GoDark DEX Blockchain Handbook

Comprehensive Guide for Bootcamp Participants

v0.1.0

2025-11-24

Table of Contents

Foundation

Development Guides

Advanced Topics

Quick Reference

01. Blockchain Engineering Subdivisions

Overview

Blockchain engineering is a multidisciplinary field that combines cryptography, distributed systems, economics, and software engineering. Understanding these subdivisions helps contextualize where GoDark DEX fits in the broader ecosystem.

Core Subfields

1. **Smart Contract Development**

Definition: Writing, testing, and deploying self-executing contracts on blockchain platforms.

Key Technologies:

- **Ethereum:** Solidity, Vyper
- **Solana:** Rust (with Anchor framework) ← **GoDark uses this**
- **Aptos/Sui:** Move
- **Cosmos:** CosmWasm (Rust)

GoDark Relevance:

- All core logic lives in Solana smart contracts (Anchor/Rust programs)
- Position management, liquidation, funding rates, vault management
- Settlement batch processing

Skills Required:

- Rust programming (for Solana)
- Anchor framework patterns
- PDA (Program Derived Address) design
- Cross-program invocations (CPIs)
- Account data serialization

---

2. **Blockchain Protocol Development**

Definition: Designing and implementing the core blockchain infrastructure, consensus mechanisms, and network protocols.

Key Concepts:

- Consensus algorithms (PoW, PoS, DPoS, PoH)
- Network architecture (P2P, gossip protocols)
- Cryptography (signatures, hashing, Merkle trees)
- State management

GoDark Relevance:

- Built on Solana (PoH + PoS hybrid consensus)
- Leverages Solana's high throughput (65,000 TPS theoretical)
- Uses Solana's account model for state management
- No protocol-level changes needed (application layer)

Skills Required:

- Distributed systems theory
- Cryptography fundamentals
- Network protocols
- Performance optimization

---

3. **DeFi (Decentralized Finance)**

Definition: Financial applications built on blockchain without intermediaries.

Key Categories:

- **DEXs:** Decentralized exchanges (Uniswap, dYdX, Drift)
- **Lending/Borrowing:** Aave, Compound
- **Derivatives:** Perpetual futures, options, structured products
- **Yield Farming:** Liquidity provision, staking rewards

GoDark's Position:

- **Perpetual Futures DEX** (derivatives category)
- Combines DEX (decentralized) with dark pool mechanics (privacy)
- High leverage (up to 1000x) for advanced traders
- Institutional-grade execution

Skills Required:

- Financial instrument understanding
- Risk management
- Economic mechanism design
- Oracle integration

---

4. **Infrastructure & DevOps**

Definition: Building and maintaining blockchain infrastructure, nodes, APIs, and developer tools.

Key Components:

- **RPC Providers:** Alchemy, QuickNode, Helius
- **Indexers:** The Graph, Helius, custom indexers
- **Node Operations:** Validator nodes, RPC nodes
- **Monitoring:** Block explorers, analytics dashboards

GoDark Relevance:

- Off-chain matching engine (infrastructure layer)
- Settlement relayer service (off-chain component)
- Oracle integration (Pyth, Switchboard)
- API gateway and WebSocket servers
- Database systems for order book and positions

Skills Required:

- System architecture
- Database design
- API development
- Monitoring and observability
- High-performance systems

---

5. **Security & Auditing**

Definition: Identifying vulnerabilities, conducting security audits, and implementing secure coding practices.

Key Areas:

- Smart contract security audits
- Penetration testing
- Formal verification
- Bug bounty programs
- Economic attack analysis

GoDark Critical Areas:

- **Liquidation Engine:** Must prevent manipulation
- **Funding Rate:** Must resist oracle manipulation
- **Settlement:** Must prevent double-spending, replay attacks
- **Vault Management:** Must prevent unauthorized withdrawals
- **Position Management:** Must prevent overflow/underflow attacks

Skills Required:

- Security mindset
- Cryptography knowledge
- Attack vector analysis
- Formal verification tools
- Economic mechanism security

---

6. **Cryptography**

Definition: Advanced cryptographic techniques for privacy, verification, and security.

Key Technologies:

- Zero-knowledge proofs (ZK-SNARKs, ZK-STARKs)
- Privacy-preserving technologies
- Signature schemes (Ed25519 for Solana)
- Hash functions (SHA-256, Keccak)

GoDark Relevance:

- Solana uses Ed25519 signatures
- Merkle trees for batch settlement verification
- Cryptographic proofs for trade integrity
- Dark pool privacy (order hiding)

Skills Required:

- Mathematical foundations
- Cryptographic protocol design
- Implementation security
- Performance optimization

---

7. **NFTs & Digital Assets**

Definition: Non-fungible tokens, token standards, and digital asset management.

Key Standards:

- **Ethereum:** ERC-20, ERC-721, ERC-1155
- **Solana:** SPL Token (fungible), Metaplex (NFTs)

GoDark Relevance:

- Uses **SPL Token** standard for USDT (quote asset)
- Token account management
- Transfer operations
- Not focused on NFTs (futures trading platform)

Skills Required:

- Token standard understanding
- Metadata management
- Marketplace mechanics

---

8. **Web3 & dApp Development**

Definition: Building user-facing applications that interact with blockchain.

Key Technologies:

- Frontend: React, Vue, Web3.js, Ethers.js
- Wallet integration: WalletConnect, Phantom, MetaMask
- State management: Redux, Zustand
- UI/UX for blockchain interactions

GoDark Relevance:

- Web UI at `app.godark.xyz`
- Wallet connection (Phantom, Solflare)
- Real-time order book (WebSocket)
- Position management UI
- Dark pool interface (no visible order book)

Skills Required:

- Frontend frameworks
- Web3 libraries (Solana Web3.js)
- Wallet integration
- Real-time data handling
- UX for complex financial products

---

9. **Blockchain Analytics & Data**

Definition: Analyzing on-chain data, building explorers, and providing insights.

Key Tools:

- Block explorers (Solscan, Solana Explorer)
- Analytics platforms (Dune Analytics, Flipside)
- Data indexing (The Graph, custom indexers)
- On-chain metrics

GoDark Relevance:

- Trade analytics and statistics
- Position tracking
- Funding rate history
- Liquidation events
- Performance metrics

Skills Required:

- Data analysis
- SQL/NoSQL databases
- GraphQL (for The Graph)
- Statistical analysis

---

10. **Research & Academia**

Definition: Advancing blockchain technology through research and academic contributions.

Key Areas:

- Consensus algorithm research
- Scalability solutions
- Economic mechanism design
- Privacy technologies
- Formal verification

GoDark Relevance:

- Dark pool mechanism design (privacy research)
- Perpetual futures pricing models
- Liquidation mechanism optimization
- Funding rate algorithms

Skills Required:

- Research methodology
- Academic writing
- Mathematical modeling
- Experimental design

---

11. **Enterprise Blockchain**

Definition: Private/permissioned blockchains for enterprise use cases.

Key Platforms:

- Hyperledger Fabric
- R3 Corda
- Enterprise Ethereum
- Private Solana deployments

GoDark Relevance:

- Not applicable (public Solana DEX)
- However, institutional users may use GoDark
- Privacy features appeal to enterprises

---

12. **Cryptocurrency Exchange Development**

Definition: Building centralized or decentralized exchanges for trading cryptocurrencies.

Key Types:

- **CEX:** Centralized exchanges (Binance, Coinbase)
- **DEX:** Decentralized exchanges (Uniswap, dYdX)
- **Hybrid:** Off-chain matching, on-chain settlement ← **GoDark is this**

GoDark's Architecture:

- **Hybrid Model:**
  - Off-chain matching engine (speed)
  - On-chain settlement (security/transparency)
  - Dark pool mechanics (privacy)
  - Perpetual futures focus (derivatives)

Skills Required:

- Order matching algorithms
- Market microstructure
- Risk management
- High-frequency trading systems
- Settlement systems

GoDark DEX: Where It Fits

GoDark DEX spans **multiple subdivisions**:

1. **Smart Contract Development** (primary)
   - Solana Anchor/Rust programs
   - All 8 core components involve smart contracts
2. **DeFi - Derivatives**
   - Perpetual futures exchange
   - High leverage trading
3. **Infrastructure & DevOps**
   - Off-chain matching engine
   - Settlement relayer
   - API/WebSocket services
4. **Security & Auditing**
   - Critical for all components
   - Economic security (liquidation, funding)
5. **Web3 & dApp Development**
   - User interface
   - Wallet integration
6. **Cryptocurrency Exchange Development**

    - Order matching
    - Settlement systems
    - Risk management

Learning Path for GoDark Developers

Foundation (All Participants)

1. **Blockchain Fundamentals**
    - What is blockchain?
    - Consensus mechanisms
    - Cryptography basics
2. **Solana-Specific**
    - Account model
    - PDAs (Program Derived Addresses)
    - Transactions and fees
    - Anchor framework
3. **DeFi Concepts**
    - Perpetual futures
    - Leverage and margin
    - Funding rates
    - Liquidation mechanics

Component-Specific (By Assignment)

Settlement Relayer Team:

- Infrastructure & DevOps
- Batch processing
- Merkle trees
- Transaction building

Position Management Team:

- Smart contract development
- Financial calculations
- State management

Liquidation Engine Team:

- Security & Auditing
- Real-time monitoring
- Economic mechanisms

Ephemeral Vault Team:

- Smart contract development
- Key management
- Session management

Funding Rate Team:

- DeFi mechanisms
- Oracle integration
- Time-series calculations

Oracle Integration Team:

- Infrastructure & DevOps
- Data validation
- Failover systems

Collateral Vault Team:

- Smart contract development
- Token program integration
- Account management

Program Upgrade Team:

- Security & Auditing
- Governance mechanisms
- State migration

Key Takeaways

1. **Blockchain engineering is multidisciplinary** - GoDark touches many subfields
2. **GoDark is primarily a DeFi application** - Perpetual futures DEX
3. **Built on Solana** - Requires Solana-specific knowledge
4. **Hybrid architecture** - Combines off-chain speed with on-chain security
5. **Security is paramount** - Financial application handling user funds

---

Next Steps

- Read **02-godark-ecosystem-role.md** to understand GoDark's position in the ecosystem
- Review **03-solana-fundamentals.md** for Solana-specific concepts
- Study **04-perpetual-futures-primer.md** for DeFi mechanics
- Check **05-component-overview.md** for your specific component

---

Last Updated: November 2025

02. GoDark DEX: Role in the Blockchain Ecosystem

Executive Summary

GoDark DEX is a **hybrid decentralized perpetual futures exchange** built on Solana that combines:

- **Dark pool privacy** (institutional-grade order hiding)
- **High-performance execution** (off-chain matching, on-chain settlement)
- **High leverage trading** (up to 1000x)
- **DeFi-native architecture** (non-custodial, transparent settlement)

Positioning in the Blockchain Landscape

1. **Layer Classification**

```
            Solana Layer 1 (Foundation)
 - Consensus: PoH + PoS
 - Throughput: 65,000 TPS (theoretical)
 - Finality: ~400ms
                    │
                    ▼
        GoDark DEX (Application Layer)
 - Smart Contracts (Anchor/Rust programs)
 - On-chain settlement
 - Position management
                    │
                    ▼
     Off-Chain Infrastructure (Matching Engine)
 - Dark pool order book
 - Millisecond-latency matching
 - Settlement relayer
```

Key Point: GoDark operates as an **application-layer protocol** on Solana, not a Layer 2 scaling solution. The "Layer 2" terminology in some docs refers to the off-chain matching layer, not a blockchain layer.

Comparison with Other Exchange Types

Centralized Exchanges (CEX)

Feature	CEX (Binance, Coinbase)	GoDark DEX
Custody	Custodial (exchange holds funds)	Non-custodial (user controls funds)
Order Book	Visible, public	Hidden (dark pool)
Settlement	Internal ledger	On-chain (Solana)
Privacy	Limited	High (dark pool)
Speed	Very fast	Fast (off-chain matching)
Transparency	Opaque	Transparent settlement
Regulation	Heavily regulated	DeFi-native

GoDark Advantage: Combines CEX-like speed with DEX-like transparency and non-custodial nature.

Traditional DEXs (AMM-based)

Feature	AMM DEX (Uniswap)	GoDark DEX
Matching	Automated Market Maker (AMM)	Order book (dark pool)
Liquidity	Liquidity pools	Order book depth
Price Discovery	Formula-based	Order matching
Order Types	Swap only	Market, Limit, Peg
Instruments	Spot trading	Perpetual futures
Leverage	None (or via lending)	Up to 1000x

Feature	AMM DEX (Uniswap)	GoDark DEX
Privacy	Public order book	Hidden orders

GoDark Advantage: Order book model provides better execution for large orders, dark pool prevents front-running.

Order Book DEXs

Feature	dYdX (v4)	Drift Protocol	GoDark DEX
Chain	Cosmos (custom)	Solana	Solana
Matching	On-chain	Off-chain	Off-chain
Order Book	Visible	Visible	**Hidden (dark pool)**
Leverage	Up to 20x	Up to 20x	**Up to 1000x**
Settlement	On-chain	On-chain	On-chain (batched)
Privacy	Public	Public	**Private**

GoDark Differentiators:

1. **Dark pool mechanics** - Orders invisible until execution
2. **Higher leverage** - 1000x vs 20x
3. **Privacy-first** - No information leakage

Ecosystem Integration Points

1. **Solana Blockchain**

GoDark's Foundation:

- Built entirely on Solana
- Uses Solana's account model
- Leverages Solana's high throughput
- Benefits from Solana's low fees

Why Solana?

- **Speed:** Sub-second finality enables fast settlement
- **Cost:** Low transaction fees (fractions of a cent)
- **Scalability:** High TPS supports batch settlements
- **Ecosystem:** Growing DeFi ecosystem

Integration Points:

- Smart contracts (Anchor programs)
- SPL Token standard (USDT)
- Solana Web3.js for frontend
- RPC providers for data access

---

2. **Oracle Networks**

GoDark Uses:

- **Primary:** Pyth Network

- **Fallback:** Switchboard

Purpose:

- Price feeds for perpetual futures
- Mark price calculation
- Liquidation price monitoring
- Funding rate calculation

Why Multiple Oracles?

- Redundancy (failover)
- Price consensus (median)
- Manipulation resistance
- Uptime reliability

Integration Pattern:

```
Oracle Feed → Price Aggregator → Mark Price → Position Management
                                        ↓
                               Funding Rate Calculator
                                        ↓
                               Liquidation Engine
```

---

3. **Wallet Infrastructure**

Supported Wallets:

- Phantom (primary)
- Solflare
- Other Solana wallets (via WalletConnect)

Integration Points:

- Wallet connection (Web3.js)
- Transaction signing
- USDT approval/delegation
- Ephemeral vault creation

User Flow:

1. Connect wallet
2. Approve USDT delegation
3. Create ephemeral vault (optional)
4. Trade with wallet or ephemeral vault

---

4. **DeFi Protocols**

GoDark's Position:

- **Standalone DEX** - Not directly integrated with other DeFi protocols
- **Composable** - Users can bridge assets from other chains
- **Future Integration Potential:**
    - Lending protocols (for additional leverage)
    - Yield farming (insurance fund)
    - Governance tokens (future)

Current Isolation:

- Self-contained perpetual futures market
- USDT-only (no multi-asset collateral yet)

- No direct DeFi composability (by design for simplicity)

---

5. **Cross-Chain Bridges**

Current State:

- Solana-only (single-chain solution)
- Users bridge assets to Solana before trading

Bridge Usage:

- Users bridge USDT from Ethereum/Polygon/etc. to Solana
- Trade on GoDark
- Bridge back if needed

Future Considerations:

- Direct bridge integration (UX improvement)
- Multi-chain settlement (complex, not planned)

Market Positioning

Target Users

1. **Professional Traders**
    - Seeking privacy (dark pool)
    - Need high leverage (1000x)
    - Want minimal market impact
2. **Market Makers**
    - Providing liquidity
    - Need hidden orders
    - Require fast execution
3. **Institutions**
    - Large block trades
    - Privacy requirements
    - Non-custodial preference
4. **Retail Traders**
    - Access to dark pool liquidity
    - High leverage trading
    - DeFi-native users

Competitive Advantages

1. **Dark Pool Privacy**
    - Unique in DeFi perpetual futures space
    - Prevents front-running
    - Reduces market impact
2. **High Leverage**
    - 1000x vs competitors' 20x
    - Attracts advanced traders
    - Higher risk/reward

3. **Hybrid Architecture**
    - Off-chain speed (milliseconds)
    - On-chain security (transparent)
    - Best of both worlds
4. **Solana Native**
    - Low fees
    - Fast settlement
    - Growing ecosystem

Technical Architecture Role

Component Ecosystem

```
┌─────────────────────────────────────────────────────────┐
│                    User Interface                       │
│               (Web UI, Mobile, API)                     │
└─────────────────────┬───────────────────────────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────────────────┐
│              API & WebSocket Gateway                    │
│         (Authentication, Rate Limiting)                 │
└─────────────────────┬───────────────────────────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────────────────┐
│         Off-Chain Matching Engine                       │
│  ┌──────────────────────────────────────┐              │
│  │ Dark Pool Order Book                 │              │
│  │ - Hidden orders                      │              │
│  │ - Price-time priority                │              │
│  │ - Matching algorithm                 │              │
│  └──────────────────────────────────────┘              │
└─────────────────────┬───────────────────────────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────────────────┐
│           Settlement Relayer                            │
│  ┌──────────────────────────────────────┐              │
│  │ Batch Accumulation (1s windows)      │              │
│  │ Net Position Calculation             │              │
│  │ Merkle Tree Generation               │              │
│  │ Transaction Building                 │              │
│  └──────────────────────────────────────┘              │
└─────────────────────┬───────────────────────────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────────────────┐
│         Solana Smart Contracts                          │
│  ┌──────────┬──────────┬──────────┬──────────┐        │
│  │ Position │Liquidation│ Funding │  Vault   │        │
│  │ Manager │  Engine  │   Rate   │ Manager  │        │
│  └──────────┴──────────┴──────────┴──────────┘        │
└─────────────────────┬───────────────────────────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────────────────┐
│              Solana Blockchain                          │
│         (Consensus, Finality, Storage)                  │
└─────────────────────────────────────────────────────────┘
```

Economic Role

Value Flow

```
Users → Deposit USDT → Trade → Pay Fees → Withdraw USDT
          ↓                                     ↑
    Collateral Vault ←──────────────────────────┘
          ↓
    Position Management
          ↓
    Settlement (on-chain)
          ↓
    Fee Collection (GoDark)
```

Fee Structure

- **Maker Fees:** Configurable (can be negative for rebates)
- **Taker Fees:** Basis points (e.g., 5-10 bps)
- **Funding Rate:** Paid hourly between longs/shorts
- **Liquidation Fee:** Paid to liquidators

Economic Security

- **Insurance Fund:** Covers bad debt from liquidations
- **Funding Rate:** Balances long/short interest
- **Liquidation:** Prevents protocol insolvency

Future Ecosystem Expansion

Potential Integrations

1. **Lending Protocols**
   - Borrow additional collateral
   - Increase leverage beyond 1000x
   - Cross-protocol composability
2. **Governance**
   - Token-based governance
   - Parameter voting
   - Protocol upgrades
3. **Multi-Asset Collateral**
   - Beyond USDT
   - SOL, ETH, BTC as collateral
   - Cross-margining
4. **Options Trading**
   - Perpetual options
   - Structured products
   - Derivatives expansion

Key Takeaways

1. **GoDark is a DeFi perpetual futures DEX** - Not a spot exchange or AMM
2. **Hybrid architecture** - Off-chain matching, on-chain settlement
3. **Privacy-first** - Dark pool mechanics unique in DeFi
4. **Solana-native** - Built entirely on Solana
5. **High leverage** - 1000x vs competitors' 20x
6. **Non-custodial** - Users control their funds
7. **Institutional-grade** - Privacy and execution quality

Next Steps

- Review **03-solana-fundamentals.md** for Solana-specific concepts
- Study **04-perpetual-futures-primer.md** for DeFi mechanics
- Check **05-component-overview.md** for component details

Last Updated: November 2025

03. Solana Fundamentals for GoDark Developers

Overview

Solana is a high-performance blockchain designed for scalability and low transaction costs. Understanding Solana's unique architecture is essential for developing GoDark DEX components. This guide covers the core concepts you'll need.

Solana Account Model

What is an Account?

In Solana, **everything is an account**. Unlike Ethereum's contract storage model, Solana uses accounts to store both data and program code.

Account Types:

1. **Data Accounts**
   - Store application state
   - Owned by programs (smart contracts)
   - Can be owned by users (wallet accounts)
   - Contains data and metadata
2. **Program Accounts**
   - Store executable code (smart contracts)
   - Immutable once deployed (unless upgradeable)
   - Executed by the Solana runtime
3. **System Accounts**
   - Native Solana programs (System Program, Token Program, etc.)
   - Special accounts with specific functionality

Account Structure

```
pub struct AccountInfo {
    pub key: &Pubkey,           // Account address
    pub lamports: &mut u64,     // SOL balance (rent)
    pub data: &mut [u8],        // Account data
    pub owner: &Pubkey,         // Program that owns this account
    pub executable: bool,       // Is this a program account?
    pub rent_epoch: u64,        // Rent exemption epoch
}
```

Key Properties:

- **key**: The account's public key (address)
- **lamports**: SOL balance (1 SOL = 1,000,000,000 lamports)
- **data**: Raw byte array storing account data
- **owner**: The program that controls this account

- **executable**: Whether this account contains executable code

Account Ownership

- **User-owned accounts**: Controlled by private keys (wallets)
- **Program-owned accounts**: Controlled by programs (smart contracts)
- **System-owned accounts**: Owned by native Solana programs

GoDark Example:

- User's wallet account: User-owned
- Position account: Program-owned (by Position Management program)
- Collateral vault: Program-owned (by Collateral Vault program)

Program Derived Addresses (PDAs)

What are PDAs?

PDAs are addresses that don't have corresponding private keys. They're deterministically derived from:

- A program ID
- A set of seeds (byte arrays)
- A bump seed (to ensure the address is off the ed25519 curve)

Why Use PDAs?

1. **Deterministic Addresses**: Same seeds = same address
2. **Program Control**: Only the program can sign for PDAs
3. **No Key Management**: No private keys to store or lose
4. **Cross-Program Invocations**: Programs can sign transactions on behalf of PDAs

PDA Derivation

```
use anchor_lang::prelude::*;

// Derive a PDA
let (pda, bump) = Pubkey::find_program_address(
    &[
        b"vault",                     // seed 1
        user_pubkey.as_ref(),         // seed 2
        program_id.as_ref(),          // program ID
    ],
    program_id,                       // program that owns this PDA
);
```

Common PDA Patterns in GoDark:

1. **User-Specific Accounts**

   ```
   // Position PDA for a user
   let (position_pda, _bump) = Pubkey::find_program_address(
       &[b"position", user_pubkey.as_ref()],
       program_id,
   );
   ```

2. **Global State Accounts**

```rust
// Global configuration PDA
let (config_pda, _bump) = Pubkey::find_program_address(
    &[b"config"],
    program_id,
);
```

3. **Token Accounts**

```rust
// Vault token account PDA
let (vault_token_pda, _bump) = Pubkey::find_program_address(
    &[b"vault", mint_pubkey.as_ref()],
    program_id,
);
```

PDA Signing

PDAs can sign transactions through **Cross-Program Invocations (CPIs)**:

```rust
// Sign with PDA seeds
let seeds = &[
    b"vault",
    user_pubkey.as_ref(),
    &[bump],
];
let signer = &[&seeds[..]];

// Use in CPI
invoke_signed(
    &instruction,
    &account_infos,
    &[signer],
)?;
```

Transactions and Instructions

Transaction Structure

A Solana transaction contains:

- **Signatures**: Required signatures (up to 64)
- **Message**: Transaction details
  - **Header**: Account metadata
  - **Account Keys**: All accounts involved
  - **Recent Blockhash**: For transaction expiration
  - **Instructions**: What to execute

Instruction Structure

```rust
pub struct Instruction {
    pub program_id: Pubkey,       // Program to execute
    pub accounts: Vec<AccountMeta>, // Accounts involved
    pub data: Vec<u8>,            // Instruction data
}
```

AccountMeta:

```
pub struct AccountMeta {
    pub pubkey: Pubkey,
    pub is_signer: bool,          // Must sign transaction
    pub is_writable: bool,        // Account data will change
}
```

Transaction Fees

- **Base Fee**: 5,000 lamports (0.000005 SOL) per transaction
- **Rent**: For account creation (can be rent-exempt)
- **Priority Fees**: Optional fees for faster processing

GoDark Consideration: Batch settlements reduce per-trade fees by combining multiple operations into one transaction.

Cross-Program Invocations (CPIs)

What are CPIs?

CPIs allow Solana programs to call other programs, similar to function calls in traditional programming.

CPI Example: Token Transfer

```
use anchor_spl::token::{self, Token, TokenAccount, Transfer};

pub fn transfer_tokens(ctx: Context<TransferTokens>, amount: u64) -> Result<()> {
    let cpi_accounts = Transfer {
        from: ctx.accounts.from.to_account_info(),
        to: ctx.accounts.to.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
    };

    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

    token::transfer(cpi_ctx, amount)?;
    Ok(())
}
```

CPI with PDA Signing

```
// Sign with PDA for CPI
let seeds = &[
    b"vault",
    user_pubkey.as_ref(),
    &[bump],
];
let signer = &[&seeds[..]];

let cpi_ctx = CpiContext::new_with_signer(
    token_program,
    cpi_accounts,
    signer,
);

token::transfer(cpi_ctx, amount)?;
```

GoDark Usage:

- Collateral vault transfers tokens via CPI
- Position management locks/unlocks collateral via CPI
- Settlement relayer executes multiple CPIs in batch

GoQuant

Anchor Framework Basics

What is Anchor?

Anchor is a framework for building Solana programs that provides:

- **IDL (Interface Definition Language)**: Type-safe program interfaces
- **Macros**: Simplify common patterns
- **Client SDKs**: TypeScript, Rust clients
- **Testing**: Built-in testing framework

Anchor Program Structure

```rust
use anchor_lang::prelude::*;

declare_id!("YourProgram11111111111111111111111111");

#[program]
pub mod your_program {
    use super::*;

    pub fn initialize(ctx: Context<Initialize>, data: u64) -> Result<()> {
        ctx.accounts.state.data = data;
        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(init, payer = user, space = 8 + 8)]
    pub state: Account<'info, State>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[account]
pub struct State {
    pub data: u64,
}
```

Key Anchor Concepts

1. `#[program]`: Marks the program module
2. `#[derive(Accounts)]`: Account validation struct
3. `#[account]`: Marks account data structures
4. `Account<'info, T>`: Type-safe account wrapper
5. `Signer<'info>`: Account that must sign
6. `Program<'info, T>`: Program account

Account Constraints

```rust
#[derive(Accounts)]
pub struct Example<'info> {
    #[account(
        init,                       // Initialize account
        payer = user,               // Who pays rent
        space = 8 + 32,             // Account size (discriminator + data)
        seeds = [b"seed"],          // PDA seeds
        bump                        // Bump seed
    )]
    pub pda: Account<'info, State>,

    #[account(mut)]                 // Account is writable
    pub user: Signer<'info>,        // Must sign

    #[account(owner = token::ID)] // Must be owned by Token Program
    pub token_account: Account<'info, TokenAccount>,

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

Error Handling

```rust
use anchor_lang::error_code;

#[error_code]
pub enum ErrorCode {
    #[msg("Insufficient funds")]
    InsufficientFunds,
    #[msg("Invalid authority")]
    InvalidAuthority,
}

// In instruction
if amount > balance {
    return Err(ErrorCode::InsufficientFunds.into());
}
```

SPL Token Integration

SPL Token Overview

SPL Token is Solana's token standard (similar to ERC-20 on Ethereum). GoDark uses USDT (SPL Token) as collateral.

Key Concepts

1. **Mint**: Token definition (like a token contract)
2. **Token Account**: Holds tokens for a user
3. **Associated Token Account (ATA)**: Standard token account address

Token Operations

Transfer Tokens:

```rust
use anchor_spl::token::{self, Transfer};

pub fn transfer(ctx: Context<TransferTokens>, amount: u64) -> Result<()> {
    let cpi_accounts = Transfer {
        from: ctx.accounts.from.to_account_info(),
        to: ctx.accounts.to.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
    token::transfer(cpi_ctx, amount)?;
    Ok(())
}
```

Mint Tokens:

```rust
use anchor_spl::token::{self, MintTo};

pub fn mint(ctx: Context<MintTokens>, amount: u64) -> Result<()> {
    let cpi_accounts = MintTo {
        mint: ctx.accounts.mint.to_account_info(),
        to: ctx.accounts.to.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
    token::mint_to(cpi_ctx, amount)?;
    Ok(())
}
```

GoDark Usage:

- Collateral vault manages USDT token accounts
- Users deposit/withdraw USDT
- Positions lock/unlock collateral via token accounts

Solana Runtime Constraints

Compute Units

- **Default**: 200,000 compute units per transaction
- **Can be increased**: Up to 1,400,000 with `ComputeBudgetInstruction`
- **Exhaustion**: Transaction fails if compute units exceeded

Optimization Tips:

- Minimize loops
- Use efficient data structures
- Batch operations when possible
- Cache expensive computations

Account Size Limits

- **Maximum**: 10 MB per account
- **Rent**: Accounts must be rent-exempt or pay rent
- **Rent-Exempt**: Minimum balance to be exempt from rent

Rent Calculation:

```rust
// Calculate rent-exempt minimum
let rent = Rent::get()?;
let space = 8 + 32; // discriminator + data
let rent_lamports = rent.minimum_balance(space);
```

Transaction Size Limits

- **Maximum**: 1,232 bytes per transaction
- **Accounts**: Up to 64 accounts per transaction
- **Instructions**: Multiple instructions per transaction

GoDark Impact:

- Batch settlements must fit within size limits
- Account ordering matters (writable accounts first)

Rent and Account Ownership

Rent System

Solana uses a **rent** system to prevent blockchain bloat:

- Accounts pay rent based on data size
- **Rent-exempt**: Accounts with minimum balance are exempt
- **Rent-paying**: Accounts below minimum pay rent periodically

Rent-Exempt Minimum

```rust
use anchor_lang::prelude::*;

let rent = Rent::get()?;
let space = 8 + 32; // Account size
let minimum_balance = rent.minimum_balance(space);
```

Account Initialization

```rust
#[account(init, payer = user, space = 8 + 32)]
pub state: Account<'info, State>,
```

- `init`: Creates new account
- `payer`: Who pays for account creation
- `space`: Account size in bytes

Closing Accounts

```rust
// Close account and refund rent
**ctx.accounts.state.to_account_info().try_borrow_mut_lamports()? -= rent;
**ctx.accounts.user.to_account_info().try_borrow_mut_lamports()? += rent;
```

Clock and Epoch Sysvars

Clock Sysvar

Provides current blockchain time:

```
use anchor_lang::solana_program::clock::Clock;

let clock = Clock::get()?;
let current_timestamp = clock.unix_timestamp;
let current_slot = clock.slot;
```

GoDark Usage:

- Timelock enforcement in upgrade system
- Funding rate calculation timing
- Session expiry in ephemeral vaults

Epoch

- **Epoch**: ~2-3 days of slots
- **Slot**: ~400ms time unit
- **Epoch Boundary**: When validator set changes

Common Solana Patterns

Pattern 1: PDA Authority

```
// Derive PDA that will be authority
let (authority_pda, bump) = Pubkey::find_program_address(
    &[b"authority"],
    program_id,
);

// Verify PDA is signer
require!(
    ctx.accounts.authority.key() == &authority_pda,
    ErrorCode::InvalidAuthority
);
```

Pattern 2: Account Initialization

```
#[account(init, payer = user, space = 8 + State::LEN)]
pub state: Account<'info, State>,

impl State {
    pub const LEN: usize = 32 + 8; // Define size
}
```

Pattern 3: Account Validation

```
#[account(
    constraint = token_account.owner == token::ID @ ErrorCode::InvalidTokenAccount,
    constraint = token_account.mint == expected_mint @ ErrorCode::InvalidMint,
)]
pub token_account: Account<'info, TokenAccount>,
```

Pattern 4: Mutability Checks

```
#[account(mut)]  // Account will be modified
pub user_account: Account<'info, UserAccount>,

#[account(mut)]
pub vault: Account<'info, Vault>,
```

GoDark-Specific Patterns

Position Account Pattern

```
#[account]
pub struct Position {
    pub user: Pubkey,
    pub market: Pubkey,
    pub size: i64,              // Positive = long, negative = short
    pub entry_price: u64,
    pub collateral: u64,
    pub leverage: u8,
    pub version: u32,           // For migrations
}
```

Vault PDA Pattern

```
// Derive vault PDA
let (vault_pda, bump) = Pubkey::find_program_address(
    &[b"vault", user_pubkey.as_ref()],
    program_id,
);

// Sign with vault PDA
let seeds = &[
    b"vault",
    user_pubkey.as_ref(),
    &[bump],
];
```

Batch Settlement Pattern

```
// Multiple instructions in one transaction
let mut instructions = Vec::new();

for trade in trades {
    instructions.push(create_settle_instruction(trade)?);
}

// Execute batch
invoke_many(&instructions, &account_infos)?;
```

Key Takeaways

1. **Everything is an account** - Data, programs, tokens all use accounts
2. **PDAs enable program control** - Deterministic addresses without private keys
3. **CPIs enable composability** - Programs call other programs
4. **Anchor simplifies development** - Type-safe, macro-based framework
5. **SPL Token is standard** - USDT uses SPL Token standard
6. **Constraints matter** - Compute units, account size, transaction size
7. **Rent system prevents bloat** - Accounts must be rent-exempt or pay rent

---

Next Steps

- Review **04-perpetual-futures-primer.md** for DeFi mechanics
- Study **05-component-overview.md** for component details
- Practice with Anchor tutorials: https://www.anchor-lang.com/

Last Updated: November 2025

04. Perpetual Futures Primer for GoDark DEX

Overview

GoDark DEX is a **perpetual futures exchange**. Understanding perpetual futures mechanics is essential for building and using the platform. This guide explains the core concepts, formulas, and mechanics specific to GoDark.

What Are Perpetual Futures?

Definition

Perpetual futures (perpetuals) are derivative contracts that:

- Have **no expiration date** (unlike traditional futures)
- Track the price of an underlying asset (e.g., BTC, ETH)
- Allow **long** (betting price goes up) or **short** (betting price goes down) positions
- Use **leverage** to amplify gains/losses
- Settle continuously through **funding rates**

Perpetual Futures vs Traditional Futures

Feature	Traditional Futures	Perpetual Futures
Expiration	Fixed expiry date	No expiration
Settlement	Physical or cash at expiry	Continuous (funding rate)
Margin	Initial + maintenance	Initial + maintenance
Leverage	Typically 10-50x	Up to 1000x (GoDark)

Perpetual Futures vs Spot Trading

Feature	Spot Trading	Perpetual Futures
Ownership	Own the asset	Contract, not asset
Leverage	None (or via lending)	Built-in leverage
Short Selling	Limited	Easy (just open short)
Funding	None	Hourly funding payments

Core Concepts

1. Long vs Short Positions

Long Position:

- Betting the price will **increase**
- Profit when price goes up
- Loss when price goes down
- Example: Open long at $50,000, close at $55,000 = +$5,000 profit

Short Position:

- Betting the price will **decrease**
- Profit when price goes down
- Loss when price goes up
- Example: Open short at $50,000, close at $45,000 = +$5,000 profit

2. Position Size

Notional Value:

```
Notional Value = Position Size × Entry Price
```

Example:

- Position Size: 1 BTC
- Entry Price: $50,000
- Notional Value: $50,000

3. Leverage

Leverage amplifies both gains and losses.

Leverage Formula:

```
Leverage = Notional Value / Collateral
```

Example:

- Collateral: $1,000 USDT
- Leverage: 10x
- Notional Value: $10,000
- Position Size: $10,000 / Entry Price

GoDark Leverage Tiers:

- 20x (conservative)
- 50x (moderate)
- 100x (aggressive)
- 500x (very aggressive)
- 1000x (maximum, high risk)

Mark Price vs Index Price

Mark Price

Mark Price is the price used for:

- PnL calculations
- Liquidation checks
- Funding rate calculations

Mark Price Sources:

- Oracle feeds (Pyth, Switchboard)
- Spot price from major exchanges
- Time-weighted average price (TWAP)

Why Mark Price?

- Prevents manipulation
- More stable than last trade price
- Fair liquidation pricing

Index Price

Index Price is the underlying asset's spot price, typically:

- Average of multiple exchanges
- Weighted by volume
- Updated frequently

GoDark Usage:

- Mark Price: From oracle feeds (Pyth/Switchboard)
- Used for all position calculations
- Updated every second for funding rate

Funding Rate Mechanics

What is Funding Rate?

Funding rate is a periodic payment between long and short positions:

- **Positive funding rate**: Longs pay shorts (more longs than shorts)
- **Negative funding rate**: Shorts pay longs (more shorts than longs)
- **Purpose**: Keeps perpetual price aligned with spot price

Funding Rate Calculation

Premium Index:

```
Premium Index = (Mark Price - Index Price) / Index Price
```

Interest Rate:

```
Interest Rate = (Interest Rate Long - Interest Rate Short) / 24
```

Funding Rate:

```
Funding Rate = Premium Index + Interest Rate
Funding Rate = Clamp(Funding Rate, -0.75%, +0.75%)  // Clamped
```

GoDark Implementation:

- Calculated every **1 second**
- Aggregated hourly
- Applied hourly to all open positions

Funding Payment

Payment Amount:

```
Funding Payment = Position Size × Mark Price × Funding Rate
```

Who Pays:

- If funding rate > 0: Longs pay shorts
- If funding rate < 0: Shorts pay longs

Example:

- Position Size: 1 BTC
- Mark Price: $50,000
- Funding Rate: 0.01% (0.0001)
- Payment: 1 × $50,000 × 0.0001 = $5
- If long: Pay $5
- If short: Receive $5

GoDark Frequency:

- Payments occur **hourly**
- Accumulated from 3,600 one-second calculations

Leverage and Margin

Initial Margin

Initial Margin is the collateral required to open a position:

```
Initial Margin = Notional Value / Leverage
```

Example:

- Notional Value: $10,000
- Leverage: 10x
- Initial Margin: $10,000 / 10 = $1,000

GoDark:

- Minimum initial margin varies by leverage tier
- Higher leverage = higher initial margin requirement

Maintenance Margin

Maintenance Margin is the minimum collateral to keep a position open:

```
Maintenance Margin = Notional Value × Maintenance Margin Rate
```

Maintenance Margin Rate:

- Typically 0.5% - 2% of notional value
- Varies by leverage tier
- Higher leverage = higher maintenance margin rate

Example:

- Notional Value: $10,000
- Maintenance Margin Rate: 1%
- Maintenance Margin: $10,000 × 0.01 = $100

Margin Ratio

Margin Ratio indicates position health:

```
Margin Ratio = (Collateral + Unrealized PnL) / Maintenance Margin
```

Interpretation:

- Margin Ratio > 1.0: Position is safe
- Margin Ratio < 1.0: Position can be liquidated
- Margin Ratio < 0.5: Immediate liquidation risk

GoDark Liquidation:

- Liquidation triggered when Margin Ratio < 1.0
- Partial liquidation possible
- Full liquidation if Margin Ratio < 0.5

PnL Calculation

Unrealized PnL

Unrealized PnL is profit/loss on open positions:

For Long Positions:

```
Unrealized PnL = Position Size × (Mark Price - Entry Price)
```

For Short Positions:

```
Unrealized PnL = Position Size × (Entry Price - Mark Price)
```

Example (Long):

- Position Size: 1 BTC
- Entry Price: $50,000
- Mark Price: $55,000
- Unrealized PnL: 1 × ($55,000 - $50,000) = +$5,000

Example (Short):

- Position Size: 1 BTC
- Entry Price: $50,000
- Mark Price: $45,000
- Unrealized PnL: 1 × ($50,000 - $45,000) = +$5,000

Realized PnL

Realized PnL is profit/loss when closing a position:

```
Realized PnL = Position Size × (Exit Price - Entry Price)  // Long
Realized PnL = Position Size × (Entry Price - Exit Price)   // Short
```

Plus Funding Payments:

```
Total Realized PnL = Realized PnL + Cumulative Funding Payments
```

Total Equity

Total Equity is your account value:

```
Total Equity = Collateral + Unrealized PnL - Unrealized Funding
```

GoDark Display:

- Real-time unrealized PnL
- Cumulative funding payments
- Total equity

Liquidation Mechanics

Liquidation Price

Liquidation Price is when a position gets liquidated:

For Long Positions:

```
Liquidation Price = Entry Price × (1 - Initial Margin Rate / Maintenance Margin Rate)
```

For Short Positions:

```
Liquidation Price = Entry Price × (1 + Initial Margin Rate / Maintenance Margin Rate)
```

Example (Long, 10x leverage):

- Entry Price: $50,000
- Initial Margin Rate: 10% (1/leverage)
- Maintenance Margin Rate: 1%
- Liquidation Price: $50,000 × (1 - 0.10 / 0.01) = $45,000

Partial vs Full Liquidation

Partial Liquidation:

- Occurs when Margin Ratio < 1.0 but > 0.5
- Liquidates enough to restore Margin Ratio to 1.0
- Remaining position stays open

Full Liquidation:

- Occurs when Margin Ratio < 0.5
- Entire position liquidated

- Remaining collateral returned (if any)

Liquidation Process

1. **Detection**: Liquidation engine monitors positions
2. **Eligibility Check**: Margin Ratio < 1.0
3. **Execution**: Liquidator executes liquidation
4. **Reward**: Liquidator receives fee (e.g., 5% of position value)
5. **Bad Debt**: If insufficient, insurance fund covers

GoDark Implementation:

- Real-time monitoring (100ms intervals)
- Automatic liquidation execution
- Liquidator rewards incentivize participation

Insurance Fund and Bad Debt

Insurance Fund

Purpose: Covers bad debt from liquidations

Bad Debt Scenarios:

- Position liquidated at worse price than expected
- Slippage during liquidation
- Market gaps (flash crashes)

Insurance Fund Sources:

- Portion of trading fees
- Liquidation penalties
- Protocol reserves

GoDark:

- Insurance fund managed on-chain
- Transparent and auditable
- Covers bad debt automatically

Bad Debt Handling

If Bad Debt Occurs:

1. Insurance fund covers the loss
2. If insurance fund insufficient: Protocol may pause
3. Emergency procedures activated

Prevention:

- Proper liquidation incentives
- Adequate insurance fund size
- Risk management parameters

Dark Pool Advantages for Perpetuals

Privacy Benefits

Order Hiding:

- Large orders don't move market

- No front-running
- Reduced market impact

GoDark Dark Pool:

- Orders invisible until execution
- Price-time priority matching
- Institutional-grade privacy

Execution Quality

Large Block Trades:

- Execute large positions without slippage
- Better fill prices
- Reduced market impact

Market Makers:

- Provide liquidity anonymously
- Better spreads
- Reduced adverse selection

Funding Rate Payment Flow

Calculation Loop

```
Every 1 Second:
1. Fetch Mark Price from oracle
2. Fetch Index Price from oracle
3. Calculate Premium Index
4. Calculate Interest Rate
5. Calculate Funding Rate
6. Store sample

Every Hour:
1. Aggregate 3,600 samples
2. Calculate average Funding Rate
3. Apply to all open positions
4. Transfer payments (longs ↔ shorts)
5. Record in history
```

Payment Distribution

For Each Position:

1. Calculate payment amount
2. Deduct from long positions (if funding > 0)
3. Credit to short positions (if funding > 0)
4. Update position collateral
5. Emit event

GoDark Implementation:

- Hourly payment distribution
- Batch processing for efficiency
- On-chain settlement

Position Lifecycle

1. Open Position

```
User → Deposit Collateral → Select Leverage → Open Position
```

Steps:

1. Deposit USDT to collateral vault
2. Select leverage tier (20x - 1000x)
3. Choose long or short
4. Specify position size
5. Position created on-chain

2. Modify Position

Actions:

- **Add Collateral**: Increase margin
- **Remove Collateral**: Decrease margin (if safe)
- **Increase Size**: Add to position
- **Decrease Size**: Partial close

Constraints:

- Must maintain maintenance margin
- Cannot remove collateral if Margin Ratio < 1.5

3. Close Position

Full Close:

- Close entire position
- Realize PnL
- Return remaining collateral
- Deduct funding payments

Partial Close:

- Reduce position size
- Realize PnL on closed portion
- Remaining position stays open

4. Liquidation

Automatic:

- Triggered by liquidation engine
- Liquidator executes
- Position closed
- Remaining collateral returned (if any)

Key Formulas Reference

Position Metrics

```
Notional Value = Position Size × Mark Price
Leverage = Notional Value / Collateral
Margin Ratio = (Collateral + Unrealized PnL) / Maintenance Margin
```

PnL Calculations

```
Unrealized PnL (Long) = Position Size × (Mark Price - Entry Price)
Unrealized PnL (Short) = Position Size × (Entry Price - Mark Price)
Realized PnL = Position Size × (Exit Price - Entry Price) + Funding Payments
```

Funding Rate

```
Premium Index = (Mark Price - Index Price) / Index Price
Funding Rate = Premium Index + Interest Rate
Funding Payment = Position Size × Mark Price × Funding Rate
```

Liquidation

```
Liquidation Price (Long) = Entry Price × (1 - Initial Margin / Maintenance Margin)
Liquidation Price (Short) = Entry Price × (1 + Initial Margin / Maintenance Margin)
```

Key Takeaways

1. **Perpetual futures have no expiration** - Unlike traditional futures
2. **Funding rate keeps price aligned** - Longs and shorts pay each other
3. **Leverage amplifies risk** - Higher leverage = higher risk
4. **Margin ratio determines safety** - < 1.0 = liquidation risk
5. **Mark price prevents manipulation** - Uses oracle feeds, not last trade
6. **Dark pool provides privacy** - Large orders don't move market
7. **Insurance fund covers bad debt** - Protects protocol solvency

---

Next Steps

- Review **05-component-overview.md** to see how these concepts are implemented
- Study **03-solana-fundamentals.md** for Solana-specific implementation details
- Check component assignments for implementation details

Last Updated: November 2025

05. GoDark DEX Component Overview

Overview

GoDark DEX consists of **8 core components** that work together to provide a high-performance perpetual futures trading platform. This guide provides a high-level overview of each component, their interactions, and shared patterns.

Component Architecture

```
┌──────────────────────────────────────────────────────────────┐
│                    User Interface Layer                      │
│              (Web UI, Mobile App, API Clients)               │
└──────────────────────────────────────────────────────────────┘
                      │
                      ▼
┌──────────────────────────────────────────────────────────────┐
│                  API & WebSocket Gateway                     │
│         (Authentication, Rate Limiting, Routing)             │
└──────────────────────────────────────────────────────────────┘
                      │
                      ▼
┌──────────────────────────────────────────────────────────────┐
│                Off-Chain Matching Engine                     │
│          (Dark Pool Order Book, Matching Algorithm)          │
└──────────────────────────────────────────────────────────────┘
                      │
                      ▼
┌──────────────────────────────────────────────────────────────┐
│              Settlement Relayer (Component 1)                │
│    (Batch Accumulation, Merkle Trees, Transaction Building)  │
└──────────────────────────────────────────────────────────────┘
                      │
         ┌────────────┴────────────┐
         ▼                         ▼
┌─────────────────┐      ┌───────────────────┐
│ On-Chain        │      │ Off-Chain         │
│ Components      │      │ Components        │
│                 │      │                   │
│ ┌───────────┐   │      │ ┌──────────────┐  │
│ │Position   │◄──┼──────┼─│Oracle        │  │
│ │Management │   │      │ │Integration   │  │
│ │(Comp 2)   │   │      │ │(Comp 6)      │  │
│ └───────────┘   │      │ └──────────────┘  │
│                 │      │                   │
│ ┌───────────┐   │      │ ┌──────────────┐  │
│ │Liquidation│◄──┼──────┼─│Funding Rate  │  │
│ │  Engine   │   │      │ │(Comp 5)      │  │
│ │(Comp 3)   │   │      │ └──────────────┘  │
│ └───────────┘   │      │                   │
│                 │      │ ┌──────────────┐  │
│ ┌───────────┐   │      │ │Ephemeral     │  │
│ │Collateral │   │      │ │Vault         │  │
│ │  Vault    │   │      │ │(Comp 4)      │  │
│ │(Comp 7)   │   │      │ └──────────────┘  │
│ └───────────┘   │      │                   │
│                 │      │ ┌──────────────┐  │
│ ┌───────────┐   │      │ │  Program     │  │
│ │ Program   │   │      │ │  Upgrade     │  │
│ │ Upgrade   │   │      │ │ (Comp 8)     │  │
│ │(Comp 8)   │   │      │ └──────────────┘  │
│ └───────────┘   │      │                   │
└─────────────────┘      └───────────────────┘
         │
         ▼
┌──────────────────────────────────────────────────────────────┐
│                     Solana Blockchain                        │
│            (Consensus, Finality, State Storage)              │
└──────────────────────────────────────────────────────────────┘
```

Component 1: Settlement Relayer & Batch Processing

Purpose

Bridges off-chain trade execution with on-chain settlement, enabling high-throughput trading while maintaining on-chain security.

Key Responsibilities

1. **Batch Accumulation**
    - Collects trades in 1-second windows
    - Groups by user and market
    - Prepares for batch settlement
2. **Net Position Calculation**
    - Calculates net position changes per user

- Reduces on-chain operations
- Optimizes transaction size

3. **Merkle Tree Generation**
    - Creates Merkle tree of all trades
    - Enables efficient verification
    - Prevents trade manipulation

4. **Transaction Building**
    - Constructs Solana transactions
    - Includes all required accounts
    - Handles compute budget

5. **Settlement Execution**
    - Submits transactions to Solana
    - Monitors confirmation
    - Retries on failure

Technology Stack

- **Language**: Rust
- **Framework**: Anchor (for on-chain program)
- **Database**: PostgreSQL (trade history)
- **Performance**: 100+ trades/second target

Integration Points

- **Input**: Off-chain matching engine (trades)
- **Output**: Position Management program (on-chain)
- **Dependencies**: Oracle Integration (for price verification)

Key Patterns

- **Batch Processing**: Accumulate trades before settlement
- **Merkle Trees**: Cryptographic verification
- **Netting**: Reduce on-chain operations
- **Retry Logic**: Handle transaction failures

Component 2: Position Management System

Purpose

Manages leveraged positions with margin calculations, PnL tracking, and position lifecycle management.

Key Responsibilities

1. **Position Creation**
    - Create PDA-based position accounts
    - Validate leverage tier
    - Lock collateral

2. **Margin Calculations**
    - Initial margin requirements
    - Maintenance margin checks
    - Margin ratio monitoring

3. **PnL Tracking**
    - Unrealized PnL calculation
    - Realized PnL on close
    - Funding rate integration
4. **Position Modifications**
    - Add/remove collateral
    - Increase/decrease size
    - Partial closes
5. **Position Closing**
    - Full position closure
    - PnL realization
    - Collateral return

Technology Stack

- **On-Chain**: Anchor program (Rust)
- **Off-Chain**: Rust service
- **Database**: PostgreSQL (position history)

Integration Points

- **Input**: Settlement Relayer (position updates)
- **Output**: Liquidation Engine (position data)
- **Dependencies**:
    - Collateral Vault (collateral management)
    - Oracle Integration (mark price)
    - Funding Rate (funding payments)

Key Patterns

- **PDA Positions**: One position account per user per market
- **Margin Calculations**: Fixed-point arithmetic
- **State Machine**: Position lifecycle states
- **Versioning**: Account version for migrations

Component 3: Liquidation Engine

Purpose

Real-time position monitoring and automatic liquidation system to protect protocol solvency.

Key Responsibilities

1. **Position Monitoring**
    - Monitor all open positions
    - Check margin ratios
    - Detect liquidation candidates
2. **Liquidation Execution**
    - Partial liquidation logic
    - Full liquidation logic
    - Transaction building
3. **Liquidator Rewards**

- Calculate rewards
- Distribute to liquidators
- Incentivize participation

4. **Insurance Fund Integration**
   - Cover bad debt
   - Monitor fund health
   - Emergency procedures

5. **Bad Debt Handling**
   - Detect bad debt scenarios
   - Insurance fund coverage
   - Protocol pause if needed

Technology Stack

- **On-Chain**: Anchor program
- **Off-Chain**: Rust service (monitoring)
- **Database**: PostgreSQL (liquidation history)

Integration Points

- **Input**: Position Management (position data)
- **Output**: Position Management (liquidation execution)
- **Dependencies**:
  - Oracle Integration (mark price)
  - Collateral Vault (collateral handling)
  - Insurance Fund (bad debt coverage)

Key Patterns

- **Real-Time Monitoring**: 100ms scan intervals
- **Priority Queue**: Liquidate most critical first
- **Partial Liquidation**: Restore margin ratio
- **Reward Mechanism**: Incentivize liquidators

Component 4: Ephemeral Vault System

Purpose

Temporary session-based wallets with delegation for gasless trading and improved UX.

Key Responsibilities

1. **Session Creation**
   - Generate ephemeral keypairs
   - Create PDA-based vaults
   - Set session expiry

2. **Delegation Management**
   - Approve delegate for trading
   - Verify delegation
   - Revoke access

3. **Auto-Deposit**
   - Monitor SOL balance

  - Auto-deposit for fees
  - Maintain minimum balance

4. **Transaction Signing**
  - Sign with ephemeral wallet
  - Handle priority fees
  - Retry logic
5. **Session Cleanup**
  - Detect expired sessions
  - Cleanup resources
  - Return remaining SOL

Technology Stack

- **On-Chain**: Anchor program
- **Off-Chain**: Rust service (key management)
- **Database**: PostgreSQL (session tracking)

Integration Points

- **Input**: User requests (session creation)
- **Output**: Settlement Relayer (signed transactions)
- **Dependencies**: Position Management (for trading)

Key Patterns

- **Ephemeral Keypairs**: Temporary wallets
- **Delegation**: Approve trading authority
- **Auto-Deposit**: Maintain SOL for fees
- **Session Expiry**: Automatic cleanup

Component 5: Funding Rate Calculation System

Purpose

Perpetual futures funding rate calculation and hourly payment distribution.

Key Responsibilities

1. **Rate Calculation**
  - Calculate every 1 second
  - Premium index calculation
  - Interest rate calculation
  - Rate clamping
2. **Hourly Aggregation**
  - Aggregate 3,600 samples
  - Calculate average rate
  - Prepare for distribution
3. **Payment Distribution**
  - Apply to all open positions
  - Longs pay shorts (or vice versa)
  - Update position collateral

4. **History Tracking**
   - Store rate history
   - Calculate statistics
   - Provide API access
5. **Oracle Integration**
   - Fetch mark price
   - Fetch index price
   - Handle oracle failures

Technology Stack

- **Off-Chain**: Rust service (calculation loop)
- **On-Chain**: Anchor program (payment distribution)
- **Database**: PostgreSQL (rate history)
- **Cache**: Redis (fast rate access)

Integration Points

- **Input**: Oracle Integration (prices)
- **Output**: Position Management (funding payments)
- **Dependencies**: Oracle Integration (price feeds)

Key Patterns

- **1-Second Loop**: Continuous calculation
- **Hourly Aggregation**: Batch payments
- **Parallel Processing**: 50+ symbols
- **Rate Clamping**: Prevent extreme rates

Component 6: Oracle Integration & Price Feeds

Purpose

Multi-oracle price feed system with validation, consensus, and failover.

Key Responsibilities

1. **Oracle Integration**
   - Pyth Network integration
   - Switchboard fallback
   - Price normalization
2. **Price Consensus**
   - Median calculation
   - Outlier detection
   - Weighted averaging
3. **Validation**
   - Confidence intervals
   - Staleness checks
   - Manipulation detection
4. **Failover**
   - Automatic failover

    - Health monitoring
    - Circuit breakers

5. **Price Distribution**
    - Cache prices (Redis)
    - WebSocket updates
    - API access

Technology Stack

- **Off-Chain**: Rust service
- **Database**: PostgreSQL (price history)
- **Cache**: Redis (current prices)
- **WebSocket**: Real-time updates

Integration Points

- **Input**: Pyth/Switchboard (price feeds)
- **Output**:
    - Funding Rate (mark price)
    - Liquidation Engine (mark price)
    - Position Management (mark price)

Key Patterns

- **Multi-Oracle**: Redundancy
- **Consensus**: Median/weighted average
- **Failover**: Automatic switching
- **Caching**: Fast price access

Component 7: Collateral Vault Management

Purpose

Non-custodial collateral vaults with SPL Token management.

Key Responsibilities

1. **Vault Creation**
    - Create PDA-based vaults
    - Create associated token accounts
    - Initialize balances

2. **Deposit/Withdraw**
    - SPL Token transfers
    - Balance tracking
    - Transaction history

3. **Collateral Locking**
    - Lock for positions
    - Unlock on close
    - CPI-callable

4. **Balance Tracking**
    - On-chain balance
    - Off-chain reconciliation

- Discrepancy detection

5. **Vault Monitoring**
    - Monitor all vaults
    - Track TVL
    - Detect anomalies

Technology Stack

- **On-Chain**: Anchor program
- **Off-Chain**: Rust service
- **Database**: PostgreSQL (vault history)

Integration Points

- **Input**: User deposits/withdrawals
- **Output**: Position Management (collateral)
- **Dependencies**: SPL Token Program

Key Patterns

- **PDA Vaults**: Deterministic addresses
- **SPL Token CPI**: Token transfers
- **Lock/Unlock**: Position collateral
- **Reconciliation**: On-chain vs off-chain

Component 8: Program Upgrade & Migration System

Purpose

Safe protocol upgrades with governance, timelock, and state migration.

Key Responsibilities

1. **Upgrade Proposals**
    - Create proposals
    - Link program buffers
    - Set timelock
2. **Multisig Governance**
    - Collect approvals
    - Threshold validation
    - Execute when ready
3. **Timelock Enforcement**
    - 48-hour minimum delay
    - Countdown monitoring
    - Prevent early execution
4. **State Migration**
    - Identify accounts to migrate
    - Transform data
    - Verify migration
5. **Rollback Capability**
    - Detect failures

  - Revert to previous version
  - Restore state

Technology Stack

- **On-Chain**: Anchor program
- **Off-Chain**: Rust service
- **Database**: PostgreSQL (upgrade history)

Integration Points

- **Input**: Governance proposals
- **Output**: All DEX programs (upgrades)
- **Dependencies**: BPF Upgradeable Loader

Key Patterns

- **Multisig**: Custom implementation
- **Timelock**: On-chain enforcement
- **Migration**: Account versioning
- **Rollback**: Emergency recovery

Component Interactions

Data Flow

```
User Trade Request
    ↓
Matching Engine (off-chain)
    ↓
Settlement Relayer (Component 1)
    ↓
Position Management (Component 2)
    ├─→ Collateral Vault (Component 7) - Lock collateral
    ├─→ Oracle Integration (Component 6) - Get mark price
    └─→ Funding Rate (Component 5) - Track for funding
    ↓
Liquidation Engine (Component 3) - Monitor position
    ├─→ Oracle Integration (Component 6) - Check mark price
    └─→ Position Management (Component 2) - Execute liquidation
```

Integration Matrix

Component	Integrates With	Purpose
Settlement Relayer	Position Management	Update positions
Position Management	Collateral Vault	Lock/unlock collateral
Position Management	Oracle Integration	Get mark price
Position Management	Funding Rate	Receive funding payments
Liquidation Engine	Position Management	Read positions
Liquidation Engine	Oracle Integration	Get mark price
Liquidation Engine	Collateral Vault	Handle liquidated collateral

Component	Integrates With	Purpose
Funding Rate	Oracle Integration	Get mark/index prices
Ephemeral Vault	Settlement Relayer	Sign transactions
Program Upgrade	All Components	Upgrade programs

Shared Patterns

1. PDA-Based Accounts

Pattern: Use PDAs for deterministic addresses

Examples:

- Position accounts: `[b"position", user_pubkey]`
- Vault accounts: `[b"vault", user_pubkey]`
- Config accounts: `[b"config"]`

Benefits:

- Deterministic addresses
- Program-controlled
- No key management

2. Cross-Program Invocations (CPIs)

Pattern: Programs call other programs

Examples:

- Collateral Vault → SPL Token Program (transfers)
- Position Management → Collateral Vault (lock/unlock)
- Settlement Relayer → Position Management (updates)

Benefits:

- Composability
- Code reuse
- Modularity

3. Error Handling

Pattern: Custom error types with clear messages

Examples:

```
#[error_code]
pub enum ErrorCode {
    #[msg("Insufficient collateral")]
    InsufficientCollateral,
    #[msg("Invalid leverage tier")]
    InvalidLeverage,
}
```

Benefits:

- Clear error messages
- Type safety
- Better debugging

4. Account Versioning

Pattern: Version field in account data

Examples:

```
#[account]
pub struct Position {
    pub version: u32,
    // ... other fields
}
```

Benefits:

- Migration support
- Backward compatibility
- Upgrade safety

5. Event Emission

Pattern: Emit events for off-chain tracking

Examples:

```
emit!(PositionOpened {
    user: user_pubkey,
    position_id: position_pda,
    size: position_size,
});
```

Benefits:

- Off-chain indexing
- Real-time updates
- Audit trail

Component-Specific Patterns

Settlement Relayer

- **Batch Processing**: Accumulate before settlement
- **Merkle Trees**: Cryptographic verification
- **Netting**: Reduce operations

Position Management

- **Fixed-Point Math**: Precise calculations
- **State Machine**: Position lifecycle
- **Margin Calculations**: Real-time monitoring

Liquidation Engine

- **Priority Queue**: Critical positions first
- **Partial Liquidation**: Restore margin ratio
- **Reward Mechanism**: Incentivize liquidators

Ephemeral Vault

- **Session Management**: Expiry and cleanup

- **Delegation**: Approve trading authority
- **Auto-Deposit**: Maintain SOL balance

Funding Rate

- **Time-Series**: 1-second samples
- **Aggregation**: Hourly averages
- **Distribution**: Batch payments

Oracle Integration

- **Multi-Source**: Pyth + Switchboard
- **Consensus**: Median/weighted average
- **Failover**: Automatic switching

Collateral Vault

- **SPL Token CPI**: Token operations
- **Reconciliation**: On-chain vs off-chain
- **Lock/Unlock**: Position collateral

Program Upgrade

- **Multisig**: Governance approval
- **Timelock**: 48-hour delay
- **Migration**: Account transformation

Key Takeaways

1. **8 Components** work together to form GoDark DEX
2. **Hybrid Architecture**: Off-chain matching, on-chain settlement
3. **Shared Patterns**: PDAs, CPIs, error handling, versioning
4. **Integration Points**: Components communicate via APIs and on-chain calls
5. **Modularity**: Each component is independently deployable
6. **Security**: Multiple layers of validation and checks

Next Steps

- Study your assigned component in detail
- Review component assignment documentation
- Understand integration points with other components
- Review **06-development-workflow.md** for development practices
- Check **07-testing-strategies.md** for testing approaches

Last Updated: November 2025

06. Development Workflow for GoDark DEX

Overview

This guide covers day-to-day development practices, tools, and workflows for building GoDark DEX components. Follow these practices to ensure code quality, consistency, and efficient collaboration.

Local Development Setup

Prerequisites

Required Software:

1. **Rust** (latest stable)

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
rustup update stable
```

2. **Solana CLI** (latest)

```
sh -c "$(curl -sSfL https://release.solana.com/stable/install)"
solana --version
```

3. **Anchor Framework**

```
cargo install --git https://github.com/coral-xyz/anchor avm --locked --force
avm install latest
avm use latest
anchor --version
```

4. **PostgreSQL** (for backend services)

```
# macOS
brew install postgresql
brew services start postgresql

# Linux
sudo apt-get install postgresql postgresql-contrib
```

5. **Node.js & Yarn** (for Anchor tests)

```
# Install Node.js (v18+)
# Install Yarn
npm install -g yarn
```

Environment Setup

Solana Configuration:

```
# Set to localnet for development
solana config set --url localhost

# Generate keypair if needed
solana-keygen new

# Airdrop SOL for testing
solana airdrop 10
```

Environment Variables:

```
# .env file
SOLANA_RPC_URL=http://localhost:8899
DATABASE_URL=postgresql://postgres:postgres@localhost/godark_dev
ANCHOR_PROVIDER_URL=http://localhost:8899
ANCHOR_WALLET=~/.config/solana/id.json
```

Project Structure Conventions

Anchor Program Structure

```
your-component/
├── Anchor.toml              # Anchor configuration
├── Cargo.toml               # Workspace Cargo.toml
├── programs/
│   └── your-component/
│       ├── src/
│       │   └── lib.rs       # Main program file
│       └── Cargo.toml       # Program dependencies
├── tests/
│   └── your-component.ts    # Anchor tests
└── migrations/              # Database migrations (if applicable)
```

Rust Backend Service Structure

```
your-service/
├── Cargo.toml               # Service dependencies
├── src/
│   ├── main.rs             # Service entry point
│   ├── lib.rs              # Library root
│   ├── error.rs            # Error types
│   ├── database.rs         # Database operations
│   ├── api.rs              # REST API
│   └── websocket.rs        # WebSocket handlers
├── migrations/
│   └── 001_initial_schema.sql
└── README.md
```

Naming Conventions

- **Files**: `snake_case.rs` (Rust), `kebab-case.ts` (TypeScript)
- **Modules**: `snake_case`
- **Structs**: `PascalCase`
- **Functions**: `snake_case`
- **Constants**: `UPPER_SNAKE_CASE`

Anchor Project Initialization

Create New Anchor Project

```
anchor init your-component
cd your-component
```

Configure Anchor.toml

```
[features]
resolution = true
skip-lint = false

[programs.localnet]
your_component = "YourProgram11111111111111111111111111"

[registry]
url = "https://api.apr.dev"

[provider]
cluster = "Localnet"
wallet = "~/.config/solana/id.json"

[scripts]
test = "yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests/**/*.ts"
```

Build Process

```
# Build program
anchor build

# Build and deploy to localnet
anchor build
anchor deploy

# Generate IDL
anchor idl parse -f programs/your-component/src/lib.rs -o target/idl/your_component.json
```

Testing Workflow

Anchor Tests (TypeScript)

Test Structure:

```
import * as anchor from "@coral-xyz/anchor";
import { Program } from "@coral-xyz/anchor";
import { YourComponent } from "../target/types/your_component";

describe("your-component", () => {
  const provider = anchor.AnchorProvider.env();
  anchor.setProvider(provider);

  const program = anchor.workspace.YourComponent as Program<YourComponent>;

  it("Initializes correctly", async () => {
    // Test code
  });
});
```

Running Tests:

```
# Run all tests
anchor test

# Run specific test file
anchor test tests/your-component.ts

# Run with verbose output
anchor test -- --verbose
```

Rust Unit Tests

Test Structure:

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

    #[test]
    fn test_calculation() {
        // Test code
    }
}
```

Running Tests:

```bash
# Run all tests
cargo test

# Run specific test
cargo test test_calculation

# Run with output
cargo test -- --nocapture
```

Integration Tests

Backend Service Tests:

```rust
#[tokio::test]
async fn test_api_endpoint() {
    // Test API endpoints
}
```

Running Integration Tests:

```bash
cargo test --test integration_test
```

Deployment Process

Localnet Deployment

```bash
# Start local validator
solana-test-validator

# In another terminal, deploy
anchor build
anchor deploy

# Verify deployment
solana program show YourProgram11111111111111111111111111
```

Devnet Deployment

```
# Switch to devnet
solana config set --url devnet

# Airdrop SOL
solana airdrop 2

# Deploy
anchor build
anchor deploy --provider.cluster devnet

# Verify
solana program show YourProgram111111111111111111111111111 --url devnet
```

Mainnet Deployment

⚠️ **Production Deployment Checklist:**

- [ ] All tests passing
- [ ] Security audit completed
- [ ] Code review approved
- [ ] Documentation updated
- [ ] Monitoring configured
- [ ] Rollback plan prepared

```
# Switch to mainnet
solana config set --url mainnet-beta

# Deploy (use upgrade authority)
anchor deploy --provider.cluster mainnet-beta
```

Debugging Techniques

Anchor Program Debugging

1. Program Logs:

```
msg!("Debug: value = {}", value);
```

View Logs:

```
solana logs
```

2. Account Inspection:

```
# View account data
solana account YourAccount111111111111111111111111111

# Decode account data
anchor account YourAccount --program-id YourProgram111111111111111111111111111
```

3. Transaction Inspection:

```
# View transaction
solana confirm <signature>

# Decode transaction
solana confirm <signature> --verbose
```

Rust Backend Debugging

1. Logging:

```rust
use tracing::{info, error, debug, warn};

info!("Processing trade: {:?}", trade);
error!("Failed to process: {}", error);
debug!("Debug info: {:?}", data);
```

2. Database Debugging:

```
# Connect to database
psql -d godark_dev

# Query tables
SELECT * FROM positions LIMIT 10;
```

3. API Debugging:

```
# Test API endpoint
curl -X POST http://localhost:3000/api/endpoint \
  -H "Content-Type: application/json" \
  -d '{"key": "value"}'
```

Common Debugging Tools

- **Solana Explorer**: https://explorer.solana.com/
- **Solscan**: https://solscan.io/
- **Anchor IDL Viewer**: View program interface
- **Transaction Decoders**: Decode transaction data

Version Control Practices

OneFlow Branching Strategy

GoDark uses **OneFlow** branching strategy for a simple, efficient git workflow.

Reference: OneFlow: A Simple Git Repository Strategy

Branch Types

1. Main Branch (`main`)

- Single long-lived branch
- Always in deployable state
- All tests passing
- Production-ready code

2. Feature Branches (`feature/XYZ-1234`)

- Short-lived branches for new features
- Branch from `main`
- Merge back to `main` when complete
- Naming: `feature/component-name-description`

Example:

```
git checkout -b feature/position-management-margin-calc
# ... make changes ...
git commit -m "Add margin calculation logic"
git push origin feature/position-management-margin-calc
# Create PR to main
```

3. Bugfix Branches (`bugfix/XYZ-1234`)

- For non-critical bug fixes
- Branch from `main`
- Merge back to `main` when fixed
- Naming: `bugfix/component-name-issue`

Example:

```
git checkout -b bugfix/liquidation-engine-priority-queue
# ... fix bug ...
git commit -m "Fix priority queue ordering"
git push origin bugfix/liquidation-engine-priority-queue
```

4. Hotfix Branches (`hotfix/XYZ-1234`)

- For critical production issues
- Branch from latest tagged release
- Fix, test, tag, merge to `main`
- Naming: `hotfix/component-name-critical-issue`

Example:

```
git checkout -b hotfix/collateral-vault-security-patch v1.2.3
# ... fix critical issue ...
git commit -m "Security patch: Fix authorization check"
git tag v1.2.4
git push origin hotfix/collateral-vault-security-patch
git push origin v1.2.4
```

5. Release Branches (`release/x.y.z`)

- For preparing releases
- Branch from `main`
- Final testing and adjustments
- Tag and merge to `main`

Example:

```
git checkout -b release/1.3.0
# ... final testing ...
git commit -m "Release 1.3.0"
git tag v1.3.0
git push origin release/1.3.0
git push origin v1.3.0
```

Workflow Example

```
main (production-ready)
  │
  ├─→ feature/position-management (develop feature)
  │     │
  │     └─→ Merge to main when complete
  │
  ├─→ bugfix/liquidation-engine (fix bug)
  │     │
  │     └─→ Merge to main when fixed
  │
  └─→ release/1.3.0 (prepare release)
        │
        └─→ Tag and merge to main
```

Branch Naming Conventions

- **Feature**: `feature/component-name-description`
- **Bugfix**: `bugfix/component-name-issue`
- **Hotfix**: `hotfix/component-name-critical-issue`
- **Release**: `release/x.y.z`

Examples:

- `feature/settlement-relayer-batch-processing`
- `bugfix/funding-rate-calculation-error`
- `hotfix/collateral-vault-authorization-bug`
- `release/1.2.0`

Merge Workflow

Pull Request Process:

1. Create feature/bugfix branch
2. Make changes and commit
3. Push branch to remote
4. Create Pull Request to `main`
5. Code review
6. Address feedback
7. Merge to `main`
8. Delete branch

Merge Commit Message:

```
Merge feature/component-name-description

- Added feature X
- Fixed issue Y
- Updated documentation
```

Code Review Guidelines

Review Checklist

Functionality:

- [ ] Code works as intended
- [ ] Edge cases handled
- [ ] Error handling appropriate
- [ ] Performance acceptable

Code Quality:

- [ ] Follows Rust/TypeScript conventions
- [ ] No code duplication
- [ ] Clear variable/function names
- [ ] Adequate comments

Security:

- [ ] Authority checks present
- [ ] Input validation
- [ ] No overflow/underflow risks
- [ ] Secure key management

Testing:

- [ ] Unit tests added
- [ ] Integration tests added
- [ ] Edge cases tested
- [ ] Tests passing

Documentation:

- [ ] Code comments added
- [ ] README updated
- [ ] API docs updated (if applicable)

Review Process

1. **Author**: Create PR with clear description
2. **Reviewer**: Review within 24 hours
3. **Feedback**: Provide constructive feedback
4. **Author**: Address feedback
5. **Approval**: At least 1 approval required
6. **Merge**: Squash and merge to `main`

Documentation Standards

Code Comments

Rust:

```rust
/// Calculates the margin ratio for a position.
///
/// # Arguments
/// * `collateral` - Current collateral amount
/// * `unrealized_pnl` - Unrealized profit/loss
/// * `maintenance_margin` - Required maintenance margin
///
/// # Returns
/// Margin ratio (collateral + pnl) / maintenance_margin
pub fn calculate_margin_ratio(
    collateral: u64,
    unrealized_pnl: i64,
    maintenance_margin: u64,
) -> Result<u64> {
    // Implementation
}
```

TypeScript:

```
/**
 * Initializes a new position account.
 * @param user - User's public key
 * @param market - Market identifier
 * @param leverage - Leverage multiplier (1-1000)
 * @returns Position account public key
 */
async function initializePosition(
  user: PublicKey,
  market: PublicKey,
  leverage: number
): Promise<PublicKey> {
  // Implementation
}
```

README Requirements

Each component should have a README.md with:

- Component overview
- Setup instructions
- Usage examples
- API documentation (if applicable)
- Testing instructions
- Deployment guide

Common Development Pitfalls and Solutions

Pitfall 1: Account Ownership Mistakes

Problem: Trying to modify account owned by wrong program

Solution:

```
// Always verify ownership
require!(
    account.owner == expected_program_id,
    ErrorCode::InvalidAccountOwner
);
```

Pitfall 2: PDA Seed Mismatches

Problem: PDA derivation fails due to seed mismatch

Solution:

```
// Use constants for seeds
const SEED_VAULT: &[u8] = b"vault";

let (pda, bump) = Pubkey::find_program_address(
    &[SEED_VAULT, user_pubkey.as_ref()],
    program_id,
);
```

Pitfall 3: Rent-Exempt Account Requirements

Problem: Account not rent-exempt, gets closed

Solution:

```
// Calculate rent-exempt minimum
let rent = Rent::get()?;
let space = 8 + State::LEN;
let minimum_balance = rent.minimum_balance(space);

// Ensure account has enough balance
require!(
    account.lamports() >= minimum_balance,
    ErrorCode::InsufficientBalance
);
```

Pitfall 4: Compute Unit Exhaustion

Problem: Transaction fails due to compute limit

Solution:

```
// Set compute budget
use solana_program::compute_budget::ComputeBudgetInstruction;

let compute_budget = ComputeBudgetInstruction::set_compute_unit_limit(400_000);
instructions.push(compute_budget);
```

Pitfall 5: Transaction Size Limits

Problem: Transaction too large (>1,232 bytes)

Solution:

- Batch operations into multiple transactions
- Reduce account data size
- Use compression techniques
- Optimize instruction data

Pitfall 6: Missing Authority Checks

Problem: Anyone can call restricted instruction

Solution:

```
// Always check authority
require!(
    ctx.accounts.authority.key() == &expected_authority,
    ErrorCode::Unauthorized
);

// Or use Anchor's Signer constraint
#[account(signer)]
pub authority: Signer<'info>,
```

Development Best Practices

1. Start Local, Test Thoroughly

- Always test on localnet first
- Verify all functionality
- Test edge cases
- Check error handling

2. Use Type Safety

- Leverage Rust's type system
- Use Anchor's type-safe wrappers
- Avoid `unwrap()` in production code
- Use `Result` types properly

3. Write Tests First (TDD)

- Write tests before implementation
- Test edge cases
- Test error conditions
- Aim for >80% coverage

4. Document As You Go

- Add comments for complex logic
- Document public APIs
- Update README with changes
- Keep docs in sync with code

5. Review Before Committing

- Review your own code
- Check for common mistakes
- Run linters/formatters
- Verify tests pass

Key Takeaways

1. **Proper Setup**: Install all required tools before starting
2. **Consistent Structure**: Follow project structure conventions
3. **Test Thoroughly**: Write tests for all functionality
4. **OneFlow Strategy**: Use simple branching workflow
5. **Code Review**: Always get code reviewed
6. **Documentation**: Keep docs updated
7. **Debugging**: Use appropriate tools for each layer

Next Steps

- Review **07-testing-strategies.md** for comprehensive testing guide
- Check **08-common-patterns-pitfalls.md** for Solana patterns
- Study **09-security-best-practices.md** for security guidelines

---

Last Updated: November 2025

07. Testing Strategies for GoDark DEX

Overview

Comprehensive testing is critical for financial applications handling user funds. This guide covers testing strategies, patterns, and best practices for Solana/Anchor programs and Rust backend services.

Testing Pyramid

```
         /\
        /  \
       / E2E \          Few, slow, expensive
      /--------\
     /          \
    / Integration \      Some, medium speed
   /--------------\
  /                \
 /   Unit Tests     \    Many, fast, cheap
/------------------\
```

GoDark Testing Strategy:

- **Unit Tests**: 70% - Fast, isolated component tests
- **Integration Tests**: 25% - Component interaction tests
- **E2E Tests**: 5% - Full system tests

Anchor Test Framework Setup

Initial Setup

Install Dependencies:

```
yarn add @coral-xyz/anchor @solana/web3.js @solana/spl-token chai mocha
```

Test Configuration (`tests/your-component.ts`):

```
import * as anchor from "@coral-xyz/anchor";
import { Program } from "@coral-xyz/anchor";
import { YourComponent } from "../target/types/your_component";
import { assert } from "chai";

describe("your-component", () => {
  const provider = anchor.AnchorProvider.env();
  anchor.setProvider(provider);

  const program = anchor.workspace.YourComponent as Program<YourComponent>;

  // Test accounts
  const user = anchor.web3.Keypair.generate();

  before(async () => {
    // Airdrop SOL for testing
    await provider.connection.confirmTransaction(
      await provider.connection.requestAirdrop(
        user.publicKey,
        10 * anchor.web3.LAMPORTS_PER_SOL
      ),
      "confirmed"
    );
  });
});
```

Running Tests

```
# Run all tests
anchor test

# Run specific test file
anchor test tests/your-component.ts

# Run with verbose output
anchor test -- --verbose

# Run with coverage (if configured)
anchor test -- --coverage
```

Unit Testing Patterns

Instruction Testing

Test Instruction Execution:

```
it("Initializes position correctly", async () => {
  const positionPDA = anchor.web3.PublicKey.findProgramAddressSync(
    [Buffer.from("position"), user.publicKey.toBuffer()],
    program.programId
  )[0];

  await program.methods
    .initializePosition(new anchor.BN(1000), 10)
    .accounts({
      position: positionPDA,
      user: user.publicKey,
      systemProgram: anchor.web3.SystemProgram.programId,
    })
    .signers([user])
    .rpc();

  const position = await program.account.position.fetch(positionPDA);
  assert.equal(position.size.toNumber(), 1000);
  assert.equal(position.leverage, 10);
});
```

Account Validation Testing

Test Account Constraints:

```
it("Fails with invalid authority", async () => {
  const invalidUser = anchor.web3.Keypair.generate();

  try {
    await program.methods
      .closePosition()
      .accounts({
        position: positionPDA,
        authority: invalidUser.publicKey, // Wrong authority
      })
      .signers([invalidUser])
      .rpc();

    assert.fail("Should have failed");
  } catch (err) {
    assert.include(err.message, "InvalidAuthority");
  }
});
```

Error Testing

Test Error Conditions:

```
it("Fails with insufficient collateral", async () => {
  try {
    await program.methods
      .openPosition(new anchor.BN(10000), 100) // Requires 1000 collateral
      .accounts({
        user: user.publicKey,
        // ... other accounts
      })
      .rpc();

    assert.fail("Should have failed");
  } catch (err) {
    assert.include(err.message, "InsufficientCollateral");
  }
});
```

Edge Case Testing

Test Boundary Conditions:

```
it("Handles maximum leverage correctly", async () => {
  // Test with 1000x leverage (maximum)
  await program.methods
    .openPosition(new anchor.BN(1000), 1000)
    .accounts({
      // ... accounts
    })
    .rpc();

  const position = await program.account.position.fetch(positionPDA);
  assert.equal(position.leverage, 1000);
});

it("Handles minimum position size", async () => {
  // Test with minimum position size
  await program.methods
    .openPosition(new anchor.BN(1), 1)
    .accounts({
      // ... accounts
    })
    .rpc();
});
```

Integration Testing

Multi-Instruction Flows

Test Complete Workflows:

```
it("Complete position lifecycle", async () => {
  // 1. Initialize
  await program.methods.initializePosition(1000, 10).rpc();

  // 2. Modify position
  await program.methods.addCollateral(new anchor.BN(500)).rpc();

  // 3. Close position
  await program.methods.closePosition().rpc();

  // Verify final state
  const position = await program.account.position.fetchNullable(positionPDA);
  assert.isNull(position); // Position should be closed
});
```

CPI Testing

Test Cross-Program Invocations:

```
it("Transfers tokens via CPI", async () => {
  const fromTokenAccount = await getAssociatedTokenAddress(mint, user.publicKey);
  const toTokenAccount = await getAssociatedTokenAddress(mint, vaultPDA, true);

  const balanceBefore = await getAccount(provider.connection, toTokenAccount);

  await program.methods
    .depositCollateral(new anchor.BN(1000))
    .accounts({
      from: fromTokenAccount,
      to: toTokenAccount,
      // ... other accounts
    })
    .rpc();

  const balanceAfter = await getAccount(provider.connection, toTokenAccount);
  assert.equal(
    balanceAfter.amount - balanceBefore.amount,
    BigInt(1000)
  );
});
```

Mock External Programs

Mock Oracle for Testing:

```
// Create mock oracle account
const mockOracle = anchor.web3.Keypair.generate();
const oracleData = {
  price: new anchor.BN(50000),
  confidence: new anchor.BN(100),
  timestamp: new anchor.BN(Date.now() / 1000),
};

// Use mock oracle in tests
await program.methods
  .updateMarkPrice()
  .accounts({
    oracle: mockOracle.publicKey,
  })
  .rpc();
```

On-Chain Testing

Localnet Deployment

Deploy to Local Validator:

```
before(async () => {
  // Start local validator (in separate terminal)
  // solana-test-validator

  // Deploy program
  await program.provider.connection.confirmTransaction(
    await program.provider.connection.requestAirdrop(
      program.provider.wallet.publicKey,
      10 * anchor.web3.LAMPORTS_PER_SOL
    ),
    "confirmed"
  );

  await program.program.methods
    .initialize()
    .rpc();
});
```

Transaction Simulation

Simulate Transactions:

```
it("Simulates transaction without executing", async () => {
  const tx = await program.methods
    .openPosition(1000, 10)
    .accounts({
      // ... accounts
    })
    .transaction();

  const simulation = await program.provider.connection.simulateTransaction(tx);

  assert.isTrue(simulation.value.err === null);
  assert.isAbove(simulation.value.logs.length, 0);
});
```

Rust Backend Testing

Unit Tests

Test Business Logic:

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

    #[test]
    fn test_margin_calculation() {
        let collateral = 1000u64;
        let leverage = 10u8;
        let initial_margin = calculate_initial_margin(collateral, leverage);

        assert_eq!(initial_margin, 10000);
    }

    #[test]
    fn test_margin_ratio() {
        let collateral = 1000u64;
        let unrealized_pnl = 500i64;
        let maintenance_margin = 500u64;

        let ratio = calculate_margin_ratio(collateral, unrealized_pnl, maintenance_margin);

        assert_eq!(ratio, 3.0); // (1000 + 500) / 500
    }
}
```

Integration Tests

Test API Endpoints:

```rust
#[tokio::test]
async fn test_create_position() {
    let app = create_test_app().await;

    let response = app
        .post("/api/positions")
        .json(&json!({
            "user": "User11111111111111111111111111111111",
            "size": 1000,
            "leverage": 10
        }))
        .send()
        .await
        .unwrap();

    assert_eq!(response.status(), 200);

    let position: Position = response.json().await.unwrap();
    assert_eq!(position.size, 1000);
}
```

Database Tests

Test Database Operations:

```rust
#[tokio::test]
async fn test_create_proposal() {
    let db = setup_test_database().await;

    let proposal = UpgradeProposal {
        id: Uuid::new_v4(),
        proposal_id: "Proposal11111111111111111111111111111111".to_string(),
        // ... other fields
    };

    db.create_proposal(&proposal).await.unwrap();

    let fetched = db.get_proposal_by_id(&proposal.proposal_id).await.unwrap();
    assert_eq!(fetched.unwrap().proposal_id, proposal.proposal_id);
}
```

Mocking Strategies

Mock Oracles

Mock Price Feed:

```rust
pub struct MockOracle {
    pub price: u64,
    pub confidence: u64,
}

impl OracleClient for MockOracle {
    async fn get_price(&self, _symbol: &str) -> Result<PriceData> {
        Ok(PriceData {
            price: self.price,
            confidence: self.confidence,
            timestamp: Utc::now(),
        })
    }
}
```

Mock External Programs

Mock SPL Token Program:

```
// Create mock token account
const mockTokenAccount = {
  mint: mint.publicKey,
  owner: user.publicKey,
  amount: new anchor.BN(10000),
};

// Use in tests
await program.methods
  .transferTokens(new anchor.BN(1000))
  .accounts({
    tokenAccount: mockTokenAccount,
    // ... other accounts
  })
  .rpc();
```

Edge Case Testing

Overflow/Underflow

Test Integer Overflow:

```
#[test]
#[should_panic(expected = "overflow")]
fn test_overflow() {
    let max: u64 = u64::MAX;
    let result = max + 1; // Should panic
}

#[test]
fn test_checked_math() {
    let a = u64::MAX;
    let b = 1u64;

    // Use checked math
    match a.checked_add(b) {
        Some(result) => panic!("Should overflow"),
        None => {} // Expected
    }
}
```

Boundary Conditions

Test Boundary Values:

```
it("Handles maximum leverage", async () => {
  await program.methods
    .openPosition(1000, 1000) // Max leverage
    .rpc();
});

it("Handles minimum position size", async () => {
  await program.methods
    .openPosition(1, 1) // Minimum
    .rpc();
});

it("Fails with leverage > 1000", async () => {
  try {
    await program.methods
      .openPosition(1000, 1001) // Too high
      .rpc();
    assert.fail("Should fail");
  } catch (err) {
    assert.include(err.message, "InvalidLeverage");
  }
});
```

Security Testing

Authorization Testing

Test Authority Checks:

```
it("Prevents unauthorized access", async () => {
  const attacker = anchor.web3.Keypair.generate();

  try {
    await program.methods
      .closePosition()
      .accounts({
        position: positionPDA,
        authority: attacker.publicKey, // Not authorized
      })
      .signers([attacker])
      .rpc();

    assert.fail("Should have failed");
  } catch (err) {
    assert.include(err.message, "Unauthorized");
  }
});
```

Input Validation Testing

Test Invalid Inputs:

```
it("Rejects invalid inputs", async () => {
  // Test negative values
  try {
    await program.methods
      .openPosition(new anchor.BN(-1000), 10)
      .rpc();
    assert.fail("Should fail");
  } catch (err) {
    assert.include(err.message, "InvalidInput");
  }

  // Test zero values
  try {
    await program.methods
      .openPosition(new anchor.BN(0), 10)
      .rpc();
    assert.fail("Should fail");
  } catch (err) {
    assert.include(err.message, "InvalidInput");
  }
});
```

Attack Vector Testing

Test Reentrancy Prevention:

```
#[test]
fn test_no_reentrancy() {
    // Test that functions cannot be called recursively
    // Anchor programs are single-threaded, but test CPI reentrancy
}
```

Test Manipulation Attempts:

```
it("Prevents oracle manipulation", async () => {
  // Try to use stale oracle price
  const staleOracle = createStaleOracle();

  try {
    await program.methods
      .updateMarkPrice()
      .accounts({
        oracle: staleOracle.publicKey,
      })
      .rpc();

    assert.fail("Should reject stale price");
  } catch (err) {
    assert.include(err.message, "StalePrice");
  }
});
```

Performance Testing

Compute Unit Testing

Test Compute Usage:

```
it("Stays within compute budget", async () => {
  const tx = await program.methods
    .batchSettle(100) // Process 100 trades
    .transaction();

  const simulation = await program.provider.connection.simulateTransaction(tx);

  assert.isBelow(
    simulation.value.unitsConsumed || 0,
    400000 // Compute budget limit
  );
});
```

Transaction Size Testing

Test Transaction Size:

```
it("Transaction fits within size limit", async () => {
  const tx = await program.methods
    .batchSettle(50) // Batch size
    .transaction();

  const serialized = tx.serialize();
  assert.isBelow(serialized.length, 1232); // Max transaction size
});
```

Load Testing

Test High Throughput:

```rust
#[tokio::test]
async fn test_high_throughput() {
    let app = create_test_app().await;

    // Create 1000 concurrent requests
    let mut handles = Vec::new();
    for i in 0..1000 {
        let app_clone = app.clone();
        handles.push(tokio::spawn(async move {
            app_clone
                .post("/api/positions")
                .json(&json!({"id": i}))
                .send()
                .await
        }));
    }

    let results = futures::future::join_all(handles).await;
    let successes = results.iter().filter(|r| r.is_ok()).count();

    assert!(successes > 950); // 95% success rate
}
```

Test Data Management

Test Fixtures

Create Reusable Test Data:

```typescript
export function createTestUser(): anchor.web3.Keypair {
  return anchor.web3.Keypair.generate();
}

export function createTestPosition(user: anchor.web3.PublicKey) {
  return {
    user: user,
    size: new anchor.BN(1000),
    leverage: 10,
    entryPrice: new anchor.BN(50000),
  };
}
```

Test Database Setup

Setup Test Database:

```rust
async fn setup_test_database() -> Arc<Database> {
    let database_url = "postgresql://postgres:postgres@localhost/godark_test";
    let pool = PgPool::connect(&database_url).await.unwrap();

    // Run migrations
    sqlx::migrate!("./migrations").run(&pool).await.unwrap();

    Arc::new(Database { pool: Arc::new(pool) })
}

#[tokio::test]
async fn test_with_clean_database() {
    let db = setup_test_database().await;
    // Test code
    // Database is cleaned after test
}
```

CI/CD Integration

GitHub Actions Example

```yaml
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Install Rust
        uses: actions-rs/toolchain@v1
        with:
          toolchain: stable

      - name: Install Solana
        run: |
          sh -c "$(curl -sSfL https://release.solana.com/stable/install)"

      - name: Install Anchor
        run: |
          cargo install --git https://github.com/coral-xyz/anchor avm --locked --force
          avm install latest
          avm use latest

      - name: Run Anchor Tests
        run: anchor test

      - name: Run Rust Tests
        run: cargo test
```

Coverage Goals and Metrics

Coverage Targets

- **Unit Tests**: >80% line coverage
- **Integration Tests**: >60% integration coverage
- **Critical Paths**: 100% coverage

Coverage Tools

Rust Coverage:

```bash
# Install cargo-tarpaulin
cargo install cargo-tarpaulin

# Generate coverage
cargo tarpaulin --out Html
```

TypeScript Coverage:

```bash
# Install nyc
npm install --save-dev nyc

# Run with coverage
nyc anchor test
```

Key Takeaways

1. **Test Pyramid**: More unit tests, fewer E2E tests
2. **Test Everything**: Functionality, errors, edge cases
3. **Mock External**: Mock oracles, external programs
4. **Security Tests**: Authorization, input validation, attacks

5. **Performance Tests**: Compute units, transaction size, throughput
6. **CI/CD**: Automate testing in pipeline
7. **Coverage**: Aim for >80% coverage

Next Steps

- Review **08-common-patterns-pitfalls.md** for testing patterns
- Check **09-security-best-practices.md** for security testing
- Practice writing tests for your component

Last Updated: November 2025

08. Common Patterns and Pitfalls in Solana/Anchor Development

Overview

This guide covers common patterns used in GoDark DEX development and pitfalls to avoid. Learning these patterns will help you write better, more secure Solana programs.

Common Patterns

Pattern 1: PDA Derivation and Management

Use Case: Creating deterministic addresses for user-specific or global state accounts.

Implementation:

```
use anchor_lang::prelude::*;

// Derive PDA
let (pda, bump) = Pubkey::find_program_address(
    &[
        b"position",              // Seed 1: account type
        user_pubkey.as_ref(),     // Seed 2: user identifier
    ],
    program_id,
);

// Verify PDA in instruction
#[account(
    seeds = [b"position", user.key().as_ref()],
    bump
)]
pub position: Account<'info, Position>,
```

GoDark Examples:

- Position accounts: `[b"position", user_pubkey]`
- Vault accounts: `[b"vault", user_pubkey, mint_pubkey]`
- Config accounts: `[b"config"]`

Best Practices:

- Use constants for seeds: `const SEED_POSITION: &[u8] = b"position";`
- Include bump in seeds for signing
- Document seed order in comments

Pattern 2: Authority Transfer Patterns

Use Case: Transferring control of accounts or programs.

Implementation:

```rust
// Transfer authority to PDA
pub fn transfer_authority(ctx: Context<TransferAuthority>) -> Result<()> {
    let new_authority = &ctx.accounts.new_authority;

    // Update authority
    ctx.accounts.account.authority = *new_authority.key;

    Ok(())
}

#[derive(Accounts)]
pub struct TransferAuthority<'info> {
    #[account(mut)]
    pub account: Account<'info, State>,
    pub current_authority: Signer<'info>,
    /// CHECK: New authority (can be PDA)
    pub new_authority: UncheckedAccount<'info>,
}
```

GoDark Usage:

- Program upgrade authority → Multisig PDA
- Vault authority → Program PDA
- Position authority → User (immutable)

Pattern 3: Account Initialization

Use Case: Creating new accounts with proper space and rent.

Implementation:

```rust
#[account(init, payer = user, space = 8 + Position::LEN)]
pub position: Account<'info, Position>,

#[account]
pub struct Position {
    pub user: Pubkey,
    pub size: u64,
    pub leverage: u8,
    // ... other fields
}

impl Position {
    pub const LEN: usize = 32 + 8 + 1; // user + size + leverage
}
```

Space Calculation:

```
// Discriminator: 8 bytes
// Data fields:
// - Pubkey: 32 bytes
// - u64: 8 bytes
// - u8: 1 byte
// Total: 8 + 32 + 8 + 1 = 49 bytes
```

GoDark Pattern:

- Always calculate space accurately
- Use constants for size: `pub const LEN: usize = ...`
- Include discriminator (8 bytes) in calculation

Pattern 4: CPI Patterns

Use Case: Calling other programs (SPL Token, other Solana programs).

Token Transfer CPI:

```
use anchor_spl::token::{self, Transfer, Token, TokenAccount};

pub fn transfer_tokens(ctx: Context<TransferTokens>, amount: u64) -> Result<()> {
    let cpi_accounts = Transfer {
        from: ctx.accounts.from.to_account_info(),
        to: ctx.accounts.to.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

    token::transfer(cpi_ctx, amount)?;
    Ok(())
}

#[derive(Accounts)]
pub struct TransferTokens<'info> {
    #[account(mut)]
    pub from: Account<'info, TokenAccount>,
    #[account(mut)]
    pub to: Account<'info, TokenAccount>,
    pub authority: Signer<'info>,
    pub token_program: Program<'info, Token>,
}
```

CPI with PDA Signing:

```
pub fn transfer_from_vault(ctx: Context<TransferFromVault>, amount: u64) -> Result<()> {
    let seeds = &[
        b"vault",
        ctx.accounts.user.key.as_ref(),
        &[ctx.bumps.vault],
    ];
    let signer = &[&seeds[..]];

    let cpi_accounts = Transfer {
        from: ctx.accounts.vault_token.to_account_info(),
        to: ctx.accounts.user_token.to_account_info(),
        authority: ctx.accounts.vault.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new_with_signer(cpi_program, cpi_accounts, signer);

    token::transfer(cpi_ctx, amount)?;
    Ok(())
}
```

GoDark Usage:

- Collateral vault → SPL Token transfers
- Position management → Lock/unlock collateral
- Settlement → Batch token operations

Pattern 5: State Migration Patterns

Use Case: Upgrading programs while preserving account data.

Implementation:

```rust
#[account]
pub struct Position {
    pub version: u32,         // Version field
    pub user: Pubkey,
    pub size: u64,
    // ... other fields
}

pub fn migrate_position(ctx: Context<MigratePosition>) -> Result<()> {
    let position = &mut ctx.accounts.position;

    // Check version
    require!(
        position.version < CURRENT_VERSION,
        ErrorCode::AlreadyMigrated
    );

    // Migrate data
    if position.version == 1 {
        // Transform from v1 to v2
        position.new_field = calculate_new_field(position);
    }

    position.version = CURRENT_VERSION;
    Ok(())
}
```

GoDark Usage:

- Program upgrade system handles migrations
- Account versioning for compatibility
- Data transformation between versions

Pattern 6: Error Handling Strategies

Use Case: Providing clear, actionable error messages.

Implementation:

```rust
use anchor_lang::error_code;

#[error_code]
pub enum ErrorCode {
    #[msg("Insufficient collateral. Required: {required}, Available: {available}")]
    InsufficientCollateral {
        required: u64,
        available: u64,
    },
    #[msg("Invalid leverage tier: {leverage}. Must be 1-1000")]
    InvalidLeverage { leverage: u8 },
    #[msg("Position not found")]
    PositionNotFound,
}

// Usage
require!(
    collateral >= required,
    ErrorCode::InsufficientCollateral {
        required,
        available: collateral
    }
);
```

Best Practices:

- Use descriptive error messages
- Include relevant context (values, constraints)
- Use error codes for programmatic handling
- Document error conditions

Pattern 7: Event Emission Patterns

Use Case: Notifying off-chain systems of on-chain events.

Implementation:

```
use anchor_lang::prelude::*;

#[event]
pub struct PositionOpened {
    pub user: Pubkey,
    pub position_id: Pubkey,
    pub size: u64,
    pub leverage: u8,
    pub entry_price: u64,
    pub timestamp: i64,
}

pub fn open_position(ctx: Context<OpenPosition>, size: u64, leverage: u8) -> Result<()> {
    // ... open position logic ...

    emit!(PositionOpened {
        user: ctx.accounts.user.key(),
        position_id: ctx.accounts.position.key(),
        size,
        leverage,
        entry_price: mark_price,
        timestamp: Clock::get()?.unix_timestamp,
    });

    Ok(())
}
```

GoDark Usage:

- Position events (open, close, modify)
- Liquidation events
- Funding rate payments
- Upgrade events

Common Pitfalls

Pitfall 1: Account Ownership Mistakes

Problem: Trying to modify account owned by wrong program.

Example (Wrong):

```
pub fn modify_account(ctx: Context<ModifyAccount>) -> Result<()> {
    // This will fail if account is owned by different program
    ctx.accounts.account.data = new_data;
    Ok(())
}
```

Solution:

```rust
pub fn modify_account(ctx: Context<ModifyAccount>) -> Result<()> {
    // Verify ownership
    require!(
        ctx.accounts.account.owner == ctx.program_id,
        ErrorCode::InvalidAccountOwner
    );

    ctx.accounts.account.data = new_data;
    Ok(())
}

// Or use Anchor's account constraint
#[account(
    owner = program_id @ ErrorCode::InvalidAccountOwner
)]
pub account: Account<'info, State>,
```

Pitfall 2: PDA Seed Mismatches

Problem: PDA derivation fails due to incorrect seeds.

Example (Wrong):

```rust
// In instruction handler
let (pda, bump) = Pubkey::find_program_address(
    &[b"vault", user_pubkey.as_ref()],
    program_id,
);

// But in account constraint
#[account(
    seeds = [b"vault", mint_pubkey.as_ref()], // Different seeds!
    bump
)]
pub vault: Account<'info, Vault>,
```

Solution:

```rust
// Use constants
const SEED_VAULT: &[u8] = b"vault";

// Derive PDA
let (pda, bump) = Pubkey::find_program_address(
    &[SEED_VAULT, user_pubkey.as_ref()],
    program_id,
);

// Use same seeds in constraint
#[account(
    seeds = [SEED_VAULT, user.key().as_ref()],
    bump
)]
pub vault: Account<'info, Vault>,
```

Pitfall 3: Rent-Exempt Account Requirements

Problem: Account not rent-exempt, gets closed by runtime.

Example (Wrong):

```rust
#[account(init, payer = user, space = 8 + 32)]
pub account: Account<'info, State>,
// Missing rent-exempt check
```

Solution:

```rust
#[account(
    init,
    payer = user,
    space = 8 + State::LEN
)]
pub account: Account<'info, State>,

// Anchor automatically ensures rent-exempt for init accounts
// For existing accounts, verify:
require!(
    account.to_account_info().lamports() >= Rent::get()?.minimum_balance(space),
    ErrorCode::InsufficientBalance
);
```

Pitfall 4: Compute Unit Exhaustion

Problem: Transaction fails due to compute limit exceeded.

Example (Wrong):

```rust
pub fn process_many(ctx: Context<ProcessMany>, items: Vec<Item>) -> Result<()> {
    for item in items.iter() {
        // Expensive operation in loop
        process_item(item)?; // May exceed compute budget
    }
    Ok(())
}
```

Solution:

```rust
// Option 1: Set higher compute budget
use solana_program::compute_budget::ComputeBudgetInstruction;

let compute_budget = ComputeBudgetInstruction::set_compute_unit_limit(400_000);
instructions.push(compute_budget);

// Option 2: Optimize algorithm
pub fn process_many(ctx: Context<ProcessMany>, items: Vec<Item>) -> Result<()> {
    // Batch process or use more efficient algorithm
    let results: Vec<_> = items.iter()
        .map(|item| process_item_optimized(item))
        .collect();
    Ok(())
}
```

Pitfall 5: Transaction Size Limits

Problem: Transaction exceeds 1,232 byte limit.

Example (Wrong):

```rust
pub fn batch_settle(ctx: Context<BatchSettle>, trades: Vec<Trade>) -> Result<()> {
    // Too many trades in one transaction
    for trade in trades.iter() {
        settle_trade(ctx, trade)?;
    }
    Ok(())
}
```

Solution:

```
// Split into multiple transactions
pub fn batch_settle(ctx: Context<BatchSettle>, trades: Vec<Trade>) -> Result<()> {
    const MAX_TRADES_PER_TX: usize = 20; // Adjust based on trade size

    for chunk in trades.chunks(MAX_TRADES_PER_TX) {
        settle_chunk(ctx, chunk)?;
    }
    Ok(())
}
```

Pitfall 6: Reentrancy Concerns

Problem: While Solana programs are single-threaded, CPI reentrancy can cause issues.

Example (Wrong):

```
pub fn withdraw(ctx: Context<Withdraw>, amount: u64) -> Result<()> {
    // Update balance first
    ctx.accounts.vault.balance -= amount;

    // Then transfer (CPI could call back)
    transfer_tokens(ctx, amount)?; // Potential reentrancy
    Ok(())
}
```

Solution:

```
// Use checks-effects-interactions pattern
pub fn withdraw(ctx: Context<Withdraw>, amount: u64) -> Result<()> {
    // 1. Checks
    require!(
        ctx.accounts.vault.balance >= amount,
        ErrorCode::InsufficientBalance
    );

    // 2. Effects (update state first)
    ctx.accounts.vault.balance -= amount;

    // 3. Interactions (external calls last)
    transfer_tokens(ctx, amount)?;

    Ok(())
}
```

Pitfall 7: Integer Overflow/Underflow

Problem: Arithmetic operations overflow without checks.

Example (Wrong):

```
pub fn add_collateral(ctx: Context<AddCollateral>, amount: u64) -> Result<()> {
    // Potential overflow
    ctx.accounts.position.collateral += amount;
    Ok(())
}
```

Solution:

```
pub fn add_collateral(ctx: Context<AddCollateral>, amount: u64) -> Result<()> {
    // Use checked arithmetic
    ctx.accounts.position.collateral = ctx.accounts.position.collateral
        .checked_add(amount)
        .ok_or(ErrorCode::Overflow)?;
    Ok(())
}

// Or use Anchor's checked math
use anchor_lang::solana_program::program_error::ProgramError;

let new_balance = ctx.accounts.position.collateral
    .checked_add(amount)
    .ok_or(ProgramError::ArithmeticOverflow)?;
```

Pitfall 8: Missing Authority Checks

Problem: Anyone can call restricted instruction.

Example (Wrong):

```
pub fn close_position(ctx: Context<ClosePosition>) -> Result<()> {
    // Missing authority check!
    ctx.accounts.position.close()?;
    Ok(())
}
```

Solution:

```
pub fn close_position(ctx: Context<ClosePosition>) -> Result<()> {
    // Check authority
    require!(
        ctx.accounts.authority.key() == &ctx.accounts.position.user,
        ErrorCode::Unauthorized
    );

    ctx.accounts.position.close()?;
    Ok(())
}

// Or use Anchor's Signer constraint
#[derive(Accounts)]
pub struct ClosePosition<'info> {
    #[account(
        mut,
        close = authority, // Closes account and returns rent
        has_one = user @ ErrorCode::Unauthorized
    )]
    pub position: Account<'info, Position>,
    #[account(mut)]
    pub authority: Signer<'info>,
}
```

Pitfall 9: Incorrect Account Ordering

Problem: Accounts not ordered correctly (writable first).

Example (Wrong):

```
#[derive(Accounts)]
pub struct Example<'info> {
    pub read_only: Account<'info, State>,      // Read-only first
    #[account(mut)]
    pub writable: Account<'info, State>,       // Writable second
}
```

Solution:

```
#[derive(Accounts)]
pub struct Example<'info> {
    #[account(mut)]                              // Writable first
    pub writable: Account<'info, State>,
    pub read_only: Account<'info, State>,        // Read-only second
}
```

Best Practice: Order accounts as:

1. Writable accounts (mut)
2. Read-only accounts
3. Signers
4. Programs

Pitfall 10: Stale Account Data

Problem: Using account data that may have changed.

Example (Wrong):

```
pub fn process(ctx: Context<Process>) -> Result<()> {
    let balance = ctx.accounts.account.balance; // Read balance

    // ... do other operations ...

    // Balance may have changed!
    ctx.accounts.account.balance = balance + amount;
    Ok(())
}
```

Solution:

```
pub fn process(ctx: Context<Process>) -> Result<()> {
    // Read and use immediately
    let current_balance = ctx.accounts.account.balance;
    ctx.accounts.account.balance = current_balance
        .checked_add(amount)
        .ok_or(ErrorCode::Overflow)?;
    Ok(())
}
```

GoDark-Specific Patterns

Pattern: Batch Settlement

Implementation:

```rust
pub fn batch_settle(ctx: Context<BatchSettle>, merkle_root: [u8; 32]) -> Result<()> {
    // Verify Merkle root
    require!(
        calculate_merkle_root(&ctx.accounts.trades) == merkle_root,
        ErrorCode::InvalidMerkleRoot
    );

    // Process net positions
    for trade in ctx.accounts.trades.iter() {
        update_position(ctx, trade)?;
    }

    Ok(())
}
```

Pattern: Margin Calculation

Implementation:

```rust
pub fn calculate_margin_ratio(
    collateral: u64,
    unrealized_pnl: i64,
    maintenance_margin: u64,
) -> Result<u64> {
    // Use fixed-point arithmetic
    let total_equity = if unrealized_pnl >= 0 {
        collateral + unrealized_pnl as u64
    } else {
        collateral.saturating_sub(unrealized_pnl.unsigned_abs())
    };

    // Calculate ratio (multiply by 100 for percentage)
    let ratio = (total_equity * 100)
        .checked_div(maintenance_margin)
        .ok_or(ErrorCode::DivisionByZero)?;

    Ok(ratio)
}
```

Pattern: Oracle Price Validation

Implementation:

```rust
pub fn validate_price(price_data: &PriceData) -> Result<()> {
    // Check staleness
    let clock = Clock::get()?;
    let max_age = 60; // 60 seconds

    require!(
        clock.unix_timestamp - price_data.timestamp < max_age,
        ErrorCode::StalePrice
    );

    // Check confidence
    let confidence_threshold = price_data.price / 100; // 1% threshold

    require!(
        price_data.confidence < confidence_threshold,
        ErrorCode::LowConfidence
    );

    Ok(())
}
```

Key Takeaways

Patterns to Use

1. **PDAs**: Deterministic addresses without keys
2. **CPI**: Call other programs for composability
3. **Account Constraints**: Validate accounts in Anchor
4. **Error Codes**: Clear, descriptive errors
5. **Events**: Emit for off-chain tracking
6. **Versioning**: Support migrations

Pitfalls to Avoid

1. **Ownership Mistakes**: Always verify account ownership
2. **Seed Mismatches**: Use constants for seeds
3. **Rent Issues**: Ensure rent-exempt accounts
4. **Compute Limits**: Optimize and set budget
5. **Transaction Size**: Batch operations
6. **Overflow**: Use checked arithmetic
7. **Authority**: Always check authorization
8. **Account Ordering**: Writable accounts first

Next Steps

- Review **09-security-best-practices.md** for security patterns
- Practice implementing these patterns in your component
- Study existing GoDark components for real examples

Last Updated: November 2025

09. Security Best Practices for GoDark DEX

Overview

Security is paramount for financial applications handling user funds. This guide covers security best practices, attack vectors, and defensive patterns for GoDark DEX development.

Solana Security Model

Authority and Ownership

Key Concepts:

- **Owner**: Program that controls an account
- **Authority**: Entity that can modify an account
- **Signer**: Account that must sign a transaction

Security Principle:

- Only the owner program can modify account data
- Authority checks must be explicit
- Signers prove identity

Account Ownership Validation

Always Verify Ownership:

```rust
pub fn modify_account(ctx: Context<ModifyAccount>) -> Result<()> {
    // Verify account is owned by this program
    require!(
        ctx.accounts.account.owner == ctx.program_id,
        ErrorCode::InvalidAccountOwner
    );

    // Verify authority
    require!(
        ctx.accounts.authority.key() == &ctx.accounts.account.authority,
        ErrorCode::Unauthorized
    );

    // Now safe to modify
    ctx.accounts.account.data = new_data;
    Ok(())
}
```

Input Validation Patterns

Validate All Inputs

Size Validation:

```rust
pub fn open_position(ctx: Context<OpenPosition>, size: u64, leverage: u8) -> Result<()> {
    // Validate size
    require!(size > 0, ErrorCode::InvalidSize);
    require!(size <= MAX_POSITION_SIZE, ErrorCode::SizeTooLarge);

    // Validate leverage
    require!(leverage >= 1, ErrorCode::InvalidLeverage);
    require!(leverage <= 1000, ErrorCode::LeverageTooHigh);

    // Validate leverage tier
    require!(
        is_valid_leverage_tier(leverage),
        ErrorCode::InvalidLeverageTier
    );

    // ... rest of logic
    Ok(())
}
```

Type Validation

Validate Account Types:

```rust
#[account(
    constraint = token_account.owner == token::ID @ ErrorCode::InvalidTokenAccount,
    constraint = token_account.mint == expected_mint @ ErrorCode::InvalidMint,
)]
pub token_account: Account<'info, TokenAccount>,
```

Range Validation

Validate Numeric Ranges:

```rust
pub fn set_funding_rate(ctx: Context<SetFundingRate>, rate: i64) -> Result<()> {
    const MIN_RATE: i64 = -7500; // -0.75% (in basis points)
    const MAX_RATE: i64 = 7500;  // +0.75%

    require!(
        rate >= MIN_RATE && rate <= MAX_RATE,
        ErrorCode::FundingRateOutOfRange
    );

    ctx.accounts.config.funding_rate = rate;
    Ok(())
}
```

Authority Checks

Who Can Call What?

Pattern: Explicit Authority Checks

```rust
pub fn close_position(ctx: Context<ClosePosition>) -> Result<()> {
    // Check 1: Must be position owner
    require!(
        ctx.accounts.authority.key() == &ctx.accounts.position.user,
        ErrorCode::Unauthorized
    );

    // Check 2: Position must be open
    require!(
        ctx.accounts.position.status == PositionStatus::Open,
        ErrorCode::PositionNotOpen
    );

    // Safe to close
    ctx.accounts.position.close()?;
    Ok(())
}
```

PDA Authority Pattern

Verify PDA Signer:

```rust
pub fn transfer_from_vault(ctx: Context<TransferFromVault>, amount: u64) -> Result<()> {
    // Derive expected PDA
    let (expected_vault, bump) = Pubkey::find_program_address(
        &[b"vault", ctx.accounts.user.key().as_ref()],
        ctx.program_id,
    );

    // Verify PDA matches
    require!(
        ctx.accounts.vault.key() == &expected_vault,
        ErrorCode::InvalidVault
    );

    // Sign with PDA
    let seeds = &[
        b"vault",
        ctx.accounts.user.key().as_ref(),
        &[bump],
    ];
    let signer = &[&seeds[..]];

    // Execute CPI with PDA signature
    transfer_tokens_cpi(ctx, amount, signer)?;
    Ok(())
}
```

Multisig Authority Pattern

Verify Multisig Approval:

```
pub fn execute_upgrade(ctx: Context<ExecuteUpgrade>) -> Result<()> {
    let proposal = &ctx.accounts.proposal;

    // Check approval threshold met
    require!(
        proposal.approvals.len() >= proposal.threshold as usize,
        ErrorCode::InsufficientApprovals
    );

    // Verify all approvers are multisig members
    let multisig = &ctx.accounts.multisig;
    for approver in proposal.approvals.iter() {
        require!(
            multisig.members.contains(approver),
            ErrorCode::InvalidApprover
        );
    }

    // Execute upgrade
    execute_upgrade_cpi(ctx)?;
    Ok(())
}
```

Reentrancy Prevention

Checks-Effects-Interactions Pattern

Order of Operations:

```
pub fn withdraw(ctx: Context<Withdraw>, amount: u64) -> Result<()> {
    // 1. CHECKS: Validate inputs and state
    require!(
        ctx.accounts.vault.balance >= amount,
        ErrorCode::InsufficientBalance
    );
    require!(
        ctx.accounts.authority.key() == &ctx.accounts.vault.authority,
        ErrorCode::Unauthorized
    );

    // 2. EFFECTS: Update state FIRST
    ctx.accounts.vault.balance -= amount;
    ctx.accounts.vault.last_withdrawal = Clock::get()?.unix_timestamp;

    // 3. INTERACTIONS: External calls LAST
    transfer_tokens_cpi(ctx, amount)?;

    Ok(())
}
```

State Locks

Prevent Concurrent Modifications:

```rust
#[account]
pub struct Position {
    pub locked: bool,        // Lock flag
    pub user: Pubkey,
    // ... other fields
}

pub fn modify_position(ctx: Context<ModifyPosition>) -> Result<()> {
    // Check lock
    require!(
        !ctx.accounts.position.locked,
        ErrorCode::PositionLocked
    );

    // Set lock
    ctx.accounts.position.locked = true;

    // Perform modification
    ctx.accounts.position.size = new_size;

    // Release lock
    ctx.accounts.position.locked = false;

    Ok(())
}
```

Integer Arithmetic Safety

Use Checked Math

Always Use Checked Operations:

```rust
// WRONG: Potential overflow
pub fn add_collateral(ctx: Context<AddCollateral>, amount: u64) -> Result<()> {
    ctx.accounts.position.collateral += amount; // May overflow!
    Ok(())
}

// CORRECT: Checked arithmetic
pub fn add_collateral(ctx: Context<AddCollateral>, amount: u64) -> Result<()> {
    ctx.accounts.position.collateral = ctx.accounts.position.collateral
        .checked_add(amount)
        .ok_or(ErrorCode::Overflow)?;
    Ok(())
}
```

Fixed-Point Arithmetic

For Financial Calculations:

```rust
// Use fixed-point math for precision
pub fn calculate_funding_payment(
    position_size: u64,
    mark_price: u64,
    funding_rate: i64, // In basis points (e.g., 100 = 0.01%)
) -> Result<i64> {
    // Multiply first to maintain precision
    let payment = (position_size as u128)
        .checked_mul(mark_price as u128)
        .ok_or(ErrorCode::Overflow)?
        .checked_mul(funding_rate.abs() as u128)
        .ok_or(ErrorCode::Overflow)?
        .checked_div(1_000_000) // Divide by 100 * 10000 (basis points * price precision)
        .ok_or(ErrorCode::DivisionByZero)?;

    Ok(if funding_rate < 0 {
        -(payment as i64)
    } else {
        payment as i64
    })
}
```

Account Validation

Ownership Validation

Verify Account Ownership:

```rust
#[account(
    owner = program_id @ ErrorCode::InvalidAccountOwner
)]
pub account: Account<'info, State>,
```

Data Format Validation

Verify Account Data Structure:

```rust
pub fn validate_account(account: &AccountInfo) -> Result<()> {
    // Check account is initialized
    require!(
        account.data_len() >= 8, // At least discriminator
        ErrorCode::AccountNotInitialized
    );

    // Check discriminator matches
    let discriminator = &account.data.borrow()[..8];
    require!(
        discriminator == State::DISCRIMINATOR,
        ErrorCode::InvalidAccountDiscriminator
    );

    Ok(())
}
```

State Validation

Verify Account State:

```
pub fn close_position(ctx: Context<ClosePosition>) -> Result<()> {
    let position = &ctx.accounts.position;

    // Check position is open
    require!(
        position.status == PositionStatus::Open,
        ErrorCode::PositionNotOpen
    );

    // Check no pending operations
    require!(
        !position.locked,
        ErrorCode::PositionLocked
    );

    // Safe to close
    position.close()?;
    Ok(())
}
```

Oracle Manipulation Prevention

Price Validation

Validate Oracle Prices:

```
pub fn validate_price(price_data: &PriceData) -> Result<()> {
    let clock = Clock::get()?;

    // Check staleness (max 60 seconds old)
    let max_age = 60;
    require!(
        clock.unix_timestamp - price_data.timestamp < max_age,
        ErrorCode::StalePrice
    );

    // Check confidence (must be < 1% of price)
    let confidence_threshold = price_data.price
        .checked_div(100)
        .ok_or(ErrorCode::DivisionByZero)?;

    require!(
        price_data.confidence < confidence_threshold,
        ErrorCode::LowConfidence
    );

    // Check price is reasonable (not zero, not extreme)
    require!(price_data.price > 0, ErrorCode::InvalidPrice);
    require!(
        price_data.price < MAX_REASONABLE_PRICE,
        ErrorCode::PriceTooHigh
    );

    Ok(())
}
```

Multi-Oracle Consensus

Use Multiple Oracles:

```rust
pub fn get_consensus_price(
    pyth_price: &PriceData,
    switchboard_price: &PriceData,
) -> Result<u64> {
    // Validate both prices
    validate_price(pyth_price)?;
    validate_price(switchboard_price)?;

    // Calculate median
    let prices = vec![pyth_price.price, switchboard_price.price];
    prices.sort();
    let median = prices[prices.len() / 2];

    // Check prices are within acceptable range (5%)
    let price_diff = if pyth_price.price > switchboard_price.price {
        pyth_price.price - switchboard_price.price
    } else {
        switchboard_price.price - pyth_price.price
    };

    let max_diff = median.checked_div(20).ok_or(ErrorCode::DivisionByZero)?; // 5%
    require!(
        price_diff < max_diff,
        ErrorCode::PriceDeviationTooHigh
    );

    Ok(median)
}
```

Economic Attack Vectors

Liquidation Manipulation

Attack: Manipulate mark price to trigger unfair liquidations.

Defense:

```rust
pub fn liquidate_position(ctx: Context<LiquidatePosition>) -> Result<()> {
    // Use time-weighted average price (TWAP) for liquidation
    let twap_price = calculate_twap(&ctx.accounts.price_history)?;

    // Verify mark price is close to TWAP (within 2%)
    let price_diff = if mark_price > twap_price {
        mark_price - twap_price
    } else {
        twap_price - mark_price
    };

    let max_diff = twap_price.checked_div(50).ok_or(ErrorCode::DivisionByZero)?; // 2%
    require!(
        price_diff < max_diff,
        ErrorCode::PriceManipulationDetected
    );

    // Proceed with liquidation
    execute_liquidation(ctx, twap_price)?;
    Ok(())
}
```

Funding Rate Attacks

Attack: Manipulate funding rate to extract value.

Defense:

```
pub fn calculate_funding_rate(
    mark_price: u64,
    index_price: u64,
) -> Result<i64> {
    // Clamp funding rate to prevent extreme values
    const MIN_RATE: i64 = -7500; // -0.75%
    const MAX_RATE: i64 = 7500;  // +0.75%

    let premium_index = calculate_premium_index(mark_price, index_price)?;
    let interest_rate = get_interest_rate()?;

    let funding_rate = premium_index + interest_rate;

    // Clamp to prevent manipulation
    let clamped_rate = funding_rate.max(MIN_RATE).min(MAX_RATE);

    Ok(clamped_rate)
}
```

Front-Running Prevention

Attack: Front-run large orders.

Defense:

- Dark pool hides orders until execution
- Batch settlement prevents front-running
- Merkle tree verification ensures trade integrity

Access Control Patterns

Role-Based Access

Implement Roles:

```
#[account]
pub struct Config {
    pub admin: Pubkey,
    pub operators: Vec<Pubkey>,
    pub emergency_pause_authority: Pubkey,
}

pub fn admin_only_operation(ctx: Context<AdminOperation>) -> Result<()> {
    require!(
        ctx.accounts.authority.key() == &ctx.accounts.config.admin,
        ErrorCode::AdminOnly
    );
    // ... operation
    Ok(())
}
```

Time-Based Access

Implement Timelocks:

```rust
pub fn execute_upgrade(ctx: Context<ExecuteUpgrade>) -> Result<()> {
    let clock = Clock::get()?;
    let proposal = &ctx.accounts.proposal;

    // Check timelock expired
    require!(
        clock.unix_timestamp >= proposal.timelock_until,
        ErrorCode::TimelockNotExpired
    );

    // Execute upgrade
    execute_upgrade_cpi(ctx)?;
    Ok(())
}
```

Secure Key Management

Never Hardcode Keys

Wrong:

```rust
const ADMIN_KEY: &str = "Admin111111111111111111111111111111111"; // DON'T DO THIS!
```

Correct:

```rust
// Use environment variables or on-chain config
pub fn get_admin(ctx: Context<GetAdmin>) -> Result<Pubkey> {
    Ok(ctx.accounts.config.admin)
}
```

Keypair Security

For Backend Services:

- Store keypairs encrypted
- Use environment variables for keys
- Rotate keys regularly
- Never commit keys to git

Example:

```rust
use solana_sdk::signature::read_keypair_file;

// Load from environment variable path
let keypair_path = std::env::var("KEYPAIR_PATH")
    .expect("KEYPAIR_PATH not set");
let keypair = read_keypair_file(&keypair_path)
    .map_err(|e| anyhow::anyhow!("Failed to read keypair: {}", e))?;
```

Audit Preparation Checklist

Code Review Checklist

- [ ] All inputs validated
- [ ] Authority checks present
- [ ] Integer overflow protection
- [ ] Account ownership verified

- [ ] Error handling comprehensive
- [ ] Oracle manipulation prevention
- [ ] Economic attack vectors considered
- [ ] Key management secure
- [ ] Documentation complete

Security Audit Points

Critical Areas:

1. **Liquidation Engine**
   - Manipulation resistance
   - Fair liquidation pricing
   - Bad debt handling
2. **Funding Rate**
   - Oracle manipulation prevention
   - Rate clamping
   - Payment distribution security
3. **Settlement**
   - Merkle tree verification
   - Batch integrity
   - Replay attack prevention
4. **Vault Management**
   - Authorization checks
   - Withdrawal limits
   - Balance reconciliation
5. **Position Management**
   - Margin calculations
   - PnL accuracy
   - State consistency

Security Review Process

Pre-Deployment Checklist

1. **Code Review**
   - Peer review completed
   - Security review completed
   - All feedback addressed
2. **Testing**
   - Unit tests passing
   - Integration tests passing
   - Security tests passing
   - Edge cases tested
3. **Documentation**
   - Security assumptions documented
   - Attack vectors documented
   - Mitigation strategies documented

4. **Audit**
    - Internal audit completed
    - External audit (if applicable)
    - Findings addressed

Post-Deployment Monitoring

- Monitor for unusual activity
- Track security metrics
- Review logs regularly
- Update security measures

Key Takeaways

1. **Validate Everything**: All inputs, accounts, states
2. **Check Authority**: Always verify who can do what
3. **Use Checked Math**: Prevent overflow/underflow
4. **Prevent Manipulation**: Oracle validation, rate clamping
5. **Secure Keys**: Never hardcode, use environment variables
6. **Audit Ready**: Document security assumptions
7. **Monitor**: Watch for attacks post-deployment

Next Steps

- Review **08-common-patterns-pitfalls.md** for implementation patterns
- Study security audit reports from other DeFi protocols
- Practice identifying attack vectors in your component

Last Updated: November 2025

10. Architecture Deep Dive: GoDark DEX

Overview

This guide provides a detailed look at the architectural patterns, design decisions, and implementation strategies used in GoDark DEX.

Hybrid Architecture

Off-Chain Matching, On-Chain Settlement

Architecture Pattern:

```
┌─────────────────────────────────────────────────────────┐
│                  Off-Chain Layer (Fast)                  │
│  ┌──────────────────────────────────────────────────┐   │
│  │ Matching Engine                                    │   │
│  │ - Order book management                            │   │
│  │ - Price-time priority matching                     │   │
│  │ - Trade execution                                  │   │
│  │ - 1000+ trades/second                              │   │
│  └──────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────┘
                     │ Trades
                     ▼
┌─────────────────────────────────────────────────────────┐
│               Settlement Relayer (Bridge)                │
│  ┌──────────────────────────────────────────────────┐   │
│  │ Batch Accumulation                                 │   │
│  │ - 1-second windows                                 │   │
│  │ - Net position calculation                         │   │
│  │ - Merkle tree generation                           │   │
│  └──────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────┘
                     │ Transactions
                     ▼
┌─────────────────────────────────────────────────────────┐
│                 On-Chain Layer (Secure)                  │
│  ┌──────────────────────────────────────────────────┐   │
│  │ Solana Programs                                    │   │
│  │ - Position Management                              │   │
│  │ - Collateral Vault                                 │   │
│  │ - Liquidation Engine                               │   │
│  │ - Funding Rate                                     │   │
│  └──────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────┘
```

Benefits:

- **Speed**: Off-chain matching enables high throughput
- **Security**: On-chain settlement ensures trustlessness
- **Cost**: Batch settlement reduces transaction fees
- **Privacy**: Dark pool hides orders until execution

Dark Pool Implementation Patterns

Order Hiding

Pattern: Orders invisible until execution

Implementation:

```
// Off-chain order book (not visible to public)
pub struct OrderBook {
    orders: BTreeMap<Price, Vec<Order>>, // Price-time priority
    // Orders not exposed via API until matched
}

// Only matched trades are revealed
pub struct Trade {
    pub price: u64,
    pub size: u64,
    pub timestamp: i64,
    // No order IDs exposed
}
```

Benefits:

- Prevents front-running
- Reduces market impact
- Protects large orders

Price-Time Priority Matching

Algorithm:

```rust
pub fn match_orders(order_book: &mut OrderBook, new_order: Order) -> Vec<Trade> {
    let mut trades = Vec::new();

    // Find matching orders (opposite side, best price)
    while let Some(matching_order) = order_book.find_best_match(&new_order) {
        let trade = execute_trade(&new_order, &matching_order);
        trades.push(trade);

        // Update order sizes
        new_order.size -= trade.size;
        matching_order.size -= trade.size;

        // Remove filled orders
        if matching_order.size == 0 {
            order_book.remove_order(matching_order);
        }
        if new_order.size == 0 {
            break;
        }
    }

    // Add remaining order to book
    if new_order.size > 0 {
        order_book.add_order(new_order);
    }

    trades
}
```

Batch Settlement Design

Merkle Trees

Purpose: Cryptographic verification of batch integrity

Implementation:

```rust
use sha2::{Sha256, Digest};

pub struct MerkleTree {
    leaves: Vec<[u8; 32]>, // Trade hashes
    root: [u8; 32],
}

impl MerkleTree {
    pub fn build(trades: &[Trade]) -> Self {
        let leaves: Vec<[u8; 32]> = trades.iter()
            .map(|trade| Self::hash_trade(trade))
            .collect();

        let root = Self::compute_root(&leaves);

        Self { leaves, root }
    }

    fn hash_trade(trade: &Trade) -> [u8; 32] {
        let mut hasher = Sha256::new();
        hasher.update(&trade.user.to_bytes());
        hasher.update(&trade.size.to_le_bytes());
        hasher.update(&trade.price.to_le_bytes());
        hasher.finalize().into()
    }

    fn compute_root(leaves: &[[u8; 32]]) -> [u8; 32] {
        // Build Merkle tree bottom-up
        let mut level = leaves.to_vec();

        while level.len() > 1 {
            let mut next_level = Vec::new();
            for chunk in level.chunks(2) {
                if chunk.len() == 2 {
                    let hash = Self::hash_pair(&chunk[0], &chunk[1]);
                    next_level.push(hash);
                } else {
                    next_level.push(chunk[0]);
                }
            }
            level = next_level;
        }

        level[0]
    }
}
```

On-Chain Verification:

```rust
pub fn verify_settlement(
    ctx: Context<VerifySettlement>,
    merkle_root: [u8; 32],
    trade_proof: Vec<[u8; 32]>, // Merkle proof path
) -> Result<()> {
    // Verify trade is in batch
    let trade_hash = hash_trade(&ctx.accounts.trade);
    let computed_root = compute_root_from_proof(trade_hash, &trade_proof);

    require!(
        computed_root == merkle_root,
        ErrorCode::InvalidMerkleProof
    );

    // Process settlement
    settle_trade(ctx, &ctx.accounts.trade)?;
    Ok(())
}
```

Netting

Purpose: Reduce on-chain operations

Algorithm:

```rust
pub struct NetPosition {
    pub user: Pubkey,
    pub market: Pubkey,
    pub net_size: i64, // Positive = long, negative = short
    pub avg_entry_price: u64,
}

pub fn calculate_net_positions(trades: &[Trade]) -> Vec<NetPosition> {
    use std::collections::HashMap;

    let mut positions: HashMap<(Pubkey, Pubkey), NetPosition> = HashMap::new();

    for trade in trades.iter() {
        let key = (trade.user, trade.market);
        let position = positions.entry(key).or_insert_with(|| NetPosition {
            user: trade.user,
            market: trade.market,
            net_size: 0,
            avg_entry_price: 0,
        });

        // Update net size
        if trade.side == Side::Long {
            position.net_size += trade.size as i64;
        } else {
            position.net_size -= trade.size as i64;
        }

        // Update average entry price (weighted average)
        let total_notional = (position.net_size.abs() as u64)
            .checked_mul(position.avg_entry_price)
            .unwrap_or(0);
        let trade_notional = trade.size.checked_mul(trade.price).unwrap_or(0);

        let new_notional = if trade.side == Side::Long {
            total_notional.checked_add(trade_notional).unwrap_or(0)
        } else {
            total_notional.checked_sub(trade_notional).unwrap_or(0)
        };

        if position.net_size != 0 {
            position.avg_entry_price = new_notional
                .checked_div(position.net_size.abs() as u64)
                .unwrap_or(0);
        }
    }

    positions.into_values().collect()
}
```

Benefits:

- Reduces transaction size
- Lowers compute units
- Decreases fees

State Management Strategies

On-Chain vs Off-Chain State

On-Chain State:

- Position accounts
- Collateral balances
- Configuration
- Critical security parameters

Off-Chain State:

- Order book
- Trade history
- User preferences

- Analytics data

Decision Criteria:

- **On-Chain**: Security-critical, needs trustlessness
- **Off-Chain**: Performance-critical, can be rebuilt

State Synchronization

Pattern: Event-driven synchronization

```rust
// On-chain event
#[event]
pub struct PositionUpdated {
    pub user: Pubkey,
    pub position_id: Pubkey,
    pub size: u64,
    pub collateral: u64,
}

// Off-chain listener
pub async fn sync_position(event: PositionUpdated) {
    // Update off-chain database
    db.update_position(event.position_id, event.size, event.collateral).await;

    // Update cache
    cache.set_position(event.position_id, event.size).await;

    // Notify WebSocket clients
    websocket.broadcast_position_update(event).await;
}
```

Event-Driven Architecture

WebSocket Notifications

Implementation:

```rust
use tokio_tungstenite::{connect_async, tungstenite::Message};

pub struct WebSocketServer {
    clients: Arc<Mutex<HashMap<Uuid, Sender>>>,
}

impl WebSocketServer {
    pub async fn broadcast_position_update(&self, update: PositionUpdate) {
        let message = serde_json::to_string(&update).unwrap();
        let clients = self.clients.lock().await;

        for (_, sender) in clients.iter() {
            let _ = sender.send(Message::Text(message.clone())).await;
        }
    }

    pub async fn subscribe_to_market(&self, client_id: Uuid, market: String) {
        // Add client to market subscription list
    }
}
```

Event Types

Position Events:

- Position opened
- Position modified
- Position closed
- Position liquidated

Market Events:

- Price updates
- Funding rate changes
- Liquidation alerts

System Events:

- Upgrades
- Pauses
- Parameter changes

Database Design Patterns

PostgreSQL Schema

Positions Table:

```
CREATE TABLE positions (
    id UUID PRIMARY KEY,
    user_pubkey TEXT NOT NULL,
    position_pda TEXT NOT NULL UNIQUE,
    market TEXT NOT NULL,
    size BIGINT NOT NULL,
    entry_price BIGINT NOT NULL,
    leverage SMALLINT NOT NULL,
    collateral BIGINT NOT NULL,
    status TEXT NOT NULL,
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP NOT NULL,
    INDEX idx_user (user_pubkey),
    INDEX idx_market (market),
    INDEX idx_status (status)
);
```

Trades Table:

```
CREATE TABLE trades (
    id UUID PRIMARY KEY,
    user_pubkey TEXT NOT NULL,
    market TEXT NOT NULL,
    side TEXT NOT NULL,
    size BIGINT NOT NULL,
    price BIGINT NOT NULL,
    timestamp TIMESTAMP NOT NULL,
    settlement_tx TEXT,
    merkle_root TEXT,
    INDEX idx_user (user_pubkey),
    INDEX idx_timestamp (timestamp),
    INDEX idx_settlement (settlement_tx)
);
```

Query Patterns

Efficient Queries:

```sql
-- Get user positions
SELECT * FROM positions
WHERE user_pubkey = $1 AND status = 'open'
ORDER BY created_at DESC;

-- Get liquidation candidates
SELECT * FROM positions
WHERE status = 'open' AND margin_ratio < 1.0
ORDER BY margin_ratio ASC
LIMIT 100;
```

Caching Strategies

Redis Caching

Cache Current Prices:

```rust
use redis::Commands;

pub struct PriceCache {
    client: redis::Client,
}

impl PriceCache {
    pub async fn set_price(&self, symbol: &str, price: u64) -> Result<()> {
        let mut conn = self.client.get_async_connection().await?;
        let key = format!("price:{}", symbol);
        conn.set_ex(&key, price.to_string(), 60).await?; // 60s TTL
        Ok(())
    }

    pub async fn get_price(&self, symbol: &str) -> Result<Option<u64>> {
        let mut conn = self.client.get_async_connection().await?;
        let key = format!("price:{}", symbol);
        let value: Option<String> = conn.get(&key).await?;
        Ok(value.map(|v| v.parse().unwrap()))
    }
}
```

In-Memory Caching

Cache Position Data:

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

pub struct PositionCache {
    positions: Arc<RwLock<HashMap<Pubkey, Position>>>,
}

impl PositionCache {
    pub async fn get(&self, position_id: &Pubkey) -> Option<Position> {
        let positions = self.positions.read().await;
        positions.get(position_id).cloned()
    }

    pub async fn set(&self, position_id: Pubkey, position: Position) {
        let mut positions = self.positions.write().await;
        positions.insert(position_id, position);
    }
}
```

Performance Optimization

Compute Unit Optimization

Techniques:

1. **Batch Operations**: Process multiple items in one instruction
2. **Efficient Algorithms**: Use O(n log n) instead of O(n²)
3. **Early Returns**: Exit early when possible
4. **Cache Computations**: Store expensive calculations

Example:

```
// Optimized margin calculation
pub fn calculate_margin_ratio_batch(
    positions: &[Position],
    prices: &HashMap<Pubkey, u64>,
) -> Vec<u64> {
    positions.iter()
        .map(|pos| {
            let mark_price = prices.get(&pos.market).unwrap_or(&0);
            let unrealized_pnl = calculate_pnl(pos, *mark_price);
            let total_equity = pos.collateral as i64 + unrealized_pnl;
            (total_equity * 100) / pos.maintenance_margin
        })
        .collect()
}
```

Transaction Batching

Batch Multiple Operations:

```
pub fn build_batch_transaction(
    operations: Vec<Operation>,
) -> Result<Transaction> {
    let mut instructions = Vec::new();

    // Set compute budget
    instructions.push(
        ComputeBudgetInstruction::set_compute_unit_limit(400_000)
    );

    // Add operations
    for op in operations.iter() {
        instructions.push(op.to_instruction()?);
    }

    // Build transaction
    let transaction = Transaction::new_with_payer(
        &instructions,
        Some(&payer.pubkey()),
    );

    Ok(transaction)
}
```

Scalability Considerations

Account Limits

Constraints:

- 64 accounts per transaction
- 1,232 bytes per transaction
- 1.4M compute units per transaction

Solutions:

- Batch operations
- Use PDAs efficiently
- Minimize account data size

Throughput Optimization

Techniques:

1. **Parallel Processing**: Process multiple markets concurrently
2. **Connection Pooling**: Reuse database connections
3. **Async Operations**: Use async/await for I/O
4. **Load Balancing**: Distribute load across instances

Monitoring and Observability

Metrics

Key Metrics:

- Trades per second
- Settlement latency
- Liquidation rate
- Funding rate accuracy
- System uptime

Implementation:

```
use prometheus::{Counter, Histogram, Registry};

pub struct Metrics {
    trades_total: Counter,
    settlement_latency: Histogram,
    liquidations_total: Counter,
}

impl Metrics {
    pub fn record_trade(&self) {
        self.trades_total.inc();
    }

    pub fn record_settlement(&self, duration: Duration) {
        self.settlement_latency.observe(duration.as_secs_f64());
    }
}
```

Logging

Structured Logging:

```
use tracing::{info, error, warn};

pub fn process_trade(trade: &Trade) -> Result<()> {
    info!(
        user = %trade.user,
        market = %trade.market,
        size = trade.size,
        price = trade.price,
        "Processing trade"
    );

    // ... process trade ...

    info!(
        trade_id = %trade.id,
        "Trade processed successfully"
    );

    Ok(())
}
```

Key Takeaways

1. **Hybrid Architecture**: Off-chain speed, on-chain security
2. **Dark Pool**: Privacy through order hiding
3. **Batch Settlement**: Merkle trees + netting for efficiency
4. **State Management**: On-chain critical, off-chain performance
5. **Event-Driven**: WebSocket for real-time updates
6. **Caching**: Redis + in-memory for performance
7. **Optimization**: Compute units, transaction size, throughput
8. **Monitoring**: Metrics and logging for observability

Next Steps

- Review **11-integration-patterns.md** for component integration
- Study your component's architecture in detail
- Understand data flow in your component

Last Updated: November 2025

11. Integration Patterns for GoDark DEX

Overview

This guide covers how GoDark DEX components integrate with each other, external services, and the Solana blockchain. Understanding these patterns is essential for building cohesive, well-integrated components.

Component Communication Patterns

On-Chain Communication (CPIs)

Pattern: Cross-Program Invocations

Example: Position Management → Collateral Vault

```
// Position Management program calls Collateral Vault
pub fn lock_collateral(ctx: Context<LockCollateral>, amount: u64) -> Result<()> {
    let cpi_accounts = LockCollateralAccounts {
        vault: ctx.accounts.vault.to_account_info(),
        user_token: ctx.accounts.user_token.to_account_info(),
        vault_token: ctx.accounts.vault_token.to_account_info(),
        authority: ctx.accounts.position_pda.to_account_info(),
        token_program: ctx.accounts.token_program.to_account_info(),
    };

    let cpi_program = ctx.accounts.collateral_vault_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

    // Sign with position PDA
    let seeds = &[
        b"position",
        ctx.accounts.user.key().as_ref(),
        &[ctx.bumps.position],
    ];
    let cpi_ctx = cpi_ctx.with_signer(&[&seeds[..]]);

    collateral_vault::cpi::lock_collateral(cpi_ctx, amount)?;
    Ok(())
}
```

Off-Chain Communication (APIs)

Pattern: REST API calls between services

Example: Settlement Relayer → Position Management Service

```
use reqwest::Client;

pub struct PositionServiceClient {
    client: Client,
    base_url: String,
}

impl PositionServiceClient {
    pub async fn get_position(&self, position_id: &str) -> Result<Position> {
        let url = format!("{}/positions/{}", self.base_url, position_id);
        let response = self.client.get(&url).send().await?;
        let position: Position = response.json().await?;
        Ok(position)
    }

    pub async fn update_position(&self, update: PositionUpdate) -> Result<()> {
        let url = format!("{}/positions", self.base_url);
        self.client.post(&url).json(&update).send().await?;
        Ok(())
    }
}
```

API Integration

REST API Patterns

Standard Endpoints:

```
use axum::{Router, Json, extract::Path};

pub fn create_api_router() -> Router {
    Router::new()
        .route("/api/positions", post(create_position))
        .route("/api/positions/:id", get(get_position))
        .route("/api/positions/:id", put(update_position))
        .route("/api/positions/:id", delete(close_position))
        .route("/api/funding-rates", get(get_funding_rates))
        .route("/api/liquidations", get(get_liquidations))
}

async fn get_position(Path(id): Path<String>) -> Json<Position> {
    // Fetch position from database
    let position = db.get_position(&id).await.unwrap();
    Json(position)
}
```

Error Handling:

```
use axum::response::IntoResponse;

pub enum ApiError {
    NotFound(String),
    BadRequest(String),
    InternalError(String),
}

impl IntoResponse for ApiError {
    fn into_response(self) -> axum::response::Response {
        let (status, message) = match self {
            ApiError::NotFound(msg) => (StatusCode::NOT_FOUND, msg),
            ApiError::BadRequest(msg) => (StatusCode::BAD_REQUEST, msg),
            ApiError::InternalError(msg) => (StatusCode::INTERNAL_SERVER_ERROR, msg),
        };

        (status, Json(json!({"error": message}))).into_response()
    }
}
```

WebSocket Integration

Real-Time Updates:

```rust
use tokio_tungstenite::{WebSocketStream, MaybeTlsStream};
use futures_util::{SinkExt, StreamExt};

pub async fn handle_websocket(
    stream: WebSocketStream<MaybeTlsStream<TcpStream>>,
) {
    let (mut sender, mut receiver) = stream.split();

    // Subscribe to updates
    while let Some(msg) = receiver.next().await {
        match msg {
            Ok(Message::Text(text)) => {
                let request: WebSocketRequest = serde_json::from_str(&text)?;

                match request {
                    WebSocketRequest::SubscribePosition { position_id } => {
                        subscribe_to_position(&mut sender, position_id).await?;
                    }
                    WebSocketRequest::SubscribeMarket { market } => {
                        subscribe_to_market(&mut sender, market).await?;
                    }
                }
            }
            Err(e) => {
                eprintln!("WebSocket error: {}", e);
                break;
            }
            _ => {}
        }
    }
}
```

Database Integration Patterns

Connection Pooling

PostgreSQL Pool:

```rust
use sqlx::postgres::{PgPool, PgPoolOptions};

pub async fn create_pool(database_url: &str) -> Result<PgPool> {
    let pool = PgPoolOptions::new()
        .max_connections(10)
        .acquire_timeout(Duration::from_secs(5))
        .connect(database_url)
        .await?;

    Ok(pool)
}
```

Transaction Management

Database Transactions:

```rust
pub async fn create_position_with_trade(
    pool: &PgPool,
    position: &Position,
    trade: &Trade,
) -> Result<()> {
    let mut tx = pool.begin().await?;

    // Insert position
    sqlx::query!(
        "INSERT INTO positions (id, user_pubkey, size, ...) VALUES ($1, $2, $3, ...)",
        position.id,
        position.user_pubkey,
        position.size,
    )
    .execute(&mut *tx)
    .await?;

    // Insert trade
    sqlx::query!(
        "INSERT INTO trades (id, position_id, size, price, ...) VALUES ($1, $2, $3, $4, ...)",
        trade.id,
        trade.position_id,
        trade.size,
        trade.price,
    )
    .execute(&mut *tx)
    .await?;

    // Commit transaction
    tx.commit().await?;
    Ok(())
}
```

Query Optimization

Efficient Queries:

```rust
// Use prepared statements
pub async fn get_user_positions(
    pool: &PgPool,
    user_pubkey: &str,
) -> Result<Vec<Position>> {
    let positions = sqlx::query_as!(
        Position,
        "SELECT * FROM positions WHERE user_pubkey = $1 AND status = 'open'",
        user_pubkey
    )
    .fetch_all(pool)
    .await?;

    Ok(positions)
}

// Use indexes
// CREATE INDEX idx_user_status ON positions(user_pubkey, status);
```

Oracle Integration

Pyth Network Integration

Price Fetching:

```rust
use pyth_solana_receiver_sdk::price_update::get_feed_id_from_hex;

pub struct PythClient {
    rpc_client: RpcClient,
    price_feed_ids: HashMap<String, Pubkey>,
}

impl PythClient {
    pub async fn get_price(&self, symbol: &str) -> Result<PriceData> {
        let feed_id = self.price_feed_ids.get(symbol)
            .ok_or_else(|| anyhow!("Unknown symbol: {}", symbol))?;

        let account_info = self.rpc_client.get_account(feed_id).await?;
        let price_data = parse_price_account(&account_info.data)?;

        Ok(PriceData {
            price: price_data.price,
            confidence: price_data.confidence,
            timestamp: price_data.publish_time,
        })
    }
}
```

Switchboard Fallback

Multi-Oracle Pattern:

```rust
pub async fn get_consensus_price(
    symbol: &str,
    pyth_client: &PythClient,
    switchboard_client: &SwitchboardClient,
) -> Result<u64> {
    // Try Pyth first
    let pyth_price = match pyth_client.get_price(symbol).await {
        Ok(price) => Some(price),
        Err(_) => None,
    };

    // Fallback to Switchboard
    let switchboard_price = match switchboard_client.get_price(symbol).await {
        Ok(price) => Some(price),
        Err(_) => None,
    };

    // Use consensus
    match (pyth_price, switchboard_price) {
        (Some(p), Some(s)) => {
            // Use median
            let prices = vec![p.price, s.price];
            prices.sort();
            Ok(prices[prices.len() / 2])
        }
        (Some(p), None) => Ok(p.price),
        (None, Some(s)) => Ok(s.price),
        (None, None) => Err(anyhow!("No oracle data available")),
    }
}
```

Wallet Integration

Phantom Wallet

Frontend Integration:

```
import { useWallet } from '@solana/wallet-adapter-react';

function TradingInterface() {
    const { publicKey, signTransaction, connected } = useWallet();

    const openPosition = async (size: number, leverage: number) => {
        if (!publicKey || !signTransaction) return;

        // Build transaction
        const transaction = await buildOpenPositionTransaction(
            publicKey,
            size,
            leverage
        );

        // Sign and send
        const signed = await signTransaction(transaction);
        const signature = await connection.sendRawTransaction(signed.serialize());
        await connection.confirmTransaction(signature);
    };

    return (
        <button onClick={() => openPosition(1000, 10)}>
            Open Position
        </button>
    );
}
```

Solflare Integration

Similar Pattern:

```
import { useWallet } from '@solana/wallet-adapter-react';

// Same API as Phantom
const { publicKey, signTransaction } = useWallet();
```

Cross-Component Data Flow

Position Lifecycle Flow

```
User Request
    ↓
Matching Engine (off-chain)
    ↓
Settlement Relayer
    ├─→ Batch Accumulation
    ├─→ Merkle Tree Generation
    └─→ Transaction Building
    ↓
Position Management (on-chain)
    ├─→ Create/Update Position Account
    ├─→ Lock Collateral (CPI → Collateral Vault)
    └─→ Emit PositionUpdated Event
    ↓
Off-Chain Services
    ├─→ Position Service (update database)
    ├─→ Liquidation Engine (monitor position)
    └─→ WebSocket (notify clients)
```

Funding Rate Flow

```
Oracle Integration
      ├─→ Fetch Mark Price (Pyth/Switchboard)
      └─→ Fetch Index Price
      ↓
Funding Rate Service
      ├─→ Calculate Premium Index
      ├─→ Calculate Interest Rate
      ├─→ Calculate Funding Rate (every 1 second)
      └─→ Aggregate Hourly
      ↓
On-Chain Distribution
      ├─→ Position Management (apply payments)
      └─→ Update Position Collateral
      ↓
Events
      └─→ FundingRatePaid Event
```

Error Propagation and Handling

Error Types

Define Error Types:

```
#[derive(Debug, thiserror::Error)]
pub enum IntegrationError {
    #[error("Oracle error: {0}")]
    OracleError(String),

    #[error("Database error: {0}")]
    DatabaseError(#[from] sqlx::Error),

    #[error("RPC error: {0}")]
    RpcError(#[from] solana_client::client_error::ClientError),

    #[error("Transaction failed: {0}")]
    TransactionError(String),
}
```

Error Handling Pattern

Propagate Errors:

```
pub async fn process_settlement(
    trades: &[Trade],
) -> Result<()> {
    // Try settlement
    let result = submit_settlement_transaction(trades).await;

    match result {
        Ok(signature) => {
            // Update database
            db.mark_trades_settled(trades, &signature).await?;
            Ok(())
        }
        Err(e) => {
            // Log error
            error!("Settlement failed: {}", e);

            // Retry logic
            if should_retry(&e) {
                retry_settlement(trades).await
            } else {
                Err(IntegrationError::TransactionError(e.to_string()))
            }
        }
    }
}
```

Transaction Dependency Management

Transaction Ordering

Dependencies:

```rust
pub struct TransactionDependency {
    pub depends_on: Vec<Signature>,
    pub transaction: Transaction,
}

pub async fn execute_with_dependencies(
    dependencies: Vec<TransactionDependency>,
) -> Result<()> {
    // Topological sort
    let sorted = topological_sort(dependencies)?;

    // Execute in order
    for dep in sorted.iter() {
        // Wait for dependencies
        for sig in dep.depends_on.iter() {
            wait_for_confirmation(sig).await?;
        }

        // Execute transaction
        execute_transaction(&dep.transaction).await?;
    }

    Ok(())
}
```

Event Coordination

Event Bus Pattern

Central Event Bus:

```rust
use tokio::sync::broadcast;

pub struct EventBus {
    sender: broadcast::Sender<Event>,
}

impl EventBus {
    pub fn new() -> Self {
        let (sender, _) = broadcast::channel(1000);
        Self { sender }
    }

    pub fn publish(&self, event: Event) -> Result<()> {
        self.sender.send(event)?;
        Ok(())
    }

    pub fn subscribe(&self) -> broadcast::Receiver<Event> {
        self.sender.subscribe()
    }
}
```

Event Handlers:

```rust
pub async fn handle_position_events(mut receiver: broadcast::Receiver<Event>) {
    while let Ok(event) = receiver.recv().await {
        match event {
            Event::PositionOpened { position_id, .. } => {
                // Update database
                db.create_position(position_id).await?;

                // Notify liquidation engine
                liquidation_engine.monitor_position(position_id).await?;
            }
            Event::PositionClosed { position_id, .. } => {
                // Update database
                db.close_position(position_id).await?;

                // Stop monitoring
                liquidation_engine.stop_monitoring(position_id).await?;
            }
            _ => {}
        }
    }
}
```

Integration Testing Strategies

Mock External Services

Mock Oracle:

```rust
pub struct MockOracle {
    prices: HashMap<String, u64>,
}

impl OracleClient for MockOracle {
    async fn get_price(&self, symbol: &str) -> Result<PriceData> {
        let price = self.prices.get(symbol)
            .ok_or_else(|| anyhow!("Price not found"))?;

        Ok(PriceData {
            price: *price,
            confidence: 100,
            timestamp: Utc::now().timestamp(),
        })
    }
}
```

Integration Test Setup

Test Environment:

```rust
#[tokio::test]
async fn test_position_lifecycle() {
    // Setup test database
    let db = setup_test_database().await;

    // Setup mock oracle
    let oracle = MockOracle::new();
    oracle.set_price("BTC", 50000);

    // Setup test RPC
    let rpc = setup_test_rpc().await;

    // Test position creation
    let position = create_position(&db, &oracle, &rpc).await?;
    assert_eq!(position.size, 1000);

    // Test position update
    update_position(&db, &oracle, &rpc, position.id).await?;

    // Test position close
    close_position(&db, &oracle, &rpc, position.id).await?;
}
```

Key Takeaways

1. **CPIs**: On-chain component communication
2. **REST APIs**: Off-chain service communication
3. **WebSockets**: Real-time updates
4. **Database**: Persistent state management
5. **Oracles**: Multi-source price feeds
6. **Wallets**: User interaction
7. **Error Handling**: Proper propagation
8. **Events**: Coordination mechanism

Next Steps

- Review **10-architecture-deep-dive.md** for architecture details
- Study your component's integration points
- Practice implementing integrations

Last Updated: November 2025

12. Tooling and Resources for GoDark DEX Development

Overview

This guide provides a comprehensive reference for essential tools, commands, and resources used in GoDark DEX development.

Development Tools

Solana CLI Commands Reference

Configuration:

```
# Set RPC endpoint
solana config set --url localhost        # Localnet
solana config set --url devnet           # Devnet
solana config set --url mainnet-beta     # Mainnet

# View current config
solana config get

# Set keypair
solana config set --keypair ~/.config/solana/id.json
```

Keypair Management:

```
# Generate new keypair
solana-keygen new

# Generate keypair with specific path
solana-keygen new --outfile ~/my-keypair.json

# View public key
solana-keygen pubkey

# View public key from file
solana-keygen pubkey ~/my-keypair.json
```

Account Management:

```
# Check balance
solana balance

# Check balance of specific account
solana balance <PUBKEY>

# Airdrop SOL (devnet/localnet only)
solana airdrop 2

# Transfer SOL
solana transfer <RECIPIENT> 1 --allow-unfunded-recipient
```

Program Management:

```
# Deploy program
solana program deploy target/deploy/your_program.so

# Upgrade program
solana program deploy --program-id <PROGRAM_ID> target/deploy/your_program.so

# Show program info
solana program show <PROGRAM_ID>

# Close program (recover rent)
solana program close <PROGRAM_ID>
```

Account Inspection:

```
# View account data
solana account <ACCOUNT_PUBKEY>

# View account data as JSON
solana account <ACCOUNT_PUBKEY> --output json

# View program account
solana account <PROGRAM_ID> --output json
```

Transaction Management:

```
# Confirm transaction
solana confirm <SIGNATURE>

# View transaction details
solana confirm <SIGNATURE> --verbose

# Get transaction history
solana transaction-history <ACCOUNT_PUBKEY>
```

Anchor CLI Commands

Project Management:

```
# Initialize new Anchor project
anchor init <project-name>

# Build program
anchor build

# Build and deploy
anchor build && anchor deploy

# Clean build artifacts
anchor clean
```

Testing:

```
# Run tests
anchor test

# Run specific test file
anchor test tests/your-test.ts

# Run with verbose output
anchor test -- --verbose

# Run with logs
anchor test --skip-local-validator
```

IDL Management:

```
# Generate IDL from program
anchor idl parse -f programs/your-program/src/lib.rs -o target/idl/your_program.json

# Initialize IDL on-chain
anchor idl init --filepath target/idl/your_program.json <PROGRAM_ID>

# Upgrade IDL
anchor idl upgrade --filepath target/idl/your_program.json <PROGRAM_ID>
```

Local Validator:

```
# Start local validator
solana-test-validator

# Start with specific features
solana-test-validator --reset

# Start with logs
solana-test-validator --log
```

Rust Tooling

Cargo Commands:

```
# Build project
cargo build

# Build release
cargo build --release

# Run tests
cargo test

# Run specific test
cargo test test_name

# Run with output
cargo test -- --nocapture

# Check code (no build)
cargo check

# Format code
cargo fmt

# Lint code
cargo clippy

# Clippy with fixes
cargo clippy --fix
```

Useful Cargo Flags:

```
# Build for specific target
cargo build --target bpf-unknown-unknown

# Build with features
cargo build --features feature-name

# Update dependencies
cargo update

# Show dependency tree
cargo tree

# Show outdated dependencies
cargo outdated
```

IDEs and Extensions

VS Code:

- **Rust Analyzer**: Language server for Rust
- **Solana**: Solana development tools
- **Anchor**: Anchor framework support
- **Error Lens**: Inline error display

Setup:

```
# Install Rust Analyzer extension
code --install-extension rust-lang.rust-analyzer

# Install Solana extension
code --install-extension solana.solana-dev
```

IntelliJ IDEA / CLion:

- Rust plugin
- Solana plugin (community)

Testing Tools

Anchor Test Framework

Setup:

```
import * as anchor from "@coral-xyz/anchor";
import { Program } from "@coral-xyz/anchor";

describe("tests", () => {
  const provider = anchor.AnchorProvider.env();
  anchor.setProvider(provider);

  const program = anchor.workspace.YourProgram;

  it("test", async () => {
    // Test code
  });
});
```

Useful Test Utilities:

```
// Airdrop SOL
await provider.connection.requestAirdrop(
  user.publicKey,
  10 * anchor.web3.LAMPORTS_PER_SOL
);

// Wait for confirmation
await provider.connection.confirmTransaction(signature, "confirmed");

// Get account data
const account = await program.account.state.fetch(statePDA);
```

Solana Test Utilities

Test Validator:

```
# Start validator
solana-test-validator

# With reset
solana-test-validator --reset

# With specific program
solana-test-validator --clone <PROGRAM_ID>
```

Transaction Simulation:

```
// Simulate transaction
const simulation = await connection.simulateTransaction(transaction);
console.log("Compute units:", simulation.value.unitsConsumed);
console.log("Logs:", simulation.value.logs);
```

Debugging Tools

Solana Explorer

URLs:

- **Mainnet**: https://explorer.solana.com/

- **Devnet**: https://explorer.solana.com/?cluster=devnet
- **Localnet**: http://localhost:8899 (if running local validator)

Features:

- View transactions
- Inspect accounts
- Check program deployments
- View token balances

Usage:

```
https://explorer.solana.com/tx/<SIGNATURE>
https://explorer.solana.com/address/<PUBKEY>
https://explorer.solana.com/account/<ACCOUNT_PUBKEY>
```

Solscan

URL: https://solscan.io/

Features:

- Transaction history
- Account analysis
- Token tracking
- Program inspection

Anchor IDL Viewer

View Program Interface:

```
# Generate IDL
anchor idl parse -f programs/your-program/src/lib.rs -o target/idl/your_program.json

# View IDL
cat target/idl/your_program.json | jq
```

Online Viewer:

- Upload IDL JSON to view program interface
- See all instructions and accounts

Transaction Decoders

Decode Transaction:

```
# Using Solana CLI
solana confirm <SIGNATURE> --verbose

# Using web3.js
const tx = await connection.getTransaction(signature, {
  maxSupportedTransactionVersion: 0
});
console.log(tx);
```

Decode Account Data:

```
use anchor_lang::AccountDeserialize;

let account_data = account_info.data.borrow();
let position = Position::try_deserialize(&mut &account_data[8..])?;
```

Resources

Official Documentation

Solana:

- **Docs**: https://docs.solana.com/
- **Cookbook**: https://solanacookbook.com/
- **API Reference**: https://docs.rs/solana-sdk/

Anchor:

- **Book**: https://www.anchor-lang.com/
- **API Docs**: https://docs.rs/anchor-lang/
- **Examples**: https://github.com/coral-xyz/anchor/tree/master/examples

SPL Token:

- **Docs**: https://spl.solana.com/token
- **Program**: https://github.com/solana-labs/solana-program-library

---

GoDark Resources

Technical Architecture:

- GoDark DEX Technical Architecture Document
- Component assignment documentation
- Architecture diagrams

Component Assignments:

- Settlement Relayer assignment
- Position Management assignment
- Liquidation Engine assignment
- Ephemeral Vault assignment
- Funding Rate assignment
- Oracle Integration assignment
- Collateral Vault assignment
- Program Upgrade assignment

Community Resources

Discord:

- Solana Discord: https://discord.gg/solana
- Anchor Discord: https://discord.gg/anchorlang

Forums:

- Solana Stack Exchange: https://solana.stackexchange.com/
- Reddit: r/solana

GitHub:

- Solana: https://github.com/solana-labs/solana

- Anchor: https://github.com/coral-xyz/anchor
- SPL: https://github.com/solana-labs/solana-program-library

Useful Scripts

Build Script

build.sh:

```
#!/bin/bash
set -e

echo "Building Anchor program..."
anchor build

echo "Generating IDL..."
anchor idl parse -f programs/your-program/src/lib.rs -o target/idl/your_program.json

echo "Build complete!"
```

Deploy Script

deploy.sh:

```
#!/bin/bash
set -e

CLUSTER=${1:-localnet}

echo "Deploying to $CLUSTER..."

if [ "$CLUSTER" = "localnet" ]; then
    solana config set --url localhost
    anchor build
    anchor deploy
elif [ "$CLUSTER" = "devnet" ]; then
    solana config set --url devnet
    anchor build
    anchor deploy --provider.cluster devnet
else
    echo "Unknown cluster: $CLUSTER"
    exit 1
fi

echo "Deployment complete!"
```

Test Script

test.sh:

```bash
#!/bin/bash
set -e

echo "Running tests..."

# Start validator in background
solana-test-validator &
VALIDATOR_PID=$!

# Wait for validator to start
sleep 5

# Run tests
anchor test

# Stop validator
kill $VALIDATOR_PID

echo "Tests complete!"
```

Environment Setup

Required Environment Variables

.env.example:

```
# Solana
SOLANA_RPC_URL=http://localhost:8899
ANCHOR_PROVIDER_URL=http://localhost:8899
ANCHOR_WALLET=~/.config/solana/id.json

# Database
DATABASE_URL=postgresql://postgres:postgres@localhost/godark_dev

# Redis
REDIS_URL=redis://localhost:6379

# API
API_PORT=3000
API_HOST=0.0.0.0

# Oracle
PYTH_RPC_URL=https://api.mainnet-beta.solana.com
SWITCHBOARD_RPC_URL=https://api.mainnet-beta.solana.com
```

Quick Reference

Common Commands

Development:

```bash
# Start local validator
solana-test-validator

# Build and deploy
anchor build && anchor deploy

# Run tests
anchor test

# Check balance
solana balance
```

Debugging:

```
# View account
solana account <PUBKEY>

# Confirm transaction
solana confirm <SIGNATURE>

# View logs
solana logs
```

Program Management:

```
# Deploy program
solana program deploy target/deploy/program.so

# Show program
solana program show <PROGRAM_ID>

# Close program
solana program close <PROGRAM_ID>
```

Key Takeaways

1. **Solana CLI**: Essential for account and program management
2. **Anchor CLI**: Simplifies program development
3. **Rust Tools**: Cargo, clippy, fmt for code quality
4. **Explorers**: Solana Explorer, Solscan for debugging
5. **Documentation**: Official docs and community resources
6. **Scripts**: Automate common tasks

Next Steps

- Review **13-quick-reference.md** for quick lookup
- Check **14-troubleshooting-guide.md** for common issues
- Set up your development environment
- Practice using these tools

Last Updated: November 2025

13. Quick Reference Guide for GoDark DEX

Overview

Quick reference cheat sheet for common operations, formulas, and commands in GoDark DEX development.

PDA Derivation Formulas

Basic PDA Derivation

```rust
use anchor_lang::prelude::*;

// Derive PDA
let (pda, bump) = Pubkey::find_program_address(
    &[
        b"seed1",
        seed2.as_ref(),
        program_id.as_ref(),
    ],
    program_id,
);

// Verify PDA in constraint
#[account(
    seeds = [b"seed1", seed2.key().as_ref()],
    bump
)]
pub pda_account: Account<'info, State>,
```

Common PDA Patterns

User-Specific:

```rust
// Position PDA
let (position_pda, bump) = Pubkey::find_program_address(
    &[b"position", user_pubkey.as_ref()],
    program_id,
);

// Vault PDA
let (vault_pda, bump) = Pubkey::find_program_address(
    &[b"vault", user_pubkey.as_ref(), mint_pubkey.as_ref()],
    program_id,
);
```

Global State:

```rust
// Config PDA
let (config_pda, bump) = Pubkey::find_program_address(
    &[b"config"],
    program_id,
);
```

Common Anchor Macros

Program Macros

```rust
// Declare program ID
declare_id!("YourProgram11111111111111111111111111");

// Program module
#[program]
pub mod your_program {
    use super::*;

    pub fn instruction(ctx: Context<Accounts>, data: u64) -> Result<()> {
        Ok(())
    }
}
```

Account Macros

```rust
// Account data structure
#[account]
pub struct State {
    pub data: u64,
}

// Account constraints
#[derive(Accounts)]
pub struct Accounts<'info> {
    #[account(init, payer = user, space = 8 + State::LEN)]
    pub state: Account<'info, State>,

    #[account(mut)]
    pub user: Signer<'info>,

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

Constraint Macros

```rust
#[account(
    init,                     // Initialize account
    payer = user,             // Who pays rent
    space = 8 + 32,           // Account size
    seeds = [b"seed"],        // PDA seeds
    bump,                     // Bump seed
    mut,                      // Account is writable
    close = user,             // Close account and return rent
    has_one = user @ ErrorCode::Mismatch, // Verify field matches
    owner = program_id @ ErrorCode::InvalidOwner, // Verify owner
)]
pub account: Account<'info, State>,
```

Account Size Calculations

Size Formula

```
Total Size = Discriminator (8 bytes) + Sum of Field Sizes
```

Common Field Sizes

```rust
// Primitive types
u8:     1 byte
u16:    2 bytes
u32:    4 bytes
u64:    8 bytes
i64:    8 bytes
bool:   1 byte

// Solana types
Pubkey: 32 bytes

// Vectors
Vec<T>: 4 bytes (length) + (T size × length)

// Strings
String: 4 bytes (length) + (1 byte × length)
```

Example Calculation

```rust
#[account]
pub struct Position {
    pub user: Pubkey,         // 32 bytes
    pub size: u64,            // 8 bytes
    pub entry_price: u64,     // 8 bytes
    pub leverage: u8,         // 1 byte
    pub version: u32,         // 4 bytes
}

impl Position {
    pub const LEN: usize = 32 + 8 + 8 + 1 + 4; // 53 bytes
    // Total account size: 8 (discriminator) + 53 = 61 bytes
}
```

Transaction Building Patterns

Basic Transaction

```rust
use solana_sdk::transaction::Transaction;

let mut transaction = Transaction::new_with_payer(
    &[instruction],
    Some(&payer.pubkey()),
);

transaction.sign(&[&payer], recent_blockhash);
```

Multiple Instructions

```rust
let mut instructions = Vec::new();

// Instruction 1
instructions.push(instruction1);

// Instruction 2
instructions.push(instruction2);

let transaction = Transaction::new_with_payer(
    &instructions,
    Some(&payer.pubkey()),
);
```

Compute Budget

```rust
use solana_program::compute_budget::ComputeBudgetInstruction;

let compute_budget = ComputeBudgetInstruction::set_compute_unit_limit(400_000);
instructions.push(compute_budget);
```

Error Code Reference

Common Error Codes

```
#[error_code]
pub enum ErrorCode {
    #[msg("Insufficient funds")]
    InsufficientFunds,

    #[msg("Unauthorized")]
    Unauthorized,

    #[msg("Invalid input")]
    InvalidInput,

    #[msg("Account not found")]
    AccountNotFound,

    #[msg("Overflow")]
    Overflow,

    #[msg("Underflow")]
    Underflow,
}
```

Solana Program Errors

```
// Common Solana errors
ProgramError::InsufficientFunds
ProgramError::InvalidAccountData
ProgramError::InvalidAccountOwner
ProgramError::AccountNotInitialized
ProgramError::ArithmeticOverflow
```

SPL Token Operations Quick Reference

Transfer Tokens

```
use anchor_spl::token::{self, Transfer};

pub fn transfer(ctx: Context<TransferTokens>, amount: u64) -> Result<()> {
    let cpi_accounts = Transfer {
        from: ctx.accounts.from.to_account_info(),
        to: ctx.accounts.to.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
    token::transfer(cpi_ctx, amount)?;
    Ok(())
}
```

Mint Tokens

```
use anchor_spl::token::{self, MintTo};

pub fn mint(ctx: Context<MintTokens>, amount: u64) -> Result<()> {
    let cpi_accounts = MintTo {
        mint: ctx.accounts.mint.to_account_info(),
        to: ctx.accounts.to.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
    token::mint_to(cpi_ctx, amount)?;
    Ok(())
}
```

Burn Tokens

```rust
use anchor_spl::token::{self, Burn};

pub fn burn(ctx: Context<BurnTokens>, amount: u64) -> Result<()> {
    let cpi_accounts = Burn {
        mint: ctx.accounts.mint.to_account_info(),
        from: ctx.accounts.from.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
    token::burn(cpi_ctx, amount)?;
    Ok(())
}
```

Oracle Price Reading Patterns

Read Price from Account

```rust
use pyth_solana_receiver_sdk::price_update::PriceUpdateV2;

pub fn read_price(price_account: &AccountInfo) -> Result<u64> {
    let price_data = PriceUpdateV2::try_from(price_account.data.borrow().as_ref())?;

    // Validate price
    require!(
        price_data.price > 0,
        ErrorCode::InvalidPrice
    );

    Ok(price_data.price as u64)
}
```

Validate Price Staleness

```rust
pub fn validate_price_staleness(price_data: &PriceData) -> Result<()> {
    let clock = Clock::get()?;
    let max_age = 60; // 60 seconds

    require!(
        clock.unix_timestamp - price_data.timestamp < max_age,
        ErrorCode::StalePrice
    );

    Ok(())
}
```

Common CLI Commands

Solana CLI

```
# Config
solana config set --url localhost
solana config get

# Keypair
solana-keygen new
solana-keygen pubkey

# Balance
solana balance
solana airdrop 2

# Program
solana program deploy target/deploy/program.so
solana program show <PROGRAM_ID>

# Account
solana account <PUBKEY>

# Transaction
solana confirm <SIGNATURE>
```

Anchor CLI

```
# Build
anchor build

# Deploy
anchor deploy

# Test
anchor test

# IDL
anchor idl parse -f programs/program/src/lib.rs -o target/idl/program.json
```

Cargo

```
# Build
cargo build

# Test
cargo test

# Format
cargo fmt

# Lint
cargo clippy

# Check
cargo check
```

Debugging Commands

View Account Data

```
# Raw data
solana account <PUBKEY>

# JSON format
solana account <PUBKEY> --output json

# Decode in program
let account_data = account_info.data.borrow();
let state = State::try_deserialize(&mut &account_data[8..])?;
```

View Transaction

```
# Basic info
solana confirm <SIGNATURE>

# Verbose
solana confirm <SIGNATURE> --verbose

# In code
const tx = await connection.getTransaction(signature, {
  maxSupportedTransactionVersion: 0
});
```

View Logs

```
# Solana logs
solana logs

# Anchor logs
anchor test --skip-local-validator

# Program logs
msg!("Debug: value = {}", value);
```

Financial Formulas

Margin Calculations

```
Initial Margin = Notional Value / Leverage
Maintenance Margin = Notional Value × Maintenance Margin Rate
Margin Ratio = (Collateral + Unrealized PnL) / Maintenance Margin
```

PnL Calculations

```
Unrealized PnL (Long) = Position Size × (Mark Price - Entry Price)
Unrealized PnL (Short) = Position Size × (Entry Price - Mark Price)
Realized PnL = Position Size × (Exit Price - Entry Price) + Funding Payments
```

Funding Rate

```
Premium Index = (Mark Price - Index Price) / Index Price
Funding Rate = Premium Index + Interest Rate
Funding Payment = Position Size × Mark Price × Funding Rate
```

Liquidation Price

```
Liquidation Price (Long) = Entry Price × (1 - Initial Margin / Maintenance Margin)
Liquidation Price (Short) = Entry Price × (1 + Initial Margin / Maintenance Margin)
```

Key Takeaways

1. **PDAs**: Use `find_program_address` for derivation
2. **Account Size**: Discriminator (8) + field sizes
3. **CPIs**: Use Anchor's CPI helpers
4. **Errors**: Use `#[error_code]` enum
5. **Formulas**: Reference financial calculations
6. **CLI**: Quick command reference

Next Steps

- Keep this guide handy for quick lookup
- Refer to detailed guides for explanations
- Practice using these patterns

Last Updated: November 2025

14. Troubleshooting Guide for GoDark DEX

Overview

Common issues, error messages, and solutions for GoDark DEX development. Use this guide to quickly resolve problems.

Build Errors and Solutions

Error: "Program ID mismatch"

Problem:

```
Error: Program ID mismatch
```

Solution:

```
# Update Anchor.toml with correct program ID
[programs.localnet]
your_program = "YourProgram1111111111111111111111111"

# Or regenerate program ID
anchor keys list
anchor keys sync
```

Error: "Account discriminator already in use"

Problem:

```
Error: Account discriminator already in use
```

Solution:

```
// Change account struct name or add version field
#[account]
pub struct PositionV2 {  // Changed name
    pub version: u32,
    // ... fields
}
```

Error: "Failed to get recent blockhash"

Problem:

```
Error: Failed to get recent blockhash
```

Solution:

```
# Check RPC connection
solana config get

# Test connection
solana balance

# If localnet, ensure validator is running
solana-test-validator
```

Error: "Insufficient funds for transaction"

Problem:

```
Error: Insufficient funds for transaction
```

Solution:

```
# Check balance
solana balance

# Airdrop SOL (devnet/localnet)
solana airdrop 2

# Check rent requirements
# Accounts need rent-exempt balance
```

Deployment Issues

Error: "Program failed to deploy"

Problem:

```
Error: Program failed to deploy
```

Solutions:

1. **Check program size:**

```
# Solana programs have size limits
ls -lh target/deploy/your_program.so
```

2. **Verify build:**

```
anchor build
cargo build-sbf
```

3. **Check RPC endpoint:**

```
solana config get
solana balance
```

Error: "Upgrade authority mismatch"

Problem:

```
Error: Upgrade authority mismatch
```

Solution:

```
# Check upgrade authority
solana program show <PROGRAM_ID>

# Set correct upgrade authority
solana program set-upgrade-authority <PROGRAM_ID> --new-upgrade-authority <AUTHORITY>
```

Error: "Program account data too large"

Problem:

```
Error: Program account data too large
```

Solution:

- Optimize program code
- Remove unused dependencies
- Split into multiple programs if needed

Transaction Failures

Error: "Insufficient funds for rent"

Problem:

```
Error: Insufficient funds for rent
```

Solution:

```
// Calculate rent-exempt minimum
let rent = Rent::get()?;
let space = 8 + State::LEN;
let minimum_balance = rent.minimum_balance(space);

// Ensure account has enough balance
require!(
    account.lamports() >= minimum_balance,
    ErrorCode::InsufficientBalance
);
```

Error: "Compute budget exceeded"

Problem:

```
Error: Compute budget exceeded
```

Solution:

```
// Set higher compute budget
use solana_program::compute_budget::ComputeBudgetInstruction;

let compute_budget = ComputeBudgetInstruction::set_compute_unit_limit(400_000);
instructions.push(compute_budget);
```

Or optimize code:

- Reduce loops
- Use efficient algorithms
- Cache expensive computations

Error: "Transaction too large"

Problem:

```
Error: Transaction too large
```

Solution:

- Reduce number of accounts
- Reduce instruction data size
- Split into multiple transactions
- Use netting to reduce operations

Error: "Account not found"

Problem:

```
Error: Account not found
```

Solution:

```
// Check account exists before using
let account_info = ctx.accounts.account.to_account_info();
require!(
    account_info.data_len() > 0,
    ErrorCode::AccountNotFound
);
```

Error: "Invalid account owner"

Problem:

```
Error: Invalid account owner
```

Solution:

```
// Verify account ownership
require!(
    ctx.accounts.account.owner == ctx.program_id,
    ErrorCode::InvalidAccountOwner
);

// Or use Anchor constraint
#[account(
    owner = program_id @ ErrorCode::InvalidAccountOwner
)]
pub account: Account<'info, State>,
```

Account Initialization Problems

Error: "Account already initialized"

Problem:

```
Error: Account already initialized
```

Solution:

```
// Check if account exists
let account = program.account.state.fetch_optional(&state_pda).await?;

if account.is_none() {
    // Initialize account
    program.methods.initialize().rpc()?;
} else {
    // Account already exists, use it
}
```

Error: "Account space calculation incorrect"

Problem:

```
Error: Account space calculation incorrect
```

Solution:

```rust
// Calculate size accurately
#[account]
pub struct State {
    pub field1: u64,    // 8 bytes
    pub field2: Pubkey, // 32 bytes
    pub field3: u8,     // 1 byte
}

impl State {
    pub const LEN: usize = 8 + 32 + 1; // 41 bytes
    // Total: 8 (discriminator) + 41 = 49 bytes
}

#[account(init, payer = user, space = 8 + State::LEN)]
pub state: Account<'info, State>,
```

PDA Derivation Issues

Error: "PDA derivation failed"

Problem:

```
Error: PDA derivation failed
```

Solution:

```rust
// Ensure seeds match exactly
const SEED: &[u8] = b"seed";

let (pda, bump) = Pubkey::find_program_address(
    &[SEED, other_seed.as_ref()],
    program_id,
);

// Use same seeds in constraint
#[account(
    seeds = [SEED, other_seed.key().as_ref()],
    bump
)]
pub pda: Account<'info, State>,
```

Error: "Invalid PDA signer"

Problem:

```
Error: Invalid PDA signer
```

Solution:

```rust
// Include bump in seeds for signing
let seeds = &[
    b"seed",
    other_seed.as_ref(),
    &[bump], // Include bump!
];
let signer = &[&seeds[..]];

// Use in CPI
let cpi_ctx = CpiContext::new_with_signer(
    program,
    accounts,
    signer,
);
```

CPI Failures

Error: "CPI call failed"

Problem:

```
Error: CPI call failed
```

Solution:

1. **Verify program ID:**

```rust
require!(
    ctx.accounts.token_program.key() == &token::ID,
    ErrorCode::InvalidProgram
);
```

2. **Check account ownership:**

```rust
require!(
    ctx.accounts.token_account.owner == &token::ID,
    ErrorCode::InvalidAccountOwner
);
```

3. **Verify signer:**

```rust
// Ensure authority is signer or PDA signer
let seeds = &[b"vault", user.key().as_ref(), &[bump]];
let signer = &[&seeds[..]];
```

Error: "Insufficient token balance"

Problem:

```
Error: Insufficient token balance
```

Solution:

```
// Check balance before transfer
let balance = ctx.accounts.from.amount;
require!(
    balance >= amount,
    ErrorCode::InsufficientBalance
);
```

Testing Issues

Error: "Test timeout"

Problem:

```
Error: Test timeout
```

Solution:

```
// Increase timeout
it("test", async () => {
  // Test code
}).timeout(60000); // 60 seconds

// Or in anchor test
anchor test -- --timeout 60000
```

Error: "Account not found in test"

Problem:

```
Error: Account not found in test
```

Solution:

```
// Initialize account before use
await program.methods
  .initialize()
  .accounts({
    state: statePDA,
    user: user.publicKey,
    systemProgram: SystemProgram.programId,
  })
  .rpc();

// Then use account
const state = await program.account.state.fetch(statePDA);
```

Error: "Transaction simulation failed"

Problem:

```
Error: Transaction simulation failed
```

Solution:

```
// Check simulation logs
const simulation = await connection.simulateTransaction(transaction);
console.log("Error:", simulation.value.err);
console.log("Logs:", simulation.value.logs);

// Fix issues based on logs
```

Performance Problems

Issue: "High compute unit usage"

Problem:
Program uses too many compute units.

Solutions:

1. **Optimize algorithms:**
   - Use O(n log n) instead of O(n²)
   - Cache expensive computations
   - Reduce loops
2. **Batch operations:**
   - Process multiple items efficiently
   - Use vectorized operations
3. **Set compute budget:**

   ```
   let compute_budget = ComputeBudgetInstruction::set_compute_unit_limit(400_000);
   ```

Issue: "Transaction too slow"

Problem:
Transactions take too long to confirm.

Solutions:

1. **Use priority fees:**

   ```
   let priority_fee = ComputeBudgetInstruction::set_compute_unit_price(1000);
   ```

2. **Optimize transaction size:**
   - Reduce number of accounts
   - Minimize instruction data
3. **Use faster RPC:**
   - Use dedicated RPC endpoint
   - Consider private RPC

Debugging Strategies

Strategy 1: Check Logs

```
// Add debug logs
msg!("Debug: value = {}", value);
msg!("Debug: account = {}", account.key());

// View logs
solana logs
```

Strategy 2: Inspect Accounts

```
# View account data
solana account <PUBKEY>

# Decode account
anchor account <ACCOUNT_NAME> --program-id <PROGRAM_ID>
```

Strategy 3: Simulate Transactions

```
// Simulate before sending
const simulation = await connection.simulateTransaction(transaction);
console.log("Compute units:", simulation.value.unitsConsumed);
console.log("Logs:", simulation.value.logs);
```

Strategy 4: Use Explorer

View Transaction:

- Solana Explorer: https://explorer.solana.com/tx/
- Solscan: https://solscan.io/tx/

View Account:

- Solana Explorer: https://explorer.solana.com/address/
- Solscan: https://solscan.io/account/

Getting Help

Where to Ask

1. **GoDark Team:**
   - Component lead
   - Technical lead
   - Discord channel
2. **Community:**
   - Solana Discord
   - Anchor Discord
   - Stack Overflow

What to Provide

When asking for help, include:

1. **Error message**: Full error text
2. **Code snippet**: Relevant code

3. **Steps to reproduce**: How to trigger the issue
4. **Environment**: Localnet/devnet/mainnet
5. **Logs**: Transaction logs or program logs
6. **Account data**: Relevant account pubkeys

Example:

````
Error: Account not found

Code:
```rust
let account = program.account.state.fetch(state_pda).await?;
````

Steps:

1. Deploy program
2. Call initialize
3. Call instruction that uses account

Environment: Localnet
Logs: [paste logs]
Account:

````


---

## Common Error Codes

### Solana Program Errors

```rust
ProgramError::InsufficientFunds
ProgramError::InvalidAccountData
ProgramError::InvalidAccountOwner
ProgramError::AccountNotInitialized
ProgramError::ArithmeticOverflow
ProgramError::InsufficientFundsForFee
ProgramError::InvalidInstructionData
ProgramError::InvalidAccountData
ProgramError::AccountDataTooSmall
ProgramError::AccountNotExecutable
ProgramError::AccountBorrowFailed
ProgramError::AccountBorrowOutstanding
ProgramError::DuplicateInstruction
ProgramError::ExecutableDataModified
ProgramError::ExecutableLamportChange
ProgramError::ExecutableAccountNotRentExempt
ProgramError::UnbalancedInstruction
ProgramError::ModifiedProgramId
ProgramError::ExternalAccountLamportSpend
ProgramError::ExternalAccountDataModified
ProgramError::ReadonlyLamportChange
ProgramError::ReadonlyDataModified
ProgramError::DuplicateAccountIndex
ProgramError::ExecutableAccountNotExecutable
ProgramError::RentEpochModified
ProgramError::NotEnoughAccountKeys
ProgramError::AccountDataSizeChanged
ProgramError::AccountNotEnoughKeys
ProgramError::AccountNotEnoughKeys
ProgramError::AccountNotEnoughKeys
````

Key Takeaways

1. **Check Basics**: RPC, balance, account existence
2. **Verify Ownership**: Account ownership and authority
3. **Validate Inputs**: Size, format, constraints
4. **Use Logs**: Debug with `msg!` macro
5. **Inspect Accounts**: Use `solana account` command

6. **Simulate First**: Test before sending
7. **Ask for Help**: Provide context and logs

Next Steps

- Refer to specific guides for detailed explanations
- Practice debugging with these strategies
- Keep this guide handy for quick reference

Last Updated: November 2025