Architectural Diagram

1. Core Program Structure

Vault Program

Responsibilities:

- Manages user crypto deposits and withdrawals
- Implements collateral safety mechanisms
- Controls access to deposited funds
- Calculates loan-to-value ratios

Key Instructions:

- `initialize_vault()` - Creates user vault PDA
- `deposit_collateral()` - Accepts crypto deposits
- `withdraw_collateral()` - Processes withdrawals
- `calculate_spending_power()` - Determines available credit

Spending Program

Responsibilities:

- Authorizes purchase transactions
- Validates sufficient collateral backing
- Updates user spending balances
- Maintains transaction history

Key Instructions:

- `authorize_purchase()` - Validates and approves spending
- `update_balance()` - Adjusts available credit
- `check_collateral_ratio()` - Ensures safe lending ratios

Yield Program

Responsibilities:

- Integrates with DeFi protocols for earning
- Auto-compounds user returns
- Tracks individual user earnings
- Manages yield strategy optimization

Key Instructions:

- `stake_collateral()` - Deploys funds to earning protocols
- `compound_earnings()` - Reinvests generated yields
- `calculate_user_share()` - Tracks individual returns

Liquidation Program

Responsibilities:

- Monitors collateral value in real-time
- Executes emergency selling when needed
- Protects platform from undercollateralization
- Notifies users of liquidation events

Key Instructions:

- `monitor_prices()` - Continuous price surveillance
- `trigger_liquidation()` - Executes emergency sales
- `notify_user()` - Sends liquidation alerts

Payment Program

Responsibilities:

- Processes merchant settlements
- Converts crypto value to fiat
- Integrates with traditional payment rails
- Manages escrow for pending transactions

Key Instructions:

- `process_merchant_payment()` - Settles with merchants
- `convert_crypto_to_fiat()` - Handles currency conversion
- `update_merchant_balance()` - Tracks merchant earnings

2. Account Architecture

Program Derived Addresses (PDAs)

User Vault PDA

- **Seeds:** `[user_pubkey, "vault", bump]`
- **Data:** Collateral amounts, deposit timestamps, withdrawal permissions
- **Owner:** Vault Program

Spending Account PDA

- **Seeds:** `[user_pubkey, "spending", bump]`
- **Data:** Available credit, transaction history, spending limits
- **Owner:** Spending Program

Yield Account PDA

- **Seeds:** `[user_pubkey, "yield", bump]`
- **Data:** Staked amounts, earned rewards, strategy allocations
- **Owner:** Yield Program

Merchant Account PDA

- **Seeds:** `[merchant_pubkey, "merchant", bump]`
- **Data:** Payment history, settlement preferences, fees earned
- **Owner:** Payment Program

Token Accounts

- **User Token Account:** Holds user's original crypto before deposit
- **Vault Token Account:** Stores collateral securely within protocol
- **Yield Token Account:** Represents staked positions in DeFi protocols
- **Merchant Token Account:** Escrow for pending merchant payments

3. Detailed Instruction Flows

Liquidation Program Flows

- **`monitor_prices()`**
    1. An off-chain keeper service calls this instruction at regular intervals.
    2. The instruction fetches the latest asset prices from a primary oracle (e.g., Pyth).
    3. It iterates through all active `User Vault PDAs`.
    4. For each vault, it calculates the current collateralization ratio: `Total Collateral Value / Loan Amount`.
    5. **Decision**: If a vault's ratio is below the defined liquidation threshold, the instruction invokes `trigger_liquidation()` for that vault.
    6. The process repeats on the next call.
- **`trigger_liquidation()`**
    1. This instruction is invoked by `monitor_prices()` or can be permissionlessly called by registered liquidators.
    2. It requires the target `User Vault PDA` and its associated `Vault Token Account`.
    3. It seizes a portion of the collateral from the `Vault Token Account`.
    4. The seized collateral is immediately sold on an integrated DEX (e.g., Jupiter) to repay the user's outstanding loan.

5. Any funds remaining after repaying the loan and a penalty fee are returned to the user's vault.
6. The `loan_amount` on the `User Vault PDA` is reduced accordingly (or set to zero if fully paid).
7. The instruction concludes by invoking `notify_user()`.

- **`notify_user()`**
  1. This instruction is called at the end of the `trigger_liquidation()` flow.
  2. It emits an on-chain event containing the details of the liquidation (e.g., amount liquidated, assets sold, final debt).
  3. Off-chain listening services capture this event and dispatch a notification (e.g., email, text message, push notification) to the affected user.

Payment Program Flows

- **`process_merchant_payment()`**
  1. A merchant's system (e.g., point-of-sale) initiates this flow once a user's purchase is approved by the `Spending Program`.
  2. The instruction verifies the transaction signature and details.
  3. It transfers the equivalent crypto value from the protocol's main treasury to the `Merchant Token Account` (escrow).
  4. It calls `update_merchant_balance()` to log the pending payment.
- **`convert_crypto_to_fiat()`**
  1. This instruction is typically executed by a keeper as part of a scheduled settlement batch (e.g., daily).
  2. It interacts via API with an integrated crypto-to-fiat off-ramp service.
  3. The crypto held in the `Merchant Token Account` is sent to the off-ramp service.
  4. The service executes the conversion and initiates a fiat transfer to the merchant's linked bank account.
- **`update_merchant_balance()`**
  1. This instruction is called by `process_merchant_payment()`.
  2. It accesses the specific `Merchant Account PDA`.
  3. It updates the `Payment history` and other data fields to record the new transaction as "pending."
  4. Once an off-chain confirmation of fiat settlement is received, the transaction status in the PDA is updated to "cleared," and the escrowed funds are released.

4. External Integration Points

Price Oracles

- **Pyth Network**: Serves as the primary source for real-time price feeds for all major cryptocurrencies used as collateral.
- **Switchboard**: Functions as a backup oracle to ensure data redundancy and for independent price validation.
- **Integration**: On-chain programs directly query these oracle accounts to retrieve prices for collateral valuation, spending power calculation, and liquidation checks.

DeFi Protocols

- **Marinade Finance**: Utilized for liquid staking of `$SOL` collateral.
- **Lido**: Provides additional liquid staking options for `$SOL` and other assets.
- **Solend**: Integrated for lending and borrowing functionalities to generate yield on deposited assets.
- **Integration**: The `Yield Program` performs automated yield farming by using Cross-Program Invocations (CPIs) to interact with these protocols.

Payment Systems

- **Traditional Rails**: Integrates with services like Stripe and direct bank transfers for merchant settlement in fiat currency.
- **Compliance**: The integration layer includes processes for KYC/AML (Know Your Customer/Anti-Money Laundering) verification to meet regulatory requirements.
- **Integration**: The `Payment Program` uses off-chain API calls to these systems to handle fiat conversion and final settlement to merchant accounts.

5. Key User Flows

This section describes the primary operational sequences from a user's perspective, detailing the program interactions at each stage.

Deposit & Setup Flow

1. The user connects their wallet to the platform front-end.
2. The `Vault Program` is invoked to create a user-specific `Vault PDA`.
3. The user transfers cryptocurrency from their wallet to the `Vault Token Account` associated with their new PDA.
4. The system calculates the user's spending power, set at 50-70% of the deposited collateral's value.
5. The `Spending Account PDA` is created or updated with the available credit amount.

Purchase Flow

1. The user initiates a purchase at a connected merchant.
2. The `Spending Program` is called to validate collateral sufficiency for the transaction amount.
3. The `Payment Program` processes the merchant settlement upon approval from the `Spending Program`.
4. The merchant receives the payment in fiat currency nearly instantly.
5. The user's spending balance on their `Spending Account PDA` is decreased accordingly.

Yield Generation Flow

1. The `Vault Program` makes a CPI to the `Yield Program`, transferring a portion of the user's collateral.
2. The `Yield Program` stakes or lends these funds in the optimal integrated DeFi protocols.
3. Generated returns are automatically compounded, with proceeds sent back to the `Vault Program` to increase the user's collateral position.
4. The `Yield Account PDA` tracks the earnings attributable to the individual user.
5. The user's increased collateral value results in a corresponding increase in their available spending power.

Emergency Liquidation Flow

1. The `Liquidation Program` continuously monitors collateral prices via oracle feeds.
2. If a user's collateral drops below the safety threshold (typically 80% Loan-to-Value), the liquidation process is triggered.
3. The program automatically sells a portion of the user's collateral on an integrated DEX.
4. The proceeds are used to repay the user's debt and maintain a healthy collateral ratio.
5. The user is notified of the liquidation event through on-chain events and off-chain communication channels.

6. Security Considerations

This section outlines the key security measures implemented at the protocol level to protect user funds and maintain system integrity.

Access Controls

- Only the verified owner of a `Vault PDA` can initiate a withdrawal of the underlying collateral.
- Spending is only authorized if backed by a valid and sufficient collateral position.

- The liquidation mechanism can only be triggered by on-chain price conditions, not by arbitrary administrative action.
- Protocol upgrades and changes to critical parameters are controlled by a multi-signature wallet.

Risk Management

- The protocol enforces conservative loan-to-value (LTV) ratios, initially set between 50-70%.
- Risk is managed through real-time price monitoring and an automated liquidation engine.
- Systemic risk is mitigated by diversifying yield generation across multiple, vetted DeFi protocols.
- An emergency pause functionality is built into the core programs to halt activity in case of a critical vulnerability or market event.

Data Integrity

- All state changes to accounts are executed exclusively through verified program instructions.
- Cross-program invocations (CPIs) are used to ensure atomic, validated state updates between programs.
- The transaction history is immutable and publicly verifiable on the blockchain.
- Regular, automated checks of all vault collateral ratios are performed to ensure platform solvency.

7. Technical Implementation Notes

This section covers specific technical details related to the protocol's construction and operation on the Solana network.

Cross-Program Invocations (CPIs)

- `Vault → Yield`: To transfer collateral for staking and yield farming.
- `Spending → Vault`: To validate collateral health before authorizing purchases.
- `Payment → Spending`: To signal a successful payment and debit the user's account.
- `Liquidation → Vault`: To seize and sell collateral during an emergency liquidation.

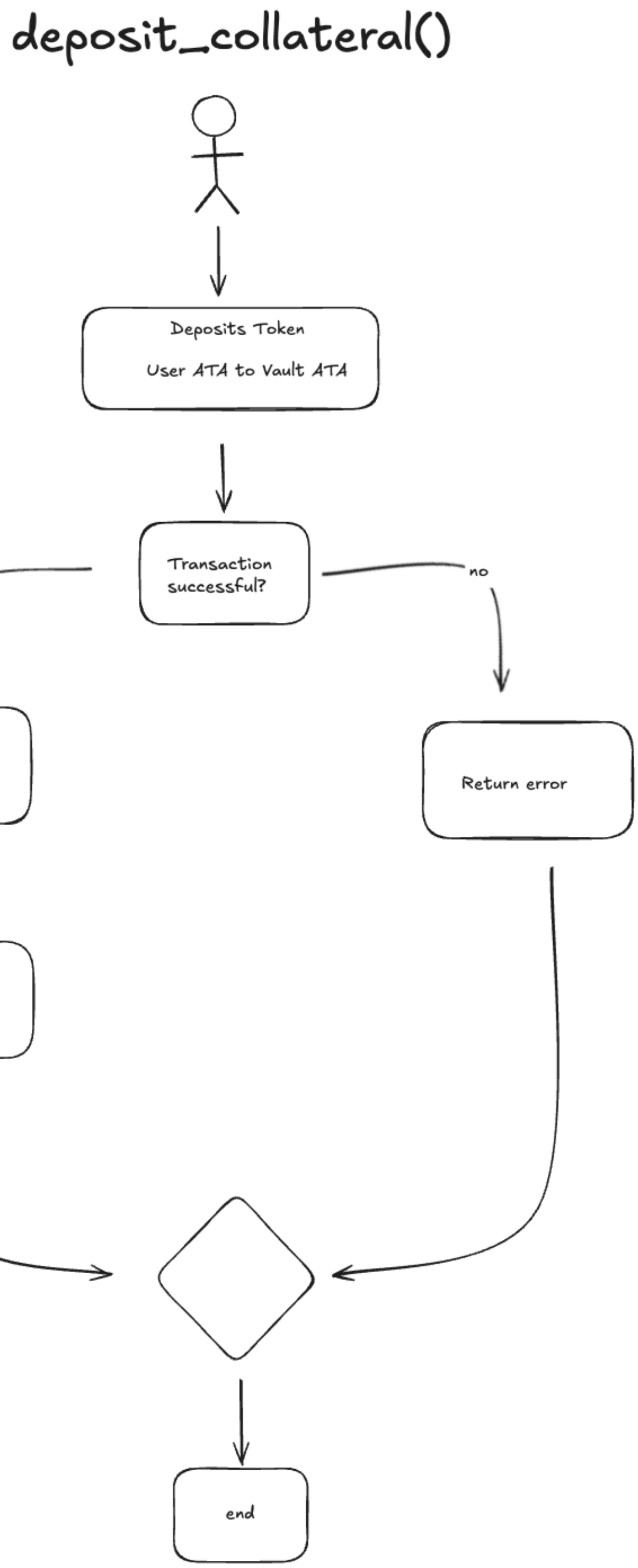
Error Handling

- The programs include robust checks and return specific errors for conditions such as:
    - Insufficient collateral errors during spending or withdrawal attempts.
    - Oracle price feed failures or data becoming stale.
    - Failures during CPIs to external DeFi protocols.
    - Network congestion and transaction timeout handling.

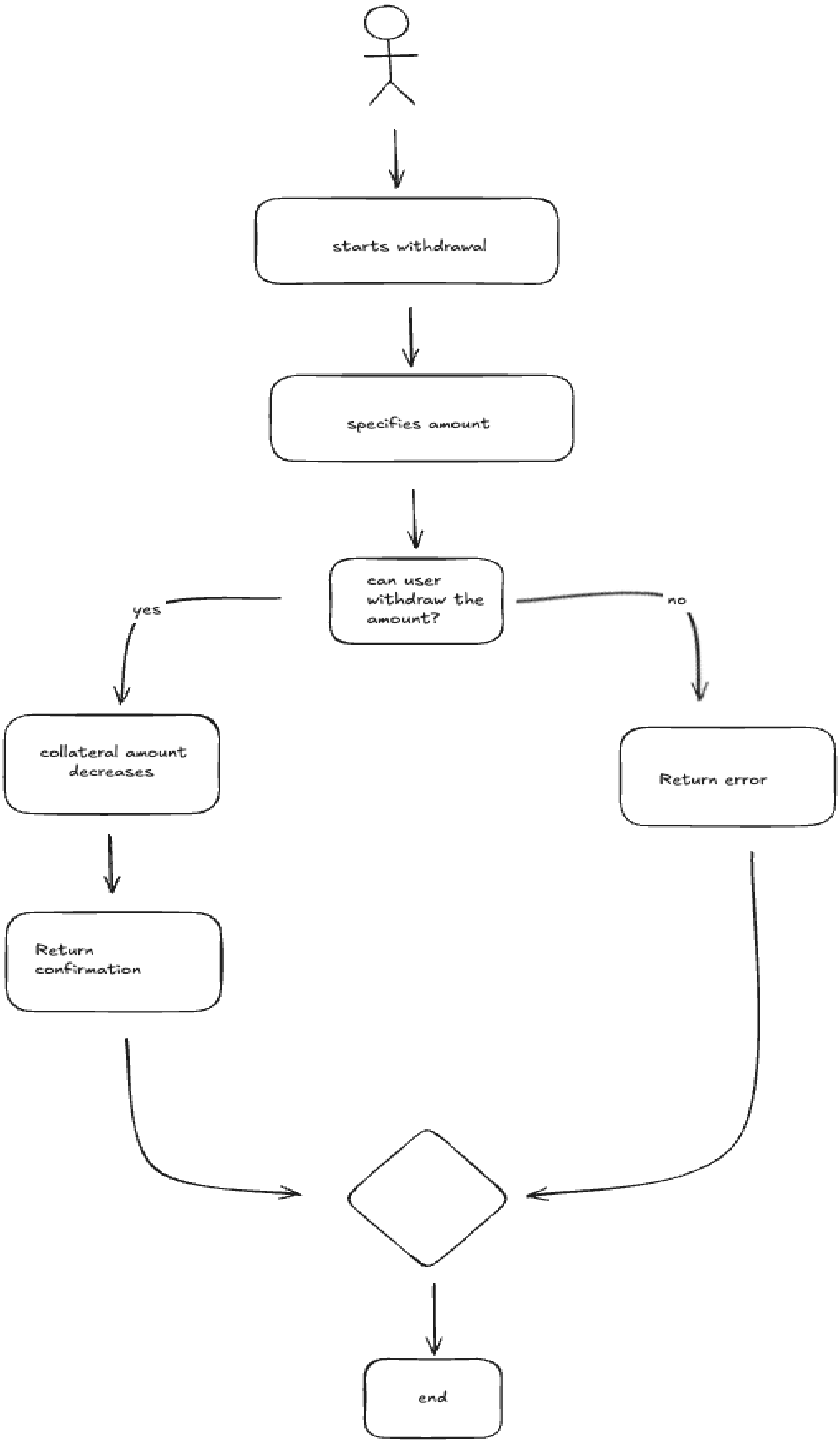
Scalability Considerations

- **Batch Processing**: Designing instructions that can process multiple accounts or events in a single transaction (e.g., batch liquidations or yield distributions).
- **Efficient PDA Derivation**: Utilizing optimized PDA derivation and off-chain caching to reduce on-chain compute load.
- **Optimized Account Rent and Storage**: Writing programs to use the minimum account space required and closing unused accounts to return rent to users.
- **Gas-efficient Instruction Design**: Minimizing the computational complexity and account loads of each instruction.

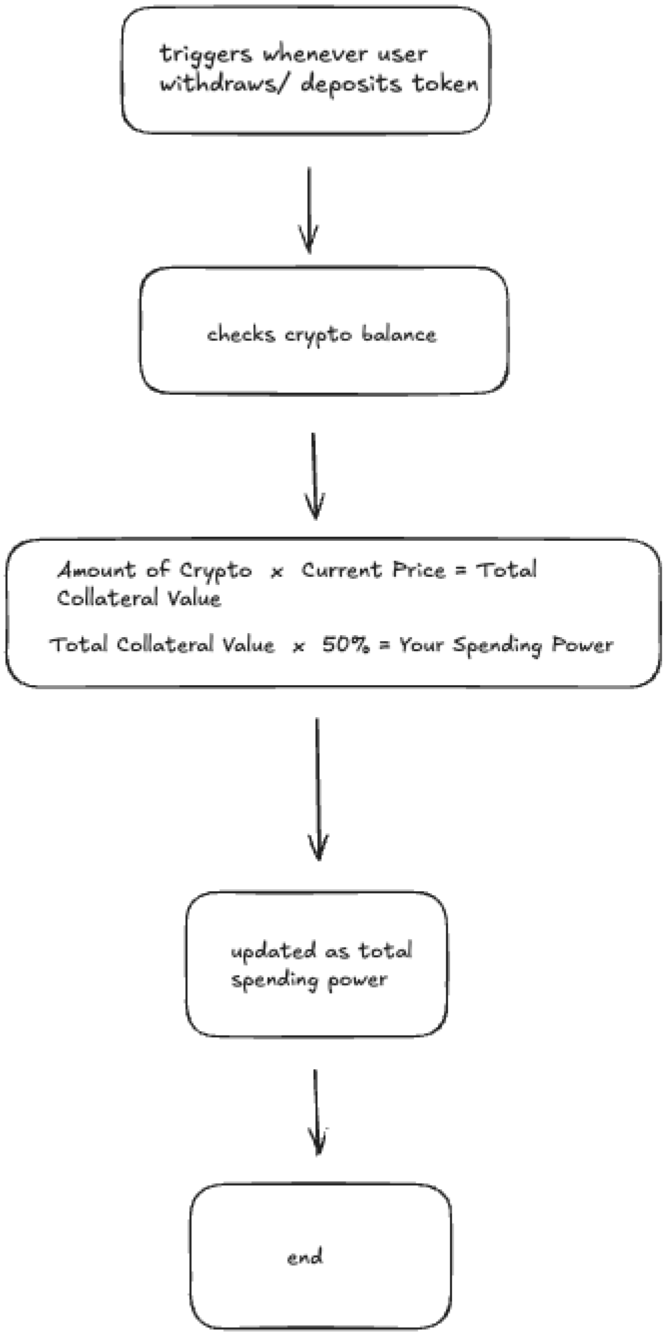
4. Detailed Flow chart



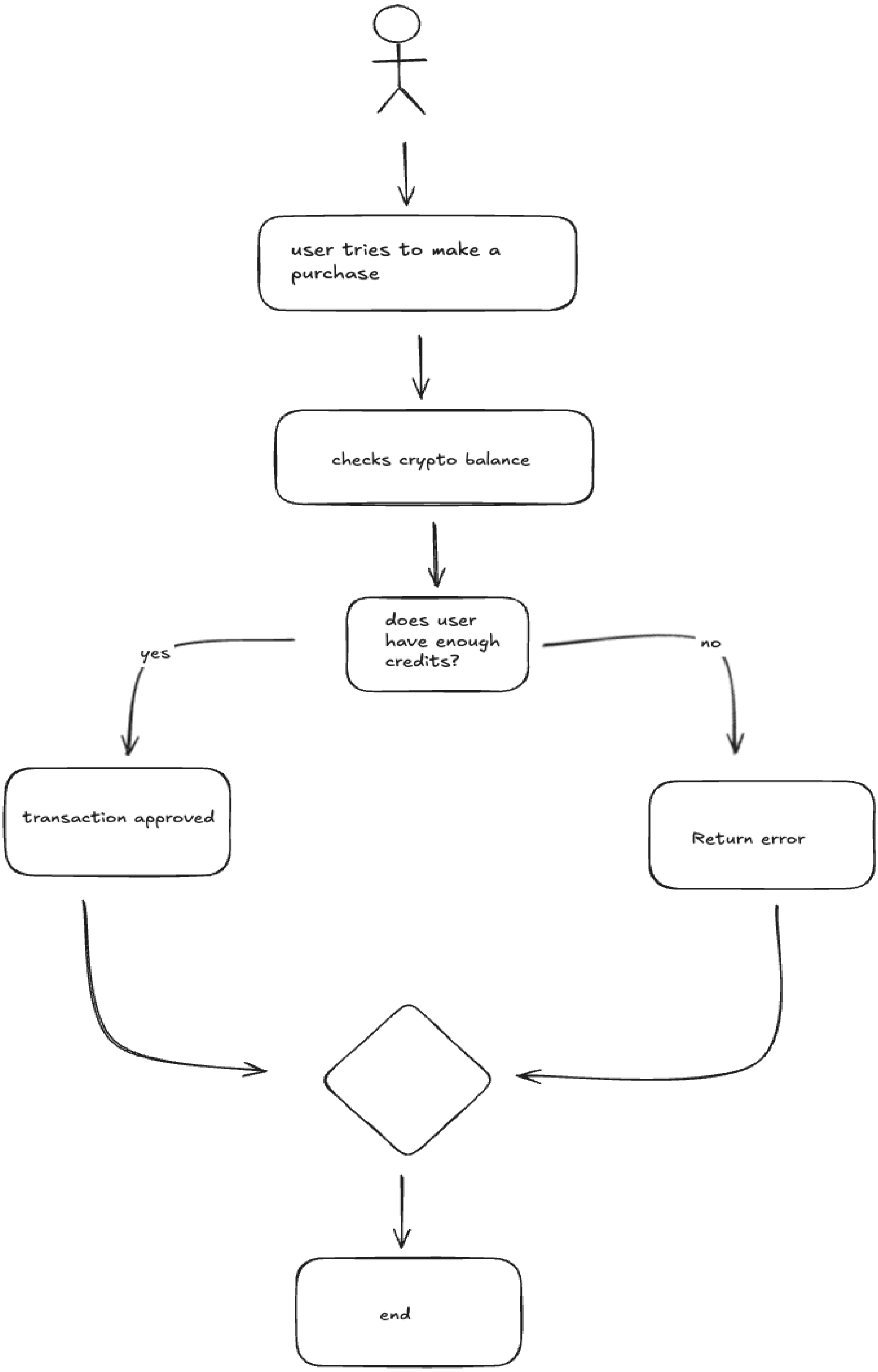
withdraw_collateral()



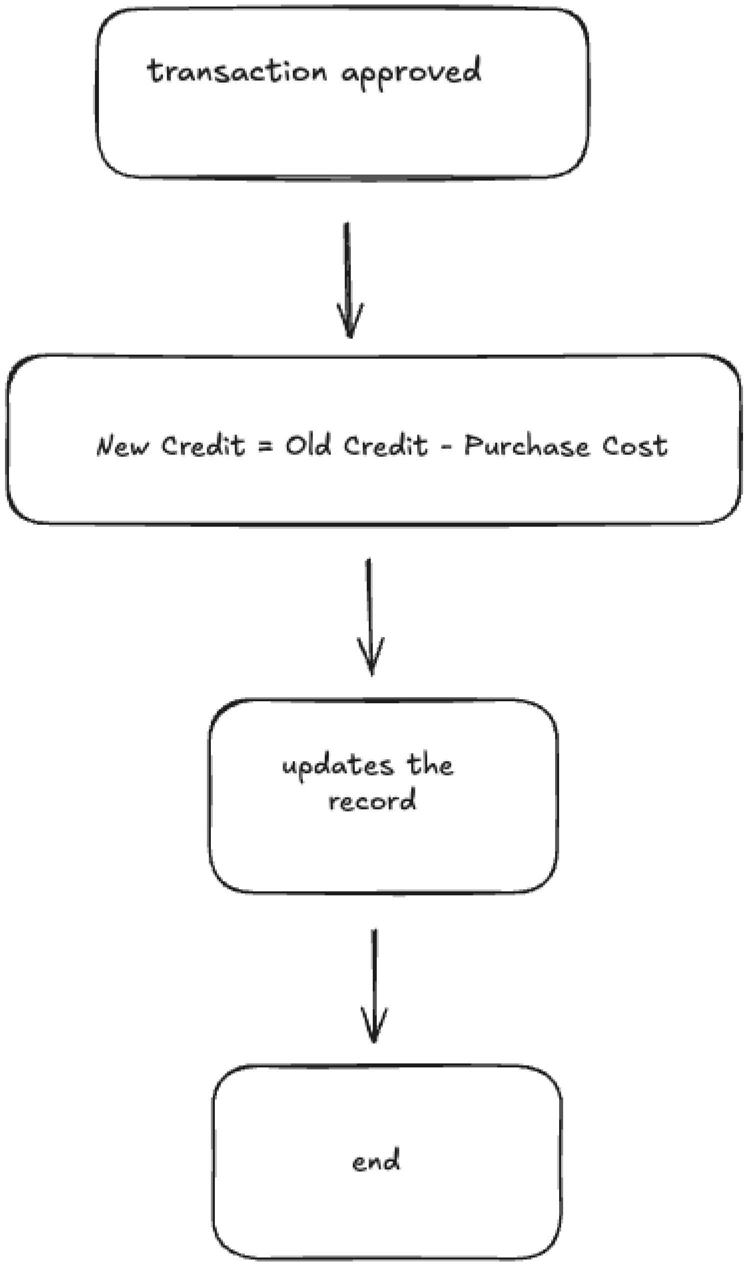
calculate_spending_power()



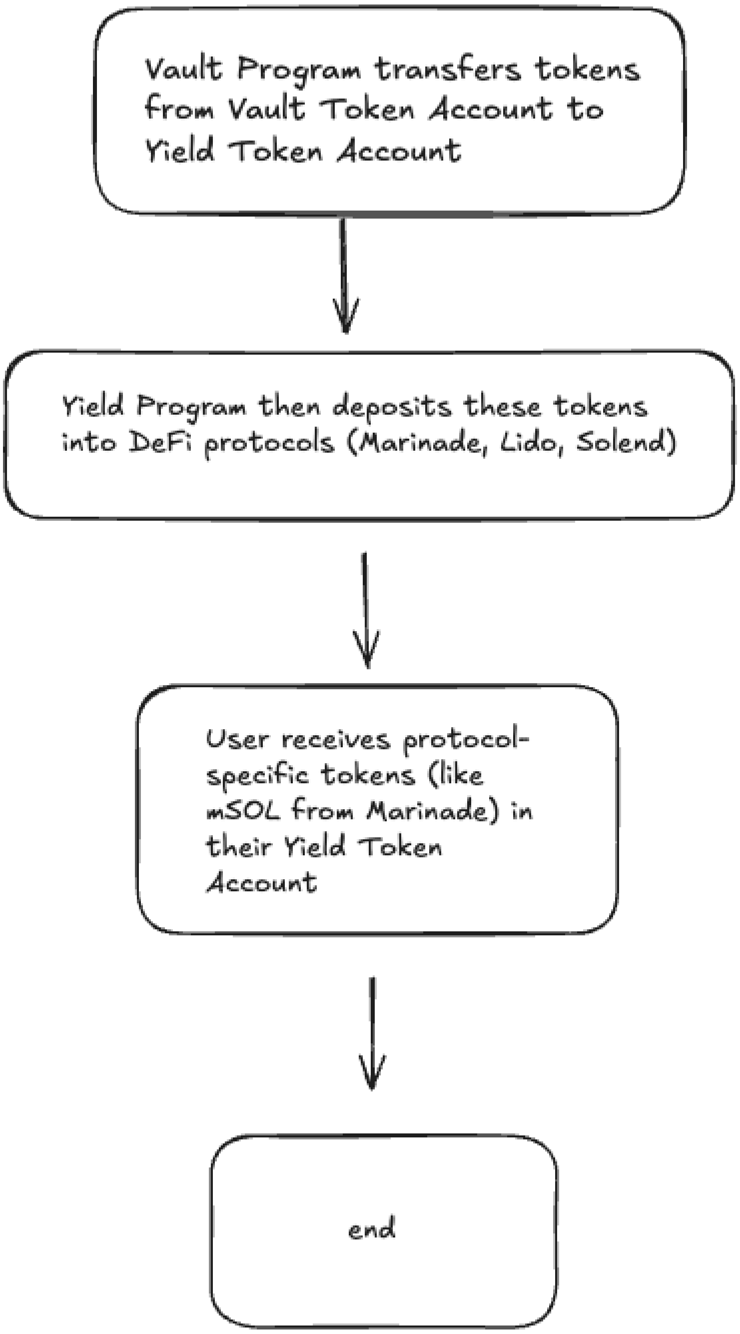
authorize_purchase()



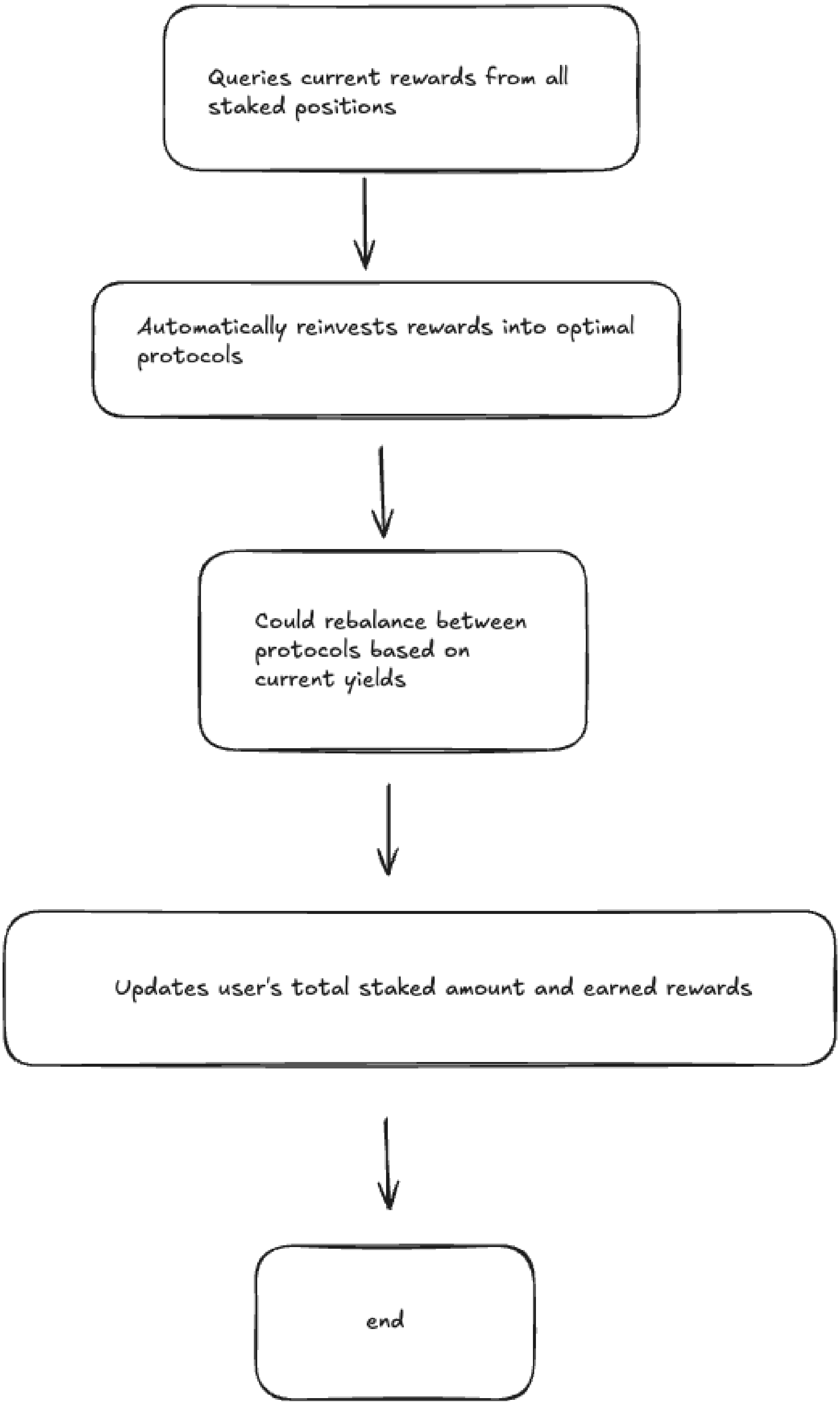
update_balance()



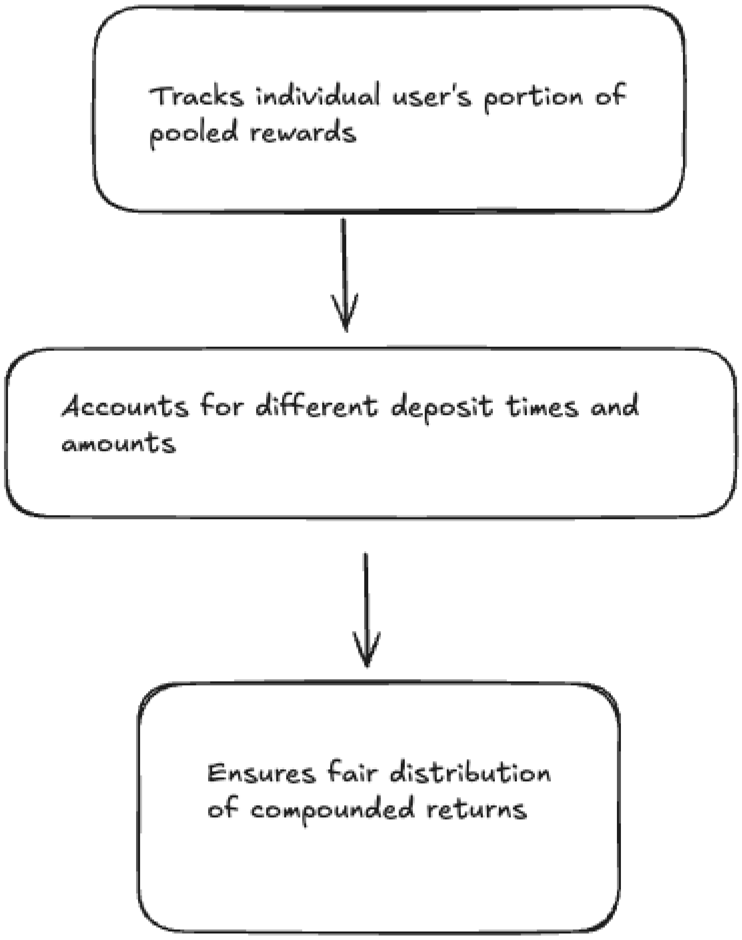
stake_collateral()



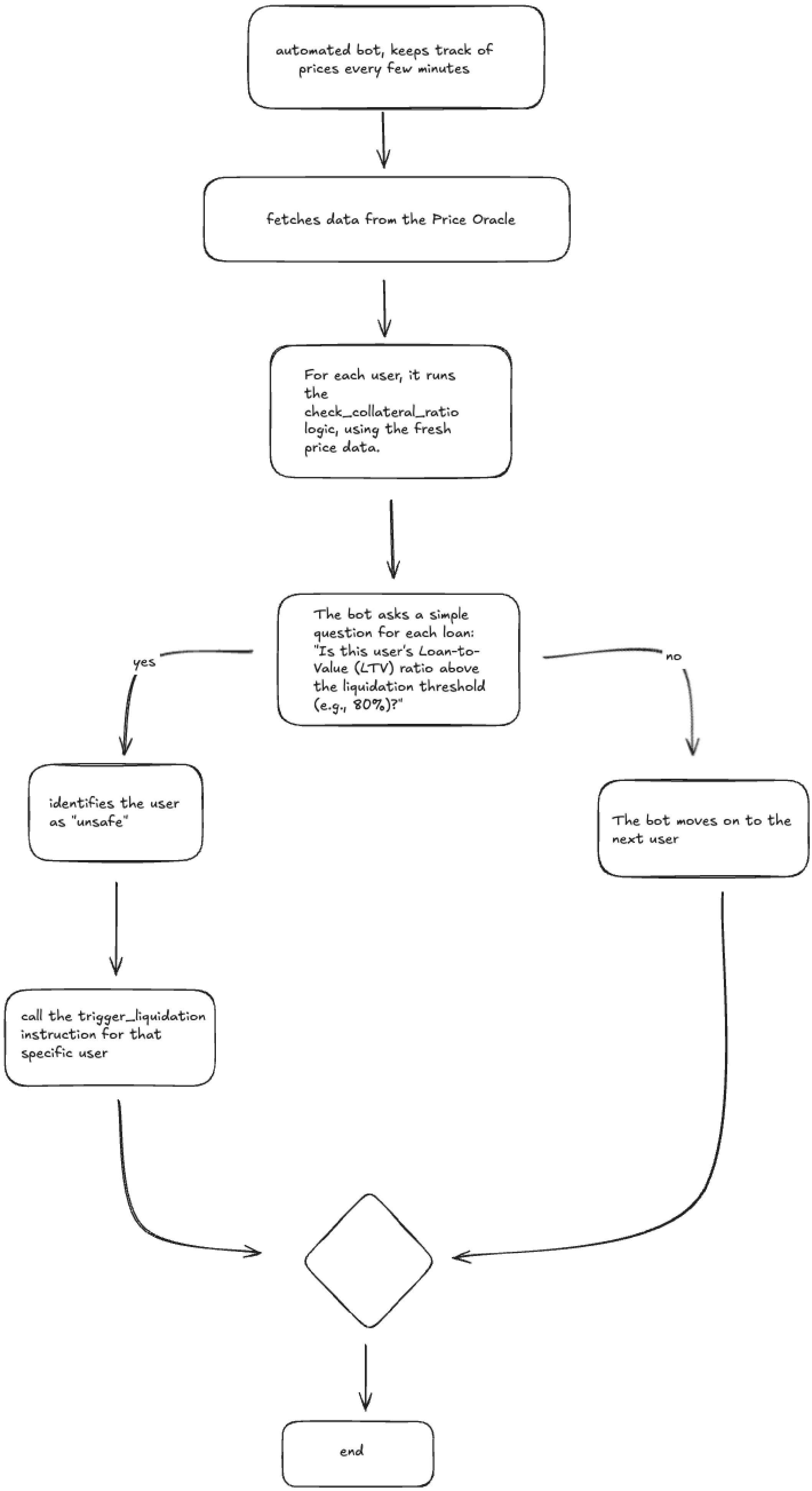
compound_earnings()



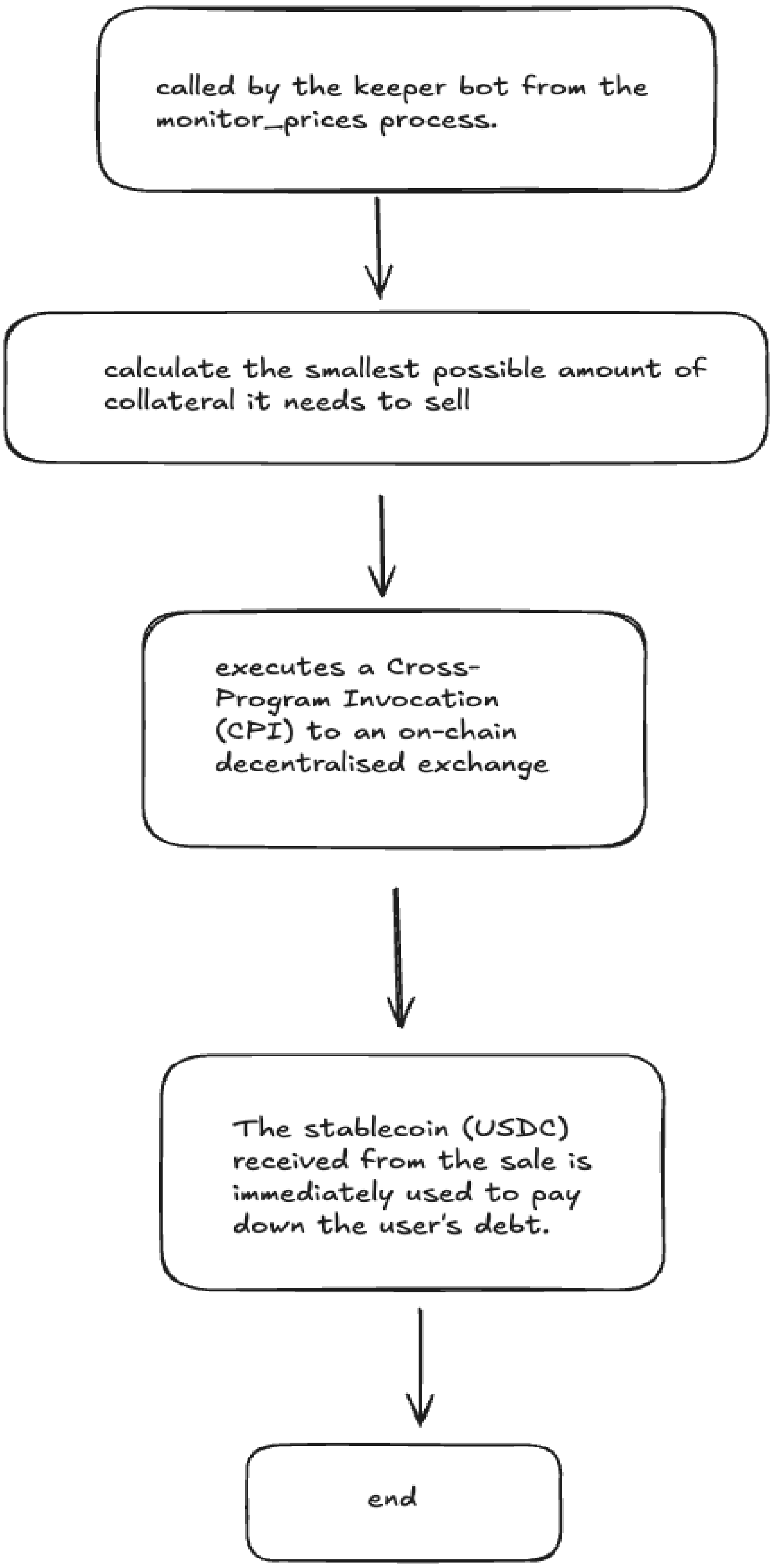
calculate_user_share()



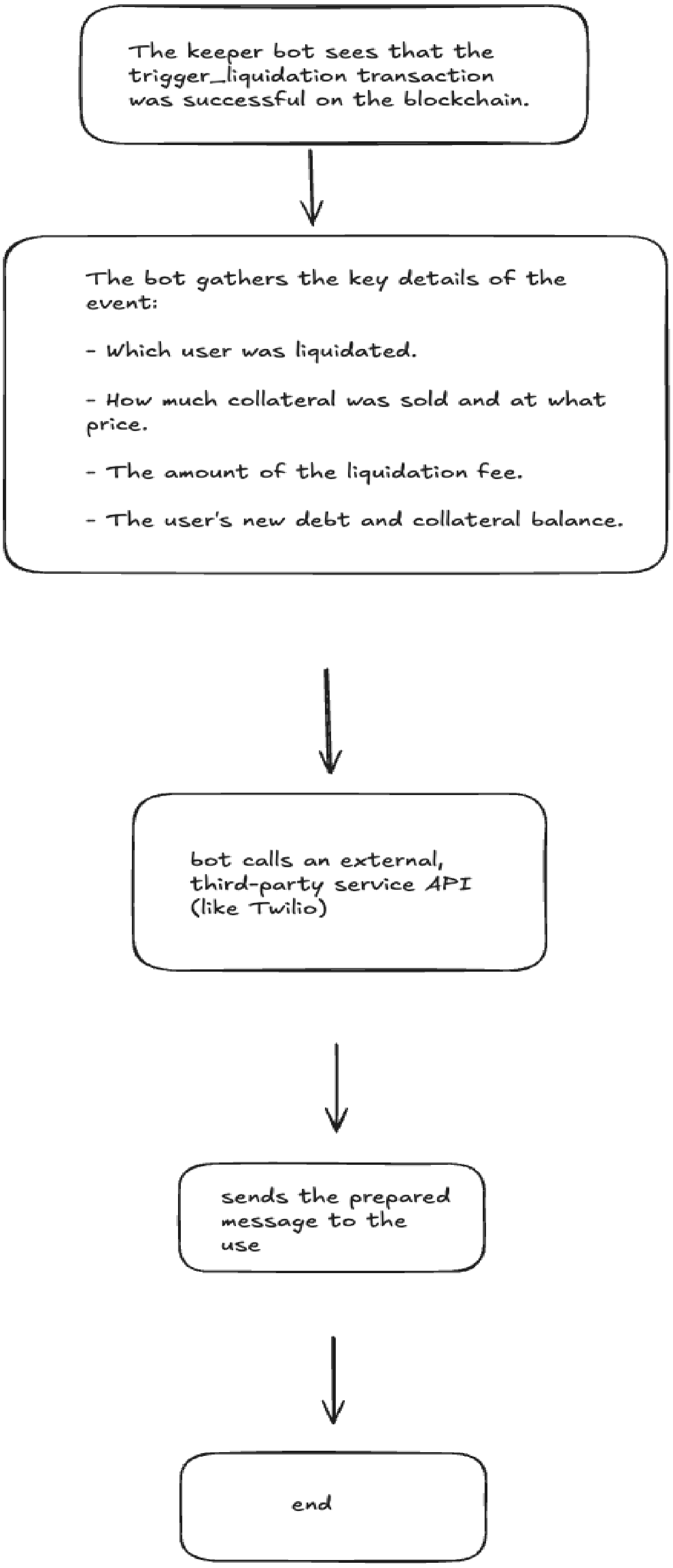
monitor_prices()



trigger_liquidation



notify_user()


The keeper bot sees that the trigger_liquidation transaction was successful on the blockchain.
The bot gathers the key details of the event:
- Which user was liquidated.
- How much collateral was sold and at what price.
- The amount of the liquidation fee.
- The user's new debt and collateral balance.
bot calls an external, third-party service API (like Twilio)
sends the prepared message to the use
end