DEX: AMM vs Orderbook Protocols

High-Level Overview: On Solana, a decentralized exchange (DEX) is implemented as an on-chain program that manages token trading without a central intermediary. Traders interact by sending transactions to the DEX program, which updates the program's accounts according to the protocol's rules. Two dominant models exist: **Automated Market Makers (AMMs)** and **Orderbook-based DEXes**. An AMM maintains a liquidity pool (with two token reserves) and uses a formula (such as the constant-product model) to price trades. In contrast, an orderbook DEX maintains a centralized limit order book where users post bids (buy) and asks (sell) at specified prices; a matching engine then pairs compatible orders.

- **AMM model:** Liquidity providers deposit token pairs into a pool. Prices are determined algorithmically by the pool's reserve balances. Traders swap directly against the pool. Commonly, a constant-product curve ($x \cdot y = k$) is used, so prices adjust according to supply and demand. AMMs are simple, always-liquid markets (if any liquidity exists), but price impact grows with trade size.
- **Orderbook model:** The DEX keeps a sorted list of active buy/sell orders. A trade (take order) is executed by matching it against the best available opposite orders. This allows fine-grained price setting (limit orders) and can support larger, more controlled trades if depth exists, but relies on a matching engine and tends to require more accounts and logic than an AMM.

Both models use multiple Solana accounts (often Program-Derived Addresses, PDAs) to store state. The AMM will have pool accounts holding reserves and LP shares; the orderbook DEX will have market accounts, vaults for token custody, and data structures for bids/asks. On Solana, programs use Anchor for convenience: each instruction has a context struct (`#[derive(Accounts)]`) declaring required accounts. The network's high throughput and account-centric design allow these protocols to process many orders or swaps quickly, but also introduce concurrency concerns (transactions touching the same accounts cannot parallelize).

AMM vs Orderbook: In summary, AMMs provide always-on liquidity at algorithmic prices and simpler UX (swap between tokens), whereas orderbook DEXes offer traditional price discovery via limit orders and often tighter spreads with deep liquidity (when available). AMMs typically fit venues where constant liquidity is needed (small trades, many users), while orderbooks suit professional trading (precise order control, large trades). Both face trade-offs in complexity and risks, but understanding their core mechanics is essential for mastery.

Deep Dive on Core Mechanics

AMM-based DEX Mechanics

Data Structures and Accounts: An AMM typically has a *Pool* account describing a trading pair and two *vault* token accounts holding the actual reserves. In Anchor, we might define:

```
#[account]
pub struct Pool {
    pub token_a_mint: Pubkey,
    pub token_b_mint: Pubkey,
    pub reserve_a: u64,
    pub reserve_b: u64,
    pub lp_token_supply: u64,
    pub fee_bps: u16, // fee in basis points
    // (plus any padding or versioning fields)
}
```

Here `reserve_a` and `reserve_b` track how many of each token the pool holds. The `lp_token_supply` tracks how many liquidity provider (LP) share tokens have been minted (representing total pool liquidity). The pool also includes parameters (like fee rate). The actual token balances live in two SPL Token accounts (the *vaults*), which are often PDAs owned by the program. Those vault accounts are referenced by their Pubkeys stored in the Pool or derived from seeds. Liquidity providers deposit into these vaults and receive LP tokens in return.

Initializing the Pool: The DEX program would provide an instruction to create a new pool. In Anchor, one could write:

```rust
#[derive(Accounts)]
pub struct InitializePool<'info> {
    #[account(init, payer = user, space = 8 + Pool::LEN)]
    pub pool: Account<'info, Pool>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
    pub token_program: Program<'info, Token>,
}

pub fn initialize_pool(
    ctx: Context<InitializePool>,
    token_a_mint: Pubkey,
    token_b_mint: Pubkey,
    fee_bps: u16,
) -> Result<()> {
    let pool = &mut ctx.accounts.pool;
    pool.token_a_mint = token_a_mint;
    pool.token_b_mint = token_b_mint;
    pool.reserve_a = 0;
    pool.reserve_b = 0;
    pool.lp_token_supply = 0;
    pool.fee_bps = fee_bps;
    // (In practice, also initialize vault PDAs here with associated token accounts)
    Ok(())
}
```

This handler creates the Pool account (a PDA or simple account) with space for its fields, sets the token mints, zeroes the reserves, and stores the fee. The actual token vaults would be created (often as PDAs) to hold the liquidity, but for brevity we omit those steps. After initialization, LPs can deposit equal-value amounts of token A and B to mint LP tokens, and traders can swap.

Swap Instruction: To perform a trade (swap), an instruction moves tokens and updates reserves. A simplified anchor handler might be:

```rust
#[derive(Accounts)]
pub struct Swap<'info> {
    #[account(mut)]
    pub pool: Account<'info, Pool>,
    #[account(mut)]
    pub user_token_a: Account<'info, TokenAccount>,
    #[account(mut)]
    pub user_token_b: Account<'info, TokenAccount>,
    #[account(mut)]
    pub pool_vault_a: Account<'info, TokenAccount>,
    #[account(mut)]
    pub pool_vault_b: Account<'info, TokenAccount>,
    pub token_program: Program<'info, Token>,
}

pub fn swap(
    ctx: Context<Swap>,
    amount_in: u64,
    min_amount_out: u64,
) -> Result<()> {
    let pool = &mut ctx.accounts.pool;
    // Compute output using constant-product: x*y = k
    // Let fee-adjusted input be
    let fee_multiplier = 10_000u64 - pool.fee_bps as u64;
    let amount_in_with_fee = amount_in * fee_multiplier / 10_000u64;
    // new_reserve_a = reserve_a + amount_in_with_fee
    // constant k = reserve_a * reserve_b (before trade)
    // output = reserve_b - (k / new_reserve_a)
    let new_reserve_a =
pool.reserve_a.checked_add(amount_in_with_fee).unwrap();
    let k = pool.reserve_a.checked_mul(pool.reserve_b).unwrap();
    let new_reserve_b = k.checked_div(new_reserve_a).unwrap();
    let amount_out = pool.reserve_b.checked_sub(new_reserve_b).unwrap();
    require!(amount_out >= min_amount_out, DEXError::SlippageExceeded);

    // Update on-chain reserves
    pool.reserve_a =
pool.reserve_a.checked_add(amount_in_with_fee).unwrap();
    pool.reserve_b = pool.reserve_b.checked_sub(amount_out).unwrap();

```

```
39      // Perform token transfers via SPL token CPI:
40      // Transfer user's token A into pool vault
41      // Transfer pool vault's token B to user
42      // (Using anchor_spl::token::transfer calls)
43      // e.g.,
44      // token::transfer(
45      //   CpiContext::new(ctx.accounts.token_program.to_account_info(),
    Transfer {
46      //       from: ctx.accounts.user_token_a.to_account_info(),
47      //       to: ctx.accounts.pool_vault_a.to_account_info(),
48      //       authority: ctx.accounts.user.to_account_info(),
49      //   }),
50      //   amount_in,
51      // )?;
52      //
53      // token::transfer(
54      //
    CpiContext::new_with_signer(ctx.accounts.token_program.to_account_info()
    Transfer {
55      //       from: ctx.accounts.pool_vault_b.to_account_info(),
56      //       to: ctx.accounts.user_token_b.to_account_info(),
57      //       authority: ctx.accounts.pool.to_account_info(),
58      //   }, &[&pool_signer_seeds[..]]),
59      //   amount_out,
60      // )?;
61      //
62      Ok(())
63  }
```

This snippet sketches the core logic: it applies a fee, enforces the constant-product invariant, and updates reserves. In practice the token transfers use CPIs to the SPL Token program (as shown in comments). After this instruction, the pool's reserves reflect the new balances, and the user has effectively swapped `amount_in` of token A for `amount_out` of token B.

AMM Curve Math: The key formula is the *constant-product invariant*: if `x = reserve_a` and `y = reserve_b`, then `x · y = k` stays constant (ignoring fees). For an incoming trade of `Δx`, the new reserves `(x+Δx')` and `(y-Δy)` satisfy `(x+Δx')*(y-Δy) = k`, solving for output `Δy`. The simplest closed-form (no fee) is `Δy = (Δx * y) / (x + Δx)`. With fees, we reduce the effective `Δx` by `(1 - fee)`. This model ensures *price*

slippage: larger `Δx` trades buy increasingly less per unit token, pushing the pool price. Liquidity providers should note *impermanent loss*: if prices diverge, LPs end up with less value than holding tokens, but they earn fees as compensation.

Orderbook-based DEX Mechanics

Data Structures and Accounts: An orderbook DEX uses several accounts per market (trading pair). The core is a `Market` account storing global parameters and pointers to sub-accounts:

```
#[account]
pub struct Market {
    pub base_mint: Pubkey,
    pub quote_mint: Pubkey,
    pub base_vault: Pubkey,
    pub quote_vault: Pubkey,
    pub bids: Pubkey,
    pub asks: Pubkey,
    pub request_queue: Pubkey,
    pub event_queue: Pubkey,
    pub lot_size: u64,
    pub tick_size: u64,
    pub fee_rate_bps: u16,
    // plus padding/version fields
}
```

- **Vaults:** `base_vault` and `quote_vault` are SPL Token accounts (usually PDAs) that hold the tokens for this market.
- **Orderbooks:** The `bids` and `asks` accounts represent the sorted order books (for buy and sell orders, respectively). Each might be a custom data structure (often a "slab" or tree) storing orders by price/time.
- **Queues:** Some implementations (like Serum/Openbook) use a `request_queue` for incoming orders and an `event_queue` to log fills, but we can abstract these.
- **OpenOrders:** Each user has an *OpenOrders* account for this market. It tracks that user's active orders and locked funds:

```rust
#[account]
pub struct OpenOrders {
    pub owner: Pubkey,
    pub market: Pubkey,
    pub base_free: u64,
    pub quote_free: u64,
    pub bids: [Option<Order>; MAX_ORDERS],
    pub asks: [Option<Order>; MAX_ORDERS],
}
```

(Here `MAX_ORDERS` is a fixed slot count, and `Order` holds a price and remaining quantity.) The OpenOrders account also tracks how much of each token is reserved to back the orders. When a user places a buy order, their quote tokens move from `quote_free` to locked; similarly for sell orders with base tokens.

Initializing a Market: An instruction sets up the market and its supporting accounts. In Anchor:

```rust
#[derive(Accounts)]
pub struct InitializeMarket<'info> {
    #[account(init, payer = user, space = 8 + Market::LEN)]
    pub market: Account<'info, Market>,
    #[account(init, payer = user, space = 8 + Orderbook::LEN)]
    pub bids: Account<'info, Orderbook>,
    #[account(init, payer = user, space = 8 + Orderbook::LEN)]
    pub asks: Account<'info, Orderbook>,
    #[account(init, payer = user, space = 8 + RequestQueue::LEN)]
    pub request_queue: Account<'info, RequestQueue>,
    #[account(init, payer = user, space = 8 + EventQueue::LEN)]
    pub event_queue: Account<'info, EventQueue>,
    #[account(mut)]
    pub base_vault: Signer<'info>,  // typically created externally as a PDA
    #[account(mut)]
    pub quote_vault: Signer<'info>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}

pub fn initialize_market(
    ctx: Context<InitializeMarket>,
    base_mint: Pubkey,
    quote_mint: Pubkey,
    lot_size: u64,
    tick_size: u64,
    fee_rate_bps: u16,
) -> Result<()> {
    let market = &mut ctx.accounts.market;
    market.base_mint = base_mint;
    market.quote_mint = quote_mint;
    market.base_vault = ctx.accounts.base_vault.key();
    market.quote_vault = ctx.accounts.quote_vault.key();
    market.bids = ctx.accounts.bids.key();
    market.asks = ctx.accounts.asks.key();
    market.request_queue = ctx.accounts.request_queue.key();
    market.event_queue = ctx.accounts.event_queue.key();
```

```
        market.lot_size = lot_size;
        market.tick_size = tick_size;
        market.fee_rate_bps = fee_rate_bps;
        Ok(())
    }
```

This sets up the market's parameters and links the sub-accounts. (In practice, the vaults would be created via associated token account or as PDAs; here we assume they are provided.) Now the market is live, with empty order books and zeroed accounts ready to receive orders.

Placing Orders: A trader submits a limit order with side (bid or ask), price, and quantity. The DEX matches it against the best orders on the opposite side. For example:

```rust
#[derive(Accounts)]
pub struct PlaceOrder<'info> {
    #[account(mut)]
    pub market: Account<'info, Market>,
    #[account(mut)]
    pub bids: Account<'info, Orderbook>,
    #[account(mut)]
    pub asks: Account<'info, Orderbook>,
    #[account(mut)]
    pub user_open_orders: Account<'info, OpenOrders>,
    #[account(mut)]
    pub user_base: Account<'info, TokenAccount>,
    #[account(mut)]
    pub user_quote: Account<'info, TokenAccount>,
    pub token_program: Program<'info, Token>,
}

pub fn place_order(
    ctx: Context<PlaceOrder>,
    side: Side,         // enum: Bid or Ask
    price: u64,
    quantity: u64,
) -> Result<()> {
    let market = &ctx.accounts.market;
    let mut remaining = quantity;
    // Pseudocode for matching:
    if side == Side::Bid {
        // Try matching with lowest ask prices
        while remaining > 0 {
            if let Some(mut best_ask) = ctx.accounts.asks.pop_best() {
                // If best ask price <= our bid price, trade
                if best_ask.price <= price {
                    let trade_qty = remaining.min(best_ask.quantity);
                    remaining -= trade_qty;
                    // Execute trade: update user balances and
OpenOrders
                    // (transfer tokens from vaults to user and vice
versa)
                    best_ask.quantity -= trade_qty;
                    // If ask fully filled, do not reinsert it
```

```
                    if best_ask.quantity > 0 {
                        ctx.accounts.asks.insert(best_ask);
                    }
                } else {
                    // No more matches
                    ctx.accounts.asks.insert(best_ask);
                    break;
                }
            } else {
                break;
            }
        }
        // If any quantity remains, add to bids orderbook
        if remaining > 0 {
            ctx.accounts.bids.insert(Order { price, quantity: remaining
});
            // Lock user's quote tokens accordingly
        }
    } else {
        // Similar logic for a sell order (ask side), matching highest
bids first
        while remaining > 0 {
            if let Some(mut best_bid) = ctx.accounts.bids.pop_best() {
                if best_bid.price >= price {
                    let trade_qty = remaining.min(best_bid.quantity);
                    remaining -= trade_qty;
                    best_bid.quantity -= trade_qty;
                    if best_bid.quantity > 0 {
                        ctx.accounts.bids.insert(best_bid);
                    }
                } else {
                    ctx.accounts.bids.insert(best_bid);
                    break;
                }
            } else {
                break;
            }
        }
        if remaining > 0 {
```

```
            ctx.accounts.asks.insert(Order { price, quantity: remaining
});
            // Lock user's base tokens accordingly
        }
    }
    Ok(())
}
```

This pseudocode shows the core idea: for a buy order (`Bid`), repeatedly take the best ask (lowest price) and match as much as possible until the incoming order is filled or no more asks at or below the bid price. Any unfilled remainder is placed into the bids book. (And symmetrically for sell orders.) In reality, each match would involve transferring tokens: the buyer's quote tokens go into the `quote_vault` and the seller's base tokens go into the `base_vault` . The matched quantities are deducted from the respective OpenOrders accounts. Partially filled orders are updated or reinserted; fully filled orders are removed from the book. The program might also push fill events into an event queue for clients to consume.

Cancelling Orders: To cancel, the user specifies an order (often by ID or by tracking it externally). The instruction would remove it from the bids or asks structure and refund the locked funds back to the user. For example:

```rust
#[derive(Accounts)]
pub struct CancelOrder<'info> {
    #[account(mut)]
    pub market: Account<'info, Market>,
    #[account(mut)]
    pub bids: Account<'info, Orderbook>,
    #[account(mut)]
    pub asks: Account<'info, Orderbook>,
    #[account(mut)]
    pub user_open_orders: Account<'info, OpenOrders>,
}

pub fn cancel_order(
    ctx: Context<CancelOrder>,
    order_id: u128,
    side: Side,
) -> Result<()> {
    if side == Side::Bid {
        // Remove from bids and unlock quote tokens
        ctx.accounts.bids.remove_order(order_id)?;
        // Credit user's quote free balance
    } else {
        // Remove from asks and unlock base tokens
        ctx.accounts.asks.remove_order(order_id)?;
        // Credit user's base free balance
    }
    Ok(())
}
```

This ensures that the order is taken out of the matching engine's consideration and the user's funds are available again. (OpenOrders typically tracks which orders belong to the user so it can safely refund them.)

Matching Engine Basics: Under the hood, the DEX's matching engine follows *price-time priority*. At any moment, the best bid is the highest buy price; the best ask is the lowest sell price. When a new order arrives, the engine checks for crossings: a buy order with price ≥ best ask (or a sell with price ≤ best bid) triggers a trade. The trade price is typically the price of the resting order (or could be midpoint, but usually taker takes maker's price). The engine iteratively fills the smaller of the two order sizes, updates both orders (or removes the filled one), and continues until no more crossing or the incoming

order is fully satisfied. A rough pseudocode loop is above. The result is a sequence of partial or full fills, with any leftovers placed into the book as a new order. This all happens in one transaction so it's atomic: either everything (matching and fund transfers) goes through or none.

Bug/Vulnerability Checklist (DEX-specific)

- **AMM DEX vulnerabilities:**
  - *Front-running & Sandwich Attacks:* Since trades are public on-chain, bots can detect a pending large swap and preempt it by placing their own trades. They may "sandwich" a user's order (buy before, sell after) to profit from the slippage the user causes.
  - *High Slippage and Liquidity Manipulation:* Large trades move the pool price significantly. An attacker can repeatedly swap in and out to temporarily skew the pool's price, potentially manipulating or draining liquidity if fees/oracles are not robust.
  - *Impermanent Loss Exploits:* LPs suffer impermanent loss when token prices diverge. While not a coding bug, any exploit that tricks LPs into locking tokens at unfavorable ratios (e.g. flash loans forcing pool imbalance) is a risk.
  - *Arithmetic or Rounding Errors:* Incorrect handling of integer math (overflow, underflow, or rounding) in the swap formula can be exploited (for example, causing 0 output or negative balances). Careful use of checked arithmetic is essential.
  - *Fee Misconfiguration:* Bugs in applying or collecting fees (e.g. forgetting to deduct the fee from the input) can drain the pool's treasury or give users unintended gain.
  - *Oracle/External Price Dependency:* Some AMMs (or adaptive curves) rely on an external price feed. If that oracle is manipulated, attackers can trade against the skewed price, extracting value.
- **Orderbook DEX vulnerabilities:**
  - *Front-running & Priority Fees:* On Solana, users can attach additional compute-unit or priority fees to prioritize transactions. Malicious actors can still front-run orders (by paying to be processed earlier) or re-order transactions in a block.
  - *Spoofing/Layering Attacks:* Traders might place large fake orders (that they intend to cancel) to mislead others about supply/demand, then cancel them, leaving genuine traders at a disadvantage. While not a protocol bug per se, the DEX must handle rapid cancellations gracefully.
  - *Incorrect Matching Engine State:* Bugs in the matching logic (e.g. not removing a filled order, double-filling, or mis-updating quantities) can lead to inconsistent state. For example, an order might remain in the book after cancellation if not properly cleared, causing "ghost" liquidity.

- *Order Slot Exhaustion:* If an OpenOrders account has limited slots, a user might exhaust them (all slots used) and be unable to place new orders, leading to funds locked unexpectedly. Handling of full order-books or full user-limits must be robust.
- *Stale Orders / Event Queue Overflow:* If the event queue or order lists grow too large (or aren't cranked/processed), orders might be matched later than expected or never settled. This could lock up funds or cause missed trades.
- *Price Tick Constraints:* Incorrect handling of tick or lot sizes (e.g. allowing a price that isn't a multiple of the tick) could break matching or cause invalid orders.
- *Thin Liquidity Risk:* Even without direct bugs, a shallow orderbook means a single large order can sweep many price levels (extreme price impact). Traders and the protocol must guard (via checks like depth or limit orders) to mitigate sudden crashes or "flash crashes."

Lending & Borrowing Protocol on Solana (Anchor & Rust)

Overview

A **Lending & Borrowing (Lending/Borrowing) protocol** is a decentralized finance (DeFi) application that enables users to deposit crypto assets as collateral and take out loans in other assets. Depositors earn interest on their supplied assets, while borrowers can access liquidity without selling their holdings. The protocol enforces over-collateralization and liquidation rules to manage risk. In general operation:

- **Depositors** supply tokens into a reserve (liquidity pool) and earn interest. Their deposit is tracked, often via a "collateral token" representation.
- **Borrowers** lock collateral and receive a loan from the reserve up to a maximum Loan-to-Value (LTV) ratio. Borrowed positions accrue interest.
- **Repayments** reduce outstanding debt, returning collateral to the borrower when repaid. If collateral value falls below a threshold, liquidations can occur.

Building this on Solana with Anchor involves defining **on-chain accounts** (e.g., *Reserve*, *Obligation*, *Vaults*) and **instructions** (Deposit, Withdraw, Borrow, Repay) that update those accounts. Anchor's account structs and CPI (cross-program invocation) utilities make it easier to handle account validation and SPL token transfers.

Below we provide an in-depth, code-centric guide to a Lending/Borrowing protocol on Solana using Anchor. Each core component is illustrated with Rust/Anchor code and carefully explained.

Core Structs

The protocol's state is stored in on-chain accounts. Key structs include **Reserve** (the lending pool) and **Obligation** (a user's position). We also manage **token vaults** as SPL token accounts under program control.

Reserve

The `Reserve` struct represents a lending pool for a specific token pair (liquidity token and collateral token). It holds configuration, interest rates, and references to vaults. For example:

```
#[account]
pub struct Reserve {
    // PDA seed bump for this reserve
    pub bump: u8,

    // The token mint for liquidity (the asset people borrow/repay)
    pub liquidity_mint: Pubkey,
    // The token mint for collateral (the asset deposited by borrowers)
    pub collateral_mint: Pubkey,

    // Vault to hold deposited collateral tokens (program-owned token account)
    pub collateral_vault: Pubkey,
    // Vault to hold available liquidity for borrowing (program-owned token account)
    pub liquidity_vault: Pubkey,

    // Total amount of collateral deposited (in collateral token units)
    pub total_collateral: u64,
    // Total amount of liquidity borrowed (in liquidity token units)
    pub total_borrowed: u64,

    // Interest rate parameters (example linear model)
    pub interest_rate_per_slot: u64,  // interest accrued per slot on borrowed amount
    pub last_update_slot: u64,        // slot when interest was last accrued
}
```

This struct uses `#[account]` so Anchor knows to store it in a Solana account. Each field means:

- `liquidity_mint` **and** `collateral_mint` : The SPL token mints involved. Borrowers put up `collateral_mint` tokens to borrow `liquidity_mint` .
- `collateral_vault` **and** `liquidity_vault` : These are PDA (program-derived) accounts that hold the actual tokens. The program owns these token accounts, so it can safely control deposits and withdrawals.
- `total_collateral` **and** `total_borrowed` : Track aggregate amounts, used for interest calculations and ensuring solvency.
- **Interest fields** (`interest_rate_per_slot` , `last_update_slot`): Simplified interest accrual model. Each slot, borrowed amounts accrue a small interest. In real protocols, interest might be more complex (e.g., variable rates based on utilization).

Whenever users interact (deposit/borrow/etc.), the program will update `total_collateral` or `total_borrowed` and accrue interest based on the elapsed slots since `last_update_slot` .

Obligation

An `Obligation` represents a user's debt position within a reserve. It tracks how much the user has borrowed and how much collateral they supplied:

```
#[account]
pub struct Obligation {
    // Owner of this obligation (the borrower)
    pub owner: Pubkey,

    // The Reserve this obligation is for
    pub reserve: Pubkey,

    // User's deposited collateral amount (in collateral token units)
    pub collateral_amount: u64,
    // User's borrowed amount (in liquidity token units)
    pub borrowed_amount: u64,

    // Last recorded slot for interest accrual on this obligation
    pub last_update_slot: u64,
}
```

Key fields explained:

- `owner`: The wallet that owns this obligation. Only the owner can repay or withdraw collateral.
- `reserve`: Pointer to the `Reserve` account this obligation is associated with. Helps link user positions to the correct pool.
- `collateral_amount` and `borrowed_amount`: The user's current locked collateral and outstanding debt, respectively.
- `last_update_slot`: We may accrue interest per-obligation; this field tracks when the obligation was last updated to apply interest on `borrowed_amount`.

The obligation account will be initialized (using Anchor's `init`) when a user first deposits collateral or borrows. Later, instructions update it.

Collateral and Liquidity Vaults

Although not new structs, it's important to note the vault accounts:

- **Collateral Vault** (`Account<'info, TokenAccount>`): Holds all collateral tokens deposited to this reserve. Mint authority belongs to nobody; the program simply holds tokens.
- **Liquidity Vault** (`Account<'info, TokenAccount>`): Holds all free liquidity available to borrowers. When borrowers take out loans, tokens move from this vault to the user. When loans are repaid, tokens return here.

These vaults are created with specific PDAs using seed arrays (e.g., `[b"collateral-vault", reserve.key().as_ref()]`) and are marked `mut` in instructions to allow transfers.

Core Instructions

The main instruction handlers implement deposit, withdraw, borrow, and repay. Each uses Anchor's `#[derive(Accounts)]` for account validation. We illustrate simplified versions of these instructions, focusing on logic.

Initialize Reserve (Setup)

Before users can borrow, an admin or governance instruction creates a `Reserve` and its vaults. Example:

```rust
#[derive(Accounts)]
pub struct InitializeReserve<'info> {
    #[account(init, payer = authority, space = 8 + std::mem::size_of::<Reserve>(),
              seeds = [b"reserve", liquidity_mint.key().as_ref(), collateral_mint.key().as_ref()],
              bump)]
    pub reserve: Account<'info, Reserve>,

    #[account(
        init, payer = authority,
        token::mint = collateral_mint, token::authority = reserve,
        seeds = [b"collateral-vault", reserve.key().as_ref()],
        bump)]
    pub collateral_vault: Account<'info, TokenAccount>,

    #[account(
        init, payer = authority,
        token::mint = liquidity_mint, token::authority = reserve,
        seeds = [b"liquidity-vault", reserve.key().as_ref()],
        bump)]
    pub liquidity_vault: Account<'info, TokenAccount>,

    pub liquidity_mint: Account<'info, Mint>,
    pub collateral_mint: Account<'info, Mint>,

    #[account(mut)]
    pub authority: Signer<'info>,      // payer for account creation
    pub rent: Sysvar<'info, Rent>,
    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
}

pub fn initialize_reserve(ctx: Context<InitializeReserve>, interest_rate_per_slot: u64) -> ProgramResult {
    let reserve = &mut ctx.accounts.reserve;
    reserve.bump = *ctx.bumps.get("reserve").unwrap();
    reserve.liquidity_mint = ctx.accounts.liquidity_mint.key();
    reserve.collateral_mint = ctx.accounts.collateral_mint.key();
    reserve.collateral_vault = ctx.accounts.collateral_vault.key();
```

```
        reserve.liquidity_vault = ctx.accounts.liquidity_vault.key();
        reserve.total_collateral = 0;
        reserve.total_borrowed = 0;
        reserve.interest_rate_per_slot = interest_rate_per_slot;
        reserve.last_update_slot = Clock::get()?.slot;
        Ok(())
    }
```

Explanation: This `initialize_reserve` instruction sets up a new lending pool. Anchor's account constraints:

- It **inits** `Reserve`, `CollateralVault`, and `LiquidityVault` accounts with PDAs (using `seeds` and `bump`).
- `token::mint = ...` and `token::authority = reserve` ensure the vaults are SPL token accounts minted by the given `Mint` and owned by the reserve's PDA.
- We store initial values (zero collateral/borrowed) and the interest rate. The vaults start empty and owned by the reserve.

Deposit (Supply Collateral)

When a user wants to use assets as collateral (or simply deposit liquidity), they call **Deposit**. This moves tokens from the user into the reserve and updates their obligation. Example:

```rust
#[derive(Accounts)]
pub struct Deposit<'info> {
    // The lending reserve
    #[account(mut)]
    pub reserve: Account<'info, Reserve>,
    // Collateral token vault owned by the program
    #[account(mut, has_one = reserve)]
    pub collateral_vault: Account<'info, TokenAccount>,

    // The user depositing collateral
    #[account(mut)]
    pub user: Signer<'info>,
    // The user's token account for the collateral mint
    #[account(mut)]
    pub user_collateral_account: Account<'info, TokenAccount>,

    // The user's obligation for this reserve (init if not exists)
    #[account(
        init_if_needed,
        payer = user,
        space = 8 + std::mem::size_of::<Obligation>(),
        seeds = [b"obligation", user.key().as_ref(),
reserve.key().as_ref()],
        bump
    )]
    pub obligation: Account<'info, Obligation>,

    // Programs
    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}

pub fn deposit(ctx: Context<Deposit>, amount: u64) -> ProgramResult {
    let reserve = &mut ctx.accounts.reserve;
    let obligation = &mut ctx.accounts.obligation;

    // Transfer collateral from user to collateral vault
    let cpi_accounts = Transfer {
        from: ctx.accounts.user_collateral_account.to_account_info(),
```

```
            to: ctx.accounts.collateral_vault.to_account_info(),
            authority: ctx.accounts.user.to_account_info(),
        };
        token::transfer(ctx.accounts.token_program.to_account_info(),
    cpi_accounts, amount)?;

        // Update reserve and obligation state
        reserve.total_collateral =
    reserve.total_collateral.checked_add(amount)
            .ok_or(ProgramError::InvalidArgument)?;
        obligation.owner = ctx.accounts.user.key();
        obligation.reserve = reserve.key();
        obligation.collateral_amount =
    obligation.collateral_amount.checked_add(amount)
            .ok_or(ProgramError::InvalidArgument)?;
        obligation.last_update_slot = Clock::get()?.slot;

        Ok(())
    }
```

Explanation: In `deposit`:

- **Accounts:** We require mutable access to the `Reserve` and its `collateral_vault` (to increase balances), the user's SPL token account, and the user as signer. We also (potentially) initialize an `Obligation` account for the user/reserve if it doesn't exist.
- **Token Transfer:** Using Anchor's CPI `transfer`, we move `amount` of collateral tokens from the user's account to the reserve's `collateral_vault`. The `authority` is the user, since they signed.
- **State Update:** We increase `reserve.total_collateral` and the `obligation.collateral_amount` by `amount`. If the obligation was just created, its initial `collateral_amount` was 0. We also set the `owner` and `reserve` fields (or they would already be correct) and update `last_update_slot`. This ties the deposit to the user's obligation record.

Withdraw (Remove Collateral)

To remove deposited collateral (if no debt or still within LTV limits), the user calls **Withdraw**. It reverses `deposit`:

```rust
#[derive(Accounts)]
pub struct Withdraw<'info> {
    #[account(mut)]
    pub reserve: Account<'info, Reserve>,
    #[account(mut, has_one = reserve)]
    pub collateral_vault: Account<'info, TokenAccount>,

    // The user withdrawing collateral
    #[account(mut)]
    pub user: Signer<'info>,
    // The user's token account for the collateral mint
    #[account(mut)]
    pub user_collateral_account: Account<'info, TokenAccount>,

    // Existing obligation for this user
    #[account(mut, has_one = owner)]
    pub obligation: Account<'info, Obligation>,

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

pub fn withdraw(ctx: Context<Withdraw>, amount: u64) -> ProgramResult {
    let reserve = &mut ctx.accounts.reserve;
    let obligation = &mut ctx.accounts.obligation;

    // Ensure user can't withdraw more than they deposited (simple check)
    if obligation.collateral_amount < amount {
        return Err(ProgramError::InsufficientFunds);
    }
    // (Omit LTV check for brevity; real code should ensure collateral after withdrawal still covers any debt.)

    // Transfer collateral from vault back to user
    let cpi_accounts = Transfer {
        from: ctx.accounts.collateral_vault.to_account_info(),
        to: ctx.accounts.user_collateral_account.to_account_info(),
        authority: ctx.accounts.reserve.to_account_info(), // Reserve is authority of vault
    };
```

```
        // We sign on behalf of the PDA (reserve) using `seeds`
        let reserve_seeds = &[b"reserve", /* + other seeds as used above */, &[reserve.bump]];
        let signer = &[&reserve_seeds[..]];
        token::transfer(
            CpiContext::new_with_signer(
                ctx.accounts.token_program.to_account_info(),
                cpi_accounts,
                signer
            ),
            amount
        )?;

        // Update state
        reserve.total_collateral = reserve.total_collateral.checked_sub(amount)
            .ok_or(ProgramError::InvalidArgument)?;
        obligation.collateral_amount = obligation.collateral_amount.checked_sub(amount)
            .ok_or(ProgramError::InvalidArgument)?;
        obligation.last_update_slot = Clock::get()?.slot;

        Ok(())
    }
```

Explanation: The `withdraw` instruction:

- Checks the user has enough collateral deposited (`obligation.collateral_amount >= amount`). A real protocol would also check that after withdrawal, the remaining collateral still covers any borrowed amount (LTV check).
- Uses a CPI `token::transfer` from the `collateral_vault` back to the user's account. Since the vault's authority is the `reserve` PDA, we use `new_with_signer` and provide the seeds/bump so the program can sign as the vault's owner.
- Decrements `reserve.total_collateral` and `obligation.collateral_amount` by `amount` .

Borrow

Borrowing requires the user has collateral locked. The `borrow` instruction moves liquidity tokens from the reserve to the borrower and updates the obligation:

```rust
#[derive(Accounts)]
pub struct Borrow<'info> {
    #[account(mut)]
    pub reserve: Account<'info, Reserve>,
    #[account(mut, has_one = reserve)]
    pub liquidity_vault: Account<'info, TokenAccount>,

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

    #[account(mut, has_one = owner)]
    pub obligation: Account<'info, Obligation>,

    // (Optional) Price oracle for checking collateral value
    // pub price_oracle: AccountInfo<'info>,

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

pub fn borrow(ctx: Context<Borrow>, amount: u64) -> ProgramResult {
    let reserve = &mut ctx.accounts.reserve;
    let obligation = &mut ctx.accounts.obligation;

    // Simple LTV check: ensure collateral covers new loan (pseudocode)
    // let collateral_value = get_price_from_oracle(...) *
obligation.collateral_amount;
    // if collateral_value < (obligation.borrowed_amount + amount) *
MAX_LTV { ... }

    // Transfer liquidity from vault to user
    let cpi_accounts = Transfer {
        from: ctx.accounts.liquidity_vault.to_account_info(),
        to: ctx.accounts.user_liquidity_account.to_account_info(),
        authority: ctx.accounts.reserve.to_account_info(),
    };
    let reserve_seeds = &[b"reserve", /* seeds as before */, &
[reserve.bump]];
    token::transfer(
```

```
38          CpiContext::new_with_signer(
39              ctx.accounts.token_program.to_account_info(),
40              cpi_accounts,
41              &[&reserve_seeds[..]],
42          ),
43          amount
44      )?;
45
46      // Update state and accrue interest if needed
47      // (In a real protocol, we'd accrue interest first; skipping for
    brevity)
48      reserve.total_borrowed = reserve.total_borrowed.checked_add(amount)
49          .ok_or(ProgramError::InvalidArgument)?;
50      obligation.borrowed_amount =
    obligation.borrowed_amount.checked_add(amount)
51          .ok_or(ProgramError::InvalidArgument)?;
52      obligation.last_update_slot = Clock::get()?.slot;
53
54      Ok(())
55  }
```

Explanation: For `borrow`:

- We (simplistically) assume any necessary collateral checks pass. In practice, the code would fetch a price oracle to ensure `(collateral_value * LTV) >= total_borrowed_after`.
- Tokens are transferred from the `liquidity_vault` to the user's token account, signed by the reserve PDA.
- We increment both `reserve.total_borrowed` and `obligation.borrowed_amount`.
- Interest accrual would typically happen here (updating borrowed amount by interest since `last_update_slot`), but we omit that complexity for clarity.

Repay

When the borrower wants to repay, the `repay` instruction transfers tokens back and reduces debt:

```rust
#[derive(Accounts)]
pub struct Repay<'info> {
    #[account(mut)]
    pub reserve: Account<'info, Reserve>,
    #[account(mut, has_one = reserve)]
    pub liquidity_vault: Account<'info, TokenAccount>,

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

    #[account(mut, has_one = owner)]
    pub obligation: Account<'info, Obligation>,

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

pub fn repay(ctx: Context<Repay>, amount: u64) -> ProgramResult {
    let reserve = &mut ctx.accounts.reserve;
    let obligation = &mut ctx.accounts.obligation;

    // Ensure user has enough debt
    if obligation.borrowed_amount < amount {
        return Err(ProgramError::InvalidArgument);
    }

    // Transfer liquidity from user back to vault
    let cpi_accounts = Transfer {
        from: ctx.accounts.user_liquidity_account.to_account_info(),
        to: ctx.accounts.liquidity_vault.to_account_info(),
        authority: ctx.accounts.user.to_account_info(),
    };
    token::transfer(ctx.accounts.token_program.to_account_info(),
cpi_accounts, amount)?;

    // Update state
    reserve.total_borrowed = reserve.total_borrowed.checked_sub(amount)
        .ok_or(ProgramError::InvalidArgument)?;
```

```
39      obligation.borrowed_amount =
    obligation.borrowed_amount.checked_sub(amount)
40          .ok_or(ProgramError::InvalidArgument)?;
41      obligation.last_update_slot = Clock::get()?.slot;
42
43      Ok(())
44  }
```

Explanation: In `repay`:

- We first check that the user is not repaying more than they borrowed. (Overpayment logic could allow capping to `borrowed_amount`.)
- Tokens are moved from the user's account to the `liquidity_vault`.
- We decrement `reserve.total_borrowed` and `obligation.borrowed_amount`.

Associated Accounts & Relationships

A solid Lending/Borrowing program carefully manages account relationships and PDAs:

- **Reserve PDA:** The `Reserve` account itself is typically a PDA derived from known seeds (e.g., a literal string and the mint addresses). Its `bump` is stored in `reserve.bump` for signing CPIs.
- **Vault PDAs:** The `collateral_vault` and `liquidity_vault` token accounts are created as PDAs using seeds that include the `reserve` key. For example:

  ```
  #[account(
      init,
      seeds = [b"collateral-vault", reserve.key().as_ref()],
      bump,
      token::mint = collateral_mint, token::authority = reserve
  )]
  pub collateral_vault: Account<'info, TokenAccount>,
  ```

  This ensures only the program (via the reserve PDA) can control these vaults.

- **Obligation Account:** Each user's position is an `Obligation` account, also a PDA with seeds like `[b"obligation", user.key().as_ref(), reserve.key().as_ref()]`. This links the obligation uniquely to a (user, reserve) pair.
- **Account Constraints:** Anchor's `#[account(...)]` attributes express relationships: e.g., `has_one = reserve` asserts that the vault's `owner` is the reserve PDA, and `has_one = owner` on `Obligation` ensures the `owner` field matches `user.key()`.

- **Token Accounts:** The user's token accounts (for collateral or liquidity) are standard `Account<TokenAccount>`. They aren't created by the program; users must supply them. The program only transfers to/from them.

All interactions revolve around these accounts. For example, the deposit instruction requires the user's collateral token account and the program's collateral vault; the borrowing instruction uses the user's liquidity token account and the protocol's liquidity vault. Keeping track of which account owns or controls tokens is crucial.

Token & Collateral Mechanics

Understanding token flow is key:

- **Depositing Collateral:** When a user deposits collateral, their tokens move from a user-owned SPL token account into the program-controlled `collateral_vault`. The program notes this in `total_collateral` and the user's `Obligation`.
- **Earning Interest:** As borrowers accrue interest on loans, that interest ideally goes back to the liquidity pool. In a simple model, interest accrues to `reserve.total_borrowed` (making it larger), meaning borrowers owe more. This effectively increases the amount of liquidity (and value) in the pool, so future repayments are slightly higher.
- **Borrowing Liquidity:** Borrowers take tokens out of `liquidity_vault` and receive them in their own token account. This decreases `total_borrowed` inventory in the vault (tracked in state) and increases `obligation.borrowed_amount`.
- **Repaying Loans:** Repayment moves tokens from user to `liquidity_vault`, decreasing both `total_borrowed` and `obligation.borrowed_amount`.
- **Withdrawing Collateral:** Users reclaim collateral by transferring tokens from `collateral_vault` back to themselves, given they still meet collateral requirements.

The protocol must carefully adjust on-chain accounting to match these token transfers. Also, every token transfer is a CPI to the SPL Token program, so the program must supply the correct signer (usually a PDA) and have the correct account references in the `Context`.

Interest Accrual (Conceptual)

While the above code snippets use a very simple model, a more robust system updates interest on each action:

- Calculate how many slots (or time) have passed since `last_update_slot`.

- Increase `total_borrowed` by `(total_borrowed * interest_rate_per_slot * slots_elapsed)`.
- Likewise, update each `Obligation.borrowed_amount` or track a global index to apply proportional interest.
- This means depositors effectively earn interest from borrowers.

Implementing precise interest mechanisms requires careful handling of fixed-point math and update ordering, but the core idea is to apply interest whenever someone interacts with the pool.

Bug and Vulnerability Checklist

Building a lending protocol securely is challenging. Below are common pitfalls and vulnerabilities to watch for:

- **Interest Rate Calculation Bugs:** Ensure interest accrual logic is correct. Errors can come from integer overflow, wrong slot/time calculations, or forgetting to update state. Always use checked math (`checked_add`, `checked_mul`, etc.) and test long-term behavior.
- **Liquidation Logic Errors:** If a borrower's collateral drops in value, the protocol should allow liquidation to protect the pool. Mistakes here can let bad debt accumulate or unfairly penalize users. Validate oracle prices, liquidation thresholds, and ensure liquidator incentives are correct.
- **Collateral Mismanagement:** Check that collateral tokens always match `total_collateral`. A bug in transfer logic could let collateral be stolen or unrecoverable. Use `has_one` constraints and PDA authorities to prevent malicious transfers.
- **Flash Loan Attacks:** Attackers can take a very large short-term loan to manipulate prices or exploit code assumptions. Mitigation strategies include re-entrancy guards, validating invariant ratios after each instruction, and cautious use of oracles.
- **Price Oracle Manipulation:** If the protocol uses external price feeds for LTV checks, ensure oracles are reliable. Use time-weighted averages or multiple sources to avoid manipulation. Always assume an attacker may push a price temporarily.
- **Incorrect Reserve Ratio Enforcement:** The protocol must enforce a maximum borrow against collateral (LTV). Bugs that skip or miscalculate this check can allow under-collateralized loans, leading to insolvency when markets move.
- **Unchecked CPI Calls:** Verify all cross-program invocations `transfer` have the correct signers and accounts. An attacker might try to call an instruction with invalid accounts to trick the program into transferring tokens to themselves.

- **Reentrancy Risks:** Although Solana's model is usually single-instruction, be mindful of callback-like patterns (e.g., if calling other programs). Use `exec` or locks if performing multi-step operations.
- **Math Precision and Rounding:** Interest and ratios often need fixed-point math. Rounding errors could accumulate unfairly. Define a clear precision (e.g., use `U64F64` or similar) and consistently apply rounding rules.
- **Account Size and Rent:** Ensure accounts (like `Obligation`) have enough space for all fields. Running out of space can cause panic. Also, if using `init_if_needed`, remember to top-up rent or require user to do so.

Liquid Staking on Solana

Overview

Liquid staking lets users stake SOL for rewards **without locking up liquidity**. On Solana, normally staking SOL requires creating a stake account and delegating it to a validator, which then *locks* the SOL for an epoch-based activation/cooldown period. A liquid staking protocol wraps this flow by pooling many users' SOL, delegating it across validators, and issuing an SPL token (e.g. stSOL) that represents the user's share of all pooled stake. This derivative token can be freely traded or used in DeFi, even while the underlying SOL is locked earning rewards. Liquid staking is important on Solana because it maximizes network security (by encouraging staking) while preserving DeFi capital efficiency. Users earn validator rewards, and the circulating value of stSOL rises over time relative to SOL.

In practice, a Solana liquid staking protocol consists of:

- **Pool State Account:** A PDA (program-derived address) holds metadata like total staked SOL, total stSOL minted, fee parameters, and the list of active validators.
- **Stake Accounts:** On-chain stake accounts hold the actual SOL and are delegated to validators via the Solana Stake Program. The protocol may create one or more stake accounts per validator. Each new deposit typically spawns or funds a stake account.
- **Validator Vote Accounts:** The public keys of validators' vote accounts (selected by the protocol) are targets for delegation. The program may update these stakes to rebalance or add new stake over time.
- **Derivative Token Mint:** An SPL token mint (the "liquid staking token") whose supply tracks total stake. The program (via the pool PDA) is the mint authority. For example, 1 stSOL initially equals 1 SOL, and as the pool earns rewards, each stSOL can be redeemed for more SOL.

- **Liquidity Buffer (Optional):** Some protocols maintain a liquidity pool of SOL<>stSOL to allow instant redemptions. Users who want immediate SOL exchange stSOL via this pool (often paying a small fee), while liquidity providers earn fees. Alternatively, unstaking directly involves deactivating stake and waiting for the unbonding period.

When a user **deposits SOL**, the protocol: (1) collects their SOL into a new stake account `transfer followed by DelegateStake to a validator`, (2) updates its total staked SOL in the pool state, and (3) mints new stSOL to the user in proportion to the deposit (maintaining the invariant `total_stSOL : total_staked_SOL`). Similarly, when a user redeems stSOL, the program *burns* those tokens and returns SOL (either from a liquidity pool or after withdrawing stake). Over time, validator rewards accumulate in the stake accounts; these rewards are effectively shared by all stSOL holders (raising the SOL-per-stSOL price).

Solana's epoch delays mean that staking operations have time lag: a newly delegated stake takes ~2 epochs to activate and earn rewards, and a withdrawal requires deactivating and waiting ~2 epochs to unlock. The liquid staking program must manage these epochal state transitions (e.g. re-delegating stake from one validator to another by deactivating then delegating in new epoch) and ensure its accounting remains correct.

Key workflow steps in a liquid staking protocol:

- **Initialize Pool:** A pool state PDA and stSOL mint are created. A list of target validators is configured (often top-N by stake weight).
- **Deposit (Stake):** User sends SOL → new stake account is created and delegated → pool state "total_staked" increases → stSOL is minted to user.
- **Mint Derivative:** Using the SPL Token program, the program mints stSOL tokens to the user's token account under the pool authority.
- **Rewards Accrue:** Over epochs, the stake accounts grow in value due to rewards. The protocol may occasionally `refresh` or re-delegate stake to maintain balanced validator weights.
- **Redeem (Unstake):** User burns stSOL → program reduces total_staked, and returns SOL. This may happen via a liquidity pool swap (immediate) or by deactivating stake accounts and transferring unlocked SOL after cooldown.
- **Update Delegations:** The program periodically or on-demand calls update instructions to delegate more stake to validators or rebalance across them, using CPI into the Stake Program.

Each step involves careful state updates and CPI (cross-program invocations) to Solana's System or Stake programs and the SPL Token program. The pool PDA typically signs these operations (as stake authority or mint authority) using known seed.

Core Architecture and Components

A typical Solana liquid staking protocol uses the following components:

- **Pool State Account (PDA):** Stores configuration (fee rates, validator list), and accounting (total staked lamports, total liquid token supply). The PDA is often used as authority for minting/burning stSOL and for stake delegation.
- **Stake Accounts:** Standard Solana stake accounts (owned by the Stake program) hold the actual locked SOL. Each stake account is initialized and then delegated to a validator vote account. The program may use one stake account per validator or split across many for flexibility.
- **Validator List:** An on-chain list of validator vote account pubkeys. The program's logic (or governance) determines which validators to use. The program must guard this list (e.g. via admin or multisig) to prevent malicious validator additions.
- **Derivative Token Mint (stSOL):** An SPL token representing ownership of staked SOL. On deposit, new tokens are minted to match the deposited SOL (typically 1:1 to start). When total SOL in the pool grows (due to rewards), each token represents slightly more SOL over time. The pool PDA acts as mint authority, so only the program can mint or burn stSOL.
- **User Token Account:** A normal SPL Token Account for stSOL owned by the user. On deposit, the user's token account is credited with stSOL; on redeem, tokens are burned from it.
- **Liquidity Pool (Optional):** Many implementations include a SOL<>stSOL liquidity pool (often a simple AMM or a concentrated liquidity design). This allows *instant* swaps: a user who wants to exit immediately pays a fee in stSOL to get SOL from the pool, and vice versa. The pool may gather fees to redistribute among liquidity providers or back to the protocol. If such a pool exists, it must be integrated carefully to preserve 1:1 price invariant (discounted by fees).

All SOL that users stake through the protocol resides in the stake accounts, not in the pool state account. The pool state only tracks balances for accounting and authority. The **Stake Program** handles the actual lockup logic: a stake must be activated in a warmup, or deactivated and withdrawn after an epoch delay. A well-designed protocol batches stakes and withdrawals to minimize overhead, often delegating to multiple validators to diversify and maximize uptime.

Importantly, the protocol enforces **fair accounting**: if a user stakes X SOL and receives Y stSOL, the invariant `Y = (X * total_stSOL) / total_staked_SOL_before` (or vice versa depending on convention) must hold. As rewards come in, *total_staked_SOL* increases (without minting new stSOL), so 1 stSOL becomes redeemable for more than 1 SOL. This mechanism must be coded precisely to avoid rounding errors or inflation bugs.

Below is a high-level view of the deposit and redeem flows:

- **Stake Deposit Flow:**
  1. User sends SOL to the program (via an instruction).
  2. Program creates (or funds) a stake account, transfers SOL into it, and delegates it to a chosen validator via the Stake Program CPI.
  3. Program updates `pool_state.total_staked` by the deposited amount.
  4. Program mints stSOL to the user using the SPL Token CPI, according to the current exchange rate.
- **Redeem (Liquid Unstake) Flow:**
  1. User calls redeem, specifying amount of stSOL to burn.
  2. Program burns that many stSOL from the user's account via SPL Token CPI.
  3. Program calculates how much SOL corresponds to the burned tokens (based on current pool balance).
  4. If a liquidity pool exists, SOL is transferred from the pool's reserve to the user (charging a fee that accrues to LPs or pool). If withdrawing directly, the program deactivates enough stake and, after epochs, withdraws SOL to a reserve account, then transfers to user.
  5. Program updates `pool_state.total_staked` and `total_stSOL` accordingly.
- **Validator Stake Update:**
  - Periodically (or on certain triggers), the program may reallocate stake. For example, it might run an `update_delegation` instruction that does a CPI into the Stake Program to delegate additional stake to a validator, or to split/merge stake accounts. This keeps stake distributed as intended and may claim staking rewards on behalf of the pool.

Overall, the architecture relies on careful use of Solana's **System Program** (for account creation and transfers), **Stake Program** (for delegation/withdrawal), and **Token Program** (for minting/burning stSOL).

Example Anchor Instructions

Below are illustrative Anchor-based code snippets for core instructions. These are simplified examples to show structure; a real implementation would include error checks and complete account validation.

Stake Deposit Instruction

When a user deposits SOL, we create/delegate a stake account and update the pool. In Anchor:

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

    /// Pool state PDA holding total_staked, total_supply, etc.
    #[account(mut)]
    pub pool_state: Account<'info, PoolState>,

    /// New stake account to be created and funded.
    #[account(init, payer = user, space = 200, owner = stake::ID)]
    pub stake_account: UncheckedAccount<'info>,

    /// Validator vote account to delegate to.
    /// CHECK: validation of correct vote account should be done in program logic.
    pub validator_vote: AccountInfo<'info>,

    pub stake_program: Program<'info, Stake>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}

pub fn stake_deposit(ctx: Context<StakeDeposit>, lamports: u64) -> ProgramResult {
    // Transfer SOL from user to the new stake account
    let ix = system_instruction::transfer(
        &ctx.accounts.user.key(),
        &ctx.accounts.stake_account.key(),
        lamports,
    );
    // User is signer, so we can invoke transfer
    invoke(&ix, &[
        ctx.accounts.user.to_account_info(),
        ctx.accounts.stake_account.to_account_info(),
        ctx.accounts.system_program.to_account_info(),
    ])?;

    // Delegate the stake account to the validator vote account.
    let delegate_ix = stake::instruction::delegate_stake(
```

```
            &ctx.accounts.stake_account.key(),
            &ctx.accounts.validator_vote.key,
            &ctx.accounts.pool_state.key(),
        );
        // The pool_state PDA is the stake authority (needs its bump for signing)
        invoke_signed(
            &delegate_ix,
            &[
                ctx.accounts.stake_account.to_account_info(),
                ctx.accounts.validator_vote.to_account_info(),
                ctx.accounts.pool_state.to_account_info(),
                ctx.accounts.rent.to_account_info(),
                ctx.accounts.system_program.to_account_info(),
            ],
            &[&[b"pool_state", &[ctx.accounts.pool_state.bump]]],
        )?;

        // Update pool accounting: increase total_staked by lamports deposited.
        ctx.accounts.pool_state.total_staked = ctx.accounts.pool_state
            .total_staked
            .checked_add(lamports)
            .unwrap();
        Ok(())
    }
```

Explanation: This instruction takes `lamports` (the amount of SOL) from the `user` and transfers it into a newly created `stake_account`. It then invokes the Solana Stake Program's `delegate_stake` to assign that stake account to `validator_vote`. The program uses the pool PDA (`pool_state`) as the stake authority (shown via `invoke_signed` with the PDA's seed). Finally, it updates the `total_staked` in the pool state. In a full implementation, one would also compute how many stSOL to mint at this point.

Mint Staking Token Instruction

After depositing SOL, the protocol mints derivative tokens (stSOL) to the user. For example:

```rust
#[derive(Accounts)]
pub struct MintToken<'info> {
    #[account(mut)]
    pub pool_state: Account<'info, PoolState>,      // authority for the mint
    #[account(mut)]
    pub mint: Account<'info, Mint>,                // the stSOL mint
    #[account(mut)]
    pub user_token_acc: Account<'info, TokenAccount>, // user's stSOL account
    pub token_program: Program<'info, Token>,
}

pub fn mint_staking_token(ctx: Context<MintToken>, amount: u64) -> ProgramResult {
    // Mint 'amount' of stSOL to the user's token account.
    let cpi_accounts = token::MintTo {
        mint: ctx.accounts.mint.to_account_info(),
        to: ctx.accounts.user_token_acc.to_account_info(),
        authority: ctx.accounts.pool_state.to_account_info(),
    };
    let cpi_ctx = CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        cpi_accounts,
    );
    token::mint_to(cpi_ctx, amount)?;
    Ok(())
}
```

Explanation: This Anchor instruction mints new tokens. Here the `pool_state` PDA is set up as the mint authority of the `mint` (the stSOL token). Calling `token::mint_to` with `amount` increases the user's `TokenAccount` balance by that many stSOL tokens. In practice, this would be called after updating `total_staked` so that the new tokens reflect the user's share of the updated stake pool. Note that the Anchor framework handles signing the mint operation with the PDA's seed.

Redeem (Liquid Unstake) Instruction

When a user wants to convert stSOL back to SOL, the program burns the tokens and transfers SOL from the pool. A simple example:

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

    #[account(mut)]
    pub pool_state: Account<'info, PoolState>,      // PDA (authority)
    #[account(mut)]
    pub mint: Account<'info, Mint>,                // stSOL mint
    #[account(mut)]
    pub user_token_acc: Account<'info, TokenAccount>, // user's stSOL balance

    /// Account holding SOL to distribute (could be a reserve or stake-derived account)
    #[account(mut)]
    pub pool_sol_account: UncheckedAccount<'info>,

    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
}

pub fn redeem(ctx: Context<Redeem>, amount: u64) -> ProgramResult {
    // Burn the user's stSOL tokens
    let burn_ctx = CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        token::Burn {
            mint: ctx.accounts.mint.to_account_info(),
            from: ctx.accounts.user_token_acc.to_account_info(),
            authority: ctx.accounts.pool_state.to_account_info(),
        },
    );
    token::burn(burn_ctx, amount)?;

    // Transfer SOL from the pool's SOL account to the user
    let ix = system_instruction::transfer(
        &ctx.accounts.pool_sol_account.key(),
        &ctx.accounts.user.key(),
        amount,
    );
```

```
        invoke(&ix, &[
            ctx.accounts.pool_sol_account.to_account_info(),
            ctx.accounts.user.to_account_info(),
            ctx.accounts.system_program.to_account_info(),
        ])?;

        // Update pool accounting (decrease total_staked)
        ctx.accounts.pool_state.total_staked = ctx.accounts.pool_state
            .total_staked
            .checked_sub(amount)
            .unwrap();
        Ok(())
    }
```

Explanation: This function first burns `amount` of stSOL from the user's token account, reducing the total token supply. It then sends `amount` lamports of SOL from a designated `pool_sol_account` to the user. In a real protocol, `pool_sol_account` might be a reserve of unlocked SOL or a special PDA that manages liquidity. After the transfer, the pool's `total_staked` is decreased by `amount` to reflect the withdrawal. If instant liquidity isn't available, similar logic could deactivate and withdraw stake from the stake accounts instead.

Update Validator Stakes Instruction

To rebalance or add stake to validators, the program invokes the Stake Program again. For example:

```rust
#[derive(Accounts)]
pub struct UpdateValidator<'info> {
    #[account(mut)]
    pub pool_state: Account<'info, PoolState>,      // PDA authority
    #[account(mut)]
    pub stake_account: UncheckedAccount<'info>,     // existing stake account
    /// CHECK: Validator vote account
    pub validator_vote: AccountInfo<'info>,
    pub stake_program: Program<'info, Stake>,
    pub clock: Sysvar<'info, Clock>,
    pub stake_history: Sysvar<'info, StakeHistory>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}

pub fn update_validator_stake(ctx: Context<UpdateValidator>) -> ProgramResult {
    // Delegate this stake account to the validator (re-delegation)
    let delegate_ix = stake::instruction::delegate_stake(
        &ctx.accounts.stake_account.key(),
        &ctx.accounts.validator_vote.key,
        &ctx.accounts.pool_state.key(),
    );
    invoke_signed(
        &delegate_ix,
        &[
            ctx.accounts.stake_account.to_account_info(),
            ctx.accounts.validator_vote.to_account_info(),
            ctx.accounts.pool_state.to_account_info(),
            ctx.accounts.clock.to_account_info(),
            ctx.accounts.stake_history.to_account_info(),
            ctx.accounts.rent.to_account_info(),
        ],
        &[&[b"pool_state", &[ctx.accounts.pool_state.bump]]],
    )?;
    Ok(())
}
```

Explanation: This instruction shows how to delegate (or redelegate) a stake account. The CPI to `delegate_stake` uses the `pool_state` PDA as authority. In practice, an update function might deactivate a stake from one validator and then call this to delegate it to another. It is also used when initially funding a stake account to a validator. The inclusion of `Clock` and `StakeHistory` sysvars satisfies the stake program's CPI requirements. Proper error handling would ensure stake accounts are active/inactive as needed before calling this.

Audit Checklist for Liquid Staking

When auditing a Solana liquid staking protocol, pay special attention to these common issues and vulnerabilities:

- **Accounting Consistency:** Ensure the math linking total staked SOL and total derivative token supply is correct. A small error in ratio calculations can allow minting more tokens than backing SOL (inflation) or locking up user funds. Check rounding and that *after every deposit or withdrawal*, `(total_staked_SOL * total_supply) == (old_total_staked + change) * (old_total_supply + minted/burned)`.
- **Unauthorized Mint/Burn:** Verify that only the pool PDA can mint or burn the staking token. If the mint authority is not securely controlled, an attacker could create free tokens. The Anchor accounts constraints should only allow pool_state (PDA) as authority.
- **Stake Activation/Withdrawal Race:** Staking on Solana has delays. The protocol must not allow instant withdrawal of stake before it's unlocked. Check that the code correctly deactivates and waits for epochs before moving funds, or properly charges higher fees for "instant" swaps via a liquidity pool. Race conditions where a user tries to withdraw twice before state updates can be dangerous.
- **Validator List Security:** If validators are chosen on-chain, ensure only authorized entities (e.g. a multisig) can change the validator list. A malicious validator could collude to divert or slashing-induce losses. The program should validate vote accounts (correct owner) and possibly enforce a minimum performance requirement off-chain.
- **Liquidity Pool Exploits:** If a liquidity pool is used for instant exits, audit that its pricing (based on stSOL price) is correct and cannot be drained by arbitrage. For example, ensure swap fees and pool invariants prevent someone from swapping a lot of stSOL to drag the price arbitrarily low. The pool should maintain 1:1 peg (minus fees) relative to on-chain stake value.
- **Impersonation or PDA Mis-sign:** All CPI calls (especially to the Stake program) rely on correct PDA seeds. A wrong seed or missing seed bump may allow an attacker to trick the contract. Verify seeds and bumps are correctly used in `invoke_signed`.

- **Stake Account Reuse or Duplication:** Ensure that stake accounts from other contexts cannot be wrongly deposited to inflate totals. For example, if the protocol allows depositing existing stake accounts, check that the owner of that account is indeed a user and that it's not already part of the pool.
- **Delayed Reward Credit:** The code must properly credit staking rewards to all stSOL holders by increasing the effective backing per token. Failing to update the price when rewards accrue could cause stale valuations. Check that after each epoch or after stake refresh, the new *total_staked_SOL* (including rewards) is reflected in pricing.
- **Edge Conditions on Cooldown:** If a user redeems stSOL and expects SOL immediately, but stake is still locked, ensure the contract either rejects the transaction or enforces that the user waits. Off-by-one epoch errors (e.g. allowing withdrawal one epoch too early) can freeze funds.
- **Fee and Reserve Handling:** Many protocols charge fees on withdrawals or swaps. Verify that fees go to the proper reserve account and cannot be lost. If the pool design includes a "reserve" stake or SOL account, ensure it's accounted for in `total_staked` or properly separated.

Stablecoins & Swaps Protocol Architecture

This document describes a robust **Stablecoins & Swaps** protocol architecture on Solana using the Anchor framework. It covers the **purpose and high-level mechanics** of the protocol, delves into **core components** (stable pools, invariant formulas, liquidity, swaps, LP tokens, fees), provides **Anchor/Rust code snippets** illustrating key structures and instructions, and concludes with an **audit checklist** highlighting typical vulnerabilities specific to stablecoin swap pools.

Overview

A **Stablecoins & Swaps** protocol enables efficient, low-slippage trading between stable-value tokens (e.g. USD-pegged coins) on Solana. It typically consists of **liquidity pools** holding pairs (or groups) of stablecoins. Traders swap one stablecoin for another through the pool, while liquidity providers (LPs) deposit assets into the pool and earn fees. The protocol maintains an **invariant formula** to price swaps and ensure the pool remains balanced. Anchor is used to write on-chain program logic, defining accounts and instructions for initializing pools, adding/removing liquidity, performing swaps, and handling fees. Overall, the goal is to allow stablecoin holders to exchange assets at near-1:1 rates with minimal slippage, leveraging smart contract pools that keep prices steady around the peg.

Core Mechanics

Stable Pools and Invariants

Stable pools are specialized automated market maker (AMM) pools tailored for assets that should trade close to a fixed ratio (e.g. 1:1). Unlike a general constant-product AMM (Uniswap-style `x * y = k`), a stable pool uses a **hybrid invariant** that keeps prices flat near the target ratio, reducing slippage for small trades. Two common invariant formulas are:

- **Constant Sum** (`x + y = K`): Implies price = 1:1 always. Adding small amounts of one token yields almost equal output of the other. However, if reserves become unbalanced (one token depleted), the model breaks (you could take all of one token for nothing). Thus, constant-sum is only safe near equal reserves.
- **Constant Product** (`x * y = K`): The classic Uniswap formula. It handles wide price ranges but has significant price slippage when reserves move far from equilibrium.
- **Hybrid (StableSwap) Curve**: Combines the above. For example, **Curve's stableswap** uses an *amplification factor A* to approximate constant-sum behavior around balanced reserves, while defaulting to constant-product when far from balance. With high *A*, the curve is very flat near 1:1; with *A = 1*, it reduces to constant-product. This provides low slippage near equal values and still prevents depletion by switching to a product curve outside a certain range.

In practice, a stable pool computes swaps by solving the chosen invariant. For a hybrid curve, this involves an equation relating token balances and an **invariant D**. On each swap, given an input amount, the contract recalculates reserves to maintain the invariant and determines the output amount. For example, in a two-token hybrid pool:

1. Compute current invariant *D* from reserves *x, y* and amplification *A*.
2. Add the input (minus fees) to one side, solve for the new opposite reserve *y'* that satisfies the invariant with the new *x'*.
3. The output amount is *amount_out = y – y'*.

This may require iterative solving (Newton's method) or a closed-form solution if available. The protocol ensures **invariants hold**, adjusting token balances and minting/burning liquidity shares accordingly.

Liquidity Provision (Minting/Redemption of LP Tokens)

Liquidity providers deposit tokens into the pool to earn a share of swap fees. When adding liquidity, LPs must usually deposit assets **proportionally** to current reserves to maintain the pool ratio. The protocol mints **LP tokens** representing the provider's share:

- **Initial Deposit**: The first LP sets the initial token ratio. The contract may define an LP mint amount (e.g. simple sum of deposited amounts or another scheme) to establish initial total supply.
- **Proportional Deposit**: Subsequent LPs deposit *amount_a* and *amount_b* such that *amount_a / reserve_a = amount_b / reserve_b*. The minted LP share is proportional to this ratio. E.g. for a constant-product pool:
  `shares = min(amount_a * total_shares / reserve_a, amount_b * total_shares / reserve_b)`.
  This ensures fair share relative to existing liquidity.
- **LP Token Mint**: After transferring user tokens into the pool's vault accounts, the contract mints LP tokens (using a mint PDA or authority) to the user. The pool tracks `total_shares` (total LP supply) to calculate future proportions.
- **Removing Liquidity**: To withdraw, a user burns LP tokens. The contract then transfers from pool vaults back to the user amounts of each token proportional to the burned share:
  `withdrawn_a = burn_shares * reserve_a / total_shares; withdrawn_b = burn_shares * reserve_b / total_shares.`
  This returns the correct share of pool assets. The LP supply (`total_shares`) is decreased by `burn_shares`.

Below is a **PoolState** structure defining pool parameters (including amplification and fees) and the Anchor account setup for adding/removing liquidity and minting/burning LP tokens. Important attributes are commented.

```rust
#[account]
pub struct PoolState {
    pub token_mint_a: Pubkey,     // Mint of token A (e.g. USDC)
    pub token_mint_b: Pubkey,     // Mint of token B (e.g. USDT)
    pub token_vault_a: Pubkey,    // Token account holding reserves of A (PDA owned by program)
    pub token_vault_b: Pubkey,    // Token account holding reserves of B
    pub lp_mint: Pubkey,          // Mint for LP token (pool share)
    pub amp: u64,                 // Amplification factor (controls flatness of the curve)
    pub fee_numerator: u64,       // Numerator for fee calculation (basis points)
    pub fee_denominator: u64,     // Denominator for fee (usually 10000 for basis points)
    pub total_shares: u64,        // Total supply of LP tokens (updated on add/remove)
    pub bump: u8,                // PDA bump seed for pool authority
}
```

```rust
#[derive(Accounts)]
pub struct InitializePool<'info> {
    #[account(init, payer = admin, space = 8 + std::mem::size_of::<PoolState>(), seeds = [b"pool"], bump)]
    pub pool: Account<'info, PoolState>,   // PDA for pool state
    /// CHECK: Vault accounts should be created and passed in
    #[account(mut)]
    pub token_vault_a: AccountInfo<'info>, // Token account for reserve A (owned by pool PDA)
    #[account(mut)]
    pub token_vault_b: AccountInfo<'info>, // Token account for reserve B
    #[account(mut)]
    pub lp_mint: Account<'info, Mint>,     // LP token mint (authority is pool PDA)
    pub admin: Signer<'info>,              // Admin/initializer
    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}

pub fn initialize_pool(
    ctx: Context<InitializePool>,
    amp_factor: u64,
    fee_numer: u64,
    fee_denom: u64
) -> ProgramResult {
    let pool = &mut ctx.accounts.pool;
    pool.token_mint_a = ctx.accounts.token_vault_a.mint; // set token mints
    pool.token_mint_b = ctx.accounts.token_vault_b.mint;
    pool.token_vault_a = ctx.accounts.token_vault_a.key();
    pool.token_vault_b = ctx.accounts.token_vault_b.key();
    pool.lp_mint = ctx.accounts.lp_mint.key();
    pool.amp = amp_factor;              // set amplification parameter
    pool.fee_numerator = fee_numer;     // e.g. 4 for 0.04%
    pool.fee_denominator = fee_denom;  // e.g. 10000 for basis points
    pool.total_shares = 0;
    pool.bump = *ctx.bumps.get("pool").unwrap();
    Ok(())
```

```
36  }
```

In **AddLiquidity** below, the user transfers tokens to the pool vaults, then LP tokens are minted. The code ensures proportional minting of LP shares:

```rust
#[derive(Accounts)]
pub struct AddLiquidity<'info> {
    #[account(mut)]
    pub pool: Account<'info, PoolState>,
    #[account(mut)]
    pub user: Signer<'info>, // The liquidity provider
    #[account(mut, constraint = token_a_account.mint == pool.token_mint_
    pub token_a_account: Account<'info, TokenAccount>, // User's token A
    #[account(mut, constraint = token_b_account.mint == pool.token_mint_
    pub token_b_account: Account<'info, TokenAccount>, // User's token B
    #[account(mut, constraint = pool_token_a.key() == pool.token_vault_a
    pub pool_token_a: Account<'info, TokenAccount>,    // Pool vault for
    #[account(mut, constraint = pool_token_b.key() == pool.token_vault_b
    pub pool_token_b: Account<'info, TokenAccount>,    // Pool vault for
    #[account(mut, constraint = user_lp_account.mint == pool.lp_mint)]
    pub user_lp_account: Account<'info, TokenAccount>, // Where LP token
minted to
    pub token_program: Program<'info, Token>,
}

pub fn add_liquidity(ctx: Context<AddLiquidity>, amount_a: u64, amount_b
u64) -> ProgramResult {
    let pool = &mut ctx.accounts.pool;
    // Transfer token A from user to pool vault
    let cpi_a = Transfer {
        from: ctx.accounts.token_a_account.to_account_info(),
        to: ctx.accounts.pool_token_a.to_account_info(),
        authority: ctx.accounts.user.to_account_info(),
    };

token::transfer(CpiContext::new(ctx.accounts.token_program.to_account_in
cpi_a), amount_a)?;
    // Transfer token B similarly
    let cpi_b = Transfer {
        from: ctx.accounts.token_b_account.to_account_info(),
        to: ctx.accounts.pool_token_b.to_account_info(),
        authority: ctx.accounts.user.to_account_info(),
    };

token::transfer(CpiContext::new(ctx.accounts.token_program.to_account_in
```

```
    cpi_b), amount_b)?;

    // Calculate LP shares to mint
    let minted_shares = if pool.total_shares == 0 {
        // First liquidity provider: create initial LP supply (simple sum of deposits)
        amount_a.checked_add(amount_b).unwrap()
    } else {
        // Maintain proportional share of pool
        let share_a = amount_a.checked_mul(pool.total_shares).unwrap()
                      / ctx.accounts.pool_token_a.amount;
        let share_b = amount_b.checked_mul(pool.total_shares).unwrap()
                      / ctx.accounts.pool_token_b.amount;
        std::cmp::min(share_a, share_b) // enforce proportional deposit
    };

    // Mint LP tokens to user, using pool PDA as authority
    let seeds = &[b"pool", &[pool.bump]];
    let signer = &[&seeds[..]];
    token::mint_to(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            MintTo {
                mint: ctx.accounts.lp_mint.to_account_info(),
                to: ctx.accounts.user_lp_account.to_account_info(),
                authority: ctx.accounts.pool.to_account_info(),
            },
            signer
        ),
        minted_shares
    )?;
    pool.total_shares = pool.total_shares.checked_add(minted_shares).unwrap();
    Ok(())
}
```

To **remove liquidity**, the user burns LP tokens and receives underlying assets. The burn and transfer operations are shown here:

```rust
#[derive(Accounts)]
pub struct RemoveLiquidity<'info> {
    #[account(mut)]
    pub pool: Account<'info, PoolState>,
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(mut, constraint = lp_account.mint == pool.lp_mint)]
    pub lp_account: Account<'info, TokenAccount>,    // User's LP token account
    #[account(mut)]
    pub lp_mint: Account<'info, Mint>,
    #[account(mut, constraint = pool_token_a.key() == pool.token_vault_a)]
    pub pool_token_a: Account<'info, TokenAccount>,
    #[account(mut, constraint = pool_token_b.key() == pool.token_vault_b)]
    pub pool_token_b: Account<'info, TokenAccount>,
    #[account(mut)]
    pub user_token_a: Account<'info, TokenAccount>,
    #[account(mut)]
    pub user_token_b: Account<'info, TokenAccount>,
    pub token_program: Program<'info, Token>,
}

pub fn remove_liquidity(ctx: Context<RemoveLiquidity>, burn_shares: u64) -> ProgramResult {
    let pool = &mut ctx.accounts.pool;
    // Burn LP tokens from user
    let seeds = &[b"pool", &[pool.bump]];
    let signer = &[&seeds[..]];
    token::burn(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            Burn {
                mint: ctx.accounts.lp_mint.to_account_info(),
                to: ctx.accounts.lp_account.to_account_info(),
                authority: ctx.accounts.pool.to_account_info(),
            },
            signer
        ),
```

```
        burn_shares
    )?;
    // Calculate amounts to withdraw (proportional to share of total supply)
    let amount_a = burn_shares
        .checked_mul(ctx.accounts.pool_token_a.amount).unwrap()
        .checked_div(pool.total_shares).unwrap();
    let amount_b = burn_shares
        .checked_mul(ctx.accounts.pool_token_b.amount).unwrap()
        .checked_div(pool.total_shares).unwrap();

    // Transfer tokens from pool vaults back to user
    token::transfer(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            Transfer {
                from: ctx.accounts.pool_token_a.to_account_info(),
                to: ctx.accounts.user_token_a.to_account_info(),
                authority: ctx.accounts.pool.to_account_info(),
            },
            signer
        ),
        amount_a
    )?;
    token::transfer(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            Transfer {
                from: ctx.accounts.pool_token_b.to_account_info(),
                to: ctx.accounts.user_token_b.to_account_info(),
                authority: ctx.accounts.pool.to_account_info(),
            },
            signer
        ),
        amount_b
    )?;
    pool.total_shares = pool.total_shares.checked_sub(burn_shares).unwrap();
    Ok(())
}
```

Swapping Tokens and Fees

When a user swaps one stablecoin for another, the protocol:

1. Takes the **input amount**, deducts a swap **fee**, and adds the remainder to the pool.
2. Computes the **output amount** by enforcing the pool's invariant.
3. Transfers the output tokens to the user, ensuring *minimum output* checks to guard against excessive slippage.

Typically a tiny fee (e.g. 0.04%) is charged on input. In code, we apply the fee as `(1 - fee_numerator/fee_denominator)`. The following snippet outlines a swap (A→B), using a constant-product calculation for illustration. A stable pool would use a similar approach but solve the hybrid invariant.

```rust
#[derive(Accounts)]
pub struct Swap<'info> {
    #[account(mut)]
    pub pool: Account<'info, PoolState>,
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(mut)]
    pub user_source: Account<'info, TokenAccount>,   // e.g. user's toke
    #[account(mut)]
    pub user_dest: Account<'info, TokenAccount>,     // user's token B
account
    #[account(mut, constraint = pool_token_source.key() ==
pool.token_vault_a)]
    pub pool_token_source: Account<'info, TokenAccount>, // pool vault f
    #[account(mut, constraint = pool_token_dest.key() == pool.token_vaul
    pub pool_token_dest: Account<'info, TokenAccount>,   // pool vault f
    pub token_program: Program<'info, Token>,
}

pub fn swap(ctx: Context<Swap>, amount_in: u64, min_amount_out: u64) ->
ProgramResult {
    let pool = &ctx.accounts.pool;
    // Compute input after fee
    let fee_numer = pool.fee_numerator;
    let fee_denom = pool.fee_denominator;
    // amount_in_after_fee = amount_in * (1 - fee)
    let amount_in_after_fee = amount_in
        .checked_mul(fee_denom.checked_sub(fee_numer).unwrap()).unwrap()
        .checked_div(fee_denom).unwrap();

    // Transfer full input from user to pool vault
    let cpi_transfer = Transfer {
        from: ctx.accounts.user_source.to_account_info(),
        to: ctx.accounts.pool_token_source.to_account_info(),
        authority: ctx.accounts.user.to_account_info(),
    };

token::transfer(CpiContext::new(ctx.accounts.token_program.to_account_in
cpi_transfer), amount_in)?;

```

```
36      // Recompute balances after input
37      let source_balance = ctx.accounts.pool_token_source.amount; // now
    includes amount_in
38      let dest_balance = ctx.accounts.pool_token_dest.amount;
39
40      // Calculate output using invariant: example for constant-product (x
41      // k = (source_balance_before * dest_balance_before)
42      let k = source_balance
43          .checked_mul(dest_balance).unwrap();
44      let new_source_balance =
    source_balance.checked_add(amount_in_after_fee).unwrap();
45      // new_dest_balance = k / new_source_balance
46      let new_dest_balance = k.checked_div(new_source_balance).unwrap();
47      // amount_out = current dest_balance - new_dest_balance
48      let amount_out = dest_balance.checked_sub(new_dest_balance).unwrap()
49      // Ensure we respect minimum amount out to protect user
50      if amount_out < min_amount_out {
51          return Err(ErrorCode::SlippageExceeded.into());
52      }
53
54      // Transfer the output tokens to user
55      let seeds = &[b"pool", &[pool.bump]];
56      let signer = &[&seeds[..]];
57      token::transfer(
58          CpiContext::new_with_signer(
59              ctx.accounts.token_program.to_account_info(),
60              Transfer {
61                  from: ctx.accounts.pool_token_dest.to_account_info(),
62                  to: ctx.accounts.user_dest.to_account_info(),
63                  authority: ctx.accounts.pool.to_account_info(),
64              },
65              signer
66          ),
67          amount_out
68      )?;
69      Ok(())
70  }
```

In a true **stable-swap curve**, the output calculation replaces the constant-product logic with the specialized curve math. One would solve for the pool invariant (often denoted *D*)

given the new deposit, then derive the output as the difference in the other token. The pseudocode above can be adapted by replacing the `k/new_source_balance` step with the solution to the stablecurve equation. In all cases, precise integer arithmetic is crucial to avoid rounding errors (using checked math or fixed-point arithmetic with large integer types).

Audit Checklist: Stable Swaps Specific Issues

When reviewing a stablecoin swap protocol, focus on math and logic vulnerabilities unique to stable pools:

- **Invariant Calculation Errors**: Ensure the stable-curve or constant-sum/product formulas are implemented correctly. Mistakes in the algebra (especially with hybrid amplification) can yield incorrect pricing, allowing arbitrage drains.
- **Division and Rounding Precision**: Integer division can truncate. In swaps, rounding down (floor) the output benefits the pool (user gets less), but rounding up could allow small arbitrage. Verify rounding direction is consistent and documented. Consider using larger precision or rational arithmetic to minimize error.
- **Zero or Minimal Reserves**: Check for divide-by-zero or impractical trades when a vault has zero or extremely small balance. Swapping or withdrawing should be blocked if reserves are insufficient to cover minimum output.
- **Fee Calculation Bugs**: Mistakes in fee logic (e.g. wrong denominator, applying fee twice, or failing to credit fees to LPs) can divert funds or yield incorrect outputs. Ensure fees are deducted on the correct side (input vs output) and that fee math (especially with basis points) does not overflow.
- **Pool Imbalance / Exploits**: A pure constant-sum pool can be drained by depositing one token until the other runs out. Even hybrid pools can behave like sum near center; test edge cases. Prevent extreme imbalances by requiring proportional deposits or limiting one-sided additions.
- **LP Minting Calculation**: If liquidity is added with mismatched ratios, the formula for minted shares must handle it correctly. Failing to take the minimum proportion (as shown) could mint excessive shares or skew the pool share math.
- **Slippage and Min-Out Checks**: Always enforce `min_amount_out` or similar slippage limits in swap instructions. Without it, price can shift before transaction finality, hurting users. Also, check that on high slippage the swap reverts properly.
- **Precision Loss in Amplification**: If implementing the stable-swap formula, using integer math for *A* and *D* requires high precision. Losing precision can allow trades that slightly violate the invariant, giving arbitrage opportunities. Testing edge cases for large amplification is crucial.
- **State Synchronization**: Ensure `total_shares` and vault balances are updated atomically. An error where LP supply isn't reduced after a withdrawal (or increased

after deposit) can desynchronize share accounting.

- **Missing Constraint Checks**: While not math, ensure account constraints prevent malicious inputs: e.g. token accounts must match expected mints, and authorities must be validated. For stable swaps, mismatched decimals or token hooks can cause logical errors if unchecked.
- **Reentrancy-like Scenarios**: Solana's model prevents classic reentrancy, but be wary of any cross-program calls (CPI) that could indirectly trigger unwanted behavior. For example, careful order of operations when transferring and updating state to avoid stale state usage.
- **Edge Case of First LP**: The logic for the first liquidity provider (when `total_shares == 0`) must be carefully implemented. If initial `minted_shares` is miscomputed, it could skew future ratios or make share math invalid.

NFT Marketplace Protocol

1. Overview

A Solana **NFT marketplace protocol** enables users to list NFTs for sale, buy listed NFTs, and cancel listings. The key actors are:

- **Seller:** Owns an NFT and wants to sell it.
- **Buyer:** Wants to purchase a listed NFT by paying the asking price.
- **Marketplace Program:** A Solana program (written in Anchor/Rust) that manages listings and transfers of NFTs and payments.

The core flow has three main operations:

- **List NFT:** Seller creates a *Listing* account (on-chain record) with sale details, and transfers their NFT into an escrow vault owned by the marketplace program (typically a PDA).
- **Buy NFT:** Buyer sends payment (SOL or a token) to the seller via the marketplace program; the program transfers the NFT from escrow to the buyer; the listing is marked closed or the account is closed.
- **Cancel Listing:** Seller cancels an existing listing before it's sold; the program transfers the NFT back from escrow to the seller, and closes the listing.

Other participants may include fee recipients if the protocol takes a marketplace commission, but the essential actors are **Seller, Buyer,** and the **Marketplace Program**.

Key accounts in the protocol:

- **Listing Account:** A PDA storing the seller's public key, the NFT mint, price, and status.
- **Vault Account:** A token account (PDA) owned by the program that temporarily holds the listed NFT in escrow.
- **Seller & Buyer Token Accounts:** Standard SPL token accounts for holding the NFT and payment tokens.

High-Level Flow

1. **Listing:**
   - Seller signs a transaction calling `CreateListing`.
   - The program creates a *Listing* account (unique PDA) with sale info, and an escrow *Vault* token account (also a PDA) with authority set to the listing PDA.
   - The NFT (1 token of supply) is transferred from the seller's NFT token account to the vault.
2. **Buying:**
   - Buyer calls `BuyListing`, sending payment (usually via SOL or an SPL payment token).
   - The program verifies the listing is active and payment covers the price.
   - Payment is forwarded to the seller.
   - The NFT is transferred from the vault to the buyer's token account.
   - The listing is marked inactive or the account is closed, finishing the sale.
3. **Canceling:**
   - Seller calls `CancelListing`.
   - The program verifies the caller is the listing's seller and listing is active.
   - The NFT is transferred from the vault back to the seller's token account.
   - The listing is marked inactive or closed.

This ensures that an NFT is held securely by the program while listed, and proper checks ensure only the rightful seller or a buyer can complete or cancel the sale.

2. Deep Dive into Core

2.1 Architecture

The marketplace program uses **Program Derived Addresses (PDAs)** to create unique accounts:

- **Listing PDA:** Derived from seeds like `["listing", seller_pubkey, nft_mint_pubkey]`. Ensures each seller+NFT pair has at most one active listing. The listing PDA stores sale details and has a bump seed for signing.
- **Vault PDA:** A token account PDA to hold the NFT, derived from either the listing PDA or similar seeds (`["vault", listing_pda]` or created via Anchor's `init` with authority = listing). This vault is owned by the marketplace program through the listing PDA authority.

The program code (in Anchor) defines account structs and instructions for each core operation. Anchor simplifies checks like ownership and PDA verification.

Program Flow in Create Listing:

- **Accounts:**
  - `seller` (Signer): must sign and will pay for new accounts.
  - `listing` (Account, init): created with PDA seeds.
  - `nft_mint` (Account): the NFT's mint.
  - `seller_nft_account` (TokenAccount): seller's current NFT token account (must hold 1 token).
  - `vault` (TokenAccount, init): new escrow token account for the NFT.
  - `token_program` , `system_program` , `rent` : standard programs/sysvars.
- **Steps:**
  1. **Create Listing Account:** `#[account(init, seeds = [b"listing", seller, nft_mint], bump, payer = seller)] pub listing: Account<'info, Listing>` sets up a new `Listing` struct on-chain.
  2. **Create Vault:** `#[account mint = nft_mint, token::authority = listing, payer = seller ] pub vault: Account<'info, TokenAccount>` creates a token account (the vault) with mint = NFT and owner = listing PDA.
  3. **Transfer NFT to Vault:** Use `anchor_spl::token::transfer` to move 1 NFT from `seller_nft_account` to `vault` .
  4. **Initialize Listing Data:** Set fields in `Listing` (seller pubkey, nft_mint, price, is_active = true, bump, vault Pubkey).

```rust
#[derive(Accounts)]
pub struct CreateListing<'info> {
    #[account(mut)]
    pub seller: Signer<'info>,
    /// The Listing PDA, unique for (seller,nft_mint).
    #[account(
        init,
        seeds = [b"listing", seller.key().as_ref(), nft_mint.key().as_ref()],
        bump,
        payer = seller,
        space = 8 + Listing::LEN
    )]
    pub listing: Account<'info, Listing>,

    /// NFT mint and token accounts
    pub nft_mint: Account<'info, Mint>,
    #[account

        token" data-dv-norm-key="mut-token">        token" data-dv-norm-key="mut-token">mint = nft_mint,


        token" data-dv-norm-key="mut-token">token::authority = seller]
    pub seller_nft_account: Account<'info, TokenAccount>,

    /// Vault token account, authority = listing PDA
    #[account

        payer = seller,
        token" data-dv-norm-key="init-payer--seller-token"> payer = seller,

        payer = seller,
        token" data-dv-norm-key="init-payer--seller-token"> token" data-dv-norm-key="init-payer--seller-token">mint = nft_mint,

token::authority = listing,]
    pub vault: Account<'info, TokenAccount>,
```

```
36
37      pub token_program: Program<'info, Token>,
38      pub system_program: Program<'info, System>,
39      pub rent: Sysvar<'info, Rent>,
40  }
41
42  pub fn create_listing(ctx: Context<CreateListing>, price: u64) ->
    Result<()> {
43      let listing = &mut ctx.accounts.listing;
44      // Initialize listing data
45      listing.seller = ctx.accounts.seller.key();
46      listing.nft_mint = ctx.accounts.nft_mint.key();
47      listing.price = price;
48      listing.is_active = true;
49      listing.bump = *ctx.bumps.get("listing").unwrap();
50      listing.vault = ctx.accounts.vault.key();
51
52      // Transfer the NFT into the vault (escrow)
53      let cpi_accounts = Transfer {
54          from: ctx.accounts.seller_nft_account.to_account_info(),
55          to: ctx.accounts.vault.to_account_info(),
56          authority: ctx.accounts.seller.to_account_info(),
57      };
58      let cpi_ctx =
    CpiContext::new(ctx.accounts.token_program.to_account_info(),
    cpi_accounts);
59      anchor_spl::token::transfer(cpi_ctx, 1)?;
60      Ok(())
```

- The `#[account(init, ...)]` attributes create new accounts with the given seeds and payers.
- We set `listing.is_active = true` to mark it as available.
- The transfer CPI moves exactly 1 NFT from seller to the vault.

2.2 Core Accounts and Their Purposes

- **Seller (Signer):** Must sign the transaction for listing or cancellation; pays rent for new accounts; receives SOL upon sale.
- **Buyer (Signer):** Signs the buy transaction; provides payment (in SOL or tokens).
- **Listing (Account):** PDA storing `{seller, nft_mint, price, is_active, bump, vault}`. Acts as the canonical record of a sale order.

- **NFT Mint (Account):** The mint of the NFT being sold.
- **Seller's NFT Token Account:** The token account currently holding the NFT (must have amount=1 on listing).
- **Vault (Account):** A PDA token account created to hold the NFT during sale. Its authority is the `listing` PDA.
- **Buyer's NFT Token Account:** Token account owned by buyer to receive NFT on purchase.
- **Payment Accounts / SOL:** If using a SPL payment token, buyer's and seller's token accounts for that mint; if using SOL, just the buyer and seller SystemAccounts and the System Program.

2.3 Core Instructions

Instruction: Create Listing

The `create_listing` instruction initializes a new listing.

Key steps and code:

```rust
#[derive(Accounts)]
pub struct CreateListing<'info> {
    #[account(mut)]
    pub seller: Signer<'info>,

    /// Create a new Listing account with PDA seeds.
    #[account(
        init,
        seeds = [b"listing", seller.key().as_ref(),
nft_mint.key().as_ref()],
        bump,
        payer = seller,
        space = 8 + Listing::LEN
    )]
    pub listing: Account<'info, Listing>,

    /// NFT mint and seller's NFT token account
    pub nft_mint: Account<'info, Mint>,
    #[account(
        mut,
        token::mint = nft_mint,
        token::authority = seller
    )]
    pub seller_nft_account: Account<'info, TokenAccount>,

    /// Vault: new token account to hold NFT, authority = listing PDA
    #[account(
        init,
        payer = seller,
        token::mint = nft_mint,
        token::authority = listing,
    )]
    pub vault: Account<'info, TokenAccount>,

    /// Programs
    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}

```

```rust
pub fn create_listing(ctx: Context<CreateListing>, price: u64) -> Result<()> {
    let listing = &mut ctx.accounts.listing;
    // Set listing details
    listing.seller = ctx.accounts.seller.key();
    listing.nft_mint = ctx.accounts.nft_mint.key();
    listing.price = price;
    listing.is_active = true;
    listing.bump = *ctx.bumps.get("listing").unwrap();
    listing.vault = ctx.accounts.vault.key();

    // Transfer 1 NFT from seller to vault (escrow)
    let cpi_accounts = Transfer {
        from: ctx.accounts.seller_nft_account.to_account_info(),
        to: ctx.accounts.vault.to_account_info(),
        authority: ctx.accounts.seller.to_account_info(),
    };
    let cpi_ctx = CpiContext::new(ctx.accounts.token_program.to_account_info(), cpi_accounts);
    anchor_spl::token::transfer(cpi_ctx, 1)?;
    Ok(())
}
```

Important points:

- `#[account(init, seeds = [b"listing", seller, nft_mint], bump)]` : Creates the **Listing PDA**. The seeds ensure uniqueness (one listing per seller+NFT).
- `Listing` struct fields: stores `seller` , `nft_mint` , `price` , a boolean `is_active` , `bump` (for PDA signing), and `vault` address.
- `#[account authority = listing ]` for `vault` : Creates a token account whose owner is the `listing` PDA. This holds the NFT in escrow.
- We use `anchor_spl::token::transfer` CPI to move the NFT. The `authority` is the seller (they sign to allow the transfer).
- After execution, the NFT is locked in `vault` until sold or canceled.

Instruction: Buy NFT

The `buy_listing` instruction allows a buyer to purchase an active listing.

```rust
#[derive(Accounts)]
pub struct BuyListing<'info> {
    #[account(mut)]
    pub buyer: Signer<'info>,

    /// The existing Listing account (must be active).
    #[account(
        mut,
        has_one = seller,
        seeds = [b"listing", listing.seller.as_ref(),
listing.nft_mint.as_ref()],
        bump = listing.bump
    )]
    pub listing: Account<'info, Listing>,

    /// NFT vault (PDA token account), authority = listing PDA
    #[account(
        mut,
        token::authority = listing,
        token::mint = listing.nft_mint
    )]
    pub vault: Account<'info, TokenAccount>,

    /// NFT token account for buyer to receive NFT
    #[account(
        init_if_needed,
        payer = buyer,
        token::mint = listing.nft_mint,
        token::authority = buyer
    )]
    pub buyer_nft_account: Account<'info, TokenAccount>,

    /// Seller to receive payment (SystemAccount if SOL)
    #[account(mut)]
    pub seller: SystemAccount<'info>,

    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
}

```

```rust
pub fn buy_listing(ctx: Context<BuyListing>) -> Result<()> {
    let listing = &mut ctx.accounts.listing;
    // Verify listing is active
    require!(listing.is_active, MyError::ListingNotActive);

    // Transfer payment from buyer to seller (using SOL in this example)
    let price = listing.price;
    let transfer_cpi = anchor_lang::system_program::Transfer {
        from: ctx.accounts.buyer.to_account_info(),
        to: ctx.accounts.seller.to_account_info(),
    };
    let cpi_ctx = CpiContext::new(ctx.accounts.system_program.to_account_info(), transfer_cpi);
    anchor_lang::system_program::transfer(cpi_ctx, price)?;

    // Transfer NFT from vault to buyer's account
    let listing_seeds = &[
        b"listing",
        listing.seller.as_ref(),
        listing.nft_mint.as_ref(),
        &[listing.bump],
    ];
    let signer = &[&listing_seeds[..]];
    let cpi_accounts = Transfer {
        from: ctx.accounts.vault.to_account_info(),
        to: ctx.accounts.buyer_nft_account.to_account_info(),
        authority: ctx.accounts.listing.to_account_info(),
    };
    let cpi_ctx = CpiContext::new_with_signer(
        ctx.accounts.token_program.to_account_info(),
        cpi_accounts,
        signer,
    );
    anchor_spl::token::transfer(cpi_ctx, 1)?;

    // Close the listing: mark inactive or automatically close if `close` attribute used
    listing.is_active = false;
```

```
76        Ok(())
77    }
```

Important points:

- `has_one = seller` : Ensures the `listing.seller` field matches the provided `seller` account.
- `listing.is_active` : Prevents buying an already sold or canceled listing.
- SOL payment: We use `system_program::transfer` CPI to send `listing.price` lamports from buyer to seller. If a token were used, a similar token transfer CPI would apply.
- NFT transfer: We provide `seeds` and `signer` to the CPI so the `listing` PDA can sign for the vault (its token authority).
- After transferring the NFT, we mark `listing.is_active = false` . Optionally, you can add `#[account(mut, close = seller)]` on the listing account to reclaim rent at the end of the instruction.

Instruction: Cancel Listing

The `cancel_listing` instruction lets a seller withdraw their NFT and close an active listing.

```rust
#[derive(Accounts)]
pub struct CancelListing<'info> {
    #[account(mut)]
    pub seller: Signer<'info>,

    /// Existing listing (must be owned by seller)
    #[account(
        mut,
        has_one = seller,
        seeds = [b"listing", seller.key().as_ref(),
listing.nft_mint.as_ref()],
        bump = listing.bump
    )]
    pub listing: Account<'info, Listing>,

    /// Vault token account (holds NFT), authority = listing PDA
    #[account(
        mut,
        token::authority = listing,
        token::mint = listing.nft_mint
    )]
    pub vault: Account<'info, TokenAccount>,

    /// Seller's token account to receive NFT back
    #[account(
        mut,
        token::mint = listing.nft_mint,
        token::authority = seller
    )]
    pub seller_nft_account: Account<'info, TokenAccount>,

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

pub fn cancel_listing(ctx: Context<CancelListing>) -> Result<()> {
    let listing = &mut ctx.accounts.listing;
    // Verify listing is active
    require!(listing.is_active, MyError::ListingNotActive);

    // Transfer NFT from vault back to seller
```

```
        let listing_seeds = &[
            b"listing",
            listing.seller.as_ref(),
            listing.nft_mint.as_ref(),
            &[listing.bump],
        ];
        let signer = &[&listing_seeds[..]];
        let cpi_accounts = Transfer {
            from: ctx.accounts.vault.to_account_info(),
            to: ctx.accounts.seller_nft_account.to_account_info(),
            authority: ctx.accounts.listing.to_account_info(),
        };
        let cpi_ctx = CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            cpi_accounts,
            signer,
        );
        anchor_spl::token::transfer(cpi_ctx, 1)?;

        // Mark listing inactive (and optionally close it)
        listing.is_active = false;
        Ok(())
    }
```

Important points:

- Only the original `seller` (PDA-authorized) can cancel.
- We check `listing.is_active` to ensure it hasn't already been sold or canceled.
- NFT Transfer: Same PDA signing logic moves the NFT from vault back to `seller_nft_account`.
- After transfer, we deactivate the listing. As with buy, you could use `#[account(mut, close = seller)]` on `listing` to reclaim rent.

2.4 Code Annotations

In each code example above, key annotated parts include:

- **PDA Initialization (`init, seeds`)**: Guarantees unique, program-owned accounts.
- **Token Account CPI**: Uses `anchor_spl::token::transfer` with appropriate `authority` (seller or listing PDA).

- **Signer Seeds (`new_with_signer`)**: Allows the listing PDA to sign for token operations on its vault.
- **Checks (e.g. `require!(listing.is_active)`)**: Prevent unauthorized actions.
- **Closing Accounts**: Optionally using `close = seller` to reclaim rent, not shown in code but recommended.

Each instruction's logic is carefully sequenced:

1. **Listing:** Initialize on-chain state, then move NFT to escrow.
2. **Buying:** Verify listing, move SOL, then move NFT, then finalize listing.
3. **Canceling:** Verify seller, then move NFT back, then finalize.

This design ensures NFTs are never lost or duplicated: they always reside either with a user or in a known escrow, and listing state prevents replay.

3. Audit Checklist / Bug Awareness

When auditing a Solana NFT marketplace protocol, consider these **protocol-specific vulnerabilities** and pitfalls:

- **NFT Ownership Verification:** Ensure the seller *actually owned* the NFT at listing time. Rely on Anchor constraints like `token::authority = seller` and require `seller_nft_account.amount == 1`. Without these checks, someone could list an NFT they don't own.
- **One-Token Assumption:** NFT is typically supply = 1. Enforce `owner_token_account.amount == 1` so fractions or semi-fungible tokens aren't misused.
- **Listing State Race:** Prevent double-spend by checking `is_active` before actions. Ensure after a buy or cancel, the listing is immediately marked inactive (or closed). Otherwise two buyers might both succeed.
- **Reentrancy-like Abuse:** While Solana doesn't have true reentrancy, ensure buying or canceling can't be called in a partially complete state. Using `is_active` and atomic state updates avoids half-done listings.
- **PDA Seed Collisions:** Use unique seeds for listing PDAs (e.g. include seller Pubkey and NFT Mint). Reusing simple seeds (like just the mint) can allow one seller to cancel or buy another's listing if PDAs collide.
- **Escrow Authority:** Vault accounts should be owned by the *listing PDA*, not the program authority directly. This enforces that only the specific listing record can move that NFT.
- **Price Manipulation:** If price currency is an SPL token with decimals, be careful of supply/decimals. Verify the correct payment token mint and that `price` units align

(e.g. if NFT priced in USDC with 6 decimals).

- **Missing Fee Handling:** If marketplace fees or royalties apply, verify the split of funds exactly (e.g. 98% to seller, 2% fee). Incomplete fee logic could allow missing payments.
- **Token Account Initializations:** Using `init_if_needed` for the buyer's NFT account handles new accounts, but ensure rent is paid (buyer pays) to prevent failures.
- **Closing vs Marking:** Decide whether to close listing accounts or keep them. Unclosed accounts cost rent forever. But closing requires care: you must not close until after token transfers.
- **Authority Checks:** Always use constraints like `has_one = seller` on `CancelListing` and `BuyListing` to ensure the provided accounts match the listing data.
- **Vault Cleanup:** Ensure the vault token account doesn't remain with 0 tokens and no close instructions. Otherwise orphan accounts accumulate. You may choose to close the vault too (returning lamports to seller or paying authority).
- **PDAs with Incrementing IDs:** If using a listing ID counter, ensure counter accounts don't overflow or get manipulated.
- **Data Tampering:** Never trust client-provided data in transactions; read price and seller from on-chain Listing account.

Cross-Chain Bridge Protocol

Protocol Overview

A cross-chain bridge enables tokens or data to move securely between Solana and other blockchains. In a typical token bridge, assets on a source chain are *locked* or *burned*, then *minted* or *released* on the destination chain, and vice versa for returning. Trust is established via an off-chain validator network or oracle set: these validators observe events on one chain and authorize corresponding actions on the other. On Solana, an Anchor-based bridge program handles on-chain logic: it locks or burns SPL tokens in a program-controlled vault and verifies validator-signed messages to mint or release assets in the opposite flow.

The high-level flow is as follows: a user requests a transfer by invoking the Anchor bridge program on Solana, which locks (escrows) their tokens and emits an event. Off-chain relayers watch this event and form a signed message attesting to the lock. This message, carrying details like amount, source and target chain, and a unique nonce, is then submitted to a bridge program on the other chain. There, validators' signatures are

verified; if valid and fresh (not replayed), the corresponding amount of tokens is minted or released to the recipient. The process works similarly in reverse, enabling tokens to flow in both directions.

Key points:

- **Asset Custody:** The bridge program on Solana holds locked tokens in a Program Derived Address (PDA) vault. It may also issue and burn wrapped tokens representing foreign assets.
- **Validator Consensus:** A set of trusted validators sign off-chain messages to attest to cross-chain transfers. The Anchor program verifies enough validator signatures (meeting a threshold) on-chain before honoring a transfer.
- **Message Integrity:** Each cross-chain message includes unique identifiers (nonce, chain IDs) to prevent replays. The program maintains state (e.g. processed nonces) to ensure one-time execution.

Core Components and Architecture

A robust cross-chain bridge on Solana comprises several on-chain and off-chain parts:

- **Validator Network (Oracle Set):** A group of nodes or accounts responsible for observing chain events and signing bridge messages. On Solana, validators might be represented by known public keys. The bridge program stores their public keys and a required signature threshold. Only when a message carries signatures from a sufficient subset does the program process it.
- **Bridge Configuration Account:** A Solana account (often a PDA) that holds bridge state such as the list of validator public keys, signature threshold, and global parameters (e.g. chain identifier, admin authority). This account is controlled by an admin or governed via multi-signature, allowing updates to validators when needed.
- **Vault Accounts (Token Escrow):** For each bridged token, the program uses a PDA-owned SPL Token account as a vault. When a user initiates a transfer from Solana to another chain, their tokens are moved into this vault (lock). Conversely, when tokens come from another chain to Solana, the program can release tokens from the vault to the recipient, or mint new ones if modeling a wrapped asset.
- **Token Mint (Wrapped Asset):** If the bridge creates a wrapped representation of a foreign token, it controls a Mint account. The program, as mint authority (often a PDA), can mint or burn tokens in response to cross-chain messages. This is typical when bridging assets that do not natively exist on Solana.
- **Bridge State (Nonces/Processed Events):** To prevent replays, the program tracks processed transfers using unique nonces or sequence numbers. This state can be

stored in a bridge state account or via seeding PDAs (for example, using transfer nonce and chain ID as seeds for a PDA that marks completion).

- **Off-Chain Relayers:** Independent services (not on Solana) listen for events from the Solana bridge program. When they detect a lock event, they coordinate with validators to form a signed message. These relayers then submit the signed data to the bridge contract on the target chain. They do the same for transfers coming from other chains into Solana.
- **User Accounts:** The end-user interacts with the bridge by calling its instructions. They must have a wallet on Solana, an SPL token account for the asset, and sufficient SOL for fees.

Communication Flow

The cross-chain communication involves multiple steps and participants. Below is a typical flow for a token transfer from Solana to an external chain, followed by the reverse flow (external to Solana):

1. **Initiation (Solana → Other Chain):**
    - The user calls the bridge program on Solana with the transfer details (amount, recipient on other chain, target chain ID).
    - The bridge program uses Anchor's token CPI (Cross-Program Invocation) to *transfer* the specified SPL tokens from the user's token account into the program's vault account. Alternatively, if the token was a wrapped foreign asset minted on Solana, it would *burn* the tokens instead.
    - The program increments a local nonce counter and emits a `TransferInitiated` event (an Anchor event log) containing the details (chain IDs, token, amount, nonce, recipient address, etc.).
2. **Relayer & Validator Signing:**
    - Off-chain relayers pick up this `TransferInitiated` event. One or more relayers bundle it into a canonical message format (often ABI-encoded or serialized JSON).
    - Validators in the network observe the event data and independently produce signatures over the message hash. These signatures prove that a quorum of validators has seen and approved the transfer request.
3. **Submission and Verification (Other Chain):**
    - A relayer sends the signed message (with validator signatures) to the target chain's bridge contract.

- The bridge contract there verifies that: 1) the signatures are valid and come from enough known validators; 2) the message format is correct; 3) the nonce/ID has not been used before.
- If checks pass, the target bridge contract mints (or releases from escrow) the corresponding token amount to the recipient's address on the target chain.

4. **Reverse Flow (Other Chain → Solana):**
    - On the other chain, a user likewise locks/burns tokens and triggers an event. Off-chain relayers collect this and form a signed message for Solana.
    - A user (or relayer) calls the Solana bridge program's `complete_transfer` instruction, providing the signed message and signatures.
    - The Solana program verifies signatures against its stored validator set and checks that the message's nonce has not been processed.
    - The program then sends SPL tokens from its vault to the recipient's Solana token account, or mints new tokens if it's a wrapped asset. It emits a `TransferCompleted` event indicating success.

Throughout this process, the bridge must ensure *atomicity* of lock/unlock steps and maintain a strict sequence of actions to avoid any inconsistencies. All message formats typically include the source chain ID, target chain ID, token identifier, amount, sender, recipient, and a unique nonce or transaction ID. Including chain IDs prevents a message meant for one destination from being replayed on the same or another chain.

Anchor Implementation Examples

Below are key Anchor-based code snippets illustrating the major components of a Solana bridge program. Each snippet is accompanied by an explanation of its role.

Validator Set Management

We define an on-chain account to store the validator public keys and the signature threshold. An admin (or multisig) is authorized to update this list. This account is a PDA or a simple stored account that holds persistent bridge configuration.

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

#[account]
pub struct BridgeConfig {
    pub admin: Pubkey,              // authority to update validators
    pub validators: Vec<Pubkey>,    // current set of trusted validators
    pub threshold: u8,              // number of signatures required
    pub nonce: u64,                 // global nonce counter for
transfers
}

#[derive(Accounts)]
pub struct UpdateValidatorSet<'info> {
    #[account(mut, has_one = admin)]
    pub config: Account<'info, BridgeConfig>,
    pub admin: Signer<'info>,
}

pub fn update_validator_set(
    ctx: Context<UpdateValidatorSet>,
    new_validators: Vec<Pubkey>,
    new_threshold: u8
) -> ProgramResult {
    let config = &mut ctx.accounts.config;
    // Only the configured admin can change the validator set
    require_keys_eq!(ctx.accounts.admin.key(), config.admin);
    // Update the validator list and threshold
    config.validators = new_validators;
    config.threshold = new_threshold;
    Ok(())
}
```

Explanation: The `BridgeConfig` account holds critical bridge state: an `admin` key authorized to make changes, the `validators` array (public keys of nodes authorized to sign messages), a `threshold` specifying how many validator signatures are required for a valid cross-chain message, and a global `nonce`. The `update_validator_set` instruction allows the admin to replace the validator list and threshold. Anchor ensures the `config` account is mutable and the signer `admin` matches the stored admin pubkey. This way, only the rightful administrator can reconfigure the bridge's validator network.

Cross-Chain Message Verification

When a cross-chain transfer message arrives (wrapped from another chain), the program must verify the attached validator signatures match the message. Below is a conceptual snippet illustrating this. In practice, verifying ECDSA signatures (for example, Ethereum signatures) on Solana requires using the `secp256k1_program`, but here we sketch the logic:

```rust
pub fn verify_message_signatures(
    message: &[u8],
    signatures: Vec<(Pubkey, [u8; 64])>,
    validators: &Vec<Pubkey>,
    threshold: u8,
) -> ProgramResult {
    let mut valid_signatures = 0;
    // Check each provided signature
    for (voter, signature) in signatures.iter() {
        // Ensure the signer is a known validator
        if !validators.contains(voter) {
            continue;
        }
        // Verify the signature over the message bytes
        // (In real code, this uses syscalls to secp256k1 or ed25519)
        if verify_signature(message, signature, voter) {
            valid_signatures += 1;
        }
    }
    // Ensure at least `threshold` validators signed
    if valid_signatures < threshold.into() {
        return Err(ProgramError::Custom(0)); // e.g. BridgeError::NotEnoughSignatures
    }
    Ok(())
}

// Placeholder for the actual signature verification logic
fn verify_signature(message: &[u8], signature: &[u8; 64], signer: &Pubkey) -> bool {
    // Actual implementation would use Solana's secp256k1 instruction or ed25519
    true
}
```

Explanation: In this example, `verify_message_signatures` takes the raw `message` bytes (the serialized transfer data) and a list of `(Pubkey, signature)` tuples. It iterates through each provided signature, checks that the signer's public key is in the bridge's known `validators` list, and then performs cryptographic signature verification. If a signature is valid and from a recognized validator, it increments `valid_signatures`.

After processing all signatures, the function requires that the count of valid signatures meets or exceeds the `threshold`. Only then is the cross-chain message considered authentic. In a real Anchor program, one would use the `secp256k1_program` (for ECDSA) or built-in ed25519 signature checks provided by Solana.

Token Locking (Escrow) and Mint/Burn Mechanisms

The bridge program must lock (transfer to a vault) or burn tokens on Solana when sending assets out, and mint or unlock tokens when bringing assets in. The example below shows an `initiate_transfer` instruction that locks user tokens in a vault and emits an event. A complementary `complete_transfer` would perform minting or release.

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

#[derive(Accounts)]
pub struct InitiateTransfer<'info> {
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(mut)]
    pub user_token_account: Account<'info, TokenAccount>,
    #[account(
        mut,
        // The vault PDA is derived from the token mint address and program ID
        seeds = [b"vault", mint.key().as_ref()],
        bump,
    )]
    pub vault_account: Account<'info, TokenAccount>,
    pub mint: Account<'info, Mint>,
    #[account(mut, has_one = admin)]
    pub config: Account<'info, BridgeConfig>,
    // Anchor simplifies calling the SPL Token program
    pub token_program: Program<'info, token::Token>,
    pub system_program: Program<'info, System>,
}

pub fn initiate_transfer(
    ctx: Context<InitiateTransfer>,
    amount: u64,
    to_chain: u64,
    recipient: [u8; 32],   // e.g. recipient address on target chain
) -> ProgramResult {
    let config = &mut ctx.accounts.config;
    // Transfer tokens from user to the program vault (locking them)
    let cpi_accounts = Transfer {
        from: ctx.accounts.user_token_account.to_account_info(),
        to: ctx.accounts.vault_account.to_account_info(),
        authority: ctx.accounts.user.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    token::transfer(CpiContext::new(cpi_program, cpi_accounts), amount)?;
```

```
        // Emit an event for relayers, including the new nonce
        let message = BridgeMessage {
            from_chain: SOLANA_CHAIN_ID,
            to_chain,
            nonce: config.nonce,
            token_address: ctx.accounts.mint.key(),
            amount,
            sender: ctx.accounts.user.key(),
            recipient,
        };
        config.nonce += 1; // increment global nonce to prevent replay
        emit!(TransferInitiated {
            message: message.clone()
        });
        Ok(())
    }
```

Explanation: The `initiate_transfer` instruction uses Anchor's CPI to the SPL Token program to move tokens. The `InitiateTransfer` accounts struct includes the user's signer and token account, the vault PDA for the token (ensured by `seeds = [b"vault", mint.key().as_ref()]`), and a mutable reference to `BridgeConfig`. Inside the function, we call `token::transfer` to move `amount` tokens from the user's account into the vault PDA (locking them). We then construct a `BridgeMessage` struct containing all relevant data (chain IDs, a unique `nonce`, token mint address, amount, and sender/recipient). After incrementing the config nonce, we emit a `TransferInitiated` event with this message. Off-chain relayers pick up this event to create the cross-chain transfer request.

For transfers coming *into* Solana from another chain, the program would implement a `complete_transfer` function. After verifying validator signatures (as shown previously), it would do one of two things depending on the token type:

- **If the token is natively an SPL token on Solana:** The vault holds actual assets. The program would transfer tokens from the vault to the recipient's token account `transfer`.
- **If the token is a wrapped foreign asset:** The program controls a Mint. It would mint new tokens to the recipient's account `mint_to`. Conversely, when sending these tokens back out of Solana, it would *burn* them `burn` instead of transferring from a vault.

For example, completing an incoming transfer by minting might look like:

```rust
pub fn complete_transfer(
    ctx: Context<CompleteTransfer>,
    message: BridgeMessage,
    signatures: Vec<(Pubkey, [u8; 64])>,
) -> ProgramResult {
    let config = &ctx.accounts.config;
    // Verify signatures on the message
    verify_message_signatures(&message.serialize(), signatures, &config.validators, config.threshold)?;
    // Prevent replay by checking nonce (this could also use a PDA keyed by nonce)
    require!(message.nonce == config.processed_nonce, ErrorCode::InvalidNonce);
    // Perform token mint or unlock
    if ctx.accounts.mint_authority.is_signer {
        // Example: mint wrapped tokens to the recipient
        let cpi_accounts = MintTo {
            mint: ctx.accounts.mint.to_account_info(),
            to: ctx.accounts.recipient_token.to_account_info(),
            authority: ctx.accounts.mint_authority.to_account_info(),
        };
        let cpi_program = ctx.accounts.token_program.to_account_info();
        token::mint_to(CpiContext::new(cpi_program, cpi_accounts), message.amount)?;
    } else {
        // Example: transfer native tokens from vault to recipient
        let cpi_accounts = Transfer {
            from: ctx.accounts.vault_account.to_account_info(),
            to: ctx.accounts.recipient_token.to_account_info(),
            authority: ctx.accounts.vault_authority.to_account_info(),
        };
        let cpi_program = ctx.accounts.token_program.to_account_info();
        token::transfer(CpiContext::new(cpi_program, cpi_accounts), message.amount)?;
    }
    // Mark nonce as processed (implementation may vary)
    // config.processed_nonce = message.nonce + 1;
    emit!(TransferCompleted { message: message.clone() });
    Ok(())
}
```

Here, `CompleteTransfer` would include accounts like the target mint, vault, recipient, and authorities. The code verifies signatures, checks the message nonce, then either mints or transfers tokens. Finally, it emits a `TransferCompleted` event to log success.

Event Emission

Anchor allows programs to emit typed events that clients can listen to. For a bridge, events like `TransferInitiated` and `TransferCompleted` are crucial signals. Below is an example of defining and emitting such an event:

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

#[event]
pub struct TransferInitiated {
    pub from_chain: u64,
    pub to_chain: u64,
    pub nonce: u64,
    pub token_address: Pubkey,
    pub amount: u64,
    pub sender: Pubkey,
    pub recipient: [u8; 32],
}

// Emitting the event inside an instruction:
emit!(TransferInitiated {
    from_chain: SOLANA_CHAIN_ID,
    to_chain: target_chain,
    nonce: config.nonce,
    token_address: ctx.accounts.mint.key(),
    amount,
    sender: ctx.accounts.user.key(),
    recipient,
});
```

Explanation: The `#[event]` macro defines a struct whose instances become log entries. Each field of the struct is recorded on-chain when `emit!` is called. In our `initiate_transfer` function, after locking the tokens, we construct `TransferInitiated` with all relevant fields and call `emit!`. Off-chain relayers subscribe to these events (by listening to the program's logs) to pick up transfer requests. Similarly, a `TransferCompleted` event can be defined and emitted after finalizing an incoming

transfer, letting external services know that the recipient's tokens have been minted or released.

Security Audit Checklist

Building a secure cross-chain bridge on Solana requires careful attention to many potential pitfalls. Below is a checklist of vulnerabilities and best practices specific to bridging protocols:

- **Replay Attacks (Missing or Improper Nonce):** Every cross-chain message must include a unique identifier (nonce or sequence number) and the contract must track processed nonces. If nonces are not managed correctly (e.g., using a global counter without per-source tracking), an attacker could replay a valid transfer message multiple times to drain tokens. Ensure the program either marks each nonce as used (via state or PDAs) or uses a strictly increasing counter per source chain.
- **Validator Set Compromise / Manipulation:** The bridge's security hinges on its validator network. Protect the `BridgeConfig` from unauthorized changes: require a multisig or on-chain governance to update the validator set. Hardcoding a single admin key is risky; consider time-locked upgrades or requiring a threshold of existing validators to add new ones. Never allow arbitrary public keys as validators without checks.
- **Insufficient Signature Verification:** Validate all parts of the message are signed. For example, an attacker might tamper with the amount or recipient if only partial data is signed. Ensure the entire message payload (amount, token address, chain IDs, nonce, sender, recipient, etc.) is hashed and verified. Use Solana's `secp256k1_program` correctly for ECDSA or ed25519 signature verification and handle malformed inputs properly.
- **Threshold & Signature Logic Bugs:** If the signature verification loop or threshold logic is flawed, it may accept fewer signatures than intended or count duplicates. Check for duplicate signatures counting only once. Ensure the threshold logic (`valid_signatures >= threshold`) is correct and the data structures (e.g., arrays or slices) cannot overflow or underflow.
- **Chain ID Mismatch:** Include chain identifiers in the message and enforce them. A message intended for one target chain (e.g., Ethereum) should not be valid on another (e.g., Solana). Hardcode or pass the expected `to_chain` value to `verify_message` . Reject any message where `to_chain` does not match the current chain's ID.
- **Token Address Validation:** If bridging multiple tokens, ensure the token address in the message matches the expected mint. An attacker could send a message with a different token address to mint the wrong asset. Typically, the bridge config holds a

mapping of supported tokens and their vaults or mints. Always verify the `token_address` from the message against a whitelist in the config.

- **Improper PDA Derivation:** Ensure that vault and authority PDAs use unambiguous seeds (e.g. include program ID, chain ID, and token mint). Incorrect seeds might allow a malicious actor to guess or collide with a PDA, potentially hijacking the vault. Always use `#[derive(ProgramAccount)]` or Anchor's `#[account(seeds = ...)]` to have the runtime enforce correct PDAs.
- **Race Conditions & Double Execution:** After verifying a message, immediately mark its nonce as used before any state changes (like minting). If an attacker tries to replay while it's still processing, early marking prevents a double-spend. Alternatively, derive a PDA for each nonce (e.g., seed with nonce) to act as a "used flag" and require it be uninitialized.
- **Liquidity Exploits:** For bridged assets, mismatches in token decimals or total supply can lead to loss or mispricing. Ensure the bridge enforces consistent decimals and that minted token supply does not exceed locked supply (if using a 1:1 model). Validate that burning and minting operations adjust supply correctly.
- **Event Monitoring Reliance:** The bridge logic often relies on events seen by off-chain relayers. If an event emission fails (e.g., due to an instruction error), the off-chain side might not notice the transfer. Make sure the smart contract always emits events after state changes, and consider on-chain events (like writing to an account) as backups.
- **Anchor-Specific Pitfalls:**
  - **Account Constraints:** Double-check that each `#[account]` constraint (like `has_one`, `seeds`, `bump`, etc.) is correct. A missing `has_one` could allow using the wrong config or vault.
  - **Signer Checks:** When requiring a signature (`Signer<'info>`), ensure it's the intended authority. For example, if the PDA (mint authority) needs to sign a CPI, you must derive the PDA seed and bump correctly and call it via `invoke_signed`.
  - **Upgrades and Data Layout:** If the program is upgradable, migrating data (like `BridgeConfig`) must preserve field order or use `#[repr(C)]`. Changing the struct order without migration logic can corrupt data.
  - **Token Program Attacks:** Ensure that the `token_program` account passed into CPIs is indeed the official SPL Token program. Although Anchor usually enforces this, if using raw account metas, a malicious caller might attempt to substitute it.
- **Time and Finality:** Solana is fast but not instant. While Solana's finality is relatively quick, if bridging with slower chains, consider time windows. For example, reject messages that appear to be too far in the past or future relative to known block times to mitigate certain types of replay or front-running.

- **Denial of Service (DOS) via Signature Flood:** If the program naively processes an unbounded list of signatures, an attacker could submit many signatures (even invalid ones) to bloat execution. Implement a reasonable limit on number of signatures or require that the relay logic sends exactly `threshold` valid ones (rejecting extras), to avoid excessive compute on-chain.
- **Incorrect Fee Handling:** Bridges often charge fees. Ensure fee logic (if any) is clear and not circumventable. For example, if collecting a fee token, confirm the vaults and burn addresses are correct and can't be tricked by address collisions.

Perpetuals Trading Protocol

Overview

A **perpetuals trading protocol** is an on-chain derivative exchange where users can take leveraged long or short positions on assets with *no expiration date*. Perpetuals differ from spot markets because they use funding rates to tether the on-chain price to an external reference (usually from an oracle). On Solana, such a protocol typically involves **liquidity pools (vaults)** for each asset, **user position accounts**, **margin/collateral tracking**, **funding rate mechanisms**, **price oracles**, and **fee management**. Users deposit collateral into the protocol (often a stablecoin), then open positions using leverage. The protocol's smart contracts (written in Rust with Anchor) handle the internal accounting: updating each position's size, entry price, collateral, and unrealized PnL. Funding rates are periodically applied to transfer value between longs and shorts to keep on-chain prices aligned with oracle prices. If a position's margin ratio falls below maintenance thresholds, the position can be liquidated to protect the protocol. Throughout, trading fees are collected and routed to protocol reserves or LPs. The Anchor framework structures these components as on-chain **program accounts** (using `#[account]`) and **instructions** (methods taking `Context<...>` with specified account parameters).

Deep Dive

Liquidity Vaults / Pools

Purpose: A vault (or liquidity pool) securely holds collateral or asset tokens and tracks global metrics. In a perpetuals protocol, each supported asset has a *Custody* or *Pool* account that stores total collateral, fee pools, and parameters (like oracle address or allowed leverage). Often there's also an underlying SPL Token account (owned by the program PDA) that physically holds deposited tokens.

Example Anchor Code: Initialize a new custody pool (vault) with a token account for collateral.

```rust
#[account]
pub struct Custody {
    pub asset_mint: Pubkey,        // Token mint of this market
    pub vault_account: Pubkey,     // PDA token account holding collateral
    pub total_collateral: u64,     // Total collateral deposited
    pub fee_collected: u64,        // Accumulated fees in protocol
    pub oracle: Pubkey,           // Associated price oracle
    pub max_leverage: u64,        // e.g. 10x = 10_000 (if fixed-point BPS)
    pub maintenance_margin: u64, // e.g. 20% = 2_000
    // ... other parameters ...
}

#[derive(Accounts)]
pub struct InitCustody<'info> {
    #[account(init, payer = admin, space = 8 + std::mem::size_of::<Custody>())]
    pub custody: Account<'info, Custody>,
    #[account(init,
        payer = admin,
        token::mint = asset_mint,
        token::authority = custody)]
    pub vault_account: Box<Account<'info, TokenAccount>>,
    pub asset_mint: Account<'info, Mint>,
    pub admin: Signer<'info>,
    pub system_program: Program<'info, System>,
    pub token_program: Program<'info, Token>,
    pub rent: Sysvar<'info, Rent>,
}

pub fn init_custody(ctx: Context<InitCustody>, oracle: Pubkey, max_leverage: u64) -> Result<()> {
    let c = &mut ctx.accounts.custody;
    c.asset_mint = ctx.accounts.asset_mint.key();
    c.vault_account = ctx.accounts.vault_account.key();
    c.total_collateral = 0;
    c.fee_collected = 0;
    c.oracle = oracle;
    c.max_leverage = max_leverage;
```

```
        c.maintenance_margin = max_leverage / 5; // e.g. 20% if 5 = 100%
        Ok(())
    }
```

Explanation: The `Custody` struct (marked with `#[account]`) holds state for one trading market. It records the token mint, the associated vault (a PDA token account), total collateral and fees, and trading parameters. The `InitCustody` context uses Anchor's `init` to create both the custody state account and the SPL token `vault_account`. On init, we set initial fields like the oracle and leverage limits. In practice, users depositing collateral will transfer tokens into `vault_account`, and the program will update `custody.total_collateral`.

User Positions

Purpose: Each trader's open position is tracked in a `Position` account. It includes the position size (in base asset units or quote-value), the direction (long/short), the entry price, and the collateral placed. Position accounts allow partial closes and multiple positions per user if desired.

Example Anchor Code: Open a new leveraged position for a user.

```rust
#[account]
pub struct Position {
    pub owner: Pubkey,        // Account that opened this position
    pub custody: Pubkey,      // Which market this position is on
    pub size: u64,            // Position size in quote units
    pub entry_price: u64,     // Price (from oracle) at entry
    pub collateral: u64,      // Collateral amount posted
    pub is_long: bool,        // true = long, false = short
    // … additional fields like cumulative funding at entry …
}

#[derive(Accounts)]
pub struct OpenPosition<'info> {
    #[account(init, payer = user, space = 8 + std::mem::size_of::<Position>())]
    pub position: Account<'info, Position>,
    #[account(mut)]
    pub custody: Account<'info, Custody>,
    #[account(mut)]
    pub user_collateral: Account<'info, TokenAccount>,
    #[account(mut)]
    pub vault_account: Account<'info, TokenAccount>, // custody vault
    pub user: Signer<'info>,
    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
}

pub fn open_position(ctx: Context<OpenPosition>, size: u64, is_long: bool) -> Result<()> {
    let custody = &mut ctx.accounts.custody;
    let price = /* fetch current price from custody.oracle, omitted */;
    let required_collateral = size / custody.max_leverage; // e.g. 10x leverage
    // Transfer collateral from user to vault
    let cpi_accounts = Transfer {
        from: ctx.accounts.user_collateral.to_account_info(),
        to: ctx.accounts.vault_account.to_account_info(),
        authority: ctx.accounts.user.to_account_info(),
    };
```

```
        let cpi_ctx =
    CpiContext::new(ctx.accounts.token_program.to_account_info(),
    cpi_accounts);
        anchor_spl::token::transfer(cpi_ctx, required_collateral)?;
        custody.total_collateral += required_collateral;
        // Initialize position account
        let pos = &mut ctx.accounts.position;
        pos.owner = *ctx.accounts.user.key;
        pos.custody = custody.key();
        pos.size = size;
        pos.entry_price = price;
        pos.collateral = required_collateral;
        pos.is_long = is_long;
        Ok(())
    }
```

Explanation: In `OpenPosition`, we init a new `Position` account. First, we fetch the current oracle price (omitted for brevity) and compute the required collateral given the protocol's max leverage. We use an SPL token CPI to transfer that collateral from the user's token account into the protocol's vault (`vault_account`). We increment the custody's `total_collateral`. Finally, we populate the `Position` fields: who owns it, which custody market, the position size, entry price, collateral, and direction. This account will be used for future updates (like marking to market or closing).

Funding Rates

Purpose: Funding rates keep the perpetual's price in line with an external index by charging one side (long or short) a periodic fee. The protocol tracks a cumulative funding factor in the `Custody` or `Pool` state and applies the difference to each position upon update or close.

Example Anchor Code: Update a market's cumulative funding given elapsed time.

```rust
#[derive(Accounts)]
pub struct UpdateFunding<'info> {
    #[account(mut)]
    pub custody: Account<'info, Custody>,
    pub clock: Sysvar<'info, Clock>,
}

pub fn update_funding(ctx: Context<UpdateFunding>) -> Result<()> {
    let custody = &mut ctx.accounts.custody;
    let current_ts = ctx.accounts.clock.unix_timestamp as u64;
    // Simple example: funding_rate in BPS per second
    let elapsed = current_ts.checked_sub(custody.last_funding_ts)
        .ok_or(ErrorCode::InvalidTimestamp)?;
    let delta_rate = custody.funding_rate_bps.checked_mul(elapsed)
        .ok_or(ErrorCode::Overflow)?;
    custody.cumulative_funding =
custody.cumulative_funding.checked_add(delta_rate)
        .ok_or(ErrorCode::Overflow)?;
    custody.last_funding_ts = current_ts;
    Ok(())
}
```

Explanation: Here we define an `UpdateFunding` instruction. It takes the `Custody` (mutable) and the Solana `Clock` sysvar. We compute the time elapsed since the last funding update and multiply it by a fixed funding rate parameter (`funding_rate_bps` per second, in basis points). This increases the market's `cumulative_funding`. In practice, when a user's position is modified (e.g. closed or partially closed), the contract would compute the funding owed by comparing the position's entry funding index to the current `cumulative_funding`. If the position is long, it pays funding to the pool when the index rose; if short, it receives funding. The key is tracking a global funding index and updating it on a schedule or event.

Margin and Leverage

Purpose: The protocol enforces collateral requirements so that no position becomes under-collateralized. **Initial margin** determines the max leverage (e.g. 10x means a user must post at least 1/10 of position value). **Maintenance margin** is a lower threshold (e.g. 20%) below which liquidation can happen. The contract checks these when positions are opened or modified.

Example Anchor Code: Verify maintenance margin before certain actions.

```rust
#[derive(Accounts)]
pub struct CheckMargin<'info> {
    pub custody: Account<'info, Custody>,
    pub position: Account<'info, Position>,
}

pub fn check_margin(ctx: Context<CheckMargin>) -> Result<()> {
    let custody = &ctx.accounts.custody;
    let pos = &ctx.accounts.position;
    // Compute current position value based on latest price
    let price = /* fetch current price from custody.oracle */;
    let position_value = pos.size; // assume size is in quote (value) terms
    let collateral = pos.collateral;
    // maintenance requirement (as BPS): collateral * 10_000 / position_value >= maintenance_margin
    let required = position_value.checked_mul(custody.maintenance_margin)
        .ok_or(ErrorCode::Overflow)?;
    let provided = collateral.checked_mul(10_000).ok_or(ErrorCode::Overflow)?;
    require!(provided >= required, ErrorCode::InsufficientCollateral);
    Ok(())
}
```

Explanation: This snippet shows how to check that a position still meets the maintenance margin. We retrieve the current price (omitted) and treat `pos.size` as the notional value. We compare the posted collateral (scaled by 10,000 for BPS) to the required maintenance threshold. If `collateral * 10000 < position_value * maintenance_margin`, it fails. Such checks would typically run before letting a position remain open; if it fails, the user must add collateral or risk liquidation.

Liquidations

Purpose: If a position's margin ratio dips below maintenance levels, the protocol allows third parties to liquidate it. Liquidation usually closes the position, seizes part of the collateral (a penalty), and gives it to a liquidation fund or the liquidator. This protects the protocol from bad debt.

Example Anchor Code: Liquidate an undercollateralized position.

```rust
#[derive(Accounts)]
pub struct Liquidate<'info> {
    #[account(mut, close = liquidator)]
    pub position: Account<'info, Position>,
    #[account(mut)]
    pub custody: Account<'info, Custody>,
    #[account(mut)]
    pub liquidator: Signer<'info>,
    #[account(mut)]
    pub liquidation_vault: Account<'info, TokenAccount>,
    pub token_program: Program<'info, Token>,
}

pub fn liquidate(ctx: Context<Liquidate>) -> Result<()> {
    let pos = &ctx.accounts.position;
    let cust = &mut ctx.accounts.custody;
    // Compute current price and value
    let price = /* fetch current price */;
    let pos_value = pos.size;
    // Determine safe threshold (similar to maintenance_margin check)
    let provided = pos.collateral.checked_mul(10_000).unwrap();
    let required =
pos_value.checked_mul(cust.maintenance_margin).unwrap();
    if provided >= required {
        return Err(ErrorCode::NotLiquidatable.into());
    }
    // Calculate liquidation penalty (e.g. 10%)
    let penalty = pos.collateral / 10;
    // Transfer penalty to liquidation vault
    let cpi = Transfer {
        from: ctx.accounts.custody.to_account_info().clone(),
        to: ctx.accounts.liquidation_vault.to_account_info().clone(),
        authority: ctx.accounts.custody.to_account_info().clone(),
    };
    let cpi_ctx =
CpiContext::new(ctx.accounts.token_program.to_account_info(), cpi);
    anchor_spl::token::transfer(cpi_ctx, penalty)?;
    cust.total_collateral =
cust.total_collateral.checked_sub(pos.collateral)
        .ok_or(ErrorCode::Overflow)?;
```

```
38      // Closing `position` account returns its rent to `liquidator` (per
    `close = liquidator`)
39      Ok(())
40  }
```

Explanation: In this instruction, a liquidator passes the `Position` account marked as `close = liquidator`, meaning Anchor will close it and send its lamports rent back to the liquidator. We first re-check that the position truly fails maintenance margin; if not, we return an error. We compute a penalty (e.g. 10% of collateral) and transfer it from the protocol custody (via CPI on the token program) to a `liquidation_vault`. We also deduct the full collateral from the market's totals. The remaining collateral (after penalty) conceptually goes to settle losses or could be returned to the original user. The account close returns leftover SOL to the liquidator, incentivizing them to initiate this. In practice, the code would fully settle the position (e.g. record realized PnL) before burning or closing it.

Oracles

Purpose: Perpetuals rely on an external price feed (e.g. Pyth, Switchboard) for the fair market price. The on-chain program reads the oracle account to set entry/exit prices and funding rates, guarding against price manipulation by frequent repricing against a known feed.

Example Anchor Code: Reading a Pyth oracle price.

```rust
#[derive(Accounts)]
pub struct ReadOraclePrice<'info> {
    /// CHECK: Oracle account, verified off-chain that this is a valid
    price feed
    pub price_feed: AccountInfo<'info>,
}

pub fn get_oracle_price(ctx: Context<ReadOraclePrice>) -> Result<u64> {
    let data = &ctx.accounts.price_feed.try_borrow_data()?;
    // Suppose we know the layout and offset for the current price
    let price = u64::from_le_bytes(data[208..216].try_into().unwrap());
    Ok(price)
}
```

Explanation: This simplistic snippet shows fetching a price from a raw account. In reality, one would use an oracle library or carefully parse the on-chain account format (for Pyth

it's usually using the pyth-sdk). The key point is that the program takes an oracle `AccountInfo`, reads its data, and extracts the current aggregated price. That price is then used by other instructions (e.g. to set `entry_price` or compute liquidation). Anchor `#[account]` cannot verify oracle contents, so we mark it `CHECK` (with caution). In an audited protocol, one must ensure only legitimate oracle accounts can be passed (often by hardcoding known oracle pubkeys).

Fee Model

Purpose: The protocol collects fees on trades and funding payments. A simple model is a fixed percentage per trade. Fees might go into a protocol-owned vault or be redistributed to liquidity providers.

Example Anchor Code: Applying a trade fee during a swap.

```rust
#[derive(Accounts)]
pub struct Trade<'info> {
    #[account(mut)]
    pub custody: Account<'info, Custody>,
    #[account(mut)]
    pub user_collateral: Account<'info, TokenAccount>,
    #[account(mut)]
    pub vault_account: Account<'info, TokenAccount>,
    pub user: Signer<'info>,
    pub token_program: Program<'info, Token>,
}

pub fn trade(ctx: Context<Trade>, amount: u64) -> Result<()> {
    // Example: charge 0.1% fee on `amount`
    let fee = amount.checked_div(1000).ok_or(ErrorCode::Overflow)?;
    // Transfer (amount - fee) from user to vault
    let to_vault = amount.checked_sub(fee).ok_or(ErrorCode::Overflow)?;
    let cpi_to_vault = Transfer {
        from: ctx.accounts.user_collateral.to_account_info(),
        to: ctx.accounts.vault_account.to_account_info(),
        authority: ctx.accounts.user.to_account_info(),
    };

anchor_spl::token::transfer(CpiContext::new(ctx.accounts.token_program.t
cpi_to_vault), to_vault)?;
    // Transfer fee to fee_collector (here, just add to custody.fee_coll
    ctx.accounts.custody.fee_collected =
ctx.accounts.custody.fee_collected.checked_add(fee)
        .ok_or(ErrorCode::Overflow)?;
    Ok(())
}
```

Explanation: In this `trade` instruction, we deduct a 0.1% fee (`amount / 1000`). The net amount (`amount - fee`) is transferred from the user's token account into the protocol vault as usual trade flow. The fee portion is added to `custody.fee_collected` , representing the protocol's earnings. In a full implementation, these fees might later be claimable by governance or distributed to liquidity providers. The key is to perform precise integer math and handle any rounding (here small rounding to zero can cause tiny dust fees, which should be tracked).

Audit Checklist

- **Funding Rate Vulnerabilities:** Ensure funding is based on a reliable external index. *Attack:* A malicious trader could manipulate on-chain price or wait for stale price before a funding update, profiting at counterparty expense. *Mitigation:* Use timely oracle data, update funding atomically with price moves, and cap funding rates.
- **Liquidation Logic Errors:** Check the maintenance margin and liquidation math carefully. *Attack:* Off-by-one or incorrect threshold could allow positions to become under-collateralized (causing losses) or be liquidated unfairly. *Mitigation:* Test extreme cases; ensure consistent rounding direction (e.g. always round up collateral requirement).
- **Front-Running Funding Payments:** If funding is applied separately from trades, bots can game it by opening/closing around funding ticks. *Attack:* Observe an impending funding payment, take an opposite position just before and exit after to capture payment. *Mitigation:* Integrate funding accrual in every trade instruction or use a timestamp check so funding can't be arbitraged away.
- **Margin Calculation Bugs:** Verify all leverage and margin formulas. *Attack:* Integer division could truncate required collateral, allowing slightly larger leveraged trades. *Mitigation:* Use fixed-point arithmetic with ample precision, require a safety cushion, and add unit tests for marginal cases.
- **PNL Accounting Flaws:** Ensure realized PnL is recorded correctly when closing or partial-close positions. *Attack:* If entry price isn't updated on rollovers, traders might profit from repeated opens/closes. *Mitigation:* After each trade, update a position's average entry price and cumulative funding to prevent "ghost" profits.
- **Collateral Withdrawals While In-Position:** Prevent users from withdrawing collateral below their required margin. *Attack:* A user could try to drain collateral after opening a position, risking others. *Mitigation:* Any withdraw instruction must re-check maintenance requirements and adjust or block if unsafe.
- **Fee Rounding and Dust:** Small remainders from fee calculations can accumulate or be lost. *Issue:* Neglecting these "dust" amounts can unbalance accounting. *Mitigation:* Track leftover dust in a reserve or distribute it periodically, and use high-precision BPS (basis points) or integer math with a large base.
- **Oracle Integrity Issues:** Although not a core bug in logic, an attack on the price oracle directly impacts funding, liquidations, and pricing. *Mitigation:* Use well-audited oracles (e.g. Pyth, Switchboard) and consider validations (e.g. max price change per block).
- **Permission or Admin Mistakes:** If administrative instructions (like pausing funding or updating parameters) lack checks, an attacker might exploit them. *Mitigation:* Restrict admin keys (multisig), and ensure state changes can't bypass necessary conditions.

- **State Synchronization:** Ensure that all interdependent state (e.g. `total_collateral`, `fee_collected`, cumulative funding) is consistently updated in each instruction. *Issue:* Missing an update in one code path (like forgetting to update global AUM after a trade) can lead to supply-demand mismatches.
