Ok(ConnectResponse {
        status: "CONNECTED".to_string(),
        wallet: wallet_public_key.to_string(),
        token: Some(session_token),
    })
}
```

Step 2: PDA Vault Creation

Smart Contract Execution:

```rust
pub fn create_ephemeral_vault(
    ctx: Context<CreateEphemeralVault>,
    approved_amount: u64
) -> Result<()> {
    let vault = &mut ctx.accounts.vault;
    let clock = Clock::get()?;

    // Initialize vault account
    vault.user_wallet = ctx.accounts.user.key();
    vault.vault_pda = ctx.accounts.vault.key();
    vault.created_at = clock.unix_timestamp;
    vault.last_activity = clock.unix_timestamp;
    vault.approved_amount = approved_amount;
    vault.used_amount = 0;
    vault.available_amount = 0;
    vault.is_active = true;
    vault.bump = *ctx.bumps.get("vault").unwrap();

    // Emit event for backend tracking
    emit!(VaultCreated {
        user: ctx.accounts.user.key(),
        vault_pda: ctx.accounts.vault.key(),
        approved_amount,
        timestamp: clock.unix_timestamp
    });

    Ok(())
}

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

    #[account(
        init,
        payer = user,
        space = 8 + std::mem::size_of::<EphemeralVault>(),
        seeds = [b"vault", user.key().as_ref()],
        bump
    )]
    pub vault: Account<'info, EphemeralVault>,

    pub system_program: Program<'info, System>,
}
```

Step 3: Deposit via Delegate Transfer

Automatic Deposit on First Trade:

```rust
pub fn auto_deposit_on_trade(
    ctx: Context<AutoDepositOnTrade>,
    order_size: u64,
    leverage: u16
) -> Result<()> {
    let vault = &mut ctx.accounts.vault;
    let market = &ctx.accounts.market;
    let clock = Clock::get()?;

    // Calculate required collateral
    let price = get_oracle_price(&ctx.accounts.oracle)?;
    let notional = order_size.checked_mul(price).ok_or(ErrorCode::MathOverflow)?;
    let required_collateral = notional.checked_div(leverage as
u64).ok_or(ErrorCode::InvalidLeverage)?;
    let initial_margin = required_collateral
        .checked_mul(market.initial_margin_ratio as u64)
        .ok_or(ErrorCode::MathOverflow)?
        .checked_div(10000)
        .ok_or(ErrorCode::MathOverflow)?;

    let total_needed =
required_collateral.checked_add(initial_margin).ok_or(ErrorCode::MathOverflow)?;

    // Check against approved amount
    require!(
        vault.used_amount + total_needed <= vault.approved_amount,
        ErrorCode::ExceedsApprovedAmount
    );

    // Transfer using delegate authority
    let authority_seeds = &[
        b"authority",
        &[ctx.bumps.authority]
    ];
    let signer = &[&authority_seeds[..]];

    token::transfer(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            token::Transfer {
                from: ctx.accounts.user_usdt_account.to_account_info(),
                to: ctx.accounts.vault_usdt_account.to_account_info(),
                authority: ctx.accounts.program_authority.to_account_info(),
            },
            signer
        ),
        total_needed
    )?;

    vault.used_amount += total_needed;
    vault.available_amount += total_needed;
    vault.last_activity = clock.unix_timestamp;

    Ok(())
}
```

Step 4: Off-Chain Trade Execution (1-Second Windows)

Matching Engine Process:

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

pub struct TradeExecutionCycle {
    cycle_duration_ms: u64,
    current_cycle: Vec<Trade>,
    matching_engine: Arc<MatchingEngine>,
}

impl TradeExecutionCycle {
    const CYCLE_DURATION_MS: u64 = 1000;

    pub fn new(matching_engine: Arc<MatchingEngine>) -> Self {
        Self {
            cycle_duration_ms: Self::CYCLE_DURATION_MS,
            current_cycle: Vec::new(),
            matching_engine,
        }
    }

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

        loop {
            timer.tick().await;
            if let Err(e) = self.execute_cycle().await {
                eprintln!("Cycle execution error: {:?}", e);
            }
        }
    }

    async fn execute_cycle(&mut self) -> Result<(), CycleError> {
        let cycle_id = self.generate_cycle_id();
        let start_time = chrono::Utc::now().timestamp_millis();

        // 1. Match all pending orders
        let matches = self.matching_engine.match_all().await?;

        // 2. Execute trades
        for order_match in matches {
            let trade = self.execute_trade(order_match).await?;
            self.current_cycle.push(trade);
        }

        // 3. At end of cycle, prepare for settlement
        if !self.current_cycle.is_empty() {
            self.prepare_settlement(&self.current_cycle).await?;
            self.current_cycle.clear();
        }

        let duration = chrono::Utc::now().timestamp_millis() - start_time;
        self.record_cycle_metrics(&cycle_id, duration, matches.len()).await?;

        Ok(())
    }

    async fn execute_trade(&self, order_match: OrderMatch) -> Result<Trade, TradeError> {
        let trade_id = generate_trade_id();

        let trade = Trade {
            id: trade_id.clone(),
            symbol: order_match.symbol,
            buyer: order_match.buy_order.user_id,
            seller: order_match.sell_order.user_id,
            buy_order_id: order_match.buy_order.id,
            sell_order_id: order_match.sell_order.id,
            price: order_match.execution_price,
            size: order_match.size,
            timestamp: chrono::Utc::now().timestamp_millis(),
            buyer_fee: self.calculate_fee(&order_match.buy_order, order_match.size,
order_match.execution_price),
            seller_fee: self.calculate_fee(&order_match.sell_order, order_match.size,
order_match.execution_price),
            status: TradeStatus::Matched,
        };

        // Store in database
        sqlx::query(
            "INSERT INTO trades (id, symbol, buyer, seller, buy_order_id, sell_order_id, price,
size, timestamp, buyer_fee, seller_fee, status)
             VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12)"
        )
        .bind(&trade.id)
        .bind(&trade.symbol)
```

```
            .bind(&trade.buyer)
            .bind(&trade.seller)
            .bind(&trade.buy_order_id)
            .bind(&trade.sell_order_id)
            .bind(trade.price)
            .bind(trade.size)
            .bind(trade.timestamp)
            .bind(trade.buyer_fee)
            .bind(trade.seller_fee)
            .bind("MATCHED")
            .execute(&self.db)
            .await?;

            // Notify users
            self.notify_trade_execution(&trade).await?;

            Ok(trade)
        }
    }
```

Step 5: Net Delta Calculation Per User

Position Netting Algorithm:

```rust
use std::collections::HashMap;

#[derive(Debug, Clone)]
struct NetPosition {
    wallet: String,
    symbol: String,
    net_size_change: f64,
    net_collateral_change: f64,
    total_fees: f64,
}

#[derive(Debug, Clone)]
struct PositionUpdate {
    wallet: String,
    symbol: String,
    size_change: f64,
    collateral_change: f64,
    fee: f64,
}

fn calculate_net_positions(trades: &[Trade]) -> HashMap<String, NetPosition> {
    let mut net_map: HashMap<String, NetPosition> = HashMap::new();

    for trade in trades {
        // Process buyer
        update_net_position(&mut net_map, PositionUpdate {
            wallet: trade.buyer.clone(),
            symbol: trade.symbol.clone(),
            size_change: trade.size,  // Positive for long
            collateral_change: -(trade.price * trade.size),  // Pay USDT
            fee: trade.buyer_fee,
        });

        // Process seller
        update_net_position(&mut net_map, PositionUpdate {
            wallet: trade.seller.clone(),
            symbol: trade.symbol.clone(),
            size_change: -trade.size,  // Negative for short
            collateral_change: trade.price * trade.size,  // Receive USDT
            fee: trade.seller_fee,
        });
    }

    net_map
}

fn update_net_position(
    map: &mut HashMap<String, NetPosition>,
    update: PositionUpdate
) {
    let key = format!("{}-{}", update.wallet, update.symbol);

    let position = map.entry(key).or_insert(NetPosition {
        wallet: update.wallet.clone(),
        symbol: update.symbol.clone(),
        net_size_change: 0.0,
        net_collateral_change: 0.0,
        total_fees: 0.0,
    });

    position.net_size_change += update.size_change;
    position.net_collateral_change += update.collateral_change;
    position.total_fees += update.fee;
}
```

Step 6: Batch Settlement (1-Second Interval)

Settlement Transaction Builder:

```rust
use solana_sdk::{
    transaction::Transaction,
    instruction::Instruction,
    compute_budget::ComputeBudgetInstruction,
    pubkey::Pubkey,
    system_program,
};
use anchor_client::Program;

async fn build_settlement_transaction(
    batch_id: &str,
    net_positions: &[NetPosition],
    merkle_root: &[u8; 32]
) -> Result<Transaction, SettlementError> {
    let mut instructions = Vec::new();

    // 1. Set compute budget
    instructions.push(
        ComputeBudgetInstruction::set_compute_unit_limit(1_400_000)
    );

    instructions.push(
        ComputeBudgetInstruction::set_compute_unit_price(1)
    );

    // 2. Record batch metadata
    let batch_pda = get_batch_pda(batch_id)?;

    let record_batch_ix = program.request()
        .accounts(godark_perps::accounts::RecordSettlementBatch {
            relayer: relayer_keypair.pubkey(),
            batch_account: batch_pda,
            system_program: system_program::ID,
        })
        .args(godark_perps::instruction::RecordSettlementBatch {
            batch_id: batch_id.as_bytes().try_into()?,
            merkle_root: *merkle_root,
            trade_count: net_positions.len() as u16,
        })
        .instructions()?;

    instructions.extend(record_batch_ix);

    // 3. Update each position
    for net_pos in net_positions {
        let wallet_pubkey = Pubkey::from_str(&net_pos.wallet)?;
        let market_pda = get_market_pda(&net_pos.symbol)?;
        let position_pda = get_position_pda(&wallet_pubkey, &market_pda)?;
        let vault_pda = get_vault_pda(&wallet_pubkey)?;
        let vault_usdt_account = get_vault_token_account(&vault_pda)?;
        let oracle_pda = get_oracle_pda(&net_pos.symbol)?;

        let settle_position_ix = program.request()
            .accounts(godark_perps::accounts::SettlePosition {
                user: wallet_pubkey,
                position: position_pda,
                market: market_pda,
                vault: vault_pda,
                vault_usdt_account,
                fee_recipient: fee_wallet,
                oracle: oracle_pda,
                system_program: system_program::ID,
                token_program: spl_token::ID,
            })
            .args(godark_perps::instruction::SettlePosition {
                size_change: (net_pos.net_size_change * 1e8) as i64,
                collateral_change: (net_pos.net_collateral_change * 1e6) as i64,
                fee: (net_pos.total_fees * 1e6) as u64,
            })
            .instructions()?;

        instructions.extend(settle_position_ix);
    }

    // Build transaction
    let recent_blockhash = rpc_client.get_latest_blockhash()?;
    let tx = Transaction::new_signed_with_payer(
        &instructions,
        Some(&relayer_keypair.pubkey()),
        &[&relayer_keypair],
        recent_blockhash,
    );
```

```
        Ok(tx)
    }
```

Step 7: On-Chain Ledger Update with Merkle Root

Smart Contract Settlement:

```rust
pub fn settle_position(
    ctx: Context<SettlePosition>,
    params: SettlePositionParams
) -> Result<()> {
    let position = &mut ctx.accounts.position;
    let market = &ctx.accounts.market;
    let vault = &mut ctx.accounts.vault;
    let oracle = &ctx.accounts.oracle;
    let clock = Clock::get()?;

    // 1. Get current mark price
    let mark_price = get_price_from_oracle(oracle)?;

    // 2. Apply pending funding
    apply_funding_payment(position, market, clock.unix_timestamp)?;

    // 3. Calculate current unrealized PnL
    let unrealized_pnl = calculate_unrealized_pnl(position, mark_price)?;

    // 4. Update position size
    let old_size = position.size;
    let new_size = old_size.checked_add(params.size_change).ok_or(ErrorCode::MathOverflow)?;

    // 5. Update entry price (weighted average)
    if new_size != 0 && (old_size > 0 && new_size > 0) || (old_size < 0 && new_size < 0) {
        // Adding to existing position
        let old_value = old_size.checked_mul(position.entry_price as
i64).ok_or(ErrorCode::MathOverflow)?;
        let new_value = params.size_change.checked_mul(mark_price as
i64).ok_or(ErrorCode::MathOverflow)?;
        let total_value = old_value.checked_add(new_value).ok_or(ErrorCode::MathOverflow)?;
        position.entry_price = (total_value / new_size) as u64;
    } else if (old_size > 0 && new_size < 0) || (old_size < 0 && new_size > 0) {
        // Flipping position
        position.realized_pnl =
position.realized_pnl.checked_add(unrealized_pnl).ok_or(ErrorCode::MathOverflow)?;
        position.entry_price = mark_price;
    } else if new_size == 0 {
        // Closing position completely
        position.realized_pnl =
position.realized_pnl.checked_add(unrealized_pnl).ok_or(ErrorCode::MathOverflow)?;
        position.entry_price = 0;
    }

    position.size = new_size;

    // 6. Update collateral
    let collateral_i64 = position.collateral as i64;
    let new_collateral = collateral_i64
        .checked_add(params.collateral_change)
        .ok_or(ErrorCode::MathOverflow)?;

    require!(new_collateral >= 0, ErrorCode::InsufficientCollateral);
    position.collateral = new_collateral as u64;

    // 7. Deduct fees
    require!(position.collateral >= params.fee, ErrorCode::InsufficientCollateral);
    position.collateral =
position.collateral.checked_sub(params.fee).ok_or(ErrorCode::MathOverflow)?;

    // 8. Transfer fees to GoDark wallet
    transfer_fees(
        &ctx.accounts.vault_usdt_account,
        &ctx.accounts.fee_recipient,
        &ctx.accounts.token_program,
        params.fee,
        vault.bump
    )?;

    // 9. Update vault accounting
    vault.used_amount = position.collateral;
    vault.last_activity = clock.unix_timestamp;

    // 10. Calculate new liquidation price
    position.liquidation_price = calculate_liquidation_price(position, market)?;

    // 11. Check margin requirements
    let margin_ratio = calculate_margin_ratio(position, mark_price)?;
    require!(
        margin_ratio >= market.maintenance_margin_ratio,
        ErrorCode::InsufficientMargin
    );
```

```
    // 12. Update position timestamp
    position.last_update_timestamp = clock.unix_timestamp;

    // 13. Emit event
    emit!(PositionSettled {
        user: ctx.accounts.user.key(),
        market: market.key(),
        size: position.size,
        entry_price: position.entry_price,
        collateral: position.collateral,
        liquidation_price: position.liquidation_price,
        timestamp: clock.unix_timestamp
    });

    Ok(())
}
```

Step 8: Vault Cleanup and Margin Return

Cleanup Process:

```
use solana_sdk::pubkey::Pubkey;
use anchor_client::Program;

async fn cleanup_vault_if_needed(user_id: &str) -> Result<(), VaultError> {
    let positions = get_open_positions(user_id).await?;

    if positions.is_empty() {
        // No open positions, can cleanup
        let vault = get_vault(user_id).await?;

        if vault.used_amount == 0 && vault.available_amount > 0 {
            // Return unused funds to user
            let user_pubkey = Pubkey::from_str(user_id)?;
            let user_usdt_account = get_user_usdt_account(user_id).await?;

            let signature = program.request()
                .accounts(godark_perps::accounts::CleanupVault {
                    user: user_pubkey,
                    vault: vault.pda,
                    vault_usdt_account: vault.usdt_account,
                    user_usdt_account,
                    token_program: spl_token::ID,
                })
                .args(godark_perps::instruction::CleanupVault {})
                .send()
                .await?;

            println!("Vault cleaned up: {}", signature);
        }
    }

    Ok(())
}
```

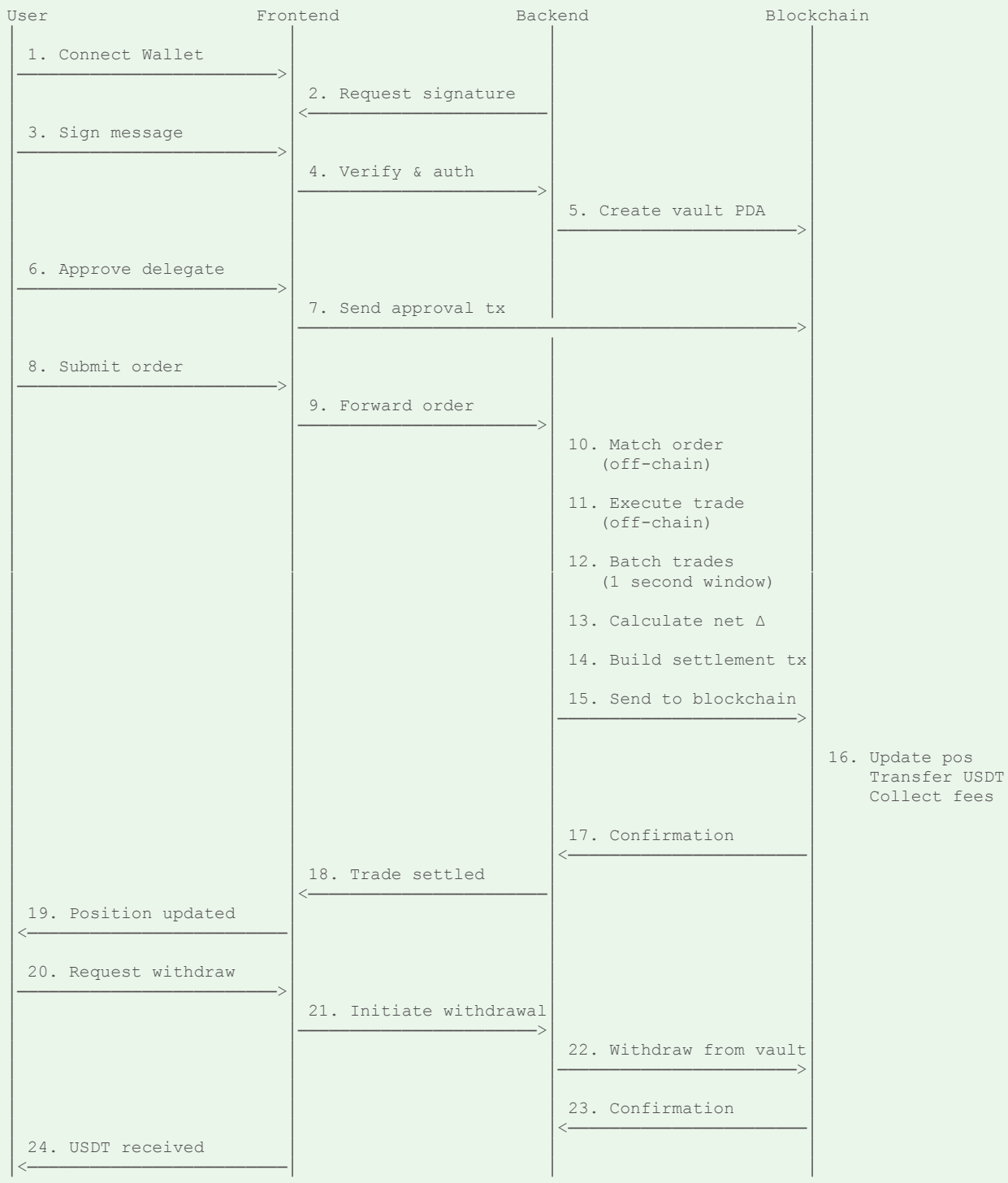
Step 9: User-Initiated Withdraw/Revoke

Withdrawal Interface:

```rust
use solana_sdk::pubkey::Pubkey;
use sqlx::PgPool;

async fn withdraw_unlocked_balance(user_id: &str, amount: Option<f64>, db: &PgPool) ->
Result<String, WithdrawalError> {
    let vault = get_vault(user_id).await?;

    // Determine withdrawal amount
    let withdraw_amount = amount.unwrap_or(vault.available_amount);

    if withdraw_amount > vault.available_amount {
        return Err(WithdrawalError::InsufficientBalance {
            available: vault.available_amount,
        });
    }

    // Execute withdrawal (backend-initiated, no user signature needed)
    let user_pubkey = Pubkey::from_str(user_id)?;
    let user_usdt_account = get_user_usdt_account(user_id).await?;
    let withdraw_amount_lamports = (withdraw_amount * 1e6) as u64;

    let signature = program.request()
        .accounts(godark_perps::accounts::WithdrawFromVault {
            user: user_pubkey,
            vault: vault.pda,
            vault_usdt_account: vault.usdt_account,
            user_usdt_account,
            token_program: spl_token::ID,
        })
        .args(godark_perps::instruction::WithdrawFromVault {
            amount: withdraw_amount_lamports,
        })
        .send()
        .await?;

    // Wait for confirmation
    rpc_client.confirm_transaction_with_spinner(&signature, &rpc_client.get_latest_blockhash()?,
CommitmentConfig::confirmed())?;

    // Update database
    sqlx::query(
        "INSERT INTO withdrawals (user_id, amount, tx_signature, timestamp, status)
         VALUES ($1, $2, $3, $4, 'COMPLETED')"
    )
    .bind(user_id)
    .bind(withdraw_amount)
    .bind(signature.to_string())
    .bind(chrono::Utc::now().timestamp_millis())
    .execute(db)
    .await?;

    Ok(signature.to_string())
}

async fn revoke_vault_access(user_id: &str, db: &PgPool) -> Result<(), VaultError> {
    // 1. Verify no active positions
    let positions = get_open_positions(user_id).await?;
    if !positions.is_empty() {
        return Err(VaultError::HasOpenPositions);
    }

    // 2. Withdraw all available balance
    let vault = get_vault(user_id).await?;
    if vault.available_amount > 0.0 {
        withdraw_unlocked_balance(user_id, None, db).await?;
    }

    // 3. User must revoke delegate approval (requires signature)
    // Frontend handles this

    // 4. Deactivate vault on backend
    let user_pubkey = Pubkey::from_str(user_id)?;

    let _signature = program.request()
        .accounts(godark_perps::accounts::DeactivateVault {
            user: user_pubkey,
            vault: vault.pda,
        })
        .args(godark_perps::instruction::DeactivateVault {})
        .send()
        .await?;

    // Update database
```

```
    sqlx::query(
        "UPDATE vaults SET status = 'REVOKED', revoked_at = $1 WHERE user_id = $2"
    )
    .bind(chrono::Utc::now().timestamp_millis())
    .bind(user_id)
    .execute(db)
    .await?;

    Ok(())
}
```

Settlement Flow Diagram



5. Security & Risk Management

Non-Custodial Design Principles

GoDark operates as a fully non-custodial protocol where users maintain control of their assets at all times.

Key Principles:

- User funds held in program-controlled PDAs (Program Derived Addresses)
- Smart contracts can only execute predefined operations
- No admin keys with withdrawal privileges
- Users can revoke access and withdraw funds at any time
- All operations require cryptographic verification

Custody Hierarchy:

State	Custodian	User Control	Withdrawal Access
Wallet Balance	User	Full	Immediate
Delegated Approval	User + Program	Revocable	Via revocation
Vault (Unlocked)	Program PDA	Withdrawal anytime	One-click
Position Collateral	Program PDA	Must close position	After closing

---

Smart Contract Security Measures

Audit Process

Multi-Tier Auditing:

```rust
use std::collections::HashMap;

struct AuditTier {
    auditors: Vec<&'static str>,
    focus: &'static str,
    duration: &'static str,
    scope: Vec<&'static str>,
}

struct BugBountyTier {
    process: &'static str,
    platform: &'static str,
    rewards: &'static str,
    ongoing: bool,
}

pub struct AuditPipeline {
    tier1: AuditTier,
    tier2: AuditTier,
    tier3: BugBountyTier,
}

impl AuditPipeline {
    pub fn new() -> Self {
        Self {
            tier1: AuditTier {
                auditors: vec!["OtterSec", "Neodyme"],
                focus: "Core smart contracts",
                duration: "4-6 weeks",
                scope: vec![
                    "Position management",
                    "Vault operations",
                    "Settlement logic",
                    "Liquidation engine",
                ],
            },
            tier2: AuditTier {
                auditors: vec!["Trail of Bits"],
                focus: "System architecture review",
                duration: "2-3 weeks",
                scope: vec![
                    "Overall design",
                    "Integration points",
                    "Oracle dependencies",
                    "Economic model",
                ],
            },
            tier3: BugBountyTier {
                process: "Community bug bounty",
                platform: "Immunefi",
                rewards: "Up to $1M",
                ongoing: true,
            },
        }
    }
}
```

Testing Requirements

Test Coverage Targets:

- Unit tests: 100% coverage
- Integration tests: All critical paths
- Fuzzing: 1M+ iterations per function
- Formal verification: Core math operations

```
// Example test structure
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_position_liquidation() {
        // Test partial liquidation
        // Test full liquidation
        // Test liquidation price calculation
        // Test insurance fund coverage
    }

    #[test]
    fn test_funding_rate_calculation() {
        // Test rate calculation
        // Test payment distribution
        // Test edge cases (extreme rates)
    }

    #[test]
    fn test_vault_operations() {
        // Test deposit
        // Test withdrawal
        // Test delegate approval
        // Test revocation
    }
}
```

Upgrade Mechanism

Immutable Core with Controlled Upgrades:

```
pub struct ProgramUpgrade {
    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
    pub status: UpgradeStatus,
}
```

Upgrade Process:

1. Multisig proposes upgrade (3/5 approval)
2. 48-hour timelock begins
3. Community notification
4. Emergency abort window
5. Automatic execution after timelock
6. Verification and rollback capability

Permission Model and Access Control

Role-Based Access Control

```rust
use std::collections::HashMap;
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub enum Role {
    User,           // Standard trader
    Liquidator,     // Can execute liquidations
    Relayer,        // Settlement relayer
    OracleUpdater,  // Can update price feeds
    Admin,          // System administration
    Emergency,      // Emergency pause only
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Restriction {
    restriction_type: String,
    value: String,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Permission {
    role: Role,
    actions: Vec<String>,
    restrictions: Vec<Restriction>,
}

pub fn get_role_permissions() -> HashMap<Role, Permission> {
    let mut permissions = HashMap::new();

    permissions.insert(Role::User, Permission {
        role: Role::User,
        actions: vec![
            "CREATE_VAULT".to_string(),
            "DEPOSIT".to_string(),
            "WITHDRAW".to_string(),
            "SUBMIT_ORDER".to_string(),
            "CANCEL_ORDER".to_string(),
            "CLOSE_POSITION".to_string(),
        ],
        restrictions: vec![
            Restriction { restriction_type: "RATE_LIMIT".to_string(), value: "60".to_string() },
            Restriction { restriction_type: "GEO".to_string(), value: "NOT_RESTRICTED".to_string()
},
        ],
    });

    permissions.insert(Role::Liquidator, Permission {
        role: Role::Liquidator,
        actions: vec![
            "LIQUIDATE_POSITION".to_string(),
            "VIEW_UNDERWATER_POSITIONS".to_string(),
        ],
        restrictions: vec![
            Restriction { restriction_type: "COLLATERAL_REQUIREMENT".to_string(), value:
"10000".to_string() },
        ],
    });

    permissions.insert(Role::Relayer, Permission {
        role: Role::Relayer,
        actions: vec![
            "SUBMIT_SETTLEMENT_BATCH".to_string(),
            "UPDATE_POSITIONS".to_string(),
            "COLLECT_FEES".to_string(),
        ],
        restrictions: vec![
            Restriction { restriction_type: "WHITELIST".to_string(), value:
"APPROVED_RELAYERS".to_string() },
        ],
    });

    permissions
}
```

Multisig Configuration

Critical Operations Require Multisig:

```
pub struct Multisig {
    pub owners: Vec<Pubkey>,            // 5 signers
    pub threshold: u8,                  // 3 required
    pub pending_transactions: Vec<PendingTx>,
}

pub struct PendingTx {
    pub transaction_id: u64,
    pub instruction: Instruction,
    pub approvals: Vec<Pubkey>,
    pub created_at: i64,
    pub expires_at: i64,
}

// Operations requiring multisig:
const MULTISIG_OPERATIONS = [
    'UPGRADE_PROGRAM',
    'CHANGE_FEE_STRUCTURE',
    'MODIFY_RISK_PARAMETERS',
    'EMERGENCY_PAUSE',
    'WITHDRAW_INSURANCE_FUND',
    'ADD_NEW_MARKET'
];
```

Liquidation Safeguards

Multi-Layer Protection

Layer 1: Early Warning System

```
pub struct MarginWarningSystem {
    warning_threshold: f64, // 150% of maintenance
}

impl MarginWarningSystem {
    const WARNING_THRESHOLD: f64 = 1.5;

    pub fn new() -> Self {
        Self {
            warning_threshold: Self::WARNING_THRESHOLD,
        }
    }

    pub async fn monitor_positions(&self) -> Result<(), MonitorError> {
        let positions = self.get_active_positions().await?;

        for position in positions {
            let margin_ratio = self.calculate_margin_ratio(&position).await?;
            let maintenance_ratio = position.maintenance_margin as f64 / 10000.0;

            if margin_ratio <= maintenance_ratio * self.warning_threshold {
                let current_price = self.get_mark_price(&position.symbol).await?;
                let liquidation_price = position.liquidation_price as f64 / 1e6;

                self.send_warning(&position.owner, MarginWarning {
                    level: WarningLevel::Critical,
                    margin_ratio,
                    liquidation_price,
                    current_price,
                    time_to_liquidation: self.estimate_time_to_liquidation(&position),
                }).await?;
            }
        }

        Ok(())
    }
}
```

Layer 2: Partial Liquidation

- Always attempt 50% liquidation first
- Preserves trader's position when possible
- Reduces insurance fund burden

Layer 3: Insurance Fund

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct InsuranceFundMetrics {
    pub total_balance: f64,
    pub target_balance: f64,       // 1% of open interest
    pub utilization_ratio: f64,    // Current usage
    pub historical_losses: f64,
    pub contribution_rate: f64,    // % of liquidation fees
}

#[derive(Debug, Clone)]
pub struct InsuranceFundHealth {
    pub is_healthy: bool,
    pub utilization_ratio: f64,
    pub coverage_ratio: f64,
    pub needs_replenishment: bool,
    pub current_balance: f64,
    pub target_balance: f64,
}

pub struct InsuranceFundManager {
    db: DatabasePool,
}

impl InsuranceFundManager {
    pub async fn check_health(&self) -> Result<InsuranceFundHealth, FundError> {
        let fund = self.get_insurance_fund().await?;
        let open_interest = self.get_total_open_interest().await?;
        let target_balance = open_interest * 0.01;

        Ok(InsuranceFundHealth {
            is_healthy: fund.total_balance >= target_balance,
            utilization_ratio: fund.bad_debt_covered / fund.total_balance,
            coverage_ratio: fund.total_balance / open_interest,
            needs_replenishment: fund.total_balance < target_balance * 0.5,
            current_balance: fund.total_balance,
            target_balance,
        })
    }

    pub async fn replenish_if_needed(&self) -> Result<(), FundError> {
        let health = self.check_health().await?;

        if health.needs_replenishment {
            // Increase liquidation fee contribution
            self.increase_fee_contribution().await?;

            // Alert governance
            self.alert_governance(GovernanceAlert {
                alert_type: "INSURANCE_FUND_LOW".to_string(),
                current_balance: health.current_balance,
                target_balance: health.target_balance,
            }).await?;
        }

        Ok(())
    }
}
```

Layer 4: Liquidation Queue

```rust
use tokio::time::{sleep, Duration};

// Prevent liquidation cascades
pub struct LiquidationQueue {
    max_concurrent: usize,
    cooldown_ms: u64,
}

impl LiquidationQueue {
    const MAX_CONCURRENT: usize = 10;
    const COOLDOWN_MS: u64 = 1000;

    pub fn new() -> Self {
        Self {
            max_concurrent: Self::MAX_CONCURRENT,
            cooldown_ms: Self::COOLDOWN_MS,
        }
    }

    pub async fn process_liquidations(&self, mut positions: Vec<Position>) -> Result<(),
LiquidationError> {
        // Sort by urgency (lowest margin ratio first)
        positions.sort_by(|a, b| {
            a.margin_ratio.partial_cmp(&b.margin_ratio).unwrap_or(std::cmp::Ordering::Equal)
        });

        // Process in batches with cooldown
        for chunk in positions.chunks(self.max_concurrent) {
            let futures: Vec<_> = chunk.iter()
                .map(|pos| self.liquidate_position(pos))
                .collect();

            // Process batch concurrently
            let _results = futures::future::join_all(futures).await;

            // Cooldown between batches
            if chunk.len() == self.max_concurrent {
                sleep(Duration::from_millis(self.cooldown_ms)).await;
            }
        }

        Ok(())
    }
}
```

Oracle Manipulation Protection

Price Feed Validation

```rust
pub struct OracleSecurityLayer {
    pyth_client: PythClient,
    switchboard_client: SwitchboardClient,
}

impl OracleSecurityLayer {
    pub async fn get_validated_price(&self, symbol: &str) -> Result<f64, OracleError> {
        // 1. Fetch from multiple sources
        let pyth_price = self.get_pyth_price(symbol).await?;
        let switchboard_price = self.get_switchboard_price(symbol).await?;
        let cex_vwap = self.get_cex_vwap(symbol).await?;

        // 2. Calculate median (resistant to outliers)
        let mut prices = vec![pyth_price, switchboard_price, cex_vwap];
        let median = self.calculate_median(&mut prices);

        // 3. Check deviation tolerance
        const MAX_DEVIATION: f64 = 0.02; // 2%

        for &price in &prices {
            let deviation = (price - median).abs() / median;

            if deviation > MAX_DEVIATION {
                self.alert_oracle_deviation(OracleDeviation {
                    symbol: symbol.to_string(),
                    median,
                    outlier: price,
                    deviation,
                    source: self.identify_source(price, &prices),
                }).await?;

                // Use only reliable sources
                let reliable_prices: Vec<f64> = prices.iter()
                    .filter(|&&p| (p - median).abs() / median <= MAX_DEVIATION)
                    .copied()
                    .collect();
                return Ok(self.calculate_median(&mut reliable_prices.clone()));
            }
        }

        Ok(median)
    }

    pub async fn detect_manipulation(&self, symbol: &str) -> Result<bool, OracleError> {
        let current_price = self.get_price(symbol).await?;
        let historical_prices = self.get_price_history(symbol, 300).await?; // 5 min

        // Check for suspicious patterns
        let avg_price: f64 = historical_prices.iter().sum::<f64>() / historical_prices.len() as
f64;
        let volatility = self.calculate_volatility(&historical_prices);
        let deviation = (current_price - avg_price).abs() / avg_price;

        // Abnormal price movement
        if deviation > volatility * 3.0 {
            self.pause_trading_if_needed(symbol).await?;
            return Ok(true);
        }

        Ok(false)
    }

    fn calculate_median(&self, prices: &mut [f64]) -> f64 {
        prices.sort_by(|a, b| a.partial_cmp(b).unwrap());
        let mid = prices.len() / 2;
        if prices.len() % 2 == 0 {
            (prices[mid - 1] + prices[mid]) / 2.0
        } else {
            prices[mid]
        }
    }
}
```

Circuit Breakers

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct CircuitBreaker {
    pub symbol: String,
    pub price_change_threshold: f64,    // 10% move triggers
    pub volume_change_threshold: f64,   // 5x normal volume
    pub time_window: u64,               // 60 seconds
    pub cooldown_period: u64,           // 300 seconds (5 min)
}

pub struct CircuitBreakerSystem {
    db: DatabasePool,
}

impl CircuitBreakerSystem {
    pub async fn check_and_trigger(&self, symbol: &str) -> Result<(), CircuitBreakerError> {
        let recent = self.get_recent_data(symbol, 60).await?;
        let historical = self.get_historical_average(symbol).await?;

        let price_change = (recent.price - recent.open_price) / recent.open_price;
        let volume_ratio = recent.volume / historical.avg_volume;

        if price_change.abs() > 0.10 || volume_ratio > 5.0 {
            self.trigger_circuit_breaker(symbol).await?;
        }

        Ok(())
    }

    pub async fn trigger_circuit_breaker(&self, symbol: &str) -> Result<(), CircuitBreakerError> {
        // 1. Pause new orders
        self.pause_market(symbol).await?;

        // 2. Allow position closures only
        self.set_reduce_only_mode(symbol).await?;

        // 3. Notify all users
        self.broadcast_alert(Alert {
            alert_type: "CIRCUIT_BREAKER".to_string(),
            symbol: symbol.to_string(),
            reason: "Abnormal price movement detected".to_string(),
            duration: 300, // 5 minutes
        }).await?;

        // 4. Schedule automatic resume
        let symbol_owned = symbol.to_string();
        let self_clone = self.clone();
        tokio::spawn(async move {
            tokio::time::sleep(Duration::from_secs(300)).await;
            if let Err(e) = self_clone.resume_market(&symbol_owned).await {
                eprintln!("Failed to resume market: {:?}", e);
            }
        });

        Ok(())
    }
}
```

Rate Limiting and DDoS Protection

Tiered Rate Limits

```typescript
interface RateLimitTier {
    name: string;
    requestsPerMinute: number;
    websocketMessagesPerSecond: number;
    burstAllowance: number;
    requirements: string[];
}

const RATE_LIMIT_TIERS: RateLimitTier[] = [
    {
        name: 'FREE',
        requestsPerMinute: 60,
        websocketMessagesPerSecond: 10,
        burstAllowance: 10,
        requirements: ['Email verified']
    },
    {
        name: 'BASIC',
        requestsPerMinute: 300,
        websocketMessagesPerSecond: 50,
        burstAllowance: 50,
        requirements: ['1,000+ DARK tokens staked']
    },
    {
        name: 'PRO',
        requestsPerMinute: 1200,
        websocketMessagesPerSecond: 200,
        burstAllowance: 200,
        requirements: ['10,000+ DARK tokens staked']
    },
    {
        name: 'MARKET_MAKER',
        requestsPerMinute: 6000,
        websocketMessagesPerSecond: 1000,
        burstAllowance: 500,
        requirements: ['Application approved', 'Dedicated support']
    }
];
```

DDoS Protection Implementation

```
class DDoSProtection {
    private readonly redis: Redis;

    async checkRateLimit(userId: string, tier: RateLimitTier): Promise<boolean> {
        const key = `ratelimit:${userId}:${Date.now() / 60000}`;
        const count = await this.redis.incr(key);
        await this.redis.expire(key, 60);

        if (count > tier.requestsPerMinute) {
            // Log suspicious activity
            await this.logRateLimitViolation(userId, count, tier);

            // Escalate if repeated violations
            const violations = await this.getRecentViolations(userId);
            if (violations > 5) {
                await this.temporaryBan(userId, 3600); // 1 hour
            }

            return false;
        }

        return true;
    }

    async detectDDoSPattern(): Promise<void> {
        const requestsPerSecond = await this.getRequestsPerSecond();
        const baselineRPS = await this.getBaselineRPS();

        if (requestsPerSecond > baselineRPS * 10) {
            // Potential DDoS attack
            await this.enableStrictMode();
            await this.alertOpsTeam({
                type: 'DDOS_SUSPECTED',
                rps: requestsPerSecond,
                baseline: baselineRPS
            });
        }
    }

    async enableStrictMode(): Promise<void> {
        // Reduce all rate limits by 50%
        // Require CAPTCHA for new connections
        // Enable IP-based blocking
        // Increase WebSocket connection cost
    }
}
```

Emergency Procedures and Circuit Breakers

Emergency Pause Mechanism

```rust
pub struct EmergencyPause {
    pub is_paused: bool,
    pub paused_at: i64,
    pub paused_by: Pubkey,              // Must be multisig
    pub reason: [u8; 128],
    pub affected_operations: Vec<Operation>,
}

pub fn emergency_pause(
    ctx: Context<EmergencyPause>,
    reason: String,
    operations: Vec<Operation>
) -> Result<()> {
    // Verify multisig authority
    require!(
        ctx.accounts.authority.owners.contains(&ctx.accounts.signer.key()),
        ErrorCode::Unauthorized
    );

    let pause = &mut ctx.accounts.emergency_pause;
    pause.is_paused = true;
    pause.paused_at = Clock::get()?.unix_timestamp;
    pause.paused_by = ctx.accounts.signer.key();
    pause.affected_operations = operations;

    // Emit event for immediate notification
    emit!(EmergencyPauseEvent {
        reason: reason.clone(),
        operations: operations.clone(),
        timestamp: pause.paused_at
    });

    Ok(())
}
```

Pause Levels:

Level	Operations Affected	Auto-Resume	Manual Override
Level 1	New orders only	5 minutes	Yes
Level 2	All trading	15 minutes	Yes (multisig)
Level 3	All except withdrawals	Manual only	Yes (multisig)
Level 4	Everything	Manual only	Yes (5/5 multisig)

Bug Bounty Program

Bounty Structure

```typescript
interface BugBounty {
    severity: 'CRITICAL' | 'HIGH' | 'MEDIUM' | 'LOW';
    reward: number;
    criteria: string[];
    examples: string[];
}

const BUG_BOUNTY_PROGRAM: Map<string, BugBounty> = new Map([
    ['CRITICAL', {
        severity: 'CRITICAL',
        reward: 1000000, // $1M USD
        criteria: [
            'Unauthorized fund extraction',
            'Infinite mint exploit',
            'Complete system compromise',
            'Oracle manipulation leading to theft'
        ],
        examples: [
            'Bypass vault withdrawal restrictions',
            'Drain insurance fund',
            'Manipulate position sizes'
        ]
    }],
    ['HIGH', {
        severity: 'HIGH',
        reward: 100000, // $100K USD
        criteria: [
            'Partial fund loss',
            'Position manipulation',
            'Settlement bypass',
            'Liquidation exploit'
        ],
        examples: [
            'Avoid liquidation incorrectly',
            'Manipulate funding rate',
            'Bypass rate limits'
        ]
    }],
    ['MEDIUM', {
        severity: 'MEDIUM',
        reward: 10000, // $10K USD
        criteria: [
            'Denial of service',
            'Information disclosure',
            'Minor economic exploits'
        ],
        examples: [
            'Crash settlement relayer',
            'Access other users orders',
            'Fee manipulation'
        ]
    }],
    ['LOW', {
        severity: 'LOW',
        reward: 1000, // $1K USD
        criteria: [
            'UI bugs affecting trading',
            'Incorrect calculations (non-exploitable)',
            'Minor security issues'
        ],
        examples: [
            'PnL display incorrect',
            'Order submission edge cases',
            'Rate limiting bypass (non-damaging)'
        ]
    }]
]);
```

Submission Process

```
class BugBountySubmission {
    async submitBug(submission: BugReport): Promise<string> {
        // 1. Validate submission
        if (!this.validateSubmission(submission)) {
            throw new Error('Invalid submission format');
        }

        // 2. Create encrypted record
        const submissionId = this.generateId();
        await this.storeEncrypted(submissionId, submission);

        // 3. Notify security team (encrypted)
        await this.notifySecurityTeam(submissionId);

        // 4. Acknowledge submitter
        return submissionId;
    }

    async evaluateAndPay(submissionId: string): Promise<void> {
        const report = await this.getReport(submissionId);

        // Security team evaluation
        const severity = await this.determineSeverity(report);
        const isValid = await this.validateExploit(report);

        if (isValid) {
            const bounty = BUG_BOUNTY_PROGRAM.get(severity);

            // Pay in DARK tokens
            await this.payBounty(report.submitter, bounty.reward);

            // Public disclosure (after fix)
            setTimeout(async () => {
                await this.publishReport(submissionId);
            }, 90 * 24 * 60 * 60 * 1000); // 90 days
        }
    }
}
```

Program Rules:

- Responsible disclosure required
- 90-day embargo before public disclosure
- No public discussion before fix
- First reporter gets full bounty
- Duplicate reports get 10% bounty
- Must not exploit on mainnet
- Must provide working PoC

Security Monitoring Dashboard

```
interface SecurityMetrics {
    // Real-time threats
    activeDDoSAttempts: number;
    suspiciousTransactions: number;
    failedAuthAttempts: number;

    // Position risk
    underwaterPositions: number;
    totalExposure: number;
    insuranceFundRatio: number;

    // Oracle health
    oracleDeviations: number;
    staleOracleFeeds: string[];

    // System health
    settlementFailures: number;
    liquidationQueueDepth: number;
    relayerUptime: number;
}

class SecurityMonitor {
    async getDashboard(): Promise<SecurityMetrics> {
        return {
            activeDDoSAttempts: await this.countDDoSAttempts(),
            suspiciousTransactions: await this.countSuspiciousTx(),
            failedAuthAttempts: await this.countFailedAuth(),
            underwaterPositions: await this.countUnderwaterPositions(),
            totalExposure: await this.calculateTotalExposure(),
            insuranceFundRatio: await this.getInsuranceFundRatio(),
            oracleDeviations: await this.countOracleDeviations(),
            staleOracleFeeds: await this.getStaleOracles(),
            settlementFailures: await this.countSettlementFailures(),
            liquidationQueueDepth: await this.getLiquidationQueueDepth(),
            relayerUptime: await this.getRelayerUptime()
        };
    }
}
```

---

6. Account & Authentication

Email/Password Account Creation Flow

GoDark requires account creation separate from wallet connection to enable multi-device access, API key management, and account recovery.

Registration Process

```rust
use serde::{Deserialize, Serialize};
use validator::Validate;

#[derive(Debug, Deserialize, Validate)]
struct AccountRegistration {
    #[validate(email)]
    email: String,
    #[validate(length(min = 8))]
    password: String,
    confirm_password: String,
    agreed_to_terms: bool,
    referral_code: Option<String>,
}

struct AccountService {
    db: DatabasePool,
    email_service: EmailService,
}

impl AccountService {
    async fn register_account(&self, registration: AccountRegistration) -> Result<Account,
AccountError> {
        // 1. Validate input
        registration.validate()?;
        self.validate_email(&registration.email)?;
        self.validate_password(&registration.password)?;

        if registration.password != registration.confirm_password {
            return Err(AccountError::PasswordMismatch);
        }

        // 2. Check if email already exists
        let email_lower = registration.email.to_lowercase();
        let existing = sqlx::query_as::<_, Account>("SELECT * FROM accounts WHERE email = $1")
            .bind(&email_lower)
            .fetch_optional(&self.db)
            .await?;

        if existing.is_some() {
            return Err(AccountError::EmailExists);
        }

        // 3. Hash password (bcrypt with cost 12)
        let password_hash = bcrypt::hash(&registration.password, 12)?;

        // 4. Generate verification token
        let verification_token = generate_random_token(32);
        let verification_expiry = Utc::now() + Duration::hours(24);

        // 5. Create account
        let account = sqlx::query_as::<_, Account>(
            "INSERT INTO accounts (email, password_hash, verification_token, verification_expiry,
             is_verified, two_factor_enabled, status, created_at)
             VALUES ($1, $2, $3, $4, false, false, 'PENDING_VERIFICATION', $5)
             RETURNING *"
        )
        .bind(&email_lower)
        .bind(&password_hash)
        .bind(&verification_token)
        .bind(verification_expiry)
        .bind(Utc::now())
        .fetch_one(&self.db)
        .await?;

        // 6. Send verification email
        self.email_service.send_verification_email(&account.email, &verification_token).await?;

        // 7. Process referral if provided
        if let Some(referral_code) = registration.referral_code {
            self.process_referral(account.id, &referral_code).await?;
        }

        Ok(account)
    }

    fn validate_email(&self, email: &str) -> Result<(), AccountError> {
        // Validator crate already checks basic format

        // Block disposable email domains
        let disposable_domains = vec!["tempmail.com", "guerrillamail.com", "10minutemail.com"];
        let domain = email.split('@').nth(1).unwrap_or("");
        if disposable_domains.contains(&domain) {
            return Err(AccountError::DisposableEmailNotAllowed);
```

```
        }

        Ok(())
    }

    fn validate_password(&self, password: &str) -> Result<(), AccountError> {
        if password.len() < 8 {
            return Err(AccountError::PasswordTooShort);
        }

        let has_uppercase = password.chars().any(|c| c.is_uppercase());
        let has_lowercase = password.chars().any(|c| c.is_lowercase());
        let has_number = password.chars().any(|c| c.is_numeric());
        let has_special = password.chars().any(|c| "!@#$%^&*(),.?\":{}|<>".contains(c));

        if !has_uppercase || !has_lowercase || !has_number || !has_special {
            return Err(AccountError::PasswordRequirementsNotMet);
        }

        // Check against common passwords
        if self.is_common_password(password) {
            return Err(AccountError::PasswordTooCommon);
        }

        Ok(())
    }
}
```

Email Verification Process

Verification Flow

```rust
struct EmailVerification {
    email_service: EmailService,
    db: DatabasePool,
}

impl EmailVerification {
    async fn send_verification_email(&self, email: &str, token: &str) -> Result<(), EmailError> {
        let verification_link = format!("https://app.godark.xyz/verify-email?token={}", token);

        self.email_service.send(EmailMessage {
            to: email.to_string(),
            subject: "Verify your GoDark account".to_string(),
            template: "email-verification".to_string(),
            data: serde_json::json!({
                "verification_link": verification_link,
                "expiry_hours": 24
            }),
        }).await?;

        Ok(())
    }

    async fn verify_email(&self, token: &str) -> Result<(), EmailError> {
        let account = sqlx::query_as::<_, Account>(
            "SELECT * FROM accounts WHERE verification_token = $1"
        )
        .bind(token)
        .fetch_optional(&self.db)
        .await?
        .ok_or(EmailError::InvalidToken)?;

        // Check expiry
        if Utc::now().timestamp_millis() > account.verification_expiry {
            return Err(EmailError::TokenExpired);
        }

        // Update account
        sqlx::query(
            "UPDATE accounts SET is_verified = true, status = 'ACTIVE',
             verified_at = $1, verification_token = NULL, verification_expiry = NULL
             WHERE id = $2"
        )
        .bind(Utc::now())
        .bind(account.id)
        .execute(&self.db)
        .await?;

        // Send welcome email
        self.send_welcome_email(&account.email).await?;

        Ok(())
    }

    async fn resend_verification(&self, email: &str) -> Result<(), EmailError> {
        let account = sqlx::query_as::<_, Account>(
            "SELECT * FROM accounts WHERE email = $1"
        )
        .bind(email)
        .fetch_optional(&self.db)
        .await?;

        // Don't reveal if email exists
        let Some(account) = account else {
            return Ok(());
        };

        if account.is_verified {
            return Err(EmailError::AlreadyVerified);
        }

        // Generate new token
        let verification_token = generate_random_token(32);
        let verification_expiry = Utc::now() + Duration::hours(24);

        sqlx::query(
            "UPDATE accounts SET verification_token = $1, verification_expiry = $2 WHERE id = $3"
        )
        .bind(&verification_token)
        .bind(verification_expiry)
        .bind(account.id)
        .execute(&self.db)
        .await?;
```

```
            self.send_verification_email(email, &verification_token).await?;

            Ok(())
        }
    }
```

2FA Setup (Authenticator App)

TOTP (Time-Based One-Time Password) Implementation

```rust
use totp_rs::{Algorithm, Secret, TOTP};
use qrcode::QrCode;
use base64::{Engine as _, engine::general_purpose};
use rand::Rng;

pub struct TwoFactorAuth {
    db: DatabasePool,
    encryption_key: Vec<u8>,
}

#[derive(Debug, Serialize)]
pub struct TwoFactorSetup {
    pub secret: String,
    pub qr_code: String,
    pub manual_entry_key: String,
}

impl TwoFactorAuth {
    pub async fn generate_secret(&self, account_id: &str) -> Result<TwoFactorSetup, TwoFactorError>
{
        // Generate secret
        let secret = Secret::generate_secret();
        let secret_base32 = secret.to_encoded().to_string();

        // Create TOTP
        let totp = TOTP::new(
            Algorithm::SHA1,
            6,
            1,
            30,
            secret.to_bytes().unwrap(),
            Some("GoDark DEX".to_string()),
            account_id.to_string(),
        )?;

        // Store encrypted secret temporarily
        let encrypted_secret = self.encrypt(&secret_base32)?;
        let now = chrono::Utc::now().timestamp_millis();

        sqlx::query(
            "INSERT INTO two_factor_setup (account_id, secret, created_at, expires_at,
is_confirmed)
             VALUES ($1, $2, $3, $4, false)"
        )
        .bind(account_id)
        .bind(&encrypted_secret)
        .bind(now)
        .bind(now + (10 * 60 * 1000)) // 10 minutes
        .execute(&self.db)
        .await?;

        // Generate QR code
        let otpauth_url = totp.get_url();
        let qr_code = QrCode::new(otpauth_url)?;
        let qr_image = qr_code.render::<image::Luma<u8>>().build();

        // Convert to data URL
        let mut png_data = Vec::new();
        image::DynamicImage::ImageLuma8(qr_image)
            .write_to(&mut std::io::Cursor::new(&mut png_data), image::ImageOutputFormat::Png)?;
        let qr_code_data_url = format!("{}",
            general_purpose::STANDARD.encode(&png_data));

        Ok(TwoFactorSetup {
            secret: secret_base32.clone(),
            qr_code: qr_code_data_url,
            manual_entry_key: secret_base32,
        })
    }

    pub async fn confirm_two_factor(&self, account_id: &str, token: &str) -> Result<Vec<String>,
TwoFactorError> {
        // Get pending setup
        let setup: TwoFactorSetupRecord = sqlx::query_as(
            "SELECT * FROM two_factor_setup WHERE account_id = $1 AND is_confirmed = false"
        )
        .bind(account_id)
        .fetch_optional(&self.db)
        .await?
        .ok_or(TwoFactorError::NoPendingSetup)?;

        // Check expiry
        if chrono::Utc::now().timestamp_millis() > setup.expires_at {
```

```
            return Err(TwoFactorError::SetupExpired);
        }

        let secret = self.decrypt(&setup.secret)?;

        // Verify token
        let totp = TOTP::new(
            Algorithm::SHA1,
            6,
            1,
            30,
            Secret::Encoded(secret.clone()).to_bytes().unwrap(),
            Some("GoDark DEX".to_string()),
            account_id.to_string(),
        )?;

        if !totp.check_current(token)? {
            return Err(TwoFactorError::InvalidCode);
        }

        // Generate backup codes
        let backup_codes = self.generate_backup_codes();
        let hashed_backup_codes: Vec<String> = backup_codes.iter()
            .map(|code| bcrypt::hash(code, 10).unwrap())
            .collect();

        // Enable 2FA for account
        sqlx::query(
            "UPDATE accounts SET two_factor_enabled = true, two_factor_secret = $1,
             backup_codes = $2, two_factor_enabled_at = $3 WHERE id = $4"
        )
        .bind(self.encrypt(&secret)?)
        .bind(&hashed_backup_codes)
        .bind(chrono::Utc::now().timestamp_millis())
        .bind(account_id)
        .execute(&self.db)
        .await?;

        // Clean up setup
        sqlx::query("DELETE FROM two_factor_setup WHERE account_id = $1")
            .bind(account_id)
            .execute(&self.db)
            .await?;

        Ok(backup_codes)
    }

    pub async fn verify_two_factor_token(&self, account_id: &str, token: &str) -> Result<bool,
TwoFactorError> {
        let account: Account = sqlx::query_as("SELECT * FROM accounts WHERE id = $1")
            .bind(account_id)
            .fetch_one(&self.db)
            .await?;

        if !account.two_factor_enabled {
            return Ok(true); // 2FA not enabled
        }

        let secret = self.decrypt(&account.two_factor_secret)?;

        // Check if it's a backup code
        for hashed_backup_code in &account.backup_codes {
            if bcrypt::verify(token, hashed_backup_code)? {
                // Remove used backup code
                sqlx::query("UPDATE accounts SET backup_codes = array_remove(backup_codes, $1)
WHERE id = $2")
                    .bind(hashed_backup_code)
                    .bind(account_id)
                    .execute(&self.db)
                    .await?;
                return Ok(true);
            }
        }

        // Verify TOTP token
        let totp = TOTP::new(
            Algorithm::SHA1,
            6,
            1,
            30,
            Secret::Encoded(secret).to_bytes().unwrap(),
            Some("GoDark DEX".to_string()),
            account_id.to_string(),
        )?;
```

```
        Ok(totp.check_current(token)?)
    }

    fn generate_backup_codes(&self) -> Vec<String> {
        let mut rng = rand::thread_rng();
        (0..10).map(|_| {
            let bytes: Vec<u8> = (0..4).map(|_| rng.gen()).collect();
            hex::encode_upper(&bytes)
        }).collect()
    }

    pub async fn disable_two_factor(&self, account_id: &str, password: &str, token: &str) ->
Result<(), TwoFactorError> {
        let account: Account = sqlx::query_as("SELECT * FROM accounts WHERE id = $1")
            .bind(account_id)
            .fetch_one(&self.db)
            .await?;

        // Verify password
        if !bcrypt::verify(password, &account.password_hash)? {
            return Err(TwoFactorError::InvalidPassword);
        }

        // Verify 2FA token
        if !self.verify_two_factor_token(account_id, token).await? {
            return Err(TwoFactorError::InvalidCode);
        }

        // Disable 2FA
        sqlx::query(
            "UPDATE accounts SET two_factor_enabled = false, two_factor_secret = NULL, backup_codes
= NULL WHERE id = $1"
        )
        .bind(account_id)
        .execute(&self.db)
        .await?;

        Ok(())
    }
}
```

Wallet Linking (One Per Account)

```
class WalletLinking {
    async linkWallet(accountId: string, walletAddress: string, signature: string): Promise<void> {
        // 1. Verify account exists
        const account = await db.accounts.findOne({ _id: accountId });
        if (!account) {
            throw new Error('Account not found');
        }

        // 2. Check if wallet already linked
        if (account.linkedWallet) {
            throw new Error('Account already has a linked wallet. Unlink first.');
        }

        // 3. Check if wallet is linked to another account
        const existingLink = await db.accounts.findOne({ linkedWallet: walletAddress });
        if (existingLink) {
            throw new Error('This wallet is already linked to another account');
        }

        // 4. Verify wallet ownership
        const message = `Link wallet to GoDark account\nAccount: ${account.email}\nTimestamp:
${Date.now()}`;
        const isValid = await this.verifySignature(walletAddress, message, signature);

        if (!isValid) {
            throw new Error('Invalid signature');
        }

        // 5. Link wallet
        await db.accounts.updateOne(
            { _id: accountId },
            {
                $set: {
                    linkedWallet: walletAddress,
                    walletLinkedAt: Date.now()
                }
            }
        );

        // 6. Create vault if doesn't exist
        await this.createVaultIfNeeded(walletAddress);
    }

    async unlinkWallet(accountId: string, password: string, twoFactorToken?: string): Promise<void>
{
        const account = await db.accounts.findOne({ _id: accountId });

        // Verify password
        const passwordValid = await bcrypt.compare(password, account.passwordHash);
        if (!passwordValid) {
            throw new Error('Invalid password');
        }

        // Verify 2FA if enabled
        if (account.twoFactorEnabled) {
            if (!twoFactorToken) {
                throw new Error('2FA code required');
            }
            const twoFactorValid = await this.twoFactorAuth.verifyTwoFactorToken(accountId,
twoFactorToken);
            if (!twoFactorValid) {
                throw new Error('Invalid 2FA code');
            }
        }

        // Check for open positions
        const openPositions = await this.getOpenPositions(account.linkedWallet);
        if (openPositions.length > 0) {
            throw new Error('Close all positions before unlinking wallet');
        }

        // Unlink wallet
        await db.accounts.updateOne(
            { _id: accountId },
            {
                $unset: {
                    linkedWallet: '',
                    walletLinkedAt: ''
                }
            }
        );
    }
```

```
    private async verifySignature(address: string, message: string, signature: string):
Promise<boolean> {
        try {
            const publicKey = new PublicKey(address);
            const messageBytes = new TextEncoder().encode(message);
            const signatureBytes = bs58.decode(signature);

            return nacl.sign.detached.verify(
                messageBytes,
                signatureBytes,
                publicKey.toBytes()
            );
        } catch (error) {
            return false;
        }
    }
}
```

Session Management (JWT Tokens)

```
interface JWTPayload {
    accountId: string;
    email: string;
    walletAddress?: string;
    role: string;
    iat: number;      // Issued at
    exp: number;      // Expiry
}

class SessionManager {
    private readonly JWT_SECRET = process.env.JWT_SECRET;
    private readonly ACCESS_TOKEN_EXPIRY = 15 * 60; // 15 minutes
    private readonly REFRESH_TOKEN_EXPIRY = 7 * 24 * 60 * 60; // 7 days

    async createSession(accountId: string): Promise<SessionTokens> {
        const account = await db.accounts.findOne({ _id: accountId });

        // Create access token
        const accessToken = jwt.sign(
            {
                accountId: account._id,
                email: account.email,
                walletAddress: account.linkedWallet,
                role: 'USER'
            },
            this.JWT_SECRET,
            { expiresIn: this.ACCESS_TOKEN_EXPIRY }
        );

        // Create refresh token
        const refreshToken = crypto.randomBytes(32).toString('hex');
        const refreshTokenHash = await bcrypt.hash(refreshToken, 10);

        // Store refresh token
        await db.refreshTokens.insert({
            accountId: account._id,
            tokenHash: refreshTokenHash,
            createdAt: Date.now(),
            expiresAt: Date.now() + (this.REFRESH_TOKEN_EXPIRY * 1000),
            ipAddress: this.getClientIP(),
            userAgent: this.getUserAgent()
        });

        return {
            accessToken,
            refreshToken,
            expiresIn: this.ACCESS_TOKEN_EXPIRY
        };
    }

    async refreshAccessToken(refreshToken: string): Promise<string> {
        // Find matching refresh token
        const tokens = await db.refreshTokens.find({
            expiresAt: { $gt: Date.now() }
        });

        let matchedToken = null;
        for (const token of tokens) {
            if (await bcrypt.compare(refreshToken, token.tokenHash)) {
                matchedToken = token;
                break;
            }
        }

        if (!matchedToken) {
            throw new Error('Invalid refresh token');
        }

        // Get account
        const account = await db.accounts.findOne({ _id: matchedToken.accountId });

        // Create new access token
        const accessToken = jwt.sign(
            {
                accountId: account._id,
                email: account.email,
                walletAddress: account.linkedWallet,
                role: 'USER'
            },
            this.JWT_SECRET,
            { expiresIn: this.ACCESS_TOKEN_EXPIRY }
        );
```

```
        return accessToken;
    }

    async revokeSession(refreshToken: string): Promise<void> {
        const tokens = await db.refreshTokens.find({});

        for (const token of tokens) {
            if (await bcrypt.compare(refreshToken, token.tokenHash)) {
                await db.refreshTokens.deleteOne({ _id: token._id });
                return;
            }
        }
    }

    async revokeAllSessions(accountId: string): Promise<void> {
        await db.refreshTokens.deleteMany({ accountId });
    }

    verifyAccessToken(token: string): JWTPayload {
        try {
            return jwt.verify(token, this.JWT_SECRET) as JWTPayload;
        } catch (error) {
            throw new Error('Invalid or expired token');
        }
    }
}
```

API Key Generation and Management

```
interface APIKey {
    id: string;
    accountId: string;
    name: string;
    apiKey: string;           // Public
    secretKeyHash: string;    // Hashed
    passphrase: string;       // User-provided
    ipWhitelist: string[];
    permissions: string[];
    createdAt: number;
    lastUsedAt: number;
    expiresAt?: number;
    isActive: boolean;
}

class APIKeyManager {
    async generateAPIKey(accountId: string, config: APIKeyConfig): Promise<APIKeyCredentials> {
        // 1. Validate API key limit (max 5 per account)
        const existingKeys = await db.apiKeys.count({ accountId, isActive: true });
        if (existingKeys >= 5) {
            throw new Error('Maximum 5 API keys per account');
        }

        // 2. Generate credentials
        const apiKey = `gq_${crypto.randomBytes(16).toString('hex')}`;
        const secretKey = crypto.randomBytes(32).toString('hex');
        const secretKeyHash = await bcrypt.hash(secretKey, 12);

        // 3. Validate passphrase
        if (!config.passphrase || config.passphrase.length < 8) {
            throw new Error('Passphrase must be at least 8 characters');
        }

        // 4. Store API key
        const keyRecord: APIKey = {
            id: crypto.randomUUID(),
            accountId,
            name: config.name,
            apiKey,
            secretKeyHash,
            passphrase: config.passphrase,
            ipWhitelist: config.ipWhitelist || [],
            permissions: config.permissions || ['READ', 'TRADE'],
            createdAt: Date.now(),
            lastUsedAt: null,
            expiresAt: config.expiresAt,
            isActive: true
        };

        await db.apiKeys.insert(keyRecord);

        // 5. Return credentials (secret shown only once)
        return {
            apiKey,
            secretKey, // ⚠️ Show only once
            passphrase: config.passphrase,
            permissions: keyRecord.permissions
        };
    }

    async authenticateAPIRequest(apiKey: string, signature: string, timestamp: number):
Promise<APIKey> {
        // 1. Find API key
        const keyRecord = await db.apiKeys.findOne({ apiKey, isActive: true });
        if (!keyRecord) {
            throw new Error('Invalid API key');
        }

        // 2. Check expiry
        if (keyRecord.expiresAt && Date.now() > keyRecord.expiresAt) {
            throw new Error('API key expired');
        }

        // 3. Check timestamp (prevent replay attacks)
        const now = Date.now();
        if (Math.abs(now - timestamp) > 5000) { // 5 second window
            throw new Error('Request timestamp too old');
        }

        // 4. Verify IP whitelist
        if (keyRecord.ipWhitelist.length > 0) {
            const clientIP = this.getClientIP();
            if (!keyRecord.ipWhitelist.includes(clientIP)) {
```

```
                throw new Error('IP address not whitelisted');
            }
        }

        // 5. Verify signature
        // Expected format: HMAC-SHA256(timestamp + method + path + body, secretKey)
        const message = `${timestamp}${this.method}${this.path}${this.body}`;

        // We can't directly verify without the secret, so we check against stored hash
        // In practice, secret would be stored encrypted and decrypted for verification
        const isValid = await this.verifyHMAC(message, signature, keyRecord.secretKeyHash);

        if (!isValid) {
            throw new Error('Invalid signature');
        }

        // 6. Update last used
        await db.apiKeys.updateOne(
            { _id: keyRecord.id },
            { $set: { lastUsedAt: Date.now() } }
        );

        return keyRecord;
    }

    async updateAPIKey(keyId: string, updates: Partial<APIKey>): Promise<void> {
        // Only allow updating: name, ipWhitelist, permissions
        const allowedUpdates = ['name', 'ipWhitelist', 'permissions'];
        const filteredUpdates = Object.keys(updates)
            .filter(key => allowedUpdates.includes(key))
            .reduce((obj, key) => {
                obj[key] = updates[key];
                return obj;
            }, {});

        await db.apiKeys.updateOne(
            { id: keyId },
            { $set: filteredUpdates }
        );
    }

    async deleteAPIKey(keyId: string, accountId: string, password: string): Promise<void> {
        // Verify password
        const account = await db.accounts.findOne({ _id: accountId });
        const passwordValid = await bcrypt.compare(password, account.passwordHash);

        if (!passwordValid) {
            throw new Error('Invalid password');
        }

        // Soft delete (deactivate)
        await db.apiKeys.updateOne(
            { id: keyId, accountId },
            {
                $set: {
                    isActive: false,
                    deletedAt: Date.now()
                }
            }
        );
    }
}
```

Password Reset and Account Recovery

```
class PasswordReset {
    async requestPasswordReset(email: string): Promise<void> {
        const account = await db.accounts.findOne({ email: email.toLowerCase() });

        // Don't reveal if email exists
        if (!account) {
            return;
        }

        // Generate reset token
        const resetToken = crypto.randomBytes(32).toString('hex');
        const resetTokenHash = await bcrypt.hash(resetToken, 10);
        const resetExpiry = Date.now() + (1 * 60 * 60 * 1000); // 1 hour

        // Store reset token
        await db.accounts.updateOne(
            { _id: account._id },
            {
                $set: {
                    resetToken: resetTokenHash,
                    resetExpiry
                }
            }
        );

        // Send reset email
        const resetLink = `https://app.godark.xyz/reset-password?token=${resetToken}`;
        await this.emailService.send({
            to: email,
            subject: 'Reset your GoDark password',
            template: 'password-reset',
            data: { resetLink, expiryHours: 1 }
        });
    }

    async resetPassword(token: string, newPassword: string): Promise<void> {
        // Find account with valid token
        const accounts = await db.accounts.find({
            resetExpiry: { $gt: Date.now() }
        });

        let matchedAccount = null;
        for (const account of accounts) {
            if (account.resetToken && await bcrypt.compare(token, account.resetToken)) {
                matchedAccount = account;
                break;
            }
        }

        if (!matchedAccount) {
            throw new Error('Invalid or expired reset token');
        }

        // Validate new password
        this.validatePassword(newPassword);

        // Hash new password
        const passwordHash = await bcrypt.hash(newPassword, 12);

        // Update password
        await db.accounts.updateOne(
            { _id: matchedAccount._id },
            {
                $set: { passwordHash },
                $unset: { resetToken: '', resetExpiry: '' }
            }
        );

        // Revoke all sessions
        await this.sessionManager.revokeAllSessions(matchedAccount._id);

        // Send confirmation email
        await this.emailService.send({
            to: matchedAccount.email,
            subject: 'Your password has been reset',
            template: 'password-reset-confirmation'
        });
    }
}
```

Account Deletion Process

```
class AccountDeletion {
    async requestAccountDeletion(accountId: string, password: string, reason?: string):
Promise<void> {
        const account = await db.accounts.findOne({ _id: accountId });

        // Verify password
        const passwordValid = await bcrypt.compare(password, account.passwordHash);
        if (!passwordValid) {
            throw new Error('Invalid password');
        }

        // Check for open positions
        if (account.linkedWallet) {
            const openPositions = await this.getOpenPositions(account.linkedWallet);
            if (openPositions.length > 0) {
                throw new Error('Close all positions before deleting account');
            }

            // Check for locked funds
            const vault = await this.getVault(account.linkedWallet);
            if (vault.used_amount > 0) {
                throw new Error('Withdraw all funds before deleting account');
            }
        }

        // Mark for deletion (7-day grace period)
        await db.accounts.updateOne(
            { _id: accountId },
            {
                $set: {
                    status: 'PENDING_DELETION',
                    deletionRequestedAt: Date.now(),
                    deletionScheduledFor: Date.now() + (7 * 24 * 60 * 60 * 1000),
                    deletionReason: reason
                }
            }
        );

        // Send confirmation email
        await this.emailService.send({
            to: account.email,
            subject: 'Account deletion scheduled',
            template: 'account-deletion-scheduled',
            data: {
                deletionDate: new Date(Date.now() + (7 * 24 * 60 * 60 * 1000)),
                cancelLink: 'https://app.godark.xyz/cancel-deletion'
            }
        });
    }

    async cancelAccountDeletion(accountId: string): Promise<void> {
        await db.accounts.updateOne(
            { _id: accountId },
            {
                $set: { status: 'ACTIVE' },
                $unset: {
                    deletionRequestedAt: '',
                    deletionScheduledFor: '',
                    deletionReason: ''
                }
            }
        );
    }

    async executeAccountDeletion(accountId: string): Promise<void> {
        const account = await db.accounts.findOne({ _id: accountId });

        if (account.status !== 'PENDING_DELETION') {
            throw new Error('Account not scheduled for deletion');
        }

        if (Date.now() < account.deletionScheduledFor) {
            throw new Error('Deletion grace period not elapsed');
        }

        // 1. Anonymize personal data
        await db.accounts.updateOne(
            { _id: accountId },
            {
                $set: {
                    email: `deleted_${accountId}@deleted.local`,
                    status: 'DELETED',
                    deletedAt: Date.now()
```

```
                },
                $unset: {
                    passwordHash: '',
                    twoFactorSecret: '',
                    backupCodes: '',
                    linkedWallet: '',
                    resetToken: '',
                    verificationToken: ''
                }
            }
        );

        // 2. Delete API keys
        await db.apiKeys.deleteMany({ accountId });

        // 3. Delete sessions
        await db.refreshTokens.deleteMany({ accountId });

        // 4. Anonymize trade history (keep for analytics)
        await db.trades.updateMany(
            { userId: accountId },
            { $set: { userId: 'DELETED_USER' } }
        );

        // 5. Delete vault if exists
        if (account.linkedWallet) {
            await this.cleanupVault(account.linkedWallet);
        }
    }
}
```

Activity Logging

```
interface ActivityLog {
    accountId: string;
    action: string;
    timestamp: number;
    ipAddress: string;
    userAgent: string;
    deviceInfo: DeviceInfo;
    location?: GeoLocation;
    success: boolean;
    metadata?: any;
}

class ActivityLogger {
    async logActivity(accountId: string, action: string, metadata?: any): Promise<void> {
        const log: ActivityLog = {
            accountId,
            action,
            timestamp: Date.now(),
            ipAddress: this.getClientIP(),
            userAgent: this.getUserAgent(),
            deviceInfo: this.parseDeviceInfo(),
            location: await this.getGeoLocation(this.getClientIP()),
            success: true,
            metadata
        };

        await db.activityLogs.insert(log);

        // Check for suspicious activity
        await this.checkSuspiciousActivity(accountId, log);
    }

    async getLastLogin(accountId: string): Promise<ActivityLog> {
        return await db.activityLogs.findOne(
            { accountId, action: 'LOGIN', success: true },
            { sort: { timestamp: -1 } }
        );
    }

    async getRecentActivity(accountId: string, limit: number = 20): Promise<ActivityLog[]> {
        return await db.activityLogs.find(
            { accountId },
            { sort: { timestamp: -1 }, limit }
        );
    }

    private async checkSuspiciousActivity(accountId: string, log: ActivityLog): Promise<void> {
        // Check for multiple failed login attempts
        if (log.action === 'LOGIN' && !log.success) {
            const recentFailed = await db.activityLogs.count({
                accountId,
                action: 'LOGIN',
                success: false,
                timestamp: { $gt: Date.now() - (15 * 60 * 1000) } // Last 15 min
            });

            if (recentFailed >= 5) {
                await this.lockAccount(accountId, '15 minutes');
                await this.alertUser(accountId, 'ACCOUNT_LOCKED');
            }
        }

        // Check for login from new location
        const recentLogins = await db.activityLogs.find({
            accountId,
            action: 'LOGIN',
            success: true,
            timestamp: { $gt: Date.now() - (30 * 24 * 60 * 60 * 1000) } // Last 30 days
        });

        const knownLocations = recentLogins.map(l => l.location?.country);

        if (log.location && !knownLocations.includes(log.location.country)) {
            await this.alertUser(accountId, 'NEW_LOCATION_LOGIN', { location: log.location });
        }
    }
}
```

7. Data Architecture

On-Chain State Management

All critical trading state is stored on the Solana blockchain for transparency, immutability, and non-custodial guarantees.

Position Accounts

```
// PDA: [b"position", user.key(), market.key()]
#[account]
pub struct UserPosition {
    // Identity
    pub owner: Pubkey,                  // 32 bytes
    pub parent_wallet: Pubkey,          // 32 bytes (if ephemeral)
    pub market: Pubkey,                 // 32 bytes

    // Position data
    pub size: i64,                      // 8 bytes (signed for long/short)
    pub entry_price: u64,               // 8 bytes (scaled by 1e6)
    pub collateral: u64,                // 8 bytes (USDT, scaled by 1e6)
    pub leverage: u16,                  // 2 bytes

    // Risk metrics
    pub liquidation_price: u64,         // 8 bytes
    pub maintenance_margin: u64,        // 8 bytes

    // PnL tracking
    pub realized_pnl: i64,              // 8 bytes
    pub unrealized_pnl: i64,            // 8 bytes

    // Funding tracking
    pub funding_index: i64,             // 8 bytes
    pub accumulated_funding: i64,       // 8 bytes
    pub last_funding_update: i64,       // 8 bytes

    // Timestamps
    pub open_timestamp: i64,            // 8 bytes
    pub last_update_timestamp: i64,     // 8 bytes

    pub bump: u8,                       // 1 byte
}
// Total: ~233 bytes + padding = 256 bytes

// Rent calculation:
// 256 bytes = 0.00179088 SOL (~$0.30 at $170/SOL)
```

Vault Accounts

```
// PDA: [b"vault", user.key()]
#[account]
pub struct EphemeralVault {
    pub user_wallet: Pubkey,            // 32 bytes
    pub vault_pda: Pubkey,              // 32 bytes
    pub created_at: i64,                // 8 bytes
    pub last_activity: i64,             // 8 bytes

    // Delegate approval tracking
    pub approved_amount: u64,           // 8 bytes (USDT)
    pub used_amount: u64,               // 8 bytes
    pub available_amount: u64,          // 8 bytes

    // Status
    pub is_active: bool,                // 1 byte
    pub bump: u8,                       // 1 byte
}
// Total: ~106 bytes = 128 bytes allocated

// Associated token account for USDT: additional ~165 bytes
```

Settlement Batch Accounts

```
// PDA: [b"batch", batch_id]
#[account]
pub struct SettlementBatch {
    pub batch_id: [u8; 32],              // 32 bytes
    pub relayer: Pubkey,                 // 32 bytes
    pub timestamp: i64,                  // 8 bytes
    pub trade_count: u16,                // 2 bytes
    pub merkle_root: [u8; 32],           // 32 bytes
    pub status: SettlementStatus,        // 1 byte
    pub bump: u8,                        // 1 byte
}
// Total: ~108 bytes = 128 bytes allocated

#[derive(AnchorSerialize, AnchorDeserialize, Clone, Copy)]
pub enum SettlementStatus {
    Pending,
    Confirmed,
    Failed,
}
```

Market Configuration Accounts

```
// PDA: [b"market", symbol_hash]
#[account]
pub struct PerpMarket {
    pub authority: Pubkey,               // 32 bytes
    pub market_id: Pubkey,               // 32 bytes
    pub symbol: [u8; 32],                // 32 bytes
    pub base_asset: [u8; 16],            // 16 bytes
    pub quote_asset: [u8; 16],           // 16 bytes

    // Vault and oracle
    pub usdt_vault: Pubkey,              // 32 bytes
    pub price_oracle: Pubkey,            // 32 bytes

    // Market parameters
    pub max_leverage: u16,               // 2 bytes
    pub maintenance_margin_ratio: u16,   // 2 bytes
    pub initial_margin_ratio: u16,       // 2 bytes
    pub maker_fee: i16,                  // 2 bytes
    pub taker_fee: u16,                  // 2 bytes

    // Market state
    pub funding_rate: i64,               // 8 bytes
    pub last_funding_update: i64,        // 8 bytes
    pub total_open_interest: u64,        // 8 bytes
    pub total_long_interest: u64,        // 8 bytes
    pub total_short_interest: u64,       // 8 bytes

    // Insurance and fees
    pub insurance_fund: Pubkey,          // 32 bytes
    pub fee_recipient: Pubkey,           // 32 bytes

    // Status
    pub is_active: bool,                 // 1 byte
    pub bump: u8,                        // 1 byte
}
// Total: ~328 bytes = 384 bytes allocated
```

Off-Chain Storage

Off-chain databases handle high-frequency data that doesn't require blockchain immutability.

PostgreSQL Schema

Accounts Table:

```
CREATE TABLE accounts (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    is_verified BOOLEAN DEFAULT FALSE,
    verification_token VARCHAR(64),
    verification_expiry BIGINT,
    two_factor_enabled BOOLEAN DEFAULT FALSE,
    two_factor_secret TEXT,
    backup_codes TEXT[],
    linked_wallet VARCHAR(44),
    status VARCHAR(50) DEFAULT 'PENDING_VERIFICATION',
    created_at BIGINT NOT NULL,
    verified_at BIGINT,
    wallet_linked_at BIGINT,
    two_factor_enabled_at BIGINT,

    INDEX idx_email (email),
    INDEX idx_linked_wallet (linked_wallet),
    INDEX idx_status (status)
);
```

Orders Table:

```
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES accounts(id),
    wallet_address VARCHAR(44) NOT NULL,
    is_ephemeral BOOLEAN DEFAULT FALSE,

    -- Order details
    symbol VARCHAR(32) NOT NULL,
    side VARCHAR(4) NOT NULL, -- BUY/SELL
    order_type VARCHAR(20) NOT NULL,
    size DECIMAL(20, 8) NOT NULL,
    limit_price DECIMAL(20, 6),

    -- Time in force
    time_in_force VARCHAR(10) NOT NULL,
    expiry_time BIGINT,

    -- Order attributes
    all_or_none BOOLEAN DEFAULT FALSE,
    min_quantity DECIMAL(20, 8),
    nbbo_protection BOOLEAN DEFAULT FALSE,

    -- Status
    status VARCHAR(20) NOT NULL,
    filled_size DECIMAL(20, 8) DEFAULT 0,
    avg_fill_price DECIMAL(20, 6) DEFAULT 0,

    -- Timestamps
    created_at BIGINT NOT NULL,
    updated_at BIGINT,
    completed_at BIGINT,

    INDEX idx_user_symbol (user_id, symbol),
    INDEX idx_status (status),
    INDEX idx_created_at (created_at),
    INDEX idx_symbol_status (symbol, status)
);
```

Trades Table:

```
CREATE TABLE trades (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    symbol VARCHAR(32) NOT NULL,

    -- Counterparties
    buyer_id UUID NOT NULL REFERENCES accounts(id),
    seller_id UUID NOT NULL REFERENCES accounts(id),
    buy_order_id UUID NOT NULL REFERENCES orders(id),
    sell_order_id UUID NOT NULL REFERENCES orders(id),

    -- Trade details
    price DECIMAL(20, 6) NOT NULL,
    size DECIMAL(20, 8) NOT NULL,
    buyer_fee DECIMAL(20, 6) NOT NULL,
    seller_fee DECIMAL(20, 6) NOT NULL,

    -- Settlement
    status VARCHAR(20) NOT NULL,
    batch_id VARCHAR(64),
    tx_signature VARCHAR(88),
    settled_at BIGINT,

    -- Timestamp
    executed_at BIGINT NOT NULL,

    INDEX idx_buyer (buyer_id, executed_at),
    INDEX idx_seller (seller_id, executed_at),
    INDEX idx_symbol (symbol, executed_at),
    INDEX idx_batch (batch_id),
    INDEX idx_status (status)
);
```

Positions Table (Mirror of On-Chain):

```
CREATE TABLE positions (
    id UUID PRIMARY KEY,
    user_id UUID NOT NULL REFERENCES accounts(id),
    wallet_address VARCHAR(44) NOT NULL,
    market_address VARCHAR(44) NOT NULL,
    symbol VARCHAR(32) NOT NULL,

    -- Position data (synced from chain)
    size DECIMAL(20, 8) NOT NULL,
    entry_price DECIMAL(20, 6) NOT NULL,
    collateral DECIMAL(20, 6) NOT NULL,
    leverage SMALLINT NOT NULL,
    liquidation_price DECIMAL(20, 6) NOT NULL,

    -- PnL
    realized_pnl DECIMAL(20, 6) DEFAULT 0,
    unrealized_pnl DECIMAL(20, 6) DEFAULT 0,

    -- Funding
    accumulated_funding DECIMAL(20, 6) DEFAULT 0,
    last_funding_update BIGINT,

    -- Status
    status VARCHAR(20) NOT NULL,
    open_timestamp BIGINT NOT NULL,
    last_update_timestamp BIGINT,
    closed_at BIGINT,

    -- Sync tracking
    last_synced_at BIGINT,
    on_chain_signature VARCHAR(88),

    INDEX idx_user_symbol (user_id, symbol),
    INDEX idx_wallet (wallet_address),
    INDEX idx_status (status),
    INDEX idx_open_timestamp (open_timestamp)
);
```

Funding Rate History:

```
CREATE TABLE funding_rates (
    id BIGSERIAL PRIMARY KEY,
    symbol VARCHAR(32) NOT NULL,
    timestamp BIGINT NOT NULL,
    funding_rate DECIMAL(20, 10) NOT NULL,
    premium_index DECIMAL(20, 10) NOT NULL,
    mark_price DECIMAL(20, 6) NOT NULL,
    index_price DECIMAL(20, 6) NOT NULL,

    INDEX idx_symbol_timestamp (symbol, timestamp),
    UNIQUE (symbol, timestamp)
);

-- Partition by month for performance
CREATE TABLE funding_rates_2025_01 PARTITION OF funding_rates
    FOR VALUES FROM (1704067200000) TO (1706745600000);
```

API Keys Table:

```
CREATE TABLE api_keys (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    account_id UUID NOT NULL REFERENCES accounts(id),
    name VARCHAR(100) NOT NULL,
    api_key VARCHAR(64) UNIQUE NOT NULL,
    secret_key_hash VARCHAR(255) NOT NULL,
    passphrase VARCHAR(255) NOT NULL,
    ip_whitelist TEXT[],
    permissions TEXT[] NOT NULL,
    created_at BIGINT NOT NULL,
    last_used_at BIGINT,
    expires_at BIGINT,
    is_active BOOLEAN DEFAULT TRUE,

    INDEX idx_account (account_id),
    INDEX idx_api_key (api_key)
);
```

Activity Logs:

```
CREATE TABLE activity_logs (
    id BIGSERIAL PRIMARY KEY,
    account_id UUID NOT NULL REFERENCES accounts(id),
    action VARCHAR(50) NOT NULL,
    timestamp BIGINT NOT NULL,
    ip_address INET NOT NULL,
    user_agent TEXT,
    device_info JSONB,
    location JSONB,
    success BOOLEAN NOT NULL,
    metadata JSONB,

    INDEX idx_account_timestamp (account_id, timestamp),
    INDEX idx_action (action),
    INDEX idx_timestamp (timestamp)
);

-- Partition by week
CREATE TABLE activity_logs_2025_w01 PARTITION OF activity_logs
    FOR VALUES FROM (1704067200000) TO (1704672000000);
```

Redis Data Structures

Order Book (Per Symbol)

```
// Redis sorted sets for price-time priority
const orderBookStructure = {
    // Buy orders sorted by price (highest first)
    [`orderbook:${symbol}:buy`]: {
        type: 'ZSET',
        score: 'price',
        members: 'orderId'
    },

    // Sell orders sorted by price (lowest first)
    [`orderbook:${symbol}:sell`]: {
        type: 'ZSET',
        score: 'price',
        members: 'orderId'
    },

    // Order details
    [`order:${orderId}`]: {
        type: 'HASH',
        fields: {
            userId: 'uuid',
            symbol: 'string',
            side: 'BUY|SELL',
            orderType: 'string',
            size: 'number',
            limitPrice: 'number',
            filledSize: 'number',
            status: 'string',
            timestamp: 'number'
        },
        ttl: 86400 // 24 hours
    }
};
```

Price Cache

```
// Latest prices for each symbol
const priceCache = {
    [`price:${symbol}:mark`]: {
        type: 'STRING',
        value: 'number',
        ttl: 1 // 1 second
    },

    [`price:${symbol}:index`]: {
        type: 'STRING',
        value: 'number',
        ttl: 1
    },

    // Price history (last 300 seconds)
    [`price:${symbol}:history`]: {
        type: 'ZSET',
        score: 'timestamp',
        members: 'price',
        maxLen: 300
    }
};
```

Session Cache

```
const sessionCache = {
    [`session:${accessToken}`]: {
        type: 'HASH',
        fields: {
            accountId: 'uuid',
            email: 'string',
            walletAddress: 'string',
            role: 'string'
        },
        ttl: 900 // 15 minutes (access token expiry)
    },

    [`user:${accountId}:positions`]: {
        type: 'HASH',
        fields: {
            [symbol]: 'positionData'
        },
        ttl: 60 // 1 minute
    }
};
```

Rate Limiting

```
const rateLimiting = {
    [`ratelimit:${userId}:${minute}`]: {
        type: 'STRING',
        value: 'count',
        ttl: 60
    },

    [`ratelimit:ws:${userId}:${second}`]: {
        type: 'STRING',
        value: 'count',
        ttl: 1
    }
};
```

Real-Time Data Feeds (WebSocket Architecture)

WebSocket Server Structure

```
class WebSocketServer {
    private connections: Map<string, WebSocket> = new Map();
    private subscriptions: Map<string, Set<string>> = new Map();
    private redis: Redis;

    async handleConnection(ws: WebSocket, request: Request): Promise<void> {
        const connectionId = crypto.randomUUID();
        this.connections.set(connectionId, ws);

        ws.on('message', async (message) => {
            await this.handleMessage(connectionId, message);
        });

        ws.on('close', () => {
            this.handleDisconnect(connectionId);
        });
    }

    async handleMessage(connectionId: string, message: any): Promise<void> {
        const data = JSON.parse(message);

        switch (data.type) {
            case 'subscribe':
                await this.handleSubscribe(connectionId, data.channels);
                break;
            case 'unsubscribe':
                await this.handleUnsubscribe(connectionId, data.channels);
                break;
            case 'ping':
                this.send(connectionId, { type: 'pong' });
                break;
        }
    }

    async handleSubscribe(connectionId: string, channels: string[]): Promise<void> {
        for (const channel of channels) {
            if (!this.subscriptions.has(channel)) {
                this.subscriptions.set(channel, new Set());

                // Subscribe to Redis pub/sub
                await this.redis.subscribe(channel);
            }

            this.subscriptions.get(channel)!.add(connectionId);
        }
    }

    async publishUpdate(channel: string, data: any): Promise<void> {
        // Publish to Redis (fan-out to all WS servers)
        await this.redis.publish(channel, JSON.stringify(data));
    }

    private async handleRedisMessage(channel: string, message: string): Promise<void> {
        const subscribers = this.subscriptions.get(channel);
        if (!subscribers) return;

        const data = JSON.parse(message);

        for (const connectionId of subscribers) {
            this.send(connectionId, data);
        }
    }
}
```

Channel Structure

```
const channels = {
    // Market data
    'trades:{symbol}': 'Trade executions',
    'funding:{symbol}': 'Funding rate updates',
    'liquidations:{symbol}': 'Liquidation events',

    // User-specific (authenticated)
    'positions:{userId}': 'Position updates',
    'orders:{userId}': 'Order status changes',
    'wallet:{userId}': 'Balance changes',

    // System-wide
    'system:status': 'System status updates',
    'system:maintenance': 'Maintenance notifications'
};
```

Historical Data Retention Policies

```
interface RetentionPolicy {
    dataType: string;
    hotStorage: string;      // Fast access
    warmStorage: string;     // Medium access
    coldStorage: string;     // Archive
    deletion: string;        // Permanent deletion
}

const RETENTION_POLICIES: RetentionPolicy[] = [
    {
        dataType: 'trades',
        hotStorage: '7 days (PostgreSQL)',
        warmStorage: '90 days (PostgreSQL)',
        coldStorage: '7 years (S3/Archive)',
        deletion: 'Never (regulatory requirement)'
    },
    {
        dataType: 'orders',
        hotStorage: '7 days (PostgreSQL)',
        warmStorage: '90 days (PostgreSQL)',
        coldStorage: '2 years (S3)',
        deletion: 'After 7 years'
    },
    {
        dataType: 'positions',
        hotStorage: '30 days (PostgreSQL)',
        warmStorage: '1 year (PostgreSQL)',
        coldStorage: '7 years (S3)',
        deletion: 'Never (regulatory requirement)'
    },
    {
        dataType: 'funding_rates',
        hotStorage: '30 days (PostgreSQL)',
        warmStorage: '1 year (PostgreSQL)',
        coldStorage: 'Indefinite (S3)',
        deletion: 'Never (historical reference)'
    },
    {
        dataType: 'activity_logs',
        hotStorage: '30 days (PostgreSQL)',
        warmStorage: '6 months (PostgreSQL)',
        coldStorage: '2 years (S3)',
        deletion: 'After 7 years'
    },
    {
        dataType: 'liquidations',
        hotStorage: '30 days (PostgreSQL)',
        warmStorage: '1 year (PostgreSQL)',
        coldStorage: 'Indefinite (S3)',
        deletion: 'Never (audit trail)'
    }
];
```

Caching Strategy

Multi-Layer Caching

```rust
use std::sync::Arc;
use std::collections::HashMap;
use tokio::sync::RwLock;
use redis::AsyncCommands;

pub struct CacheManager {
    l1_cache: Arc<RwLock<HashMap<String, (String, i64)>>>,  // In-memory with expiry
    l2_cache: redis::aio::ConnectionManager,                 // Redis
    db: DatabasePool,                                        // PostgreSQL
}

impl CacheManager {
    pub fn new(redis_client: redis::aio::ConnectionManager, db: DatabasePool) -> Self {
        Self {
            l1_cache: Arc::new(RwLock::new(HashMap::new())),
            l2_cache: redis_client,
            db,
        }
    }

    pub async fn get(&mut self, key: &str) -> Result<Option<String>, CacheError> {
        // L1: In-memory cache (fastest)
        {
            let cache = self.l1_cache.read().await;
            if let Some((value, expiry)) = cache.get(key) {
                if chrono::Utc::now().timestamp() < *expiry {
                    return Ok(Some(value.clone()));
                }
            }
        }

        // L2: Redis cache (fast)
        let redis_value: Option<String> = self.l2_cache.get(key).await?;
        if let Some(value) = redis_value {
            // Cache in L1 for 1 min
            let expiry = chrono::Utc::now().timestamp() + 60;
            let mut cache = self.l1_cache.write().await;
            cache.insert(key.to_string(), (value.clone(), expiry));
            return Ok(Some(value));
        }

        // L3: Database (slower)
        let db_value = self.fetch_from_database(key).await?;
        if let Some(value) = db_value {
            let value_json = serde_json::to_string(&value)?;

            // Cache in Redis for 5 min
            self.l2_cache.set_ex(key, &value_json, 300).await?;

            // Cache in L1 for 1 min
            let expiry = chrono::Utc::now().timestamp() + 60;
            let mut cache = self.l1_cache.write().await;
            cache.insert(key.to_string(), (value_json.clone(), expiry));

            return Ok(Some(value_json));
        }

        Ok(None)
    }

    pub async fn set(&mut self, key: &str, value: &str, ttl: Option<usize>) -> Result<(),
CacheError> {
        let ttl_l1 = ttl.unwrap_or(60);
        let ttl_l2 = ttl.unwrap_or(300);

        // Write to L1
        let expiry = chrono::Utc::now().timestamp() + ttl_l1 as i64;
        let mut cache = self.l1_cache.write().await;
        cache.insert(key.to_string(), (value.to_string(), expiry));
        drop(cache);

        // Write to L2 (Redis)
        self.l2_cache.set_ex(key, value, ttl_l2).await?;

        // Database is source of truth, updated separately
        Ok(())
    }

    pub async fn invalidate(&mut self, key: &str) -> Result<(), CacheError> {
        // Remove from L1
        let mut cache = self.l1_cache.write().await;
        cache.remove(key);
        drop(cache);
```

```
        // Remove from L2
        self.l2_cache.del(key).await?;

        Ok(())
    }
}
```

Cache Warming

```
pub struct CacheWarmer {
    cache: Arc<CacheManager>,
    db: DatabasePool,
}

impl CacheWarmer {
    pub async fn warm_caches(&self) -> Result<(), CacheError> {
        // Warm price cache
        let symbols = self.get_all_symbols().await?;
        for symbol in symbols {
            let price = self.fetch_price(&symbol).await?;
            self.cache.set(&format!("price:{}:mark", symbol), &price.to_string(), None).await?;
        }

        // Warm top positions
        let active_users = self.get_active_users().await?;
        for user_id in active_users {
            let positions = self.fetch_positions(&user_id).await?;
            let positions_json = serde_json::to_string(&positions)?;
            self.cache.set(&format!("user:{}:positions", user_id), &positions_json, None).await?;
        }

        // Warm market stats
        for symbol in symbols {
            let stats = self.calculate_market_stats(&symbol).await?;
            let stats_json = serde_json::to_string(&stats)?;
            self.cache.set(&format!("stats:{}", symbol), &stats_json, None).await?;
        }

        Ok(())
    }
}
```

Data Backup and Recovery

Backup Strategy

```typescript
interface BackupPlan {
    source: string;
    destination: string;
    frequency: string;
    retention: string;
    encryption: boolean;
}

const BACKUP_PLANS: BackupPlan[] = [
    {
        source: 'PostgreSQL (Hot)',
        destination: 'S3 + Glacier',
        frequency: 'Every 6 hours',
        retention: '30 days hot, 1 year glacier',
        encryption: true
    },
    {
        source: 'Redis (State)',
        destination: 'S3',
        frequency: 'Every 1 hour',
        retention: '7 days',
        encryption: true
    },
    {
        source: 'Smart Contract State',
        destination: 'Archive Node + S3',
        frequency: 'Real-time (automatic)',
        retention: 'Indefinite',
        encryption: false // Public blockchain
    }
];
```

Disaster Recovery

```
class DisasterRecovery {
    async createBackup(): Promise<string> {
        const backupId = `backup-${Date.now()}`;

        // 1. Dump PostgreSQL
        await exec(`pg_dump -Fc godark_db > /backups/${backupId}/postgres.dump`);

        // 2. Snapshot Redis
        await this.redis.bgsave();
        await exec(`cp /var/lib/redis/dump.rdb /backups/${backupId}/redis.rdb`);

        // 3. Export Solana account snapshots
        await this.exportSolanaAccounts(backupId);

        // 4. Compress and encrypt
        await exec(`tar -czf /backups/${backupId}.tar.gz /backups/${backupId}`);
        await exec(`openssl enc -aes-256-cbc -salt -in /backups/${backupId}.tar.gz -out
/backups/${backupId}.enc`);

        // 5. Upload to S3
        await this.s3.upload({
            Bucket: 'godark-backups',
            Key: `${backupId}.enc`,
            Body: fs.createReadStream(`/backups/${backupId}.enc`)
        });

        return backupId;
    }

    async restoreFromBackup(backupId: string): Promise<void> {
        // 1. Download from S3
        await this.s3.download(backupId);

        // 2. Decrypt and extract
        await exec(`openssl enc -aes-256-cbc -d -in ${backupId}.enc -out ${backupId}.tar.gz`);
        await exec(`tar -xzf ${backupId}.tar.gz`);

        // 3. Restore PostgreSQL
        await exec(`pg_restore -d godark_db ${backupId}/postgres.dump`);

        // 4. Restore Redis
        await exec(`cp ${backupId}/redis.rdb /var/lib/redis/dump.rdb`);
        await this.redis.shutdown();
        await this.redis.start();

        // 5. Verify data integrity
        await this.verifyRestore();
    }

    async performDRDrill(): Promise<DRTestReport> {
        // Regular disaster recovery testing
        const testId = `dr-test-${Date.now()}`;

        // Create test backup
        const backupId = await this.createBackup();

        // Spin up test environment
        const testEnv = await this.createTestEnvironment();

        // Restore to test environment
        await this.restoreToEnvironment(testEnv, backupId);

        // Verify functionality
        const verifications = await this.runVerificationTests(testEnv);

        // Cleanup
        await this.destroyTestEnvironment(testEnv);

        return {
            testId,
            timestamp: Date.now(),
            backupId,
            success: verifications.every(v => v.passed),
            duration: verifications.reduce((acc, v) => acc + v.duration, 0),
            results: verifications
        };
    }
}
```

8. Performance Requirements

Throughput Targets

GoDark is designed to handle institutional-grade trading volumes with minimal latency.

Trade Execution Throughput

Target Metrics:

```
const THROUGHPUT_TARGETS = {
    tradesPerSecond: {
        target: 100,
        peak: 500,
        sustained: 200
    },
    ordersPerSecond: {
        target: 1000,
        peak: 5000,
        sustained: 2000
    },
    matchingLatency: {
        p50: 5,     // 5ms median
        p95: 15,    // 15ms at 95th percentile
        p99: 50     // 50ms at 99th percentile
    },
    settlementLatency: {
        target: 1000,  // 1 second batch window
        p95: 1500,
        p99: 2000
    }
};
```

Concurrent User Support

Capacity Planning:

User Tier	Concurrent Users	Requests/Min per User	Total Load
Free	5,000	60	300K req/min
Basic	2,000	300	600K req/min
Pro	500	1,200	600K req/min
Market Maker	50	6,000	300K req/min
Total	**7,550**	**-**	**1.8M req/min**

```rust
pub struct CapacityManager {
    max_concurrent_users: usize,
    max_requests_per_minute: usize,
    db: DatabasePool,
    metrics: Arc<MetricsCollector>,
}

#[derive(Debug, Serialize)]
pub struct CapacityStatus {
    pub users: ResourceStatus,
    pub requests: ResourceStatus,
    pub should_scale_up: bool,
    pub should_scale_down: bool,
}

#[derive(Debug, Serialize)]
pub struct ResourceStatus {
    pub current: usize,
    pub max: usize,
    pub utilization: f64,
    pub available: usize,
}

impl CapacityManager {
    const MAX_CONCURRENT_USERS: usize = 10000;
    const MAX_REQUESTS_PER_MINUTE: usize = 2000000; // 2M

    pub fn new(db: DatabasePool, metrics: Arc<MetricsCollector>) -> Self {
        Self {
            max_concurrent_users: Self::MAX_CONCURRENT_USERS,
            max_requests_per_minute: Self::MAX_REQUESTS_PER_MINUTE,
            db,
            metrics,
        }
    }

    pub async fn check_capacity(&self) -> Result<CapacityStatus, CapacityError> {
        let current_users = self.get_current_user_count().await?;
        let current_rpm = self.get_current_rpm().await?;

        Ok(CapacityStatus {
            users: ResourceStatus {
                current: current_users,
                max: self.max_concurrent_users,
                utilization: current_users as f64 / self.max_concurrent_users as f64,
                available: self.max_concurrent_users.saturating_sub(current_users),
            },
            requests: ResourceStatus {
                current: current_rpm,
                max: self.max_requests_per_minute,
                utilization: current_rpm as f64 / self.max_requests_per_minute as f64,
                available: self.max_requests_per_minute.saturating_sub(current_rpm),
            },
            should_scale_up: current_users > (self.max_concurrent_users as f64 * 0.8) as usize,
            should_scale_down: current_users < (self.max_concurrent_users as f64 * 0.3) as usize,
        })
    }
}
```

Database Performance

PostgreSQL Targets:

```sql
-- Query performance targets
SELECT
    query_type,
    target_p50_ms,
    target_p95_ms,
    target_p99_ms
FROM performance_targets;

/*
query_type              | target_p50_ms | target_p95_ms | target_p99_ms
------------------------+---------------+---------------+--------------
get_user_positions      | 10            | 25            | 50
get_order_history       | 20            | 50            | 100
insert_trade            | 5             | 10            | 20
update_position         | 8             | 15            | 30
get_market_stats        | 15            | 40            | 80
*/
```

Redis Performance:

```javascript
const REDIS_TARGETS = {
    operations: {
        get: { p50: 0.5, p99: 2 },      // milliseconds
        set: { p50: 0.8, p99: 3 },
        zadd: { p50: 1, p99: 5 },
        hget: { p50: 0.6, p99: 2.5 }
    },
    throughput: {
        opsPerSecond: 100000,
        maxMemory: '32GB',
        evictionPolicy: 'allkeys-lru'
    },
    availability: {
        uptime: 0.9999,  // 99.99%
        failoverTime: 30  // seconds
    }
};
```

API Rate Limits per Tier

```rust
use std::collections::HashMap;
use once_cell::sync::Lazy;

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct RateLimitConfig {
    pub tier: String,
    pub rest_api: RestAPILimits,
    pub websocket: WebSocketLimits,
    pub trading: TradingLimits,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct RestAPILimits {
    pub requests_per_minute: u32,
    pub requests_per_second: u32,
    pub burst_allowance: u32,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct WebSocketLimits {
    pub messages_per_second: u32,
    pub max_connections: u32,
    pub max_subscriptions: u32,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct TradingLimits {
    pub orders_per_minute: u32,
    pub cancels_per_minute: u32,
    pub max_open_orders: u32,
}

pub static RATE_LIMITS: Lazy<HashMap<String, RateLimitConfig>> = Lazy::new(|| {
    let mut limits = HashMap::new();

    limits.insert("FREE".to_string(), RateLimitConfig {
        tier: "FREE".to_string(),
        rest_api: RestAPILimits {
            requests_per_minute: 60,
            requests_per_second: 2,
            burst_allowance: 10,
        },
        websocket: WebSocketLimits {
            messages_per_second: 10,
            max_connections: 2,
            max_subscriptions: 10,
        },
        trading: TradingLimits {
            orders_per_minute: 30,
            cancels_per_minute: 60,
            max_open_orders: 20,
        },
    });

    limits.insert("BASIC".to_string(), RateLimitConfig {
        tier: "BASIC".to_string(),
        rest_api: RestAPILimits {
            requests_per_minute: 300,
            requests_per_second: 10,
            burst_allowance: 50,
        },
        websocket: WebSocketLimits {
            messages_per_second: 50,
            max_connections: 5,
            max_subscriptions: 50,
        },
        trading: TradingLimits {
            orders_per_minute: 150,
            cancels_per_minute: 300,
            max_open_orders: 100,
        },
    });

    limits.insert("PRO".to_string(), RateLimitConfig {
        tier: "PRO".to_string(),
        rest_api: RestAPILimits {
            requests_per_minute: 1200,
            requests_per_second: 40,
            burst_allowance: 200,
        },
        websocket: WebSocketLimits {
            messages_per_second: 200,
            max_connections: 10,
```

```
            max_subscriptions: 200,
        },
        trading: TradingLimits {
            orders_per_minute: 600,
            cancels_per_minute: 1200,
            max_open_orders: 500,
        },
    });

    limits.insert("MARKET_MAKER".to_string(), RateLimitConfig {
        tier: "MARKET_MAKER".to_string(),
        rest_api: RestAPILimits {
            requests_per_minute: 6000,
            requests_per_second: 200,
            burst_allowance: 500,
        },
        websocket: WebSocketLimits {
            messages_per_second: 1000,
            max_connections: 20,
            max_subscriptions: 1000,
        },
        trading: TradingLimits {
            orders_per_minute: 3000,
            cancels_per_minute: 6000,
            max_open_orders: 5000,
        },
    });

    limits
});
```

Latency Targets

End-to-End Latency Budget

```typescript
interface LatencyBudget {
    component: string;
    target_p50: number;
    target_p95: number;
    target_p99: number;
    unit: 'ms';
}

const LATENCY_BUDGETS: LatencyBudget[] = [
    // Order submission flow
    { component: 'API Gateway', target_p50: 2, target_p95: 5, target_p99: 10, unit: 'ms' },
    { component: 'Authentication', target_p50: 1, target_p95: 3, target_p99: 5, unit: 'ms' },
    { component: 'Order Validation', target_p50: 2, target_p95: 5, target_p99: 10, unit: 'ms' },
    { component: 'Matching Engine', target_p50: 5, target_p95: 15, target_p99: 50, unit: 'ms' },
    { component: 'Database Write', target_p50: 5, target_p95: 10, target_p99: 20, unit: 'ms' },
    { component: 'WebSocket Notification', target_p50: 3, target_p95: 8, target_p99: 15, unit: 'ms'
},

    // Total order to acknowledgment
    { component: 'Order Submission Total', target_p50: 20, target_p95: 50, target_p99: 100, unit:
'ms' },

    // Settlement flow
    { component: 'Batch Creation', target_p50: 10, target_p95: 25, target_p99: 50, unit: 'ms' },
    { component: 'Transaction Build', target_p50: 50, target_p95: 100, target_p99: 200, unit: 'ms'
},
    { component: 'Solana Confirmation', target_p50: 400, target_p95: 800, target_p99: 1500, unit:
'ms' },
    { component: 'Position Update', target_p50: 20, target_p95: 50, target_p99: 100, unit: 'ms' },

    // Total trade to settlement
    { component: 'Settlement Total', target_p50: 1000, target_p95: 1500, target_p99: 2000, unit:
'ms' },

    // API endpoints
    { component: 'GET /markets', target_p50: 10, target_p95: 25, target_p99: 50, unit: 'ms' },
    { component: 'GET /positions', target_p50: 15, target_p95: 40, target_p99: 80, unit: 'ms' },
    { component: 'GET /orders', target_p50: 20, target_p95: 50, target_p99: 100, unit: 'ms' },
    { component: 'POST /orders', target_p50: 25, target_p95: 60, target_p99: 120, unit: 'ms' },
    { component: 'DELETE /orders/:id', target_p50: 15, target_p95: 35, target_p99: 70, unit: 'ms'
},

    // WebSocket
    { component: 'WS Message Delivery', target_p50: 5, target_p95: 15, target_p99: 50, unit: 'ms'
},
    { component: 'WS Ping/Pong', target_p50: 3, target_p95: 8, target_p99: 15, unit: 'ms' }
];
```

Geographic Latency Targets

```
const GEOGRAPHIC_LATENCY = {
    regions: [
        {
            region: 'North America (US East)',
            distance: '0 km',
            targetLatency: { p50: 20, p95: 50, p99: 100 }
        },
        {
            region: 'North America (US West)',
            distance: '4,000 km',
            targetLatency: { p50: 60, p95: 100, p99: 150 }
        },
        {
            region: 'Europe (London)',
            distance: '5,500 km',
            targetLatency: { p50: 80, p95: 120, p99: 180 }
        },
        {
            region: 'Asia (Singapore)',
            distance: '15,000 km',
            targetLatency: { p50: 150, p95: 200, p99: 300 }
        },
        {
            region: 'Asia (Tokyo)',
            distance: '11,000 km',
            targetLatency: { p50: 120, p95: 170, p99: 250 }
        }
    ],
    mitigations: [
        'CloudFlare CDN for static assets',
        'Regional API endpoints (future)',
        'WebSocket connection pooling',
        'Optimized protocol (protobuf for WS)'
    ]
};
```

Funding Rate Calculation (1-Second Intervals)

```rust
use std::time::Instant;
use tokio::task::JoinSet;

pub struct FundingRatePerformance {
    calculation_interval_ms: u64,
    target_calculation_time_ms: u64,
    oracle: Arc<ConsolidatedOracle>,
    redis: Arc<RedisPool>,
    alert_service: Arc<AlertService>,
}

#[derive(Debug, Serialize)]
pub struct PerformanceMetrics {
    pub symbol_count: usize,
    pub duration_ms: f64,
    pub average_per_symbol: f64,
    pub updates_per_second: f64,
    pub target_met: bool,
}

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

impl FundingRatePerformance {
    const CALCULATION_INTERVAL_MS: u64 = 1000;
    const TARGET_CALCULATION_TIME_MS: u64 = 100;

    pub fn new(
        oracle: Arc<ConsolidatedOracle>,
        redis: Arc<RedisPool>,
        alert_service: Arc<AlertService>,
    ) -> Self {
        Self {
            calculation_interval_ms: Self::CALCULATION_INTERVAL_MS,
            target_calculation_time_ms: Self::TARGET_CALCULATION_TIME_MS,
            oracle,
            redis,
            alert_service,
        }
    }

    pub async fn calculate_funding_rates(&self) -> Result<PerformanceMetrics, FundingError> {
        let start_time = Instant::now();
        let symbols = self.get_all_active_symbols().await?;

        // Parallel calculation for all symbols
        let mut join_set = JoinSet::new();
        for symbol in symbols.clone() {
            let self_clone = self.clone();
            join_set.spawn(async move {
                self_clone.calculate_for_symbol(&symbol).await
            });
        }

        let mut calculations = Vec::new();
        while let Some(result) = join_set.join_next().await {
            calculations.push(result??);
        }

        let duration = start_time.elapsed().as_millis() as f64;

        // Performance check
        if duration > self.target_calculation_time_ms as f64 {
            self.alert_service.send_alert(Alert {
                alert_type: "FUNDING_RATE_SLOW".to_string(),
                duration,
                target: self.target_calculation_time_ms as f64,
                symbol_count: symbols.len(),
            }).await?;
        }

        Ok(PerformanceMetrics {
            symbol_count: symbols.len(),
            duration_ms: duration,
            average_per_symbol: duration / symbols.len() as f64,
            updates_per_second: 1000.0 / duration,
```

```
            target_met: duration <= self.target_calculation_time_ms as f64,
        })
    }

    async fn calculate_for_symbol(&self, symbol: &str) -> Result<FundingRate, FundingError> {
        // Optimized parallel data fetching
        let (mark_price, index_price, _prev_rate) = tokio::join!(
            self.get_mark_price(symbol),
            self.get_index_price(symbol),
            self.get_previous_funding_rate(symbol),
        );

        let mark_price = mark_price?;
        let index_price = index_price?;

        // Fast calculation (< 1ms per symbol)
        let premium_index = (mark_price - index_price) / index_price;
        let interest_rate = 0.01 / (24.0 * 3600.0); // Daily rate per second
        let funding_rate = premium_index + interest_rate;

        // Clamp to limits
        let clamped_rate = funding_rate.max(-0.0005).min(0.0005);

        // Store in Redis (fire and forget)
        let redis = self.redis.clone();
        let symbol_owned = symbol.to_string();
        tokio::spawn(async move {
            if let Err(e) = Self::store_funding_rate(&redis, &symbol_owned, clamped_rate).await {
                eprintln!("Failed to store funding rate: {:?}", e);
            }
        });

        Ok(FundingRate {
            symbol: symbol.to_string(),
            rate: clamped_rate,
            premium_index,
            mark_price,
            index_price,
            timestamp: chrono::Utc::now().timestamp_millis(),
        })
    }
}
```

Performance Optimization:

```rust
// Batch processing for efficiency
pub struct BatchFundingRateCalculator {
    batch_size: usize,
    oracle: Arc<ConsolidatedOracle>,
    redis: Arc<RedisPool>,
}

impl BatchFundingRateCalculator {
    const BATCH_SIZE: usize = 50;

    pub fn new(oracle: Arc<ConsolidatedOracle>, redis: Arc<RedisPool>) -> Self {
        Self {
            batch_size: Self::BATCH_SIZE,
            oracle,
            redis,
        }
    }

    pub async fn calculate_all_markets(&self, symbols: Vec<String>) -> Result<(), FundingError> {
        // Process in batches to avoid overwhelming the system
        for chunk in symbols.chunks(self.batch_size) {
            let mut join_set = JoinSet::new();

            for symbol in chunk {
                let symbol_owned = symbol.clone();
                let self_clone = self.clone();
                join_set.spawn(async move {
                    self_clone.calculate_funding_rate(&symbol_owned).await
                });
            }

            // Wait for batch to complete
            while let Some(result) = join_set.join_next().await {
                result??;
            }

            // Small delay between batches if needed
            if chunk.len() == self.batch_size {
                tokio::time::sleep(tokio::time::Duration::from_millis(10)).await;
            }
        }

        Ok(())
    }
}
```

Real-Time Liquidation Monitoring

```rust
use std::collections::{HashMap, HashSet};

pub struct LiquidationMonitorPerformance {
    check_interval_ms: u64,
    target_check_time_ms: u64,
    cache: Arc<CacheManager>,
    oracle: Arc<ConsolidatedOracle>,
    liquidation_engine: Arc<LiquidationEngine>,
    alert_service: Arc<AlertService>,
    redis: Arc<RedisPool>,
}

#[derive(Debug, Serialize)]
pub struct MonitoringMetrics {
    pub position_count: usize,
    pub symbol_count: usize,
    pub duration_ms: f64,
    pub average_per_position: f64,
    pub liquidations_triggered: usize,
    pub target_met: bool,
}

impl LiquidationMonitorPerformance {
    const CHECK_INTERVAL_MS: u64 = 1000;  // Every second
    const TARGET_CHECK_TIME_MS: u64 = 500; // Must complete in 500ms

    pub fn new(
        cache: Arc<CacheManager>,
        oracle: Arc<ConsolidatedOracle>,
        liquidation_engine: Arc<LiquidationEngine>,
        alert_service: Arc<AlertService>,
        redis: Arc<RedisPool>,
    ) -> Self {
        Self {
            check_interval_ms: Self::CHECK_INTERVAL_MS,
            target_check_time_ms: Self::TARGET_CHECK_TIME_MS,
            cache,
            oracle,
            liquidation_engine,
            alert_service,
            redis,
        }
    }

    pub async fn monitor_all_positions(&self) -> Result<MonitoringMetrics, MonitorError> {
        let start_time = Instant::now();

        // 1. Get all open positions (cached)
        let positions = self.get_cached_open_positions().await?;

        // 2. Get current prices for all symbols (batch fetch)
        let unique_symbols: HashSet<String> = positions.iter()
            .map(|p| p.symbol.clone())
            .collect();
        let price_map = self.batch_get_prices(unique_symbols.iter().cloned().collect()).await?;

        // 3. Check each position in parallel
        let mut join_set = JoinSet::new();
        for position in positions.clone() {
            let price = price_map.get(&position.symbol).cloned();
            let self_clone = self.clone();
            join_set.spawn(async move {
                self_clone.check_position(position, price).await
            });
        }

        let mut checks = Vec::new();
        while let Some(result) = join_set.join_next().await {
            if let Ok(Ok(check_result)) = result {
                checks.push(check_result);
            }
        }

        let duration = start_time.elapsed().as_millis() as f64;

        // 4. Collect positions needing liquidation
        let liquidations: Vec<_> = checks.into_iter()
            .filter(|r| r.needs_liquidation)
            .collect();

        // 5. Execute liquidations (fire and forget)
        if !liquidations.is_empty() {
            let liquidation_engine = self.liquidation_engine.clone();
```

```
            tokio::spawn(async move {
                if let Err(e) = liquidation_engine.execute_liquidations(liquidations).await {
                    eprintln!("Liquidation execution failed: {:?}", e);
                }
            });
        }

        // 6. Performance tracking
        if duration > self.target_check_time_ms as f64 {
            self.alert_service.send_alert(Alert {
                alert_type: "SLOW_MONITORING".to_string(),
                duration,
                position_count: positions.len(),
                symbol_count: unique_symbols.len(),
                liquidation_count: liquidations.len(),
            }).await?;
        }

        Ok(MonitoringMetrics {
            position_count: positions.len(),
            symbol_count: unique_symbols.len(),
            duration_ms: duration,
            average_per_position: if positions.is_empty() { 0.0 } else { duration / positions.len()
as f64 },
            liquidations_triggered: liquidations.len(),
            target_met: duration <= self.target_check_time_ms as f64,
        })
    }

    async fn batch_get_prices(&self, symbols: Vec<String>) -> Result<HashMap<String, f64>,
RedisError> {
        // Batch fetch from Redis (single round trip using pipeline)
        let mut pipe = redis::pipe();
        for symbol in &symbols {
            pipe.get(format!("price:{}:mark", symbol));
        }

        let results: Vec<Option<String>> = pipe.query_async(&mut self.redis.clone()).await?;

        let mut price_map = HashMap::new();
        for (symbol, result) in symbols.iter().zip(results.iter()) {
            if let Some(price_str) = result {
                if let Ok(price) = price_str.parse::<f64>() {
                    price_map.insert(symbol.clone(), price);
                }
            }
        }

        Ok(price_map)
    }
}
```

Scalability Considerations

Horizontal Scaling Strategy

```
const SCALING_ARCHITECTURE = {
    components: {
        apiGateway: {
            type: 'Stateless',
            scaling: 'Horizontal',
            instances: {
                min: 3,
                max: 20,
                targetCPU: 70
            }
        },
        matchingEngine: {
            type: 'Stateful (per symbol)',
            scaling: 'Horizontal + Sharding',
            sharding: {
                strategy: 'By symbol',
                shardsPerInstance: 10,
                rebalancing: 'Dynamic'
            }
        },
        settlementRelayer: {
            type: 'Stateful (leader election)',
            scaling: 'Active-Passive',
            instances: {
                active: 1,
                passive: 2,
                failoverTime: '< 30s'
            }
        },
        liquidationEngine: {
            type: 'Stateless',
            scaling: 'Horizontal',
            instances: {
                min: 2,
                max: 10,
                targetPositions: 5000
            }
        },
        websocketServer: {
            type: 'Stateful (connections)',
            scaling: 'Horizontal',
            instances: {
                min: 3,
                max: 20,
                connectionsPerInstance: 5000
            }
        }
    },
    databases: {
        postgresql: {
            type: 'Primary-Replica',
            instances: {
                primary: 1,
                replicas: 3,
                readLoadBalancing: true
            },
            scaling: {
                vertical: 'Up to 64 vCPU, 256GB RAM',
                horizontal: 'Sharding by user_id (future)'
            }
        },
        redis: {
            type: 'Cluster',
            instances: {
                masters: 6,
                replicasPerMaster: 2
            },
            scaling: {
                addShards: 'Linear performance increase',
                maxShards: 16
            }
        }
    }
};
```

Load Balancing

```
class LoadBalancer {
    private readonly HEALTH_CHECK_INTERVAL = 10000; // 10 seconds

    async routeRequest(request: Request): Promise<Response> {
        // 1. Get healthy instances
        const instances = await this.getHealthyInstances(request.service);

        if (instances.length === 0) {
            throw new Error('No healthy instances available');
        }

        // 2. Choose instance based on strategy
        const instance = this.selectInstance(instances, request);

        // 3. Route request
        try {
            return await this.forwardRequest(instance, request);
        } catch (error) {
            // 4. Retry on different instance
            return await this.retryOnDifferentInstance(instances, request);
        }
    }

    private selectInstance(instances: Instance[], request: Request): Instance {
        switch (request.service) {
            case 'matching-engine':
                // Hash-based routing for symbol affinity
                return this.consistentHash(request.symbol, instances);

            case 'api-gateway':
                // Least connections
                return this.leastConnections(instances);

            case 'websocket':
                // Least connections + geographic proximity
                return this.geoAwareLeastConnections(instances, request.ip);

            default:
                // Round robin
                return this.roundRobin(instances);
        }
    }
}
```

Performance Monitoring and Metrics

Key Performance Indicators (KPIs)

```
interface PerformanceKPIs {
    // Throughput
    tradesPerSecond: Metric;
    ordersPerSecond: Metric;
    settlementsPerSecond: Metric;

    // Latency
    orderSubmissionLatency: LatencyMetric;
    matchingLatency: LatencyMetric;
    settlementLatency: LatencyMetric;
    apiResponseTime: LatencyMetric;
    websocketLatency: LatencyMetric;

    // Availability
    uptime: Metric;
    apiAvailability: Metric;
    websocketAvailability: Metric;
    settlementSuccessRate: Metric;

    // Resource utilization
    cpuUtilization: Metric;
    memoryUtilization: Metric;
    diskIOPS: Metric;
    networkBandwidth: Metric;

    // Business metrics
    activeUsers: Metric;
    concurrentConnections: Metric;
    totalOpenPositions: Metric;
    totalOpenInterest: Metric;
}

class PerformanceMonitor {
    async collectMetrics(): Promise<PerformanceKPIs> {
        return {
            // Throughput (last minute)
            tradesPerSecond: await this.calculateRate('trades', 60),
            ordersPerSecond: await this.calculateRate('orders', 60),
            settlementsPerSecond: await this.calculateRate('settlements', 60),

            // Latency (p50, p95, p99)
            orderSubmissionLatency: await this.calculateLatency('order_submission'),
            matchingLatency: await this.calculateLatency('matching'),
            settlementLatency: await this.calculateLatency('settlement'),
            apiResponseTime: await this.calculateLatency('api'),
            websocketLatency: await this.calculateLatency('websocket'),

            // Availability (last hour)
            uptime: await this.calculateUptime(3600),
            apiAvailability: await this.calculateAvailability('api', 3600),
            websocketAvailability: await this.calculateAvailability('websocket', 3600),
            settlementSuccessRate: await this.calculateSuccessRate('settlement', 3600),

            // Resources (current)
            cpuUtilization: await this.getCPUUtilization(),
            memoryUtilization: await this.getMemoryUtilization(),
            diskIOPS: await this.getDiskIOPS(),
            networkBandwidth: await this.getNetworkBandwidth(),

            // Business (current)
            activeUsers: await this.getActiveUsers(),
            concurrentConnections: await this.getConcurrentConnections(),
            totalOpenPositions: await this.getTotalOpenPositions(),
            totalOpenInterest: await this.getTotalOpenInterest()
        };
    }

    async checkSLAs(): Promise<SLAStatus[]> {
        const slas: SLA[] = [
            { metric: 'uptime', target: 0.999, current: await this.getUptime() },
            { metric: 'api_latency_p95', target: 100, current: await this.getAPILatencyP95() },
            { metric: 'settlement_success', target: 0.99, current: await
this.getSettlementSuccessRate() },
            { metric: 'trades_per_second', target: 100, current: await this.getTradesPerSecond() }
        ];

        return slas.map(sla => ({
            ...sla,
            met: this.isSLAMet(sla),
            breachDuration: this.getBreachDuration(sla)
        }));
    }
}
```

Alerting Thresholds

```
const ALERT_THRESHOLDS = {
    critical: {
        apiLatencyP95: 500,           // ms
        settlementLatency: 5000,      // ms
        errorRate: 0.05,              // 5%
        uptime: 0.99,                 // 99%
        cpuUtilization: 90,           // %
        memoryUtilization: 90,        // %
        diskUtilization: 85,          // %
    },
    warning: {
        apiLatencyP95: 200,
        settlementLatency: 2000,
        errorRate: 0.02,
        uptime: 0.995,
        cpuUtilization: 75,
        memoryUtilization: 75,
        diskUtilization: 70,
    },
    actions: {
        critical: [
            'Page on-call engineer',
            'Create incident ticket',
            'Trigger auto-scaling',
            'Send user notification'
        ],
        warning: [
            'Send Slack alert',
            'Log to monitoring',
            'Prepare for scaling'
        ]
    }
};
```

---

9. UI/UX Overview (Brief)

Note: The UI/UX design is being developed separately by the design team. This section provides a high-level overview of the interface components and their integration with the backend architecture.

Key Interface Components

Trading Interface (app.godark.xyz/trade)

Primary Components:

- Order entry form with leverage selector
- Symbol selector with search and favorites
- Real-time chart integration (TradingView or custom)
- Position display with PnL tracking
- Order management (working orders, history)
- Funding rate display with countdown timer
- Account balance and margin metrics

Backend Integration Points:

```
// API endpoints used by trading interface
const TRADING_INTERFACE_APIs = {
    // Real-time data
    websocket: 'wss://api.godark.xyz/v1/ws',
    channels: [
        'positions:{userId}',
        'orders:{userId}',
        'trades:{symbol}',
        'funding:{symbol}'
    ],

    // REST APIs
    submitOrder: 'POST /api/v1/orders',
    cancelOrder: 'DELETE /api/v1/orders/:id',
    getPositions: 'GET /api/v1/positions',
    getOrders: 'GET /api/v1/orders',
    getMarkets: 'GET /api/v1/markets',
    getBalance: 'GET /api/v1/account/balance'
};
```

Stats Dashboard (stats.godark.xyz)

Three Metric Sections:

1. Execution Quality & Savings

- Slippage and market impact saved (USDT)
- MEV avoided (USDT)
- Cumulative charts with daily granularity
- Data published at midnight UTC (T-2 days)

2. GoDark Market Data

- Matched volume (USDT)
- Liquidity submitted (USDT)
- Buy/Sell ratio (%)
- Average time to fill (seconds)
- Average trade size (USDT)
- Average order size (USDT)
- Matched trade count
- Order count

3. Operational Transparency

- Fees collected (USDT)
- Average settlement finality time (seconds)
- Failed settlements count
- System downtime (minutes)
- Average API response time (seconds)

Backend Integration:

```
// Stats API endpoints
const STATS_APIs = {
    getExecutionQuality: 'GET /api/v1/stats/execution-quality',
    getMarketData: 'GET /api/v1/stats/market-data',
    getOperationalMetrics: 'GET /api/v1/stats/operational',

    // Query parameters
    params: {
        startDate: 'YYYY-MM-DD',
        endDate: 'YYYY-MM-DD',
        symbol: 'optional',
        aggregation: 'daily' // or hourly
    }
};
```

Admin Panel (app.godark.xyz/admin)

Sections:

1. Linked Wallet

- Display current linked wallet address
- Link/unlink wallet functionality
- One wallet per account limitation enforced

2. API Key Management

- Table of existing API keys
- Columns: Key name, API key, Secret key, Passphrase, IP whitelist
- Edit key name and IP whitelist
- Delete keys (with password confirmation)
- Create new API key (max 5 per account)

3. Account Management

- Email address (display only)
- Change password
- Enable/disable 2FA
- Delete account (with confirmation and grace period)

4. Activity Log

- Recent logins with device and IP information
- Account actions audit trail
- Security alerts

Backend Integration:

```
const ADMIN_APIs = {
    // Wallet management
    linkWallet: 'POST /api/v1/wallet/link',
    unlinkWallet: 'POST /api/v1/wallet/unlink',
    getWalletInfo: 'GET /api/v1/wallet/info',

    // API keys
    createAPIKey: 'POST /api/v1/api-keys',
    listAPIKeys: 'GET /api/v1/api-keys',
    updateAPIKey: 'PUT /api/v1/api-keys/:id',
    deleteAPIKey: 'DELETE /api/v1/api-keys/:id',

    // Account
    changePassword: 'POST /api/v1/account/change-password',
    enable2FA: 'POST /api/v1/account/2fa/enable',
    disable2FA: 'POST /api/v1/account/2fa/disable',
    deleteAccount: 'POST /api/v1/account/delete',

    // Activity
    getActivity: 'GET /api/v1/account/activity'
};
```

Referrals System (Modal popup)

Features:

- User's unique referral code
- Referral link generator
- Referral statistics (sign-ups, trading volume)
- Reward tracking
- Social sharing buttons

Backend Integration:

```
const REFERRAL_APIs = {
    getReferralCode: 'GET /api/v1/referrals/code',
    getReferralStats: 'GET /api/v1/referrals/stats',
    getRewards: 'GET /api/v1/referrals/rewards'
};
```

User Flows

1. New User Onboarding

```
Step 1: Registration
├── Enter email and password
├── Agree to terms
├── Submit registration
└── Receive verification email

Step 2: Email Verification
├── Click link in email
├── Account activated
└── Redirect to login

Step 3: Login
├── Enter email and password
├── (Optional) Enter 2FA code
└── Session created

Step 4: Wallet Connection
├── Click "Connect Wallet"
├── Choose wallet provider (Phantom/Trust/Solflare)
├── OR create new wallet (Google/Apple/X/Discord sign-in)
├── Sign authorization message
└── Wallet linked to account

Step 5: Fund Account
├── Authorize USDT amount for trading
├── Sign delegate approval transaction
├── Ephemeral vault created
└── Ready to trade
```

2. Trading Flow

```
Order Submission
├── Select symbol
├── Choose side (Buy/Sell)
├── Set order type (Market/Limit/Peg)
├── Enter size
├── Set leverage
├── (Optional) Set order attributes
├── Review order details
├── Submit
└── Receive confirmation

Order Matching (Backend)
├── Order enters matching engine
├── Price-time priority matching
├── Trade execution
└── WebSocket notification sent

Settlement (Backend)
├── Trade batched with others
├── Net position calculated
├── Settlement transaction sent to Solana
├── On-chain confirmation
└── Position updated

User Notification
├── WebSocket update received
├── Position displayed in UI
├── PnL calculation shown
└── Order moved to history
```

3. Position Management Flow

```
Monitor Position
├── View real-time PnL
├── Check margin ratio
├── Monitor liquidation price
├── Track funding payments
└── Adjust if needed

Close Position
├── Click "Close"
├── Choose close type (Market/Limit)
├── Confirm closure
├── Order submitted
├── Matched and settled
├── Collateral released
└── Available for withdrawal

Withdraw Funds
├── Click "Withdraw"
├── Enter amount (or "Max")
├── Confirm (no signature needed for unlocked funds)
├── Backend initiates withdrawal
├── On-chain transfer
└── USDT received in wallet
```

Basic vs Advanced Mode

Basic Interface Mode

Simplified for Retail Traders:

- Area line chart (instead of candlesticks)
- No order book or trades feed visible
- Best bid/ask displayed at top of chart
- Only Market and Limit orders
- Quick leverage selector: 1x, 10x, 100x, 1000x
- Quantity slider (instead of manual input)
- GTC only (no complex time-in-force)

- TP/SL (Take Profit / Stop Loss) integration
- No advanced attributes (NBBO, Min Qty removed)

Backend Differences:

- Same APIs used
- UI simplifies complexity
- Some order attributes set to defaults automatically

Advanced Interface Mode

Full Feature Set for Professional Traders:

- Candlestick charts with indicators
- Order book depth visualization
- Recent trades feed
- All order types (Market, Limit, Peg to Mid/Bid/Ask)
- All time-in-force options (IOC, FOK, GTD, GTC)
- All order attributes (AON, Min Qty, NBBO Protection)
- Advanced position management
- Multiple chart layouts
- Hotkey support

Mobile Responsiveness

Responsive Design Considerations:

- Mobile-first approach
- Touch-optimized controls
- Simplified navigation on mobile
- Progressive disclosure of features
- Native app consideration (future)

Mobile-Specific Features:

- Swipe gestures for tab navigation
- Collapsible sections
- Bottom sheet modals
- Biometric authentication support

Integration Points with Backend APIs

WebSocket Message Flow

```
// Client subscribes to channels
client.send(JSON.stringify({
    type: 'subscribe',
    channels: ['positions:user123', 'orders:user123', 'funding:BTC-USDT-PERP']
}));

// Server sends updates
{
    type: 'position_update',
    data: {
        symbol: 'BTC-USDT-PERP',
        size: 1.5,
        unrealizedPnl: 1250.50,
        markPrice: 45123.00,
        liquidationPrice: 40100.00
    }
}

{
    type: 'order_update',
    data: {
        orderId: 'abc123',
        status: 'FILLED',
        filledSize: 0.5,
        avgFillPrice: 45125.00
    }
}

{
    type: 'funding_update',
    data: {
        symbol: 'BTC-USDT-PERP',
        fundingRate: 0.00012,
        nextFundingTime: 1698768000000,
        countdown: 3568
    }
}
```

State Management

Frontend State:

```
interface AppState {
    // User state
    user: {
        accountId: string;
        email: string;
        walletAddress: string;
        isAuthenticated: boolean;
    };

    // Trading state
    trading: {
        selectedSymbol: string;
        orderForm: OrderFormState;
        positions: Position[];
        orders: Order[];
        balance: Balance;
    };

    // Market data
    markets: {
        symbols: Symbol[];
        prices: Map<string, Price>;
        fundingRates: Map<string, FundingRate>;
    };

    // UI state
    ui: {
        mode: 'BASIC' | 'ADVANCED';
        theme: 'LIGHT' | 'DARK';
        chartLayout: LayoutConfig;
        notifications: Notification[];
    };
}
```

Error Handling

User-Friendly Error Messages:

```
const ERROR_MESSAGES: Map<string, string> = new Map([
    ['INSUFFICIENT_COLLATERAL', 'Insufficient funds. Please deposit more USDT.'],
    ['POSITION_SIZE_EXCEEDED', 'Order size exceeds maximum allowed for your leverage.'],
    ['MARKET_CLOSED', 'This market is temporarily closed.'],
    ['RATE_LIMIT_EXCEEDED', 'Too many requests. Please wait a moment.'],
    ['INVALID_PRICE', 'Invalid price. Please check your order.'],
    ['LIQUIDATION_PENDING', 'Your position is being liquidated.'],
    ['SETTLEMENT_FAILED', 'Settlement failed. Retrying...'],
    ['NETWORK_ERROR', 'Network error. Please check your connection.']
]);
```

Performance Optimization

Frontend Optimizations:

- Virtual scrolling for order tables
- Debounced API calls
- Optimistic UI updates
- Cached market data
- Lazy loading of components
- Code splitting per route
- Service worker for offline support

Bundle Size Targets:

- Initial bundle: < 300KB gzipped
- Route chunks: < 100KB each
- Total JS: < 1MB
- First contentful paint: < 1.5s
- Time to interactive: < 3s

Accessibility

WCAG 2.1 AA Compliance:

- Keyboard navigation support
- Screen reader compatibility
- Sufficient color contrast
- Focus indicators
- Alt text for images
- ARIA labels for interactive elements

Browser Support

Supported Browsers:

- Chrome 90+
- Firefox 88+
- Safari 14+
- Edge 90+

Mobile Browsers:

- iOS Safari 14+
- Chrome Mobile 90+

---

10. Token Economics (DARK Token)

Overview

The DARK token is the native utility token of the GoDark DEX ecosystem, designed to align incentives between the platform, traders, and token holders.

Token Details:

- Name: GoDark Token
- Symbol: DARK
- Blockchain: Solana (SPL Token)
- Total Supply: 1,000,000,000 DARK (fixed, no inflation)
- Decimals: 9

Token Utilities

1. Staking Yield (Revenue Share)

Mechanism:

```
interface StakingTier {
    minStake: number;
    revenueSharePercent: number;
    lockPeriod: number; // days
    apr: number; // estimated
}

const STAKING_TIERS: StakingTier[] = [
    {
        minStake: 1000,
        revenueSharePercent: 5,
        lockPeriod: 0,    // Flexible
        apr: 8
    },
    {
        minStake: 10000,
        revenueSharePercent: 10,
        lockPeriod: 30,
        apr: 12
    },
    {
        minStake: 50000,
        revenueSharePercent: 15,
        lockPeriod: 90,
        apr: 18
    },
    {
        minStake: 100000,
        revenueSharePercent: 20,
        lockPeriod: 180,
        apr: 25
    }
];
```

Revenue Distribution:

```rust
// Weekly revenue distribution
pub struct RevenueDistribution {
    db: Arc<DatabasePool>,
    token_program: Arc<TokenProgramClient>,
}

#[derive(Debug, Serialize)]
pub struct DistributionRecord {
    pub total_fees: f64,
    pub stakers_share: f64,
    pub total_staked: f64,
    pub timestamp: i64,
}

impl RevenueDistribution {
    pub fn new(db: Arc<DatabasePool>, token_program: Arc<TokenProgramClient>) -> Self {
        Self { db, token_program }
    }

    pub async fn distribute_weekly_revenue(&self) -> Result<(), DistributionError> {
        // 1. Calculate total fees collected
        let total_fees = self.get_total_fees_collected(7).await?; // Last 7 days

        // 2. Get staker shares
        let stakers_share = total_fees * 0.40; // 40% to stakers

        // 3. Calculate each staker's portion
        let stakes = self.get_all_stakes().await?;
        let total_staked: f64 = stakes.iter().map(|s| s.amount).sum();

        for stake in &stakes {
            let user_share = (stake.amount / total_staked) * stakers_share;

            // 4. Distribute USDT rewards
            self.distribute_reward(&stake.user_id, user_share).await?;
        }

        // 5. Record distribution
        self.record_distribution(DistributionRecord {
            total_fees,
            stakers_share,
            total_staked,
            timestamp: chrono::Utc::now().timestamp(),
        }).await?;

        Ok(())
    }
}
```

2. Fee Rebates and Discounts

Fee Structure:

DARK Staked	Maker Fee	Taker Fee	Discount
0	-0.02%	0.05%	0%
1,000 - 9,999	-0.02%	0.0475%	5%
10,000 - 49,999	-0.02%	0.045%	10%
50,000 - 99,999	-0.02%	0.0425%	15%
100,000+	-0.02%	0.04%	20%

Implementation:

```rust
pub async fn calculate_fee(
    user_id: &str,
    trade_size: f64,
    is_maker: bool,
    staking_service: &StakingService,
) -> Result<f64, FeeError> {
    let staked_amount = staking_service.get_staked_amount(user_id).await?;
    let tier = get_fee_tier(staked_amount);

    let base_fee = if is_maker { tier.maker_fee } else { tier.taker_fee };
    let fee_amount = trade_size * base_fee.abs();

    Ok(if is_maker { -fee_amount } else { fee_amount }) // Negative for rebate
}
```

3. Airdrops to Holders

Airdrop Events:

- Token launch airdrop to early users
- Monthly trading volume rewards
- Snapshot-based distributions for governance votes
- Partnership token airdrops (ecosystem tokens)

Distribution Formula:

```
interface AirdropCalculation {
    userAllocation = (userTokens / totalCirculating) * totalAirdrop;

    // With minimum holding requirement
    if (holdingDuration < 30 days) {
        userAllocation *= 0.5; // 50% penalty
    }

    // With trading volume multiplier
    if (tradingVolume > threshold) {
        userAllocation *= (1 + tradingBonus);
    }
}
```

4. Burn-and-Buyback Mechanism

Burn Schedule:

```rust
pub struct BurnMechanism {
    db: Arc<DatabasePool>,
    token_program: Arc<TokenProgramClient>,
    dex_integration: Arc<DexIntegration>,
    notification_service: Arc<NotificationService>,
}

#[derive(Debug, Serialize)]
pub struct BurnAnnouncement {
    pub amount: f64,
    pub usd_value: f64,
    pub new_circulating: f64,
    pub timestamp: i64,
}

impl BurnMechanism {
    pub fn new(
        db: Arc<DatabasePool>,
        token_program: Arc<TokenProgramClient>,
        dex_integration: Arc<DexIntegration>,
        notification_service: Arc<NotificationService>,
    ) -> Self {
        Self {
            db,
            token_program,
            dex_integration,
            notification_service,
        }
    }

    pub async fn execute_burn_cycle(&self) -> Result<(), BurnError> {
        // Every month
        let fee_revenue = self.get_monthly_fee_revenue().await?;
        let buyback_amount = fee_revenue * 0.30; // 30% for buyback

        // 1. Market buy DARK tokens
        let tokens_bought = self.market_buy_dark(buyback_amount).await?;

        // 2. Burn tokens (send to null address)
        self.burn_tokens(tokens_bought).await?;

        // 3. Update circulating supply
        self.update_circulating_supply().await?;

        // 4. Announce to community
        self.announce_burn(BurnAnnouncement {
            amount: tokens_bought,
            usd_value: buyback_amount,
            new_circulating: self.get_circulating_supply().await?,
            timestamp: chrono::Utc::now().timestamp(),
        }).await?;

        Ok(())
    }
}
```

Burn Impact:

```javascript
// Projected token burn over time
const BURN_PROJECTION = {
    year1: {
        estimatedFees: 50_000_000, // $50M
        buybackPercent: 0.30,
        buybackAmount: 15_000_000, // $15M
        estimatedBurn: 15_000_000, // 1.5% of supply
        newCirculating: 985_000_000
    },
    year5: {
        estimatedFees: 500_000_000,
        cumulativeBurn: 150_000_000, // 15% of supply
        newCirculating: 850_000_000
    }
};
```

5. Bug Bounty Payments

Bug bounty rewards paid in DARK tokens:

- Critical: up to 1M DARK ($1M value)
- High: up to 100K DARK
- Medium: up to 10K DARK
- Low: up to 1K DARK

Token Allocation

Total Supply: 1,000,000,000 DARK

```
const TOKEN_ALLOCATION = {
    foundation: {
        amount: 200_000_000,      // 20%
        purpose: 'Protocol development, partnerships',
        vesting: 'Linear over 4 years',
        cliff: 'None'
    },
    goQuant: {
        amount: 150_000_000,      // 15%
        purpose: 'Strategic reserve for GoQuant',
        vesting: 'Linear over 4 years',
        cliff: '1 year'
    },
    team: {
        amount: 150_000_000,      // 15%
        purpose: 'Core team and advisors',
        vesting: 'Linear over 4 years',
        cliff: '1 year'
    },
    incentives: {
        amount: 300_000_000,      // 30%
        purpose: 'Liquidity mining, market making',
        vesting: 'Linear over 5 years',
        cliff: 'None'
    },
    community: {
        amount: 150_000_000,      // 15%
        purpose: 'Bootcamp, bounties, airdrops',
        vesting: 'Linear over 3 years',
        cliff: 'None'
    },
    publicSale: {
        amount: 50_000_000,       // 5%
        purpose: 'Public token sale',
        vesting: '20% TGE, rest over 6 months',
        cliff: 'None'
    }
};
```

Vesting Schedule Visualization:

```
Month   0    6    12   18   24   30   36   42   48
        ┌────┬────┬────┬────┬────┬────┬────┬────┬────┐
Team    │ 0  │ 0  │12.5%│ 25%│37.5%│ 50%│62.5%│ 75%│ 100%│
        └────┴────┴────┴────┴────┴────┴────┴────┴────┘
             Cliff         Linear vesting over 4 years

        ┌────┬────┬────┬────┬────┬────┬────┬────┬────┐
Comm    │ 33%│ 33%│ 66%│ 66%│100%│    │    │    │    │
        └────┴────┴────┴────┴────┴────┴────┴────┴────┘
             No cliff, linear over 3 years
```

Revenue Distribution Model

Fee Revenue Breakdown:

```
const REVENUE_DISTRIBUTION = {
    stakers: {
        percent: 40,
        description: 'Distributed to DARK token stakers'
    },
    buyback: {
        percent: 30,
        description: 'Buy DARK from market and burn'
    },
    treasury: {
        percent: 20,
        description: 'Protocol treasury for development'
    },
    team: {
        percent: 10,
        description: 'Team compensation and operations'
    }
};

// Example with $1M monthly fees
const monthlyRevenue = 1_000_000;
const distribution = {
    toStakers: 400_000,    // $400k in USDT
    toBuyback: 300_000,    // $300k to buy & burn DARK
    toTreasury: 200_000,   // $200k for development
    toTeam: 100_000        // $100k for team
};
```

Staking Mechanisms

Staking Contract

```rust
#[account]
pub struct StakeAccount {
    pub owner: Pubkey,
    pub amount: u64,
    pub tier: StakeTier,
    pub locked_until: i64,
    pub rewards_earned: u64,
    pub last_claim: i64,
    pub created_at: i64,
}

pub fn stake_tokens(
    ctx: Context<StakeTokens>,
    amount: u64,
    lock_period: u32
) -> Result<()> {
    let stake_account = &mut ctx.accounts.stake_account;
    let clock = Clock::get()?;

    // Transfer DARK to staking vault
    token::transfer(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            token::Transfer {
                from: ctx.accounts.user_token_account.to_account_info(),
                to: ctx.accounts.staking_vault.to_account_info(),
                authority: ctx.accounts.user.to_account_info(),
            }
        ),
        amount
    )?;

    // Record stake
    stake_account.owner = ctx.accounts.user.key();
    stake_account.amount = amount;
    stake_account.tier = determine_tier(amount, lock_period);
    stake_account.locked_until = clock.unix_timestamp + (lock_period as i64 * 86400);
    stake_account.created_at = clock.unix_timestamp;

    Ok(())
}

pub fn claim_rewards(ctx: Context<ClaimRewards>) -> Result<()> {
    let stake_account = &mut ctx.accounts.stake_account;
    let clock = Clock::get()?;

    // Calculate rewards
    let rewards = calculate_rewards(stake_account)?;

    // Transfer USDT rewards
    let seeds = &[
        b"rewards_vault",
        &[ctx.bumps.rewards_vault]
    ];

    token::transfer(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            token::Transfer {
                from: ctx.accounts.rewards_vault.to_account_info(),
                to: ctx.accounts.user_usdt_account.to_account_info(),
                authority: ctx.accounts.rewards_vault.to_account_info(),
            },
            &[&seeds[..]]
        ),
        rewards
    )?;

    stake_account.rewards_earned += rewards;
    stake_account.last_claim = clock.unix_timestamp;

    Ok(())
}
```

Unstaking with Lock Period

```rust
pub struct UnstakingManager {
    db: Arc<DatabasePool>,
    token_program: Arc<TokenProgramClient>,
}

impl UnstakingManager {
    pub fn new(db: Arc<DatabasePool>, token_program: Arc<TokenProgramClient>) -> Self {
        Self { db, token_program }
    }

    pub async fn unstake(&self, user_id: &str, amount: f64) -> Result<(), UnstakeError> {
        let stake = self.get_stake(user_id).await?;

        // Check lock period
        let now = chrono::Utc::now().timestamp();
        if now < stake.locked_until {
            return Err(UnstakeError::TokensLocked {
                unlock_date: chrono::DateTime::from_timestamp(stake.locked_until, 0),
            });
        }

        // Check unstaking amount
        if amount > stake.amount {
            return Err(UnstakeError::InsufficientStake);
        }

        // Apply early unstaking penalty if applicable
        let penalty = self.calculate_penalty(&stake, amount)?;
        let net_amount = amount - penalty;

        // Execute unstaking
        self.execute_unstake(user_id, net_amount, penalty).await?;

        // Update staking record
        let mut updated_stake = stake;
        updated_stake.amount -= amount;
        self.update_stake(updated_stake).await?;

        Ok(())
    }

    fn calculate_penalty(&self, stake: &Stake, amount: f64) -> Result<f64, UnstakeError> {
        // No penalty if lock period completed
        let now = chrono::Utc::now().timestamp();
        if now >= stake.locked_until {
            return Ok(0.0);
        }

        // Progressive penalty based on remaining lock time
        let remaining_days = (stake.locked_until - now) as f64 / (24.0 * 60.0 * 60.0);
        let penalty_rate = (remaining_days / stake.lock_period as f64 * 0.20).min(0.20);

        Ok(amount * penalty_rate) // Max 20% penalty
    }
}
```

Governance (Limited)

Governance Rights:

```typescript
interface GovernanceProposal {
    id: string;
    title: string;
    description: string;
    category: 'MARKET_LISTING' | 'FEE_CHANGE' | 'PARAMETER_ADJUSTMENT';
    proposer: string;
    requiredStake: number;      // 100,000 DARK to propose
    votingPeriod: number;       // 7 days
    quorum: number;             // 10% of staked supply
    threshold: number;          // 66% approval
    status: 'PENDING' | 'ACTIVE' | 'PASSED' | 'REJECTED';
}
```

```rust
pub struct GovernanceSystem {
    db: Arc<DatabasePool>,
    staking_service: Arc<StakingService>,
}

#[derive(Debug, Serialize, Deserialize)]
pub struct GovernanceProposal {
    pub id: String,
    pub title: String,
    pub description: String,
    pub category: ProposalCategory,
    pub proposer: String,
    pub required_stake: f64,      // 100,000 DARK to propose
    pub voting_period: i64,       // 7 days
    pub quorum: f64,              // 10% of staked supply
    pub threshold: f64,           // 66% approval
    pub status: ProposalStatus,
}

#[derive(Debug, Serialize, Deserialize)]
pub enum ProposalCategory {
    MarketListing,
    FeeChange,
    ParameterAdjustment,
}

#[derive(Debug, Serialize, Deserialize)]
pub enum ProposalStatus {
    Pending,
    Active,
    Passed,
    Rejected,
}

impl GovernanceSystem {
    pub fn new(db: Arc<DatabasePool>, staking_service: Arc<StakingService>) -> Self {
        Self { db, staking_service }
    }

    pub async fn create_proposal(&self, proposal: GovernanceProposal) -> Result<String,
GovernanceError> {
        // Verify proposer has enough staked
        let proposer_stake = self.staking_service.get_staked_amount(&proposal.proposer).await?;

        if proposer_stake < proposal.required_stake {
            return Err(GovernanceError::InsufficientStake);
        }

        // Create proposal
        let proposal_id = self.store_proposal(proposal).await?;

        // Start voting period
        self.start_voting(&proposal_id).await?;

        Ok(proposal_id)
    }

    pub async fn vote(&self, proposal_id: &str, user_id: &str, support: bool) -> Result<(),
GovernanceError> {
        let _proposal = self.get_proposal(proposal_id).await?;
        let user_stake = self.staking_service.get_staked_amount(user_id).await?;

        // Voting power = staked amount
        self.record_vote(VoteRecord {
            proposal_id: proposal_id.to_string(),
            user_id: user_id.to_string(),
            voting_power: user_stake,
            support,
            timestamp: chrono::Utc::now().timestamp(),
```

```
            }).await?;

            Ok(())
        }

        pub async fn execute_proposal(&self, proposal_id: &str) -> Result<(), GovernanceError> {
            let mut proposal = self.get_proposal(proposal_id).await?;
            let results = self.tally_votes(proposal_id).await?;

            // Check quorum
            if results.total_votes < results.required_quorum {
                proposal.status = ProposalStatus::Rejected;
                self.update_proposal(&proposal).await?;
                return Ok(());
            }

            // Check threshold
            if results.support_percent < proposal.threshold {
                proposal.status = ProposalStatus::Rejected;
                self.update_proposal(&proposal).await?;
                return Ok(());
            }

            // Execute proposal
            proposal.status = ProposalStatus::Passed;
            self.implement_proposal(&proposal).await?;

            Ok(())
        }
    }
```

Governable Parameters:

- New market listings (after security review)
- Fee structure adjustments (within bounds)
- Staking tier requirements
- Revenue distribution percentages
- Governance parameters themselves

Non-Governable (Immutable):

- Core smart contract logic
- Security parameters
- Token supply
- Multisig requirements

Comparison to Similar Tokens

Feature	DARK	HYPE	ASTER	RAY	MANTA	JUP	PUMP
Chain	Solana	Custom L1	Solana	Solana	ETH L2	Solana	Solana
Fee Discount	✅ 20%	✅	✅	✅ 10%	✅	✅	❌
Staking Yield	✅ 8-25%	✅	✅	✅	✅	✅	❌
Governance	✅ Limited	✅ Full	✅ Limited	✅ Full	✅ Full	✅ Full	❌
Buyback/Burn	✅ 30%	✅	✅	✅	✅	✅	✅
Max Supply	1B	1B	10B	555M	1B	10B	Unlimited
Primary Use	Perps DEX	Perps DEX	Perps DEX	AMM DEX	Privacy L2	Aggregator	Launchpad

Unique Differentiators:

- **DARK**: Dark pool focus, highest leverage (1000x), ephemeral wallets
- **HYPE**: Custom L1, zero gas fees, highest TVL

- **ASTER**: Cross-chain bridges, institutional focus
- **RAY**: Oldest Solana DEX, deepest liquidity
- **MANTA**: Zero-knowledge privacy, modular L2
- **JUP**: Largest aggregator, best prices
- **PUMP**: Memecoin launchpad, viral growth

Token Launch Plan

Phase 1: TGE (Token Generation Event)

- Initial DEX offering (IDO) on Jupiter/Raydium
- 5% of supply (50M DARK)
- Price discovery through bonding curve
- Initial liquidity: $500K
- Launch partners: Jupiter, Raydium, Phantom

Phase 2: Liquidity Bootstrapping (Month 1-3)

- High APY liquidity mining (100%+ APY)
- Trading competitions
- Referral bonuses
- Early adopter airdrops

Phase 3: Utility Activation (Month 3-6)

- Staking program launch
- Fee discounts go live
- First revenue distribution
- First buyback and burn

Phase 4: Governance (Month 6-12)

- Governance proposals enabled
- Community voting begins
- DAO treasury formed
- Long-term roadmap

---

11. Technology Stack

Blockchain Layer

Solana

Core Blockchain:

- Network: Solana Mainnet Beta
- Consensus: Proof of History (PoH) + Proof of Stake (PoS)
- Block time: ~400ms
- Throughput: 65,000 TPS (theoretical)
- Finality: ~12 seconds (confirmed)

Why Solana:

- High performance for trading applications
- Low transaction costs (~$0.00025 per transaction)
- Native support for high-frequency operations
- Robust ecosystem and tooling
- Growing DeFi infrastructure

Solana Program Development:

```
// Technology: Anchor Framework
// Version: 0.29.0+
// Language: Rust 1.70+

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

declare_id!("GoDark11111111111111111111111111111111111111");

#[program]
pub mod godark_perps {
    use super::*;

    // Program instructions
    pub fn initialize_market(ctx: Context<InitializeMarket>, params: MarketParams) -> Result<()> {
        // Market initialization logic
    }

    pub fn open_position(ctx: Context<OpenPosition>, params: PositionParams) -> Result<()> {
        // Position opening logic
    }

    // Additional instructions...
}
```

SPL Token (USDT)

Token Standard: SPL Token Program

- USDT: Circle's USD Coin or Tether wrapped on Solana
- Mint address: (Production address TBD)
- Decimals: 6
- All collateral and fees in USDT

Token Operations:

```
// Transfer USDT
token::transfer(
    CpiContext::new(
        token_program.to_account_info(),
        Transfer {
            from: user_account,
            to: vault_account,
            authority: user,
        }
    ),
    amount
)?;
```

Oracle Integration

Primary: Pyth Network

- Real-time price feeds
- Sub-second updates
- Confidence intervals
- Price: ~$0.01 per update

```rust
use solana_client::rpc_client::RpcClient;
use solana_sdk::pubkey::Pubkey;
use pyth_sdk_solana::load_price_feed_from_account_info;

pub struct PythClient {
    rpc_client: RpcClient,
}

impl PythClient {
    pub fn new(rpc_url: &str) -> Self {
        Self {
            rpc_client: RpcClient::new(rpc_url.to_string()),
        }
    }

    pub async fn get_price(&self, price_feed_id: &Pubkey) -> Result<f64, PythError> {
        let account_info = self.rpc_client.get_account(price_feed_id)?;
        let price_feed = load_price_feed_from_account_info(&price_feed_id, &account_info)?;

        let price = price_feed.get_current_price()
            .ok_or(PythError::PriceUnavailable)?;

        Ok(price.price as f64 / 10_f64.powi(price.expo))
    }
}
```

Secondary: Switchboard

- Decentralized oracle network
- Backup price source
- Customizable feeds
- Community-driven

Price Validation:

```rust
// Use median of both oracles
pub async fn get_validated_price(
    pyth_client: &PythClient,
    switchboard_client: &SwitchboardClient,
    symbol: &str,
) -> Result<f64, OracleError> {
    let pyth_price = pyth_client.get_price(symbol).await?;
    let switchboard_price = switchboard_client.get_price(symbol).await?;

    let mut prices = vec![pyth_price, switchboard_price];
    prices.sort_by(|a, b| a.partial_cmp(b).unwrap());

    // Return median
    let validated_price = if prices.len() % 2 == 0 {
        (prices[prices.len() / 2 - 1] + prices[prices.len() / 2]) / 2.0
    } else {
        prices[prices.len() / 2]
    };

    Ok(validated_price)
}
```

Backend Services

Rust

Framework: Axum + Tokio (Async Runtime)

```
# Cargo.toml
[dependencies]
axum = "0.7"
tokio = { version = "1.35", features = ["full"] }
tower = "0.4"
tower-http = { version = "0.5", features = ["cors", "trace"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
sqlx = { version = "0.7", features = ["postgres", "runtime-tokio-native-tls"] }
redis = { version = "0.24", features = ["tokio-comp", "connection-manager"] }
anchor-client = "0.29"
solana-sdk = "1.17"
jsonwebtoken = "9.0"
bcrypt = "0.15"
validator = "0.16"
```

Server Architecture:

```
// Main application server
use axum::{
    Router,
    routing::{get, post, delete},
    middleware,
};
use tower_http::cors::CorsLayer;
use std::net::SocketAddr;

#[tokio::main]
async fn main() {
    // Initialize application state
    let app_state = AppState::new().await;

    // Build router
    let app = Router::new()
        .route("/health", get(health_check))
        .nest("/api/v1/auth", auth_routes())
        .nest("/api/v1/orders", order_routes())
        .nest("/api/v1/positions", position_routes())
        .nest("/api/v1/markets", market_routes())
        .layer(CorsLayer::permissive())
        .layer(middleware::from_fn(rate_limit_middleware))
        .layer(middleware::from_fn(auth_middleware))
        .with_state(app_state);

    // Start server
    let addr = SocketAddr::from(([0, 0, 0, 0], 3000));
    println!("GoDark API running on {}", addr);

    axum::Server::bind(&addr)
        .serve(app.into_make_service())
        .await
        .unwrap();
}
```

Matching Engine

Custom Implementation in Rust:

```rust
// High-performance order matching
use std::collections::HashMap;
use tokio::sync::RwLock;
use redis::AsyncCommands;

pub struct MatchingEngine {
    order_books: RwLock<HashMap<String, OrderBook>>,
    redis: redis::aio::ConnectionManager,
}

impl MatchingEngine {
    pub async fn new(redis_url: &str) -> Self {
        let client = redis::Client::open(redis_url).unwrap();
        let redis = redis::aio::ConnectionManager::new(client).await.unwrap();

        Self {
            order_books: RwLock::new(HashMap::new()),
            redis,
        }
    }

    pub async fn process_order(&self, order: Order) -> Result<Vec<Trade>, MatchError> {
        // Get order book
        let mut books = self.order_books.write().await;
        let order_book = books.entry(order.symbol.clone())
            .or_insert_with(|| OrderBook::new(order.symbol.clone()));

        // Match order
        let trades = order_book.match_order(order).await?;

        // Persist trades to Redis
        self.persist_trades(&trades).await?;

        // Emit to settlement queue
        self.emit_to_settlement(&trades).await?;

        Ok(trades)
    }
}
```

Performance Optimizations:

- In-memory order books with RwLock
- Redis for persistence
- Lock-free data structures where possible
- Batch processing
- CPU affinity for critical threads
- Zero-copy serialization with bincode

Settlement Relayer

Dedicated Service:

```rust
// Settlement relayer service
use anchor_client::{Client, Cluster};
use solana_sdk::{signature::Keypair, signer::Signer};

pub struct SettlementRelayer {
    client: Client,
    program: anchor_client::Program,
    relayer_keypair: Keypair,
}

impl SettlementRelayer {
    pub async fn start(&self) {
        // Start batch timer (1 second intervals)
        let mut interval = tokio::time::interval(Duration::from_secs(1));

        loop {
            interval.tick().await;
            if let Err(e) = self.settle_batch().await {
                eprintln!("Settlement error: {}", e);
            }
        }
    }

    async fn settle_batch(&self) -> Result<(), SettlementError> {
        let pending_trades = self.get_pending_trades().await?;
        if pending_trades.is_empty() {
            return Ok(());
        }

        let tx = self.build_settlement_transaction(&pending_trades).await?;
        let signature = self.send_and_confirm(tx).await?;

        self.mark_trades_settled(&pending_trades, signature).await?;
        Ok(())
    }
}
```

Databases

PostgreSQL

Version: PostgreSQL 15+
Configuration:

```yaml
# postgresql.conf
max_connections: 200
shared_buffers: 16GB
effective_cache_size: 48GB
maintenance_work_mem: 2GB
checkpoint_completion_target: 0.9
wal_buffers: 16MB
default_statistics_target: 100
random_page_cost: 1.1
effective_io_concurrency: 200
work_mem: 20MB
min_wal_size: 1GB
max_wal_size: 4GB
max_worker_processes: 8
max_parallel_workers_per_gather: 4
max_parallel_workers: 8
max_parallel_maintenance_workers: 4
```

Connection Pooling:

```rust
use sqlx::postgres::{PgPoolOptions, PgPool};
use std::time::Duration;

pub async fn create_pg_pool() -> Result<PgPool, sqlx::Error> {
    let database_url = std::env::var("DATABASE_URL")
        .expect("DATABASE_URL must be set");

    let pool = PgPoolOptions::new()
        .max_connections(20)                    // Max connections
        .idle_timeout(Duration::from_secs(30))  // 30 seconds idle timeout
        .acquire_timeout(Duration::from_secs(2)) // 2 seconds connection timeout
        .connect(&database_url)
        .await?;

    Ok(pool)
}
```

Backup Strategy:

- Continuous archiving with WAL
- Point-in-time recovery enabled
- Daily full backups
- Hourly incremental backups
- Replication to standby server

Redis

Version: Redis 7.0+
Configuration:

```
# redis.conf
maxmemory 32gb
maxmemory-policy allkeys-lru
save ""
appendonly yes
appendfsync everysec
tcp-backlog 511
timeout 0
tcp-keepalive 300
```

Cluster Setup:

```rust
use redis::cluster::{ClusterClient, ClusterConnection};
use redis::ConnectionInfo;

pub fn create_redis_cluster() -> Result<ClusterConnection, redis::RedisError> {
    let password = std::env::var("REDIS_PASSWORD")
        .expect("REDIS_PASSWORD must be set");

    let nodes = vec![
        "redis://redis-1:6379",
        "redis://redis-2:6379",
        "redis://redis-3:6379",
        "redis://redis-4:6379",
        "redis://redis-5:6379",
        "redis://redis-6:6379",
    ];

    let client = ClusterClient::builder(nodes)
        .password(password)
        .build()?;

    let connection = client.get_connection()?;

    Ok(connection)
}
```

Use Cases:

- Order book storage
- Price caching
- Session management
- Rate limiting
- Real-time leaderboards

Infrastructure

On-Premise Solana RPC Nodes

Hardware Requirements:

```
Server Specifications:
  CPU: AMD Threadripper 3970X (32 cores) or better
  RAM: 256GB ECC
  Storage: 2TB NVMe SSD (RAID 1)
  Network: 10Gbps
  OS: Ubuntu 22.04 LTS

RPC Configuration:
  - Full archive node
  - WebSocket support enabled
  - Custom rate limits
  - Priority fee optimization
```

Node Setup:

```
# Install Solana
sh -c "$(curl -sSfL https://release.solana.com/v1.17.0/install)"

# Configure validator
solana-validator \
    --identity validator-keypair.json \
    --vote-account vote-keypair.json \
    --ledger /mnt/ledger \
    --rpc-port 8899 \
    --rpc-bind-address 0.0.0.0 \
    --dynamic-port-range 8000-8020 \
    --entrypoint entrypoint.mainnet-beta.solana.com:8001 \
    --expected-genesis-hash 5eykt4UsFv8P8NJdTREpY1vzqKqZKvdpKuc147dw2N9d \
    --wal-recovery-mode skip_any_corrupted_record \
    --limit-ledger-size
```

RPC Endpoints:

- Primary: `https://rpc-1.godark.internal`
- Backup: `https://rpc-2.godark.internal`
- Public: `https://rpc.godark.xyz` (rate limited)

Load Balancers

Technology: HAProxy

```
# haproxy.cfg
frontend api_frontend
    bind *:443 ssl crt /etc/ssl/certs/godark.pem
    default_backend api_backend

backend api_backend
    balance leastconn
    option httpchk GET /health
    server api1 10.0.1.10:3000 check
    server api2 10.0.1.11:3000 check
    server api3 10.0.1.12:3000 check
```

Features:

- SSL termination
- Health checks
- Sticky sessions for WebSocket
- Rate limiting
- Geographic routing

Monitoring

Prometheus + Grafana

```
# prometheus.yml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'godark-api'
    static_configs:
      - targets: ['api-1:9090', 'api-2:9090']

  - job_name: 'postgres'
    static_configs:
      - targets: ['postgres-1:9187']

  - job_name: 'redis'
    static_configs:
      - targets: ['redis-1:9121']

  - job_name: 'solana-node'
    static_configs:
      - targets: ['rpc-1:9100']
```

Grafana Dashboards:

- System Overview
- Trading Metrics
- Database Performance
- Blockchain State
- User Activity
- Security Events

Alerting:

```
# alerts.yml
groups:
  - name: critical
    rules:
      - alert: HighAPILatency
        expr: api_latency_p95 > 200
        for: 5m
        annotations:
          summary: "API latency too high"

      - alert: SettlementFailed
        expr: settlement_failures > 3
        for: 1m
        annotations:
          summary: "Multiple settlement failures"
```

Development Tools

Anchor CLI

Installation:

```
cargo install --git https://github.com/coral-xyz/anchor avm --locked --force
avm install 0.29.0
avm use 0.29.0
```

Project Structure:

```
godark-perps/
├── Anchor.toml
├── Cargo.toml
├── programs/
│   └── godark-perps/
│       ├── Cargo.toml
│       └── src/
│           ├── lib.rs
│           ├── instructions/
│           ├── state/
│           └── errors.rs
├── tests/
│   └── godark-perps.ts
└── migrations/
    └── deploy.ts
```

Common Commands:

```
# Build program
anchor build

# Test
anchor test

# Deploy to devnet
anchor deploy --provider.cluster devnet

# Deploy to mainnet
anchor deploy --provider.cluster mainnet
```

Solana CLI

Common Operations:

```
# Check balance
solana balance

# Transfer SOL
solana transfer <recipient> <amount>

# View account
solana account <address>

# Program deployment
solana program deploy <program.so>

# Create token
spl-token create-token

# Create token account
spl-token create-account <token-mint>
```

Git / GitHub

Repository Structure:

```
godark-monorepo/
├── packages/
│   ├── smart-contracts/     # Anchor programs
│   ├── backend-api/         # Node.js API
│   ├── matching-engine/     # Order matching
│   ├── settlement-relayer/  # Settlement service
│   ├── frontend/            # React UI
│   └── shared/              # Shared types
├── .github/
│   └── workflows/
│       ├── test.yml
│       ├── deploy-testnet.yml
│       └── deploy-mainnet.yml
└── docs/
```

Branch Strategy:

- `main`: Production
- `develop`: Integration
- `feature/*`: Feature branches
- `hotfix/*`: Emergency fixes

CI/CD Pipeline

GitHub Actions:

```yaml
# .github/workflows/deploy.yml
name: Deploy to Mainnet

on:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Install Rust
        uses: actions-rs/toolchain@v1
      - name: Run tests
        run: anchor test

  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Build programs
        run: anchor build
      - name: Upload artifacts
        uses: actions/upload-artifact@v3

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to mainnet
        run: anchor deploy --provider.cluster mainnet
        env:
          ANCHOR_WALLET: ${{ secrets.DEPLOY_WALLET }}
```

Deployment Checklist:

- [ ] All tests passing
- [ ] Security audit completed
- [ ] Mainnet balance sufficient
- [ ] Multisig approval obtained
- [ ] Rollback plan prepared
- [ ] Monitoring alerts configured

- [ ] User notification sent

Additional Tools & Libraries

Backend Dependencies:

```
{
  "security": {
    "helmet": "^7.0.0",
    "express-rate-limit": "^6.10.0",
    "bcrypt": "^5.1.0",
    "jsonwebtoken": "^9.0.2"
  },
  "validation": {
    "zod": "^3.22.0",
    "validator": "^13.11.0"
  },
  "logging": {
    "winston": "^3.10.0",
    "morgan": "^1.10.0"
  },
  "testing": {
    "jest": "^29.6.0",
    "@types/jest": "^29.5.0",
    "supertest": "^6.3.0"
  }
}
```

Solana SDK:

```
import {
    Connection,
    Keypair,
    PublicKey,
    Transaction,
    sendAndConfirmTransaction
} from '@solana/web3.js';
import * as anchor from '@coral-xyz/anchor';
import { Program } from '@coral-xyz/anchor';
```

Monitoring & Observability:

- Sentry: Error tracking
- DataDog: APM
- Grafana: Dashboards
- Prometheus: Metrics collection
- ELK Stack: Log aggregation

---

12. Deployment & Operations

Development Phases

Phase 1: Testnet Deployment (Months 1-2)

Objectives:

- Deploy all smart contracts to Solana Devnet
- Test core functionality in isolation
- Conduct internal security review
- Performance testing with simulated load

Deliverables:

```
const PHASE_1_DELIVERABLES = {
    smartContracts: [
        'Market initialization',
        'Position management',
        'Vault operations',
        'Settlement logic',
        'Liquidation engine'
    ],
    backend: [
        'REST API (all endpoints)',
        'WebSocket server',
        'Matching engine',
        'Settlement relayer',
        'Liquidation monitor'
    ],
    infrastructure: [
        'Devnet RPC node',
        'PostgreSQL database',
        'Redis cluster',
        'Monitoring stack'
    ],
    testing: [
        'Unit tests (100% coverage)',
        'Integration tests',
        'Load tests (100 TPS)',
        'Security audit (preliminary)'
    ]
};
```

Success Criteria:

- All tests passing
- 100+ simulated users
- 1000+ test trades executed
- Zero critical bugs
- API latency < 100ms p95
- Settlement success rate > 99%

Phase 2: Mainnet Beta (Months 3-4)

Objectives:

- Deploy to Solana Mainnet with limited access
- Invite selected beta testers
- Test with real value (limited amounts)
- Monitor system behavior under real conditions

Beta Program:

```
interface BetaProgram {
    participants: {
        internalTeam: 10,
        earlyAdopters: 50,
        marketMakers: 5,
        total: 65
    },
    limits: {
        maxDepositPerUser: 1000,      // USDT
        maxPositionSize: 500,         // USDT
        maxLeverage: 20,              // Conservative
        totalTVL: 50000               // USDT
    },
    monitoring: {
        24x7: true,
        manualApproval: true,
        emergencyPause: true
    }
}
```

Beta Phases:

```
Week 1-2: Internal testing (10 users, $10K TVL)
├── Core team trades
├── System monitoring
└── Bug fixes

Week 3-4: Limited beta (25 users, $25K TVL)
├── Invite early supporters
├── Collect feedback
└── Optimize UX

Week 5-6: Expanded beta (50 users, $50K TVL)
├── Add market makers
├── Increase limits
└── Stress testing

Week 7-8: Pre-launch prep
├── Final audit
├── Documentation
└── Marketing preparation
```

Success Criteria:

- Zero settlement failures
- No security incidents
- User feedback positive (>4/5 stars)
- All edge cases handled
- Ready for public launch

Phase 3: Full Production (Month 5+)

Launch Checklist:

```
const LAUNCH_CHECKLIST = {
    technical: [
        '✓ Smart contracts audited (OtterSec + Neodyme)',
        '✓ Bug bounty program live',
        '✓ Mainnet RPC nodes operational',
        '✓ Database replication configured',
        '✓ Monitoring and alerting active',
        '✓ Disaster recovery tested',
        '✓ Rate limiting configured',
        '✓ Geo-restrictions implemented'
    ],
    legal: [
        '✓ Terms of service published',
        '✓ Privacy policy published',
        '✓ Legal entity established',
        '✓ Compliance review completed'
    ],
    marketing: [
        '✓ Website live',
        '✓ Documentation complete',
        '✓ Social media accounts created',
        '✓ Community channels active',
        '✓ Launch announcement prepared'
    ],
    operations: [
        '✓ On-call rotation established',
        '✓ Runbooks documented',
        '✓ Support channels ready',
        '✓ Incident response plan tested'
    ]
};
```

Launch Day:

1. Pre-launch verification (T-2 hours)
2. System health check (T-1 hour)
3. Remove beta restrictions (T-0)
4. Announce publicly

5. Monitor closely (first 24 hours)
6. Daily check-ins (first week)

Post-Launch Monitoring:

```
const POST_LAUNCH_KPIS = {
    day1: {
        users: 100,
        trades: 500,
        volume: 50000,
        uptime: 0.99
    },
    week1: {
        users: 500,
        trades: 5000,
        volume: 500000,
        uptime: 0.995
    },
    month1: {
        users: 2000,
        trades: 50000,
        volume: 5000000,
        uptime: 0.999
    }
};
```

Environment Setup

Development Environment

Local Development:

```
# Setup script
#!/bin/bash

# 1. Install dependencies
yarn install

# 2. Start local Solana validator
solana-test-validator \
    --reset \
    --ledger /tmp/test-ledger \
    --rpc-port 8899

# 3. Deploy programs
anchor deploy

# 4. Start PostgreSQL
docker-compose up -d postgres

# 5. Run migrations
yarn db:migrate

# 6. Start Redis
docker-compose up -d redis

# 7. Start API server
yarn dev:api

# 8. Start matching engine
yarn dev:matching

# 9. Start frontend
yarn dev:frontend
```

Docker Compose:

```
version: '3.8'

services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: godark_dev
      POSTGRES_USER: dev
      POSTGRES_PASSWORD: devpass
    ports:
      - "5432:5432"
    volumes:
      - pg_data:/var/lib/postgresql/data

  redis:
    image: redis:7
    ports:
      - "6379:6379"
    command: redis-server --appendonly yes

  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
    environment:
      GF_SECURITY_ADMIN_PASSWORD: admin

volumes:
  pg_data:
```

Staging Environment

Configuration:

```
# staging.env
NODE_ENV=staging
API_PORT=3000
SOLANA_CLUSTER=devnet
SOLANA_RPC_URL=https://api.devnet.solana.com

DATABASE_URL=postgresql://staging:xxx@postgres-staging/godark
REDIS_URL=redis://redis-staging:6379

LOG_LEVEL=debug
ENABLE_METRICS=true
ENABLE_TRACING=true
```

Purpose:

- Final testing before production
- Mirrors production environment
- Safe for breaking changes
- Used for QA and integration tests

Production Environment

Multi-Region Architecture:

```
Primary Region (US-East-1)
├── API Servers (3x)
├── Matching Engine (1x active, 2x standby)
├── Settlement Relayer (1x active, 2x standby)
├── PostgreSQL Primary (1x)
├── PostgreSQL Replicas (2x)
├── Redis Cluster (6 nodes)
└── Solana RPC Nodes (2x)

Backup Region (US-West-2)
├── API Servers (2x)
├── PostgreSQL Replica (1x)
├── Redis Cluster (3 nodes)
└── Solana RPC Node (1x)
```

Environment Variables:

```
# production.env
NODE_ENV=production
API_PORT=3000
SOLANA_CLUSTER=mainnet-beta
SOLANA_RPC_URL=https://rpc-1.godark.internal

DATABASE_URL=postgresql://prod:xxx@postgres-prod/godark
REDIS_URL=redis://redis-prod:6379

LOG_LEVEL=info
ENABLE_METRICS=true
SENTRY_DSN=https://xxx@sentry.io/xxx

# Security
JWT_SECRET=xxx
API_KEY_SALT=xxx
ENCRYPTION_KEY=xxx

# Rate Limiting
RATE_LIMIT_WINDOW_MS=60000
RATE_LIMIT_MAX_REQUESTS=60
```

Deployment Procedures

Smart Contract Deployment

Deployment Script:

```
#!/bin/bash
# deploy-contracts.sh

set -e

echo "🚀 Deploying GoDark Smart Contracts to Mainnet"

# 1. Verify we're on correct branch
BRANCH=$(git branch --show-current)
if [ "$BRANCH" != "main" ]; then
    echo "❌ Must be on main branch"
    exit 1
fi

# 2. Build programs
echo "📦 Building programs..."
anchor build

# 3. Run tests
echo "🧪 Running tests..."
anchor test

# 4. Get program IDs
PROGRAM_ID=$(solana address -k target/deploy/godark_perps-keypair.json)
echo "Program ID: $PROGRAM_ID"

# 5. Verify sufficient SOL for deployment
BALANCE=$(solana balance)
echo "Deployer balance: $BALANCE"

# 6. Deploy
echo "📤 Deploying to mainnet..."
anchor deploy --provider.cluster mainnet --program-name godark_perps

# 7. Verify deployment
echo "✅ Verifying deployment..."
solana program show $PROGRAM_ID

# 8. Initialize markets
echo "🎯 Initializing markets..."
ts-node scripts/initialize-markets.ts

# 9. Set up governance
echo "🗳️ Setting up multisig..."
ts-node scripts/setup-governance.ts

echo "✅ Deployment complete!"
```

Rollback Plan:

```
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 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(())
}
```

Program Upgrades

Upgrade Process:

```rust
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
        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
        self.notification_service.notify_community(Notification {
            notification_type: NotificationType::ProgramUpgrade,
            proposal_id: proposal_id.clone(),
            changes: "See: github.com/godark/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
        self.wait_for_timelock(proposal_id).await?;

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

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

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

        Ok(())
    }
}
```

Database Migrations

Migration Tool: node-pg-migrate

```
// migrations/1699999999999_initial_schema.ts
import { MigrationBuilder } from 'node-pg-migrate';

export async function up(pgm: MigrationBuilder): Promise<void> {
    // Create accounts table
    pgm.createTable('accounts', {
        id: { type: 'uuid', primaryKey: true, default: pgm.func('gen_random_uuid()') },
        email: { type: 'varchar(255)', notNull: true, unique: true },
        password_hash: { type: 'varchar(255)', notNull: true },
        created_at: { type: 'bigint', notNull: true }
    });

    // Create indexes
    pgm.createIndex('accounts', 'email');

    // Create orders table
    pgm.createTable('orders', {
        // ...schema
    });
}

export async function down(pgm: MigrationBuilder): Promise<void> {
    pgm.dropTable('orders');
    pgm.dropTable('accounts');
}
```

Migration Execution:

```
# Run migrations
npm run migrate up

# Rollback last migration
npm run migrate down

# Check migration status
npm run migrate status
```

Monitoring and Alerting

System Health Checks

```rust
use reqwest;
use std::time::Instant;

pub struct HealthMonitor {
    db: Arc<DatabasePool>,
    redis: Arc<RedisPool>,
    solana_client: Arc<RpcClient>,
    matching_engine: Arc<MatchingEngine>,
    settlement_relayer: Arc<SettlementRelayer>,
}

#[derive(Debug, Serialize)]
pub struct HealthReport {
    pub status: HealthStatus,
    pub checks: Vec<HealthCheck>,
    pub timestamp: i64,
}

#[derive(Debug, Serialize)]
pub enum HealthStatus {
    Healthy,
    Degraded,
}

#[derive(Debug, Serialize)]
pub struct HealthCheck {
    pub service: String,
    pub healthy: bool,
    pub latency: Option<u64>,
    pub error: Option<String>,
}

impl HealthMonitor {
    pub async fn perform_health_check(&self) -> Result<HealthReport, MonitorError> {
        let checks = tokio::join!(
            self.check_api(),
            self.check_database(),
            self.check_redis(),
            self.check_solana_rpc(),
            self.check_matching_engine(),
            self.check_settlement_relayer()
        );

        let checks_vec = vec![checks.0, checks.1, checks.2, checks.3, checks.4, checks.5];
        let all_healthy = checks_vec.iter().all(|c| c.healthy);

        Ok(HealthReport {
            status: if all_healthy { HealthStatus::Healthy } else { HealthStatus::Degraded },
            checks: checks_vec,
            timestamp: chrono::Utc::now().timestamp(),
        })
    }

    async fn check_api(&self) -> HealthCheck {
        let start = Instant::now();
        match reqwest::get("http://localhost:3000/health").await {
            Ok(_) => HealthCheck {
                service: "API".to_string(),
                healthy: true,
                latency: Some(start.elapsed().as_millis() as u64),
                error: None,
            },
            Err(e) => HealthCheck {
                service: "API".to_string(),
                healthy: false,
                latency: None,
                error: Some(e.to_string()),
            },
        }
    }

    // Similar checks for other services...
}
```

Health Endpoint:

```rust
// Health endpoint with Axum
use axum::{extract::State, http::StatusCode, Json, response::IntoResponse};

pub async fn health_handler(
    State(health_monitor): State<Arc<HealthMonitor>>,
) -> impl IntoResponse {
    match health_monitor.perform_health_check().await {
        Ok(health) => {
            let status = match health.status {
                HealthStatus::Healthy => StatusCode::OK,
                HealthStatus::Degraded => StatusCode::SERVICE_UNAVAILABLE,
            };
            (status, Json(health))
        }
        Err(e) => (
            StatusCode::INTERNAL_SERVER_ERROR,
            Json(serde_json::json!({ "error": e.to_string() }))
        ),
    }
}
```

Settlement Failure Alerts

```rust
pub struct SettlementMonitor {
    db: Arc<DatabasePool>,
    alert_service: Arc<AlertService>,
}

#[derive(Debug, Serialize)]
pub struct Alert {
    pub severity: AlertSeverity,
    pub title: String,
    pub description: String,
    pub actions: Vec<String>,
}

#[derive(Debug, Serialize)]
pub enum AlertSeverity {
    Critical,
    Warning,
    Info,
}

impl SettlementMonitor {
    pub async fn monitor_settlements(&self) -> Result<(), MonitorError> {
        let failures = self.get_recent_failures(300).await?; // Last 5 min

        if failures.len() > 3 {
            self.alert_service.send_alert(Alert {
                severity: AlertSeverity::Critical,
                title: "Multiple settlement failures".to_string(),
                description: format!("{} settlements failed in last 5 minutes", failures.len()),
                actions: vec![
                    "Check Solana RPC health".to_string(),
                    "Verify relayer balance".to_string(),
                    "Review transaction logs".to_string(),
                ],
            }).await?;
        }

        Ok(())
    }
}
```

Liquidation Monitoring

```rust
pub struct LiquidationMonitor {
    db: Arc<DatabasePool>,
    alert_service: Arc<AlertService>,
    insurance_fund: Arc<InsuranceFundManager>,
}

impl LiquidationMonitor {
    pub async fn monitor_liquidations(&self) -> Result<(), MonitorError> {
        let underwater_positions = self.get_underwater_positions().await?;

        if underwater_positions.len() > 10 {
            let insurance_balance = self.insurance_fund.get_balance().await?;
            let total_exposure = self.calculate_total_exposure(&underwater_positions)?;

            self.alert_service.send_alert(Alert {
                severity: AlertSeverity::Warning,
                title: "High liquidation risk".to_string(),
                description: format!("{} positions underwater", underwater_positions.len()),
                actions: vec![
                    format!("Insurance fund balance: ${:.2}", insurance_balance),
                    format!("Total exposure: ${:.2}", total_exposure),
                ],
            }).await?;
        }

        Ok(())
    }
}
```

Performance Metrics

Key Metrics Dashboard:

```javascript
const PERFORMANCE_METRICS = {
    // Throughput
    'api.requests_per_second': 'Counter',
    'trades.per_second': 'Counter',
    'orders.per_second': 'Counter',

    // Latency
    'api.response_time': 'Histogram',
    'matching.latency': 'Histogram',
    'settlement.latency': 'Histogram',

    // Errors
    'api.errors': 'Counter',
    'settlement.failures': 'Counter',
    'liquidation.failures': 'Counter',

    // Business
    'users.active': 'Gauge',
    'positions.open': 'Gauge',
    'total.open_interest': 'Gauge',
    'tvl': 'Gauge'
};
```

Incident Response Procedures

Incident Severity Levels

```
enum IncidentSeverity {
    P0 = 'CRITICAL',      // System down, funds at risk
    P1 = 'HIGH',          // Major feature broken
    P2 = 'MEDIUM',        // Minor issue, workaround available
    P3 = 'LOW'            // Cosmetic or minor bug
}

const RESPONSE_TIMES = {
    P0: '15 minutes',
    P1: '1 hour',
    P2: '4 hours',
    P3: '24 hours'
};
```

Incident Response Workflow

```
1. Detection
   ├── Automated alert
   ├── User report
   └── Monitoring system

2. Triage (Within 15 min for P0)
   ├── Assess severity
   ├── Page on-call engineer
   └── Create incident ticket

3. Investigation
   ├── Check logs
   ├── Review metrics
   ├── Identify root cause
   └── Document findings

4. Resolution
   ├── Apply fix
   ├── Deploy hotfix
   ├── Verify resolution
   └── Monitor closely

5. Post-Mortem
   ├── Write incident report
   ├── Identify improvements
   ├── Update runbooks
   └── Share learnings
```

System Downtime Handling

Planned Maintenance:

```rust
pub async fn schedule_maintenance(
    window: MaintenanceWindow,
    notification_service: &NotificationService,
    system_manager: &SystemManager,
) -> Result<(), MaintenanceError> {
    // 1. Announce 48 hours in advance
    notification_service.announce_maintenance(MaintenanceAnnouncement {
        start_time: window.start,
        duration: window.duration,
        reason: window.reason.clone(),
    }).await?;

    // 2. At maintenance start
    system_manager.pause_new_orders().await?;
    system_manager.close_existing_orders().await?;
    system_manager.enable_withdrawals_only().await?;

    // 3. Perform maintenance
    system_manager.execute_maintenance(window.tasks).await?;

    // 4. Resume operations
    system_manager.verify_system_health().await?;
    system_manager.resume_full_operations().await?;

    // 5. Announce completion
    notification_service.announce_completion().await?;

    Ok(())
}
```

Unplanned Outage:

```rust
pub async fn handle_outage(
    incident_service: &IncidentService,
    notification_service: &NotificationService,
) -> Result<(), OutageError> {
    // 1. Detect outage
    let outage = incident_service.detect_outage().await?;

    // 2. Activate incident response
    incident_service.activate_incident_response(&outage).await?;

    // 3. Communicate to users
    notification_service.broadcast_status(StatusUpdate {
        status: SystemStatus::Investigating,
        message: "We are investigating an issue...".to_string(),
    }).await?;

    // 4. Implement fix
    incident_service.implement_fix(&outage).await?;

    // 5. Post-mortem
    incident_service.write_post_mortem(&outage).await?;

    Ok(())
}
```

Backup and Disaster Recovery

RTO/RPO Targets:

```
const DISASTER_RECOVERY_TARGETS = {
    RTO: 4 * 60 * 60,          // Recovery Time Objective: 4 hours
    RPO: 15 * 60,              // Recovery Point Objective: 15 minutes

    backupFrequency: {
        postgres: '6 hours',
        redis: '1 hour',
        solanaState: 'real-time'
    },

    retentionPeriod: {
        hot: 7,                // days
        warm: 30,              // days
        cold: 365 * 7          // 7 years
    }
};
```

Disaster Recovery Drill:

```
async function performDRDrill(): Promise<DRReport> {
    console.log('🚨 Starting DR drill...');

    // 1. Simulate failure
    const failure = await simulateFailure();

    // 2. Activate DR procedures
    const start = Date.now();
    await activateDRProcedures(failure);

    // 3. Restore from backup
    await restoreFromBackup();

    // 4. Verify integrity
    const verificationResults = await verifyDataIntegrity();

    // 5. Measure recovery time
    const recoveryTime = Date.now() - start;

    return {
        failureType: failure.type,
        recoveryTime,
        rtoMet: recoveryTime < DISASTER_RECOVERY_TARGETS.RTO * 1000,
        dataLoss: verificationResults.dataLoss,
        rpoMet: verificationResults.dataLoss < DISASTER_RECOVERY_TARGETS.RPO,
        success: verificationResults.success
    };
}
```

Upgrade and Migration Strategy

Zero-Downtime Deployments:

```typescript
async function deployZeroDowntime(newVersion: string): Promise<void> {
    // 1. Deploy new version alongside old
    await deployNewVersion(newVersion);

    // 2. Run health checks
    await verifyNewVersion();

    // 3. Gradually shift traffic (10% -> 50% -> 100%)
    await gradualTrafficShift(newVersion);

    // 4. Monitor for issues
    await monitorDeployment(newVersion);

    // 5. Rollback if needed
    if (await detectIssues()) {
        await rollback();
    }

    // 6. Decommission old version
    await removeOldVersion();
}
```

---

13. Compliance & Legal

Geo-Restrictions

GoDark implements comprehensive geographic restrictions to comply with regulatory requirements and reduce legal risk.

Restricted Regions

Complete List of Blocked Jurisdictions:

North America:

- United States (all states and territories)
- Canada: Quebec and Ontario only

Central and South America:

- Bolivia
- Ecuador
- Cuba

Europe:

- United Kingdom
- France
- Belarus
- Russia
- Ukraine (entire country)
- Crimea (UA-40)
- Donetsk (UA-14)
- Luhansk (UA-09)

Asia:

- China (mainland, excluding Hong Kong and Macau)
- North Korea
- Iran
- Iraq
- Syria
- Bangladesh
- Myanmar
- Nepal

Africa:

- Algeria
- Morocco
- Egypt

IP Blocking Implementation

Geographic Detection Service

```rust
use lru::LruCache;
use std::collections::HashSet;
use std::num::NonZeroUsize;
use std::sync::{Arc, Mutex};
use reqwest;
use serde::{Deserialize, Serialize};

pub struct GeoRestrictionService {
    geo_cache: Arc<Mutex<LruCache<String, GeoLocation>>>,
    blocked_countries: HashSet<String>,
    blocked_regions: HashSet<String>,
    http_client: reqwest::Client,
    db: Arc<DatabasePool>,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct GeoLocation {
    pub ip: String,
    pub country: String,
    pub region: Option<String>,
    pub city: Option<String>,
    pub latitude: f64,
    pub longitude: f64,
    pub timestamp: i64,
}

#[derive(Debug, Serialize)]
pub struct AccessCheckResult {
    pub allowed: bool,
    pub country: String,
    pub region: Option<String>,
    pub is_vpn: bool,
    pub reason: Option<String>,
}

impl GeoRestrictionService {
    pub fn new(db: Arc<DatabasePool>) -> Self {
        let blocked_countries = HashSet::from([
            "US", "CA-QC", "CA-ON", "BO", "EC", "CU",
            "GB", "FR", "BY", "RU", "UA",
            "CN", "KP", "IR", "IQ", "SY", "BD", "MM", "NP",
            "DZ", "MA", "EG"
        ].map(String::from));

        let blocked_regions = HashSet::from([
            "UA-40",  // Crimea
            "UA-14",  // Donetsk
            "UA-09"   // Luhansk
        ].map(String::from));

        Self {
            geo_cache: Arc::new(Mutex::new(LruCache::new(NonZeroUsize::new(10000).unwrap()))),
            blocked_countries,
            blocked_regions,
            http_client: reqwest::Client::new(),
            db,
        }
    }

    pub async fn check_access(&self, ip: &str) -> Result<AccessCheckResult, GeoError> {
        // 1. Check cache
        let mut cache = self.geo_cache.lock().unwrap();
        let geo = if let Some(cached) = cache.get(ip) {
            cached.clone()
        } else {
            drop(cache);
            // 2. Lookup if not cached
            let geo = self.lookup_geo_location(ip).await?;
            let mut cache = self.geo_cache.lock().unwrap();
            cache.put(ip.to_string(), geo.clone());
            geo
        };

        // 3. Check if blocked
        let is_blocked = self.is_restricted(&geo);

        // 4. Check for VPN/Proxy
        let is_vpn = self.detect_vpn(ip).await?;

        // 5. Log attempt if blocked
        if is_blocked || is_vpn {
            self.log_blocked_access(ip, &geo, is_vpn).await?;
        }
```

```
        Ok(AccessCheckResult {
            allowed: !is_blocked && !is_vpn,
            country: geo.country.clone(),
            region: geo.region.clone(),
            is_vpn,
            reason: if is_blocked {
                Some("RESTRICTED_REGION".to_string())
            } else if is_vpn {
                Some("VPN_DETECTED".to_string())
            } else {
                None
            },
        })
    }

    async fn lookup_geo_location(&self, ip: &str) -> Result<GeoLocation, GeoError> {
        // Try MaxMind GeoIP2 first
        let maxmind_id = std::env::var("MAXMIND_ACCOUNT_ID").ok();
        let maxmind_key = std::env::var("MAXMIND_LICENSE_KEY").ok();

        if let (Some(id), Some(key)) = (maxmind_id, maxmind_key) {
            if let Ok(response) = self.http_client
                .get(&format!("https://geoip.maxmind.com/geoip/v2.1/city/{}", ip))
                .basic_auth(id, Some(key))
                .send()
                .await
            {
                if let Ok(data) = response.json::<serde_json::Value>().await {
                    return Ok(GeoLocation {
                        ip: ip.to_string(),
                        country: data["country"]["iso_code"].as_str().unwrap_or("").to_string(),
                        region: data["subdivisions"][0]["iso_code"].as_str().map(String::from),
                        city: data["city"]["names"]["en"].as_str().map(String::from),
                        latitude: data["location"]["latitude"].as_f64().unwrap_or(0.0),
                        longitude: data["location"]["longitude"].as_f64().unwrap_or(0.0),
                        timestamp: chrono::Utc::now().timestamp(),
                    });
                }
            }
        }

        // Fallback to IP-API (free tier)
        let response = self.http_client
            .get(&format!("http://ip-api.com/json/{}", ip))
            .send()
            .await?
            .json::<serde_json::Value>()
            .await?;

        Ok(GeoLocation {
            ip: ip.to_string(),
            country: response["countryCode"].as_str().unwrap_or("").to_string(),
            region: response["region"].as_str().map(String::from),
            city: response["city"].as_str().map(String::from),
            latitude: response["lat"].as_f64().unwrap_or(0.0),
            longitude: response["lon"].as_f64().unwrap_or(0.0),
            timestamp: chrono::Utc::now().timestamp(),
        })
    }

    fn is_restricted(&self, geo: &GeoLocation) -> bool {
        // Check country-level restriction
        if self.blocked_countries.contains(&geo.country) {
            return true;
        }

        // Check region-level restriction
        if let Some(region) = &geo.region {
            let region_code = format!("{}-{}", geo.country, region);
            if self.blocked_regions.contains(&region_code) {
                return true;
            }
        }

        false
    }

    async fn detect_vpn(&self, ip: &str) -> Result<bool, GeoError> {
        let ipqs_key = match std::env::var("IPQS_KEY") {
            Ok(key) => key,
            Err(_) => return Ok(false),
        };

        match self.http_client
```

```
            .get(&format!("https://ipqualityscore.com/api/json/ip/{}/{}", ipqs_key, ip))
            .query(&[("strictness", "1"), ("allow_public_access_points", "true")])
            .send()
            .await
        {
            Ok(response) => {
                if let Ok(data) = response.json::<serde_json::Value>().await {
                    Ok(data["vpn"].as_bool().unwrap_or(false) ||
                       data["tor"].as_bool().unwrap_or(false) ||
                       data["proxy"].as_bool().unwrap_or(false) ||
                       data["bot_status"].as_bool().unwrap_or(false))
                } else {
                    Ok(false)
                }
            }
            Err(e) => {
                // Don't block on detection service failure
                eprintln!("VPN detection failed: {:?}", e);
                Ok(false)
            }
        }
    }

    async fn log_blocked_access(&self, ip: &str, geo: &GeoLocation, is_vpn: bool) -> Result<(),
GeoError> {
        sqlx::query!(
            r#"
            INSERT INTO blocked_access (ip, country, region, city, is_vpn, timestamp)
            VALUES ($1, $2, $3, $4, $5, $6)
            "#,
            ip,
            &geo.country,
            geo.region.as_ref(),
            geo.city.as_ref(),
            is_vpn,
            chrono::Utc::now().timestamp()
        )
        .execute(&*self.db)
        .await?;

        Ok(())
    }
}
```

Middleware Implementation

```
// Express middleware for geo-restriction
export function geoRestrictionMiddleware(geoService: GeoRestrictionService) {
    return async (req: Request, res: Response, next: NextFunction) => {
        const ip = req.ip || req.socket.remoteAddress;

        // Skip check for health endpoints
        if (req.path === '/health' || req.path === '/status') {
            return next();
        }

        try {
            const accessCheck = await geoService.checkAccess(ip);

            if (!accessCheck.allowed) {
                return res.status(451).json({
                    error: 'REGION_RESTRICTED',
                    message: 'Access to GoDark is not available in your region',
                    country: accessCheck.country,
                    reason: accessCheck.reason
                });
            }

            // Attach geo info to request
            req.geo = accessCheck;
            next();
        } catch (error) {
            // Log error but don't block user on detection failure
            console.error('Geo-restriction check failed:', error);
            next();
        }
    };
}
```

VPN/Proxy Detection

Multi-Layer Detection

```
class VPNDetection {
    private readonly DETECTION_SERVICES = [
        'IPQualityScore',
        'IPHub',
        'VPNBlocker'
    ];

    async detectVPN(ip: string): Promise<VPNDetectionResult> {
        // Run multiple detection services in parallel
        const results = await Promise.allSettled([
            this.checkIPQS(ip),
            this.checkIPHub(ip),
            this.checkVPNBlocker(ip)
        ]);

        // Count positive detections
        const positiveDetections = results.filter(
            r => r.status === 'fulfilled' && r.value === true
        ).length;

        // Require 2+ services to agree
        const isVPN = positiveDetections >= 2;

        return {
            isVPN,
            confidence: positiveDetections / results.length,
            services: results.map((r, i) => ({
                service: this.DETECTION_SERVICES[i],
                result: r.status === 'fulfilled' ? r.value : 'error'
            }))
        };
    }

    private async checkIPQS(ip: string): Promise<boolean> {
        const response = await axios.get(
            `https://ipqualityscore.com/api/json/ip/${process.env.IPQS_KEY}/${ip}`
        );

        return response.data.vpn ||
               response.data.tor ||
               response.data.proxy;
    }

    private async checkIPHub(ip: string): Promise<boolean> {
        const response = await axios.get(`https://v2.api.iphub.info/ip/${ip}`, {
            headers: { 'X-Key': process.env.IPHUB_KEY }
        });

        // block = 1 means VPN/proxy
        return response.data.block === 1;
    }

    private async checkVPNBlocker(ip: string): Promise<boolean> {
        const response = await axios.get(
            `https://vpnblocker.net/api/check/${ip}`,
            { params: { key: process.env.VPNBLOCKER_KEY } }
        );

        return response.data.host === 'true';
    }
}
```

Allowlist for Known Good IPs

```
// Some legitimate users may use corporate VPNs
class IPAllowlist {
    private allowlist: Set<string>;

    async isAllowlisted(ip: string): Promise<boolean> {
        // Check if IP is in allowlist
        if (this.allowlist.has(ip)) {
            return true;
        }

        // Check if user has been manually verified
        const user = await this.getUserByIP(ip);
        if (user?.vpnVerified) {
            return true;
        }

        return false;
    }

    async requestAllowlist(userId: string, ip: string, reason: string): Promise<void> {
        await db.allowlistRequests.insert({
            userId,
            ip,
            reason,
            status: 'PENDING',
            requestedAt: Date.now()
        });

        // Notify admin team for review
        await this.notifyAdminTeam({
            type: 'ALLOWLIST_REQUEST',
            userId,
            ip,
            reason
        });
    }
}
```

KYC Requirements

Risk-Based KYC Approach

```typescript
interface KYCRequirement {
    level: 'NONE' | 'BASIC' | 'ENHANCED';
    triggers: string[];
    documentation: string[];
    verificationTime: string;
}

const KYC_TIERS: KYCRequirement[] = [
    {
        level: 'NONE',
        triggers: [
            'Volume < $10,000/month',
            'Balance < $5,000'
        ],
        documentation: [],
        verificationTime: 'N/A'
    },
    {
        level: 'BASIC',
        triggers: [
            'Volume $10,000 - $100,000/month',
            'Balance > $5,000',
            'First withdrawal > $5,000'
        ],
        documentation: [
            'Government ID',
            'Proof of address',
            'Selfie verification'
        ],
        verificationTime: '24-48 hours'
    },
    {
        level: 'ENHANCED',
        triggers: [
            'Volume > $100,000/month',
            'Balance > $50,000',
            'Suspicious activity flagged'
        ],
        documentation: [
            'All BASIC requirements',
            'Source of funds',
            'Bank statements',
            'Tax returns (optional)',
            'Video verification call'
        ],
        verificationTime: '5-7 business days'
    }
];

class KYCService {
    async checkKYCRequirement(userId: string): Promise<KYCRequirement> {
        const user = await db.accounts.findOne({ _id: userId });
        const stats = await this.getUserStats(userId);

        // Check triggers for enhanced KYC
        if (stats.monthlyVolume > 100000 || stats.currentBalance > 50000) {
            return KYC_TIERS.find(t => t.level === 'ENHANCED');
        }

        // Check triggers for basic KYC
        if (stats.monthlyVolume > 10000 || stats.currentBalance > 5000) {
            return KYC_TIERS.find(t => t.level === 'BASIC');
        }

        // No KYC required
        return KYC_TIERS.find(t => t.level === 'NONE');
    }

    async enforceKYCRequirement(userId: string): Promise<void> {
        const requirement = await this.checkKYCRequirement(userId);
        const user = await db.accounts.findOne({ _id: userId });

        if (requirement.level === 'NONE') {
            return; // No action needed
        }

        if (user.kycLevel < requirement.level) {
            // Restrict certain actions until KYC completed
            await this.restrictUser(userId, {
                canDeposit: true,
                canTrade: requirement.level === 'BASIC', // Allow trading for BASIC
                canWithdraw: false, // No withdrawal until verified
                reason: `${requirement.level} KYC required`
```

```
            });

            // Notify user
            await this.notifyKYCRequired(userId, requirement);
        }
    }
}
```

KYC Provider Integration

```
// Integration with Persona, Onfido, or similar
class KYCProvider {
    async createVerificationSession(userId: string): Promise<string> {
        const response = await axios.post('https://api.withpersona.com/api/v1/inquiries', {
            data: {
                type: 'inquiry',
                attributes: {
                    'inquiry-template-id': process.env.PERSONA_TEMPLATE_ID,
                    'reference-id': userId
                }
            }
        }, {
            headers: {
                'Authorization': `Bearer ${process.env.PERSONA_API_KEY}`,
                'Persona-Version': '2023-01-05'
            }
        });

        return response.data.data.attributes['session-token'];
    }

    async checkVerificationStatus(userId: string): Promise<KYCStatus> {
        const response = await axios.get(
            `https://api.withpersona.com/api/v1/inquiries?filter[reference-id]=${userId}`,
            {
                headers: {
                    'Authorization': `Bearer ${process.env.PERSONA_API_KEY}`
                }
            }
        );

        const inquiry = response.data.data[0];

        return {
            status: inquiry.attributes.status,
            level: this.mapStatusToLevel(inquiry.attributes.status),
            completedAt: inquiry.attributes['completed-at'],
            fields: inquiry.attributes.fields
        };
    }
}
```

Terms of Service Integration

Acceptance Tracking

```
interface TOSAcceptance {
    userId: string;
    version: string;
    acceptedAt: number;
    ipAddress: string;
    userAgent: string;
}

class TOSService {
    private readonly CURRENT_VERSION = 'v1.0';

    async requireAcceptance(userId: string): Promise<boolean> {
        const latestAcceptance = await db.tosAcceptances.findOne(
            { userId },
            { sort: { acceptedAt: -1 } }
        );

        // Check if user has accepted current version
        if (!latestAcceptance || latestAcceptance.version !== this.CURRENT_VERSION) {
            return true; // Acceptance required
        }

        return false;
    }

    async recordAcceptance(userId: string, ip: string, userAgent: string): Promise<void> {
        await db.tosAcceptances.insert({
            userId,
            version: this.CURRENT_VERSION,
            acceptedAt: Date.now(),
            ipAddress: ip,
            userAgent
        });

        // Update user account
        await db.accounts.updateOne(
            { _id: userId },
            { $set: { tosAccepted: true, tosVersion: this.CURRENT_VERSION } }
        );
    }

    async notifyTOSUpdate(version: string): Promise<void> {
        // Get all active users
        const users = await db.accounts.find({ status: 'ACTIVE' });

        for (const user of users) {
            // Send email notification
            await this.emailService.send({
                to: user.email,
                subject: 'Updated Terms of Service',
                template: 'tos-update',
                data: {
                    version,
                    changesUrl: `https://godark.xyz/legal/tos/${version}/changes`,
                    acceptUrl: `https://app.godark.xyz/accept-tos`
                }
            });
        }
    }
}
```

TOS Content

```
# GoDark DEX Terms of Service

Last Updated: [DATE]
Version: v1.0

## 1. Acceptance of Terms

By accessing or using GoDark DEX, you agree to be bound by these Terms of Service...

## 2. Eligibility

You must:
- Be at least 18 years old
- Not be a resident of restricted jurisdictions
- Comply with all applicable laws
- Not use VPN to circumvent geo-restrictions

## 3. Risks

Trading perpetual futures involves significant risk:
- High leverage can result in total loss of funds
- Prices are volatile and unpredictable
- Platform may experience downtime or technical issues
- Smart contracts may contain undiscovered vulnerabilities

## 4. Non-Custodial Nature

- You maintain control of your private keys
- GoDark cannot recover lost keys
- You are responsible for securing your account

## 5. Fees

- Trading fees as disclosed on the platform
- Funding rates paid/received hourly
- Network transaction fees (gas)

## 6. Prohibited Activities

You may not:
- Manipulate markets
- Engage in wash trading
- Use automated systems without approval
- Circumvent security measures

## 7. Limitation of Liability

GoDark is provided "as is" without warranties...

## 8. Dispute Resolution

Any disputes shall be resolved through binding arbitration...

[Full terms continue...]
```

Data Privacy and GDPR Considerations

Privacy Policy

```
interface PrivacyCompliance {
    dataCollected: string[];
    purpose: string;
    retention: string;
    sharing: string[];
    userRights: string[];
}

const PRIVACY_POLICY: PrivacyCompliance = {
    dataCollected: [
        'Email address',
        'Wallet address',
        'IP address',
        'Trading activity',
        'Device information',
        'KYC documents (if applicable)'
    ],
    purpose: 'Provide trading services, comply with regulations, prevent fraud',
    retention: '7 years (regulatory requirement)',
    sharing: [
        'Service providers (hosting, KYC)',
        'Law enforcement (if required)',
        'Never sold to third parties'
    ],
    userRights: [
        'Access your data',
        'Correct inaccurate data',
        'Delete your data (with limitations)',
        'Export your data',
        'Opt-out of marketing'
    ]
};
```

Data Export

```
class DataExportService {
    async exportUserData(userId: string): Promise<UserDataExport> {
        const user = await db.accounts.findOne({ _id: userId });
        const orders = await db.orders.find({ user_id: userId });
        const trades = await db.trades.find({
            $or: [{ buyer_id: userId }, { seller_id: userId }]
        });
        const positions = await db.positions.find({ user_id: userId });

        return {
            account: {
                email: user.email,
                createdAt: user.created_at,
                kycLevel: user.kyc_level
            },
            orders: orders.map(o => ({
                id: o.id,
                symbol: o.symbol,
                side: o.side,
                size: o.size,
                price: o.price,
                timestamp: o.created_at
            })),
            trades: trades.map(t => ({
                id: t.id,
                symbol: t.symbol,
                price: t.price,
                size: t.size,
                fee: t.fee,
                timestamp: t.executed_at
            })),
            positions: positions.map(p => ({
                symbol: p.symbol,
                size: p.size,
                entryPrice: p.entry_price,
                realizedPnl: p.realized_pnl,
                openedAt: p.open_timestamp
            }))
        };
    }
}
```

AML/CTF Compliance Measures

Transaction Monitoring

```
class AMLMonitoring {
    async monitorTransaction(transaction: Transaction): Promise<AMLAlert[]> {
        const alerts: AMLAlert[] = [];

        // 1. Check transaction amount
        if (transaction.amount > 10000) {
            alerts.push({
                type: 'LARGE_TRANSACTION',
                severity: 'MEDIUM',
                amount: transaction.amount,
                threshold: 10000
            });
        }

        // 2. Check rapid succession
        const recentTxs = await this.getRecentTransactions(transaction.userId, 3600);
        if (recentTxs.length > 10) {
            alerts.push({
                type: 'RAPID_TRADING',
                severity: 'LOW',
                count: recentTxs.length,
                timeWindow: '1 hour'
            });
        }

        // 3. Check for structuring (breaking up large amounts)
        const dayTxs = await this.getRecentTransactions(transaction.userId, 86400);
        const totalAmount = dayTxs.reduce((sum, tx) => sum + tx.amount, 0);
        if (totalAmount > 50000 && dayTxs.length > 20) {
            alerts.push({
                type: 'POSSIBLE_STRUCTURING',
                severity: 'HIGH',
                totalAmount,
                transactionCount: dayTxs.length
            });
        }

        // 4. Check against OFAC sanctions list
        const isSanctioned = await this.checkSanctionsList(transaction.walletAddress);
        if (isSanctioned) {
            alerts.push({
                type: 'SANCTIONED_WALLET',
                severity: 'CRITICAL',
                wallet: transaction.walletAddress
            });

            // Immediately freeze account
            await this.freezeAccount(transaction.userId);
        }

        return alerts;
    }

    private async checkSanctionsList(wallet: string): Promise<boolean> {
        // Check against Chainalysis, TRM Labs, or similar
        try {
            const response = await axios.post('https://api.chainalysis.com/api/risk/v2/entities', {
                address: wallet
            }, {
                headers: { 'X-API-Key': process.env.CHAINALYSIS_KEY }
            });

            return response.data.risk === 'severe';
        } catch (error) {
            console.error('Sanctions check failed:', error);
            return false;
        }
    }
}
```

Regulatory Status and Disclaimers

Legal Disclaimers

```
const LEGAL_DISCLAIMERS = {
    notSecurities: `
        DARK tokens are utility tokens and are not securities.
        They have not been registered with any securities commission.
    `,

    notInvestmentAdvice: `
        Nothing on this platform constitutes investment advice.
        Trading involves substantial risk of loss.
    `,

    noGuarantees: `
        Past performance does not guarantee future results.
        GoDark makes no guarantees about profitability.
    `,

    technicalRisks: `
        Smart contracts may contain bugs or vulnerabilities.
        The platform may experience downtime or data loss.
    `,

    regulatoryRisks: `
        Regulatory status of cryptocurrencies is uncertain and evolving.
        Future regulations may impact the platform's operations.
    `
};
```

Legal Entity Structure

```
GoDark Foundation (Panama)
├── Purpose: Protocol governance and development
├── Structure: Non-profit foundation
└── Assets: DARK token treasury

GoDark Technologies Inc. (Delaware)
├── Purpose: Commercial operations
├── Structure: C-Corporation
└── Services: Platform hosting and support

Subsidiaries (Future):
├── GoDark EU (Estonia) - European operations
├── GoDark Asia (Singapore) - Asian operations
└── GoDark MENA (Dubai) - Middle East operations
```

Regulatory Compliance Roadmap

```
const COMPLIANCE_ROADMAP = {
    phase1: {
        timeline: 'Launch',
        items: [
            'TOS and Privacy Policy',
            'Geo-restrictions',
            'Basic AML monitoring',
            'OFAC sanctions screening'
        ]
    },
    phase2: {
        timeline: 'Month 6',
        items: [
            'Risk-based KYC',
            'Enhanced transaction monitoring',
            'SAR filing procedures',
            'Audit trail implementation'
        ]
    },
    phase3: {
        timeline: 'Month 12',
        items: [
            'MSB registration (if required)',
            'State-by-state licensing review',
            'ISO 27001 certification',
            'SOC 2 Type II audit'
        ]
    },
    phase4: {
        timeline: 'Year 2',
        items: [
            'MiCA compliance (EU)',
            'FCA review (if UK expansion)',
            'MAS license (Singapore)',
            'Full regulatory compliance globally'
        ]
    }
};
```