cleevio

# DERIVERSE

## Analysis

# Description

Deriverse is a service that works with NFT derivatives. That is, a user can long or short an NFT collection. Also there is the possibility of providing liquidity to liquidity pools, which are created using Uniswap v3.

**BE**

Deriverse will be a fork of Perp v2 protocol, which has GPL-3.0-or-later license. This license means that everyone can use the code, duplicate it, change it, but code must be open source.

**FE**

Application in Next.js, communication via web3.js library with our smart contracts.

# Networks

## Perp v2 protocol runs on Optimism network.

Optimism is the EVM equivalent. For development that means we can use EVM tools and develop contracts in Solidity. Optimism is a layer 2 optimistic rollup network and it is the second most popular optimism rollup solution on Ethereum with a 10% share of the entire Ethereum layer 2 market. Optimism doesn't have a native crypto coin. Instead of native coin Optimism uses wrapped ETH (wETH) as its base currency. Here is a link to wETH smart contract.

Second option is Arbitrum. The main differences between Arbitrum and Optimism are that Arbitrum uses multiple-round fraud proofs and Optimism uses single-round fraud proofs, Optimism is EVM compatible only, Arbitrum is as well, but has its own Arbitrum Virtual Machine (AVM). And Optimism supports only Solidity, Arbitrum supports all EVM programming languages. Arbitrum as settlement currency uses ETH, not wETH as Optimism. Arbitrum has a significantly bigger ecosystem with total value locked (TVL) $1.5 billion and more active dApss than Optimism. Optimism has TVL $300 million.

# Arbitrum vs Optimism in detail

Optimism and Arbitrum are two of the largest layer-two (L2) solutions that utilize Optimistic rollup technology to scale the Ethereum network. Optimistic rollups work by executing transactions on a layer-2 rollup chain while a node, called a sequencer, rolls up and subsequently posts transaction state data to layer-1. This method of processing transactions has the advantage of compressing the data posted to the Ethereum mainnet while also amortizing gas fees among transactions in each rollup batch. Optimistic Rollups rely on fraud proofs to dispute fraudulent transactions, should such cases occur. So, if someone says that the data is invalid, then the computations will be checked. They can be verified through cryptography and, if it turns out that fraud has taken place, the fraudulent transactions will be rolled back while those who committed the fraud get ejected.

**Optimistic rollups have a longer fund withdrawal period due to their security model (up to 14 days).**

Currently, on both Optimism and Arbitrum, centralized sequencers must be trusted to post valid transaction data to layer-1. This can be a security risk as it must be assumed there is at least one honest party between a sequencer and validator.

# Arbitrum vs Optimism in detail

## Tokens bridge

Both platforms apply bridges to interact with other blockchains and ensure the flow of tokens. However, Arbitrum uses a permissionless bridge for all tokens, whereas Optimism deploys dedicated bridges based on the market demands.

## Base differences

Optimism uses single-round fraud proofs executed on layer-1. When dealing with suspicious transactions, Optimism sends the entire transaction again through the EVM, so the fraud proof verification is instant. At the same time, the result is higher cost, since on-chain Layer 1 execution requires more gas. Moreover, the Layer 2 fee is limited by the Layer 1 gas block. Optimism doesn't have a native crypto coin. Instead of native coin Optimism uses wrapped ETH (wETH) as its base currency.

Arbitrum's multi-round fraud proofing is the more advanced of the two, with it being cheaper and more efficient than single-round proofing.
ArbOS is the operating system that runs on top of the AVM and is responsible for ensuring the execution of contracts on the Arbitrum chain. ArbOS exists and runs completely on L2 and manages the execution of Smart Contracts on the EVM (Ethereum Virtual Machine) just as they would be executed on Ethereum.

# Arbitrum vs Optimism in detail

## Decentralization

Optimism is governed by **Optimism Collective DAO**.
**Arbitrum on the other hand, is not DAO governed, with the protocol run solely by Offchain Labs.**

## Tooling

Optimism supports all EVM compatible developers tools.
Arbitrum network has support for most popular Ethereum development tools and suites (Web3.js, Truffle, Hardhat, Infura, Moralis).

## Fees

Optimism has the capacity to process up to 2,000 transactions per second. According to L2 Fees data, Optimism transaction fees are slightly higher compared to Arbitrum and range from $0,6 to $0,9.

Arbitrum allows for 40,000 transactions per second, with gas fees ranging between $0,5-0,7 according to Layer 2 data aggregator L2 Fees.

# Arbitrum vs Optimism in detail

## Optimism +

- EVM equivalent
- Simplicity
- More decentralized
- Perpv2 protocol is well tested

## Arbitrum +

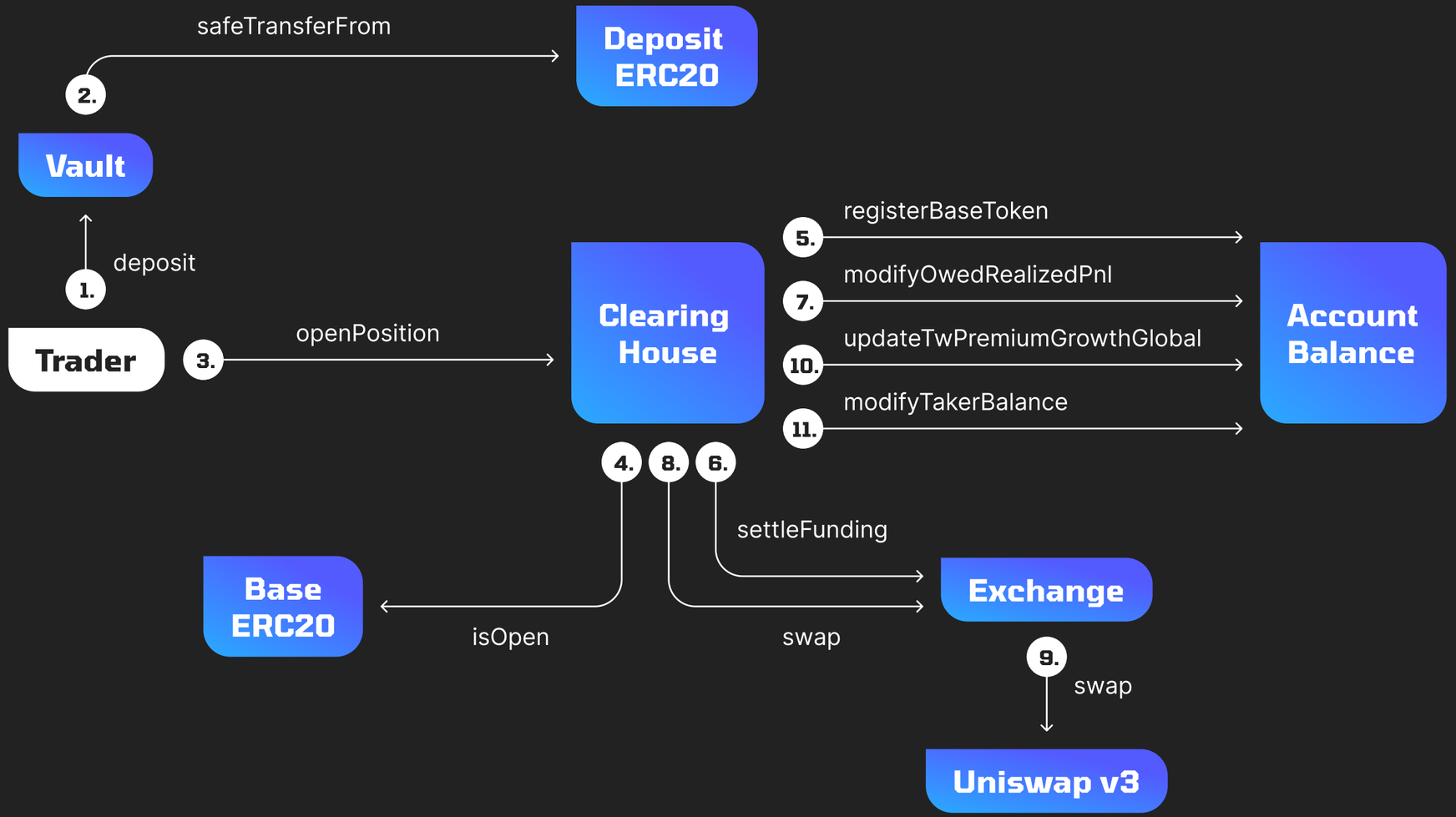
- Higher security due to multi-round fraud proofing
- Nitro runs Geth at layer 2 on top of Ethereum, and can prove fraud over the core engine of Geth compiled to WASM. **(Currently on devnet)**
- Bigger ecosystem
- Bigger user base
- Layer 1 block gas limit doesn't matter as Layer 2 transactions are never entirely executed on Layer 1

# Functionality

## Deposit on open position (user wants to go long on BAYC)

User deposit wETH. ClearingHouse mints vETH using max leverage. This doesn’t mean the user has to use the maximum leverage when opening a position, it only gives the users the opportunity to do so by issuing the maximum number of tokens that the user may or may not use. For example, if a user deposits 1,000 ETH, the protocol would issue 10,000 vETH. User decides to go long. ClearingHouse trades his vETH for vBAYC token. Protocol uses Uniswap v3, there uses the correct pair pool - in our case vETH/vBAYC.

# Functionality

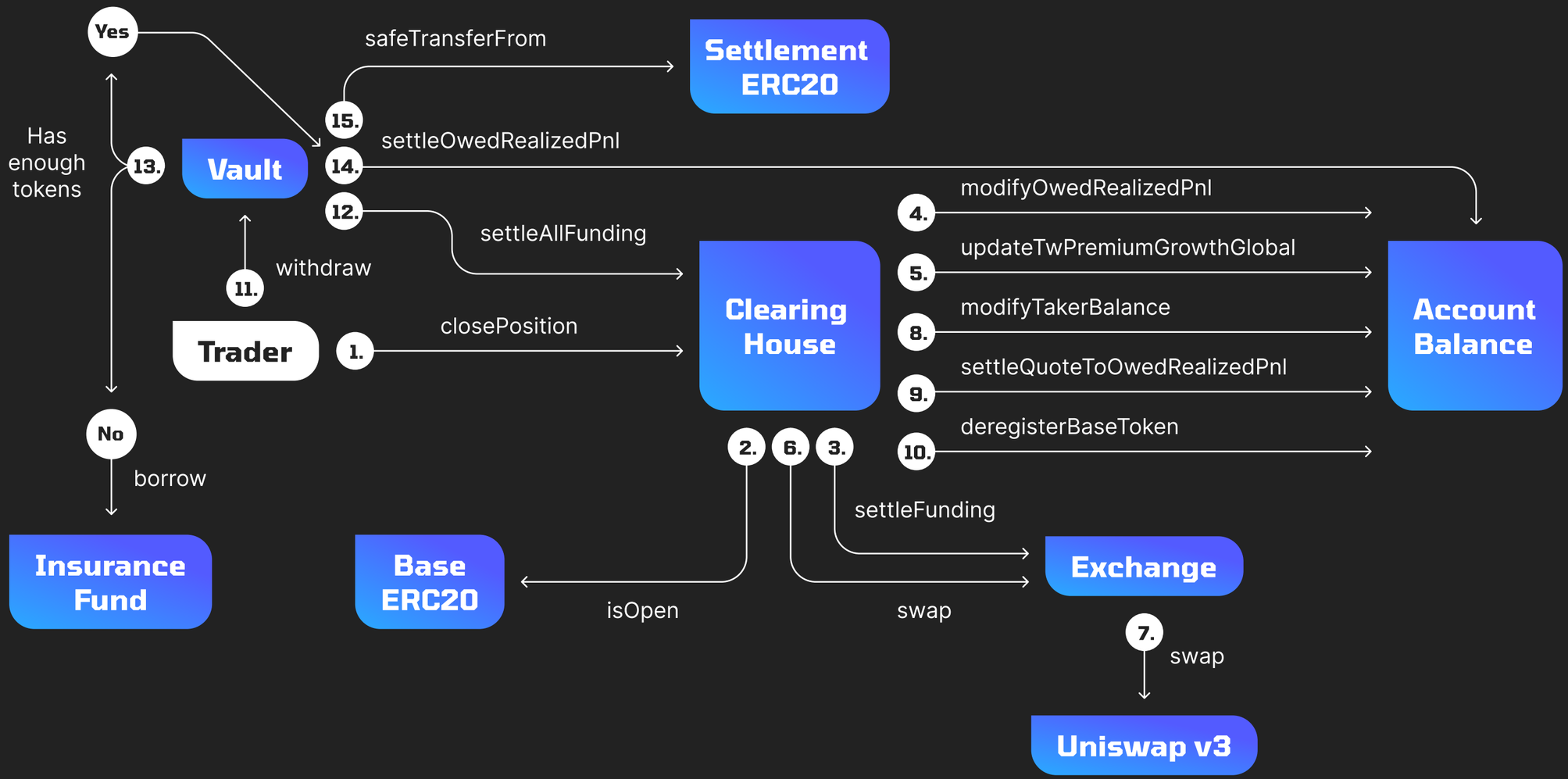


# Functionality

## Withdraw and close position (user wants to close his position on BAYC)

Once the user decides to close his position, he can trade his vBAYC back to vETH. This gives the user a profit or a loss, depending on what has happened to the BAYC price since the user opened the position.

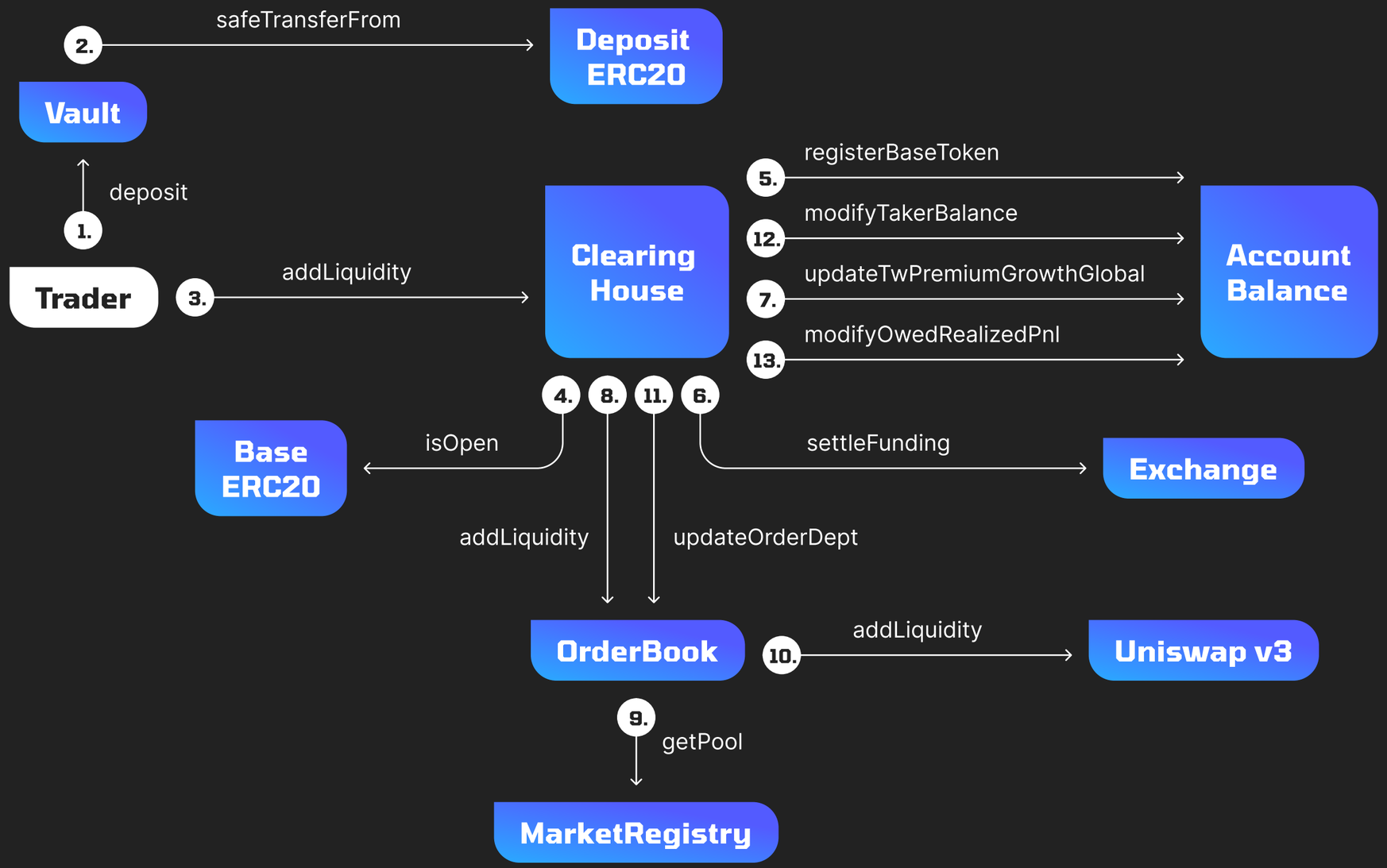
# Functionality



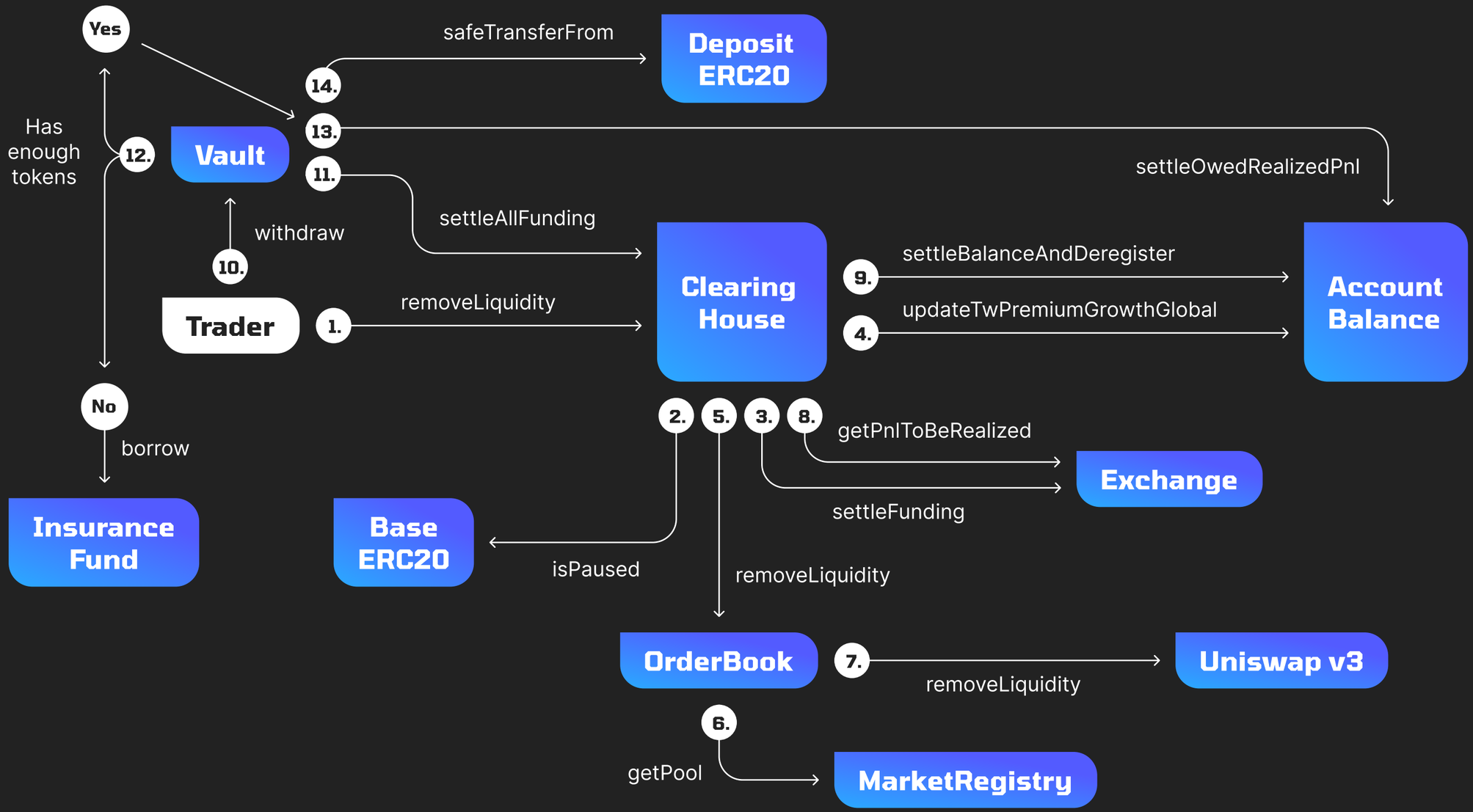
# Makers

Makers provide liquidity. They earn fees for each trade in the pool where they provide liquidity. When makers decide to provide liquidity, they make a wETH deposit into the ClearingHouse smart contract. It mines vETH and adds liquidity to the Uniswap v3 pool. Example - if a user wants to provide liquidity to vETH-vBAYC, the protocol creates liquidity in the pool with the correct disposition. I.e. If BAYC costs 50 wETH and someone provides 100 wETH, there will be 50 vETH and 1 vBAYC in the pool.

# Add Liquidity



# Remove liquidity



# Margin model

The protocol uses cross margin. All the client's funds are in one pool and all his positions use the pool as collateral. The advantage is that he doesn't have to add margin on different positions, the disadvantage is that he just needs one of the positions to go down a lot and thus for the others he can go into liquidation because he won't have a big enough collateral.

## B0Q1

B0Q1 stands for "BaseToken is Token0, and QuoteToken is Token1". In UniswapV3, a pool has two tokens, one is designated as `token0`, and the other one `token1`. The order is decided by `(address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA)`
The price of the pool is always token1/token0. The price range liquidity is added to is also based on this configuration.

Assuming we want to have pools such as vETH:vBAYC or vCRYPTOPUNK:vETH, the price a UniswapV3Pool returns can be denominated in ETH, BAYC or CRYPTOPUNK, depending on which is token0 and token1, which creates extra efforts.

To make it simpler, we force the ClearingHouse to only support UniswapV3Pool whose token0 must be base, and token1 must be quote. This can be done by only deploying a BaseToken when its address is smaller than our QuoteToken (not using create2).

Summary - token0 is baseToken (vBAYC) and token1 is quoteToken (vETH).

# Description of Perp v2 parts

Perp v2 protocol consists of 3 parts.

**Curie Periphery** | **Curio Core** | **Oracle**

## MetaTxGateway.sol

Used as a gateway for meta transactions for whitelisted contracts. A meta transaction is a regular Ethereum transaction which contains another transaction. The actual transaction is signed by a user and then sent to an operator, there is no blockchain interaction required from the user side. The operator takes this signed transaction and submits it to the blockchain paying for the gas fees himself. Owner can add a contract address to the whitelist. Only contracts in the whitelist can be executed by this gateway. This prevents the gateway from being abused to execute arbitrary meta txs. See: EIP712 standard

## lens/PerpPortal.sol

Contains view functions for other curie contracts

**Depending on the (initialization):**

ClearingHouse
ClearingHouseConfig
Exchange
OrderBook
MarketRegistry
AccountBalance
Vault
InsuranceFund

# Curie contracts

## AccountBalance.sol

Where most balances of a trader are recorded, such as margin ratio, position size, position value, etc. This contract is pretty much filled with view functions, so the comments (or Natspec, if there is) in the contract code should be sufficient.

**Depending on the (initialization):**
ClearingHouseConfig
OrderBook

## Vault.sol

Store all collaterals. Perp v2 supports only USDC. Contains functions for deposit, withdrawal or liquidate collateral. WETH instead of ETH.

**Depending on the (initialization):**
InsuranceFund
ClearingHouseConfig
AccountBalance
Exchange

## MarketRegistry.sol

Is responsible for adding pools to uniswap. Contract can set up an exchange and uniswap fee ratio, insurance fund fee ratio and maximum orders for a specified base token.

**Depending on the (initialization):**
UniswapV3 factory
QuoteToken (virtual ETH)

## CollateralManager.sol

Store settings:

- Maximum collateral tokens per account
- Liquidation ratio
- Max margin
- Insurance fund fee ratio

**Depending on the (initialization):**
ClearingHouseConfig
Vault

# Curie contracts

## Exchange.sol

Contract is responsible for swapping virtual tokens on Uniswap and settling funding.

**Depending on the (initialization):**

MarketRegistry
OrderBook
ClearingHouseConfig

**Clearinghouse.sol**

Is responsible for minting and burning virtual tokens called v-tokens that are held by the contract on behalf of the user.

**Depending on the (initialization):**

ClearingHouseConfig
Vault
QuoteToken
UniswapV3 factory
Exchange
AccountBalance
InsuranceFund

# Oracle contracts

Perp v2 using as a price oracle Chainlink / Band protocol. Price feed contracts are responsible for fetching price from external sources, saving price in cache and calculating TWAP (time weighted average price) for selected intervals. Contracts are deployed for each market pair, implementing interface IPriceFeedV2.sol and base contract CachedTwap.sol. External data source is passed to the price feed contract in the constructor.

## Main functions

**PriceFeedV2.sol**
*getPrice(uint256 interval)*: Fetch actual price from external source. When interval is provided, TWAP value is calculated from cache data and returned.

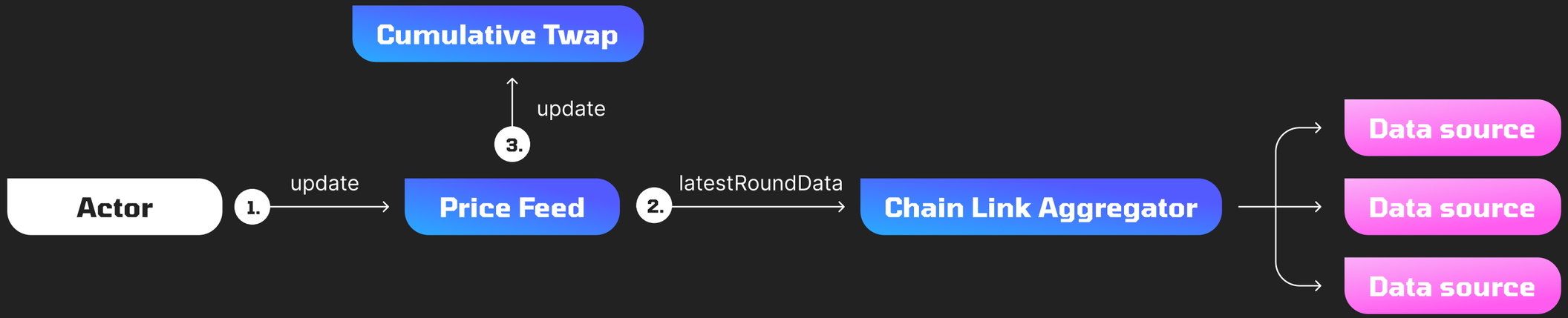
*cacheTwap(uint256 interval)*: Fetch actual price from external source. When interval is provided, TWAP value is calculated and saved to cache.

**Added to {PriceFeed}.sol**
*update():* Fetch actual price from external source and add to cache.

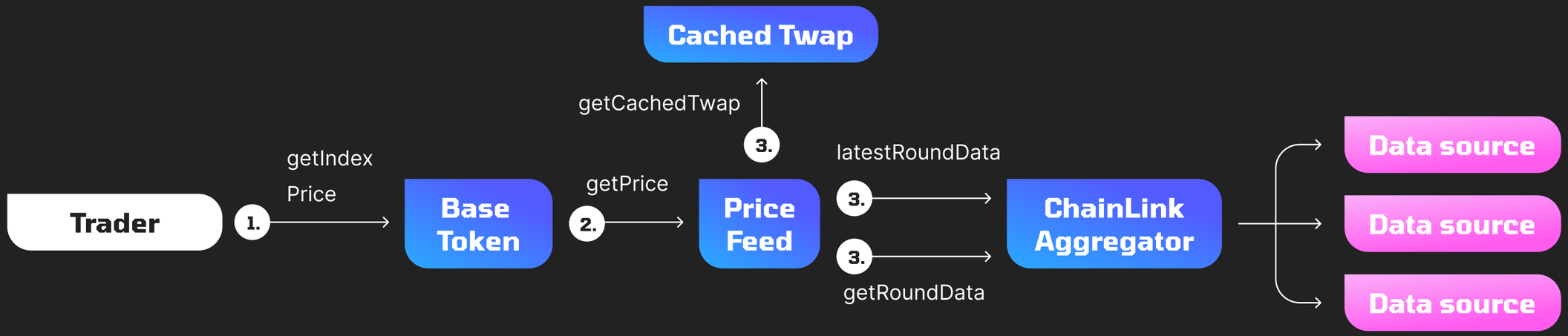
**CumulativeTwap.sol**
*_update(uint256 price, uint256 lastUpdatedTimestamp)*: Push latest observed price to Observation[256] state

*_calculateTwapPrice(*
*uint256 interval,*
*uint256 latestPrice,*
*uint256 latestUpdatedTimestamp*
*)*: Calculates TWAP by providing parameters and surrounding data from Observation[256] state.

# Update price




# Get price



# Tokens overview

**quoteToken**
We need to set quoteToken to vETH, which is the token we need to implement and deploy to the network. Then address to the token must be defined in initialization of ClearingHouse.sol smart contract as param “quoteTokenArg”.

**baseToken**
BaseToken is our virtual token which represents the NFT collections. For every collection we will support we want to create our own token and deploy it. Within BaseToken.sol contract initialization we set everything we need. As priceFeed param we set a link to the contract (oracle), which will give us a price.

**settlementToken**
Within the implementation there is a modifier on the deposit that checks if the deposited token is the same as the settlementToken. SettlementToken must be initialized in InsuranceFund.sol smart contract as param “tokenArg”. We will use wETH (in case we will be on Optimism).

# Code description

## Deposit

Application calls function deposit on Vault.sol smart contract. Function is external, that means that can be called only outside of function. Function has three modifiers.

- whenNotPaused - it’s possible to pause functionality of the contract, this function can be called only when smart contract is not paused
- noReentrant - defense against reentrant attack
- onlySettlementOrCollateralToken - verify if user deposits settlementToken

```
function deposit (address token, uint256 amount)
    external
    override
    whenNotPaused
    nonReentrant
    onlySettlementOrCollateralToken(token)
{
    // input requirement checks:
    //   token: here
    //   amount: here

    address from = _msgSender();
    _deposit(from, from, token, amount);
}
```

# Code description

SettlementToken is set in InsuranceFund.sol in initialization as tokenArg

```
function initialize (address tokenArg) external initializer {
        // token address is not contract
        require(tokenArg.isContract(), "IF_TNC");

        __ReentrancyGuard_init();
        __OwnerPausable_init();

        _token = tokenArg;
    }
```

# Code description

```
    function initialize(
        address insuranceFundArg,
        address clearingHouseConfigArg,
        address accountBalanceArg,
        address exchangeArg
    ) external initializer {
        address settlementTokenArg =
IInsuranceFund(insuranceFundArg) .getToken();
        uint8 decimalsArg =
IERC20Metadata(settlementTokenArg) .decimals();

        // invalid settlementToken decimals
        require(decimalsArg <= 18, "V_ISTD");
        // ClearingHouseConfig address is not contract
        require(clearingHouseConfigArg.isContract(), "V_CHCNC");
        // accountBalance address is not contract
        require(accountBalanceArg.isContract(), "V_ABNC");
        // exchange address is not contract
        require(exchangeArg.isContract(), "V_ENC");

        __ReentrancyGuard_init();
        __OwnerPausable_init();

        // update states
        _decimals = decimalsArg;
        _settlementToken = settlementTokenArg;
        _insuranceFund = insuranceFundArg;
        _clearingHouseConfig = clearingHouseConfigArg;
        _accountBalance = accountBalanceArg;
        _exchange = exchangeArg;
    }
```

And then is set in Vault.sol in initialization, where we have to send an InsuranceFound.sol address. Vault smart contract then calling InsuranceFound via interface.

# Code description

In the deposit function we get the address of a sender and then we call _deposit function.

```
function _deposit(
    address from,
    address to,
    address token,
    uint256 amount
) internal {
    // V_ZA: Zero amount
    require(amount > 0, "V_ZA");
    _transferTokenIn(token, from, amount);
    _checkDepositCapAndRegister(token, to, amount);
}
```

Amount cannot be negative. Then being called the _transferTokenIn function. It’s an internal function.

```
function _transferTokenIn(
    address token,
    address from,
    uint256 amount
) internal {
    uint256 balanceBefore =
IERC20Metadata(token).balanceOf(address(this));

SafeERC20Upgradeable.safeTransferFrom(IERC20Up
gradeable(token), from, address(this), amount);

require((IERC20Metadata(token).balanceOf(address(t
his)).sub(balanceBefore)) == amount, "V_IBA");
}
```

# Code description

```
    function _checkDepositCapAndRegister(
        address token,
        address to,
        uint256 amount
    ) internal {
        if (token == _settlementToken) {
            uint256 settlementTokenBalanceCap =

IClearingHouseConfig(_clearingHouseConfig).getSettl
ementTokenBalanceCap();

require(IERC20Metadata(token).balanceOf(address(thi
s)) <= settlementTokenBalanceCap, "V_GTSTBC");
        } else {
            uint256 depositCap =
ICollateralManager(_collateralManager).getCollateralC
onfig(token).depositCap;

require(IERC20Metadata(token).balanceOf(address(thi
s)) <=
depositCap, "V_GTDC");
        }

        _modifyBalance(to, token, amount.toInt256());
        emit Deposited(token, to, amount);
    }
```

Basically on the first line of function, we get balanceBefore for later verification. On the second line we process transfer and on the third line we verify that balance is correct.
What is important here, that tokens are deposited into Vault smart contract. That means all tokens are owned and they are locked by Vault smart contract.

Another function here within _deposit function is _checkDepositCapAndRegister(token, to, amount).

# Code description

If the token is settlementToken (USDC), we get settlementTokenBalanceCap, which is the max value of settlement token balance. This number can be changed by the owner of the contract in ClearingHouseConfig.sol with the function setSettlementTokenBalanceCap. ClearingHouseConfig will be breakdowned in the next part of the analysis.

If the token is not settlementToken and it is different collateral, we will get max deposit value in CollateralManager from CollateralConfig. Every collateral must have its own Collateral Config, so every Collateral can have a different max deposit value.

One of the last functions is _modifyBalance. If a user deposited a settlementToken, we just update his balance in VaulStorage.sol.

```
// key: trader, token address
 mapping(address => mapping(address => int256))
internal _balance;
```

# Code description

In case of non-settlementToken, we store collateral in VaultStorage as well.

```
// trader => collateral token
// collateral token registry of each trader
mapping(address => address[]) internal
_collateralTokensMap;
```

As the last line is emitting an event. For this event we will listen to on the FE of an application. Once FE receives the event, it can approve a user deposit processed successfully.


```
// trader => collateral token
// collateral token registry of each trader
mapping(address => address[]) internal
_collateralTokensMap;
```

# Withdraw

When withdrawing collaterals, one can withdraw the amount up to your freeCollateral. This makes sure that positions are always sufficiently collateralized.

First of all we will check how much collateral a trader can withdraw with function getFreeCollateral in Vault.sol

```
function getFreeCollateral(address trader) public view
override returns (uint256) {
        return
            PerpMath
                .max(getFreeCollateralByRatio(trader,
IClearingHouseConfig(_clearingHouseConfig).getImR
atio()), 0)
                .toUint256();
    }
```

# Withdraw

As we can see, we are using ImRation for calculation, which is set in ClearingHouseConfig.
If a user has enough collateral, we can call the withdraw function on Vault.sol. Function is external, that means it can be called only outside of the smart contract.

```
function withdraw(address token, uint256 amount)
    external
    override
    whenNotPaused
    nonReentrant
    onlySettlementOrCollateralToken(token)
  {
    // input requirement checks:
    //   token: here
    //   amount: here

    address to = _msgSender();
    _settleAndDecreaseBalance(to, token, amount);

SafeERC20Upgradeable.safeTransfer(IERC20Upgrade
able(token), to, amount);
    emit Withdrawn(token, to, amount);
  }
```

As a token we have to send a collateral token address, same as in the deposit function. As the amount send the amount the user wants to withdraw, but be aware it cannot be more than function freeCollateral returned.
On function are the same modifiers as on deposit. For description of modifiers, please check their description in the Deposit chapter.

_msgSender function gets the address of an user, who makes the withdrawal.
As the first thing we have to call _settleAndDecreaseBalance function.

# Withdraw

*_settleAndDecreaseBalance function*

```
    IClearingHouse(_clearingHouse).settleAllFunding(to);

        uint256 freeCollateral =
getFreeCollateralByToken(to, token);
        require(freeCollateral >= amount, "V_NEFC");

        int256 deltaBalance = amount.toInt256().neg256();
        if (token == _settlementToken) {
            uint256 vaultBalanceX10_S =
IERC20Metadata(token).balanceOf(address(this));
            if (vaultBalanceX10_S < amount) {
                uint256 borrowedAmountX10_S = amount -
vaultBalanceX10_S;

IInsuranceFund(_insuranceFund).borrow(borrowedAm
ountX10_S);
                _totalDebt += borrowedAmountX10_S;
            }

            int256 owedRealizedPnlX10_18 =
IAccountBalance(_accountBalance).settleOwedRealiz
edPnl(to);
            deltaBalance =
deltaBalance.add(owedRealizedPnlX10_18.formatSettl
ementToken(_decimals));
        }

        _modifyBalance(to, token, deltaBalance);
```

As the first thing we have to settle all fundings. This function is inside ClearingHouse. As the first thing we get traders' base tokens, which are stored in AccountBalanceStorageV1.sol. We will get it from AccontBalance smart contract, function getBaseTokens(address trader)

```
    mapping(address => address[]) internal
_baseTokensMap;
```

# Withdraw

```
function _settleFunding(address trader, address
baseToken)
        internal
        returns (Funding.Growth memory
fundingGrowthGlobal)
    {
        int256 fundingPayment;
        (fundingPayment, fundingGrowthGlobal) =
IExchange(_exchange).settleFunding(trader,
baseToken);

        if (fundingPayment != 0) {
            _modifyOwedRealizedPnl(trader,
fundingPayment.neg256());
            emit FundingPaymentSettled(trader, baseToken,
fundingPayment);
        }


IAccountBalance(_accountBalance).updateTwPremiu
mGrowthGlobal(
            trader,
            baseToken,
            fundingGrowthGlobal.twPremiumX96
        );
        return fundingGrowthGlobal;
    }
```

Then we call _settleFunding with the trade's address and address of baseToken.

# Withdraw

User has derivatives and he wants to exchange them for USDC (in case of Perp v2).
On the second line is being called settleFunding in Exchange smart contract. This function can be called only from the ClearingHouse contract. Function in Exchange.sol verifies if base token really exists, then we call function _getFundingGrowthGlobalAndTwaps, this function calculates the up-to-date globalFundingGrowth and twaps and passes them out. After the funding update, a smart contract emits a FundingUpdate event, which we should be able to catch in FE of an application.

```
emit FundingUpdated(baseToken, markTwap,
indexTwap);
```

If funding payments are not zero, smart contract calls in _settleFunding function modifyOwedRealizedPnl function in AccountBalance contract. This function can be called only from the ClearingHouse smart contract. Function will add an amount to the _owedRealizedPnlMap in AccountBalanceStorageV1 smart contract and emit an event about it.

```
emit PnlRealized(trader, amount);
```

# Withdraw

As the last thing in _settleFunding we have to update _accountMarketMap as well.

```
    // first key: trader, second key: baseToken
    mapping(address => mapping(address =>
AccountMarket.Info)) internal _accountMarketMap;
```

Now we have settled all the funding and we will continue with the rest of the _settleAndDecreaseBalance function. As the first thing we will check if the user has really enough collateral.Next step is to check if there are enough tokens in Vault. In case there are not enough tokens, we have to borrow them from the InsuranceFound smart contract.

```
        uint256 vaultBalanceX10_S =
IERC20Metadata(token).balanceOf(address(this));
        if (vaultBalanceX10_S < amount) {
            uint256 borrowedAmountX10_S = amount -
vaultBalanceX10_S;

IInsuranceFund(_insuranceFund).borrow(borrowedAm
ountX10_S);
            _totalDebt += borrowedAmountX10_S;
        }
```

# Withdraw

Then we settle both the withdrawn amount and owedRealizedPnl to collateral.

```
        int256 owedRealizedPnlX10_18 =
IAccountBalance(_accountBalance).settleOwedRealiz
edPnl(to);
            deltaBalance =
deltaBalance.add(owedRealizedPnlX10_18.formatSettl
ementToken(_decimals));
```

Last step is modie balance with the _modifyBalance function in Vault.sol smart contract. Modify balance function is only for internal purposes, that means can be called only from inside of the smart contract itself or smart contracts which inherit the Vault.sol.

# ClearingHouseConfig.sol

ClearingHouseConfig is used by ClearingHouse, Vault and AccountBalance contracts.

What we set in ClearingHouseConfig (in brackets are default values):

- Initial-margin ratio (10%)
- Minimum-margin ratio (6.25%)
- Initial penalty ratio (2.5%)
- Partial close ratio (25%)
- Max funding rate (10%)
- Twap interval (15 minutes)
- Settlement token balance cap (0)
- Max markets per account (255)

All values are possible to change.

All changes always emit an event, which we can catch on FE of an application.

```
    event TwapIntervalChanged(uint256 twapInterval);
    event LiquidationPenaltyRatioChanged(uint24
liquidationPenaltyRatio);
    event PartialCloseRatioChanged(uint24
partialCloseRatio);
    event MaxMarketsPerAccountChanged(uint8
maxMarketsPerAccount);
    event SettlementTokenBalanceCapChanged(uint256
cap);
    event MaxFundingRateChanged(uint24 rate);
    event BackstopLiquidityProviderChanged(address
indexed account, bool indexed isProvider);
```

list of events

# Open Position

If we want to open or adjust (increase or reduce) a position, we have to call the openPosition function in ClearingHouse smart contract.

As a parameter of a function is struct OpenPositionParams.

```
struct OpenPositionParams {
    address baseToken;
    bool isBaseToQuote;
    bool isExactInput;
    uint256 amount;
    uint256 oppositeAmountBound;
    uint256 deadline;
    uint160 sqrtPriceLimitX96;
    bytes32 referralCode;
}
```

# Open Position

**Parameters**

- baseToken: the address of the base token; specifies which market you want to trade in
- isBaseToQuote: true for shorting the base token asset, false for longing the base token asset
- isExactInput: for specifying exactInput or exactOutput ; similar to UniSwap V2's specs
- amount: the amount specified. Depending on the isExactInput parameter, this can be either the input amount or output amount.
- oppositeAmountBound: the restriction on how many token to receive/pay, depending on isBaseToQuote & isExactInput
  - isBaseToQuote && isExactInput: want more output quote as possible, so we set a lower bound of output quote
  - isBaseToQuote && !isExactInput: want less input base as possible, so we set a upper bound of input base
  - !isBaseToQuote && isExactInput: want more output base as possible, so we set a lower bound of output base
  - !isBaseToQuote && !isExactInput: want less input quote as possible, so we set a upper bound of input quote
- deadline: the restriction on when this tx should be executed; otherwise, it fails
- sqrtPriceLimitX96: the restriction on the ending price after the swap. 0 for no limit. This is the same as sqrtPriceLimitX96 in the UniSwap V3 contract.
- referralCode: the referral code for partners

**Returns**

deltaBase: the amount of base token exchanged
deltaQuote: the amount of quote token exchanged

# Open Position

Add a liquidity function as the first thing checks, if the market is open. It is checked within ClearingHouse, where the contract calls the interface of BaseToken.

```
    function _checkMarketOpen(address baseToken)
internal view {
        // CH_BC: Market not opened
        require(IBaseToken(baseToken).isOpen(),
"CH_MNO");
    }
```

In case this is the user's first time, we will store his address with a base token.

```
    // trader => baseTokens
    // base token registry of each trader
    mapping(address => address[]) internal
_baseTokensMap;
```

Then the contract settles funding and opens a position. An input to the openPosition function has to be InternalOpenPositionParam.

```
struct InternalOpenPositionParams {
    address trader;
    address baseToken;
    bool isBaseToQuote;
    bool isExactInput;
    bool isClose;
    uint256 amount;
    uint160 sqrtPriceLimitX96;
    bool isLiquidation;
}
```

Trader is found by msgSender function.
BaseToken, isBaseToQuote, isExactInput, amount and sqrtPriceLimitX96 are copied from OpenPositionParams.
Inside the openPosition function swap has been processed.
This functionality is implemented in an Exchange contract. If the user opens position, he swaps settlement token/collateral for base token.
The last function in OpenPosition is check slippage.

# Close Position

**Parameters**

- baseToken: the address of the base token; specifies which market you want to trade in
- sqrtPriceLimitX96: the restriction on the ending price after the swap. 0 for no limit. This is the same as sqrtPriceLimitX96 in the UniSwap V3 contract.
  - `oppositeAmountBound`: the restriction on how many token to receive/pay, depending on `isBaseToQuote` & `isExactInput`
  - `isBaseToQuote` && `isExactInput`: want more output quote as possible, so we set a lower bound of output quote
  - `isBaseToQuote` && `!isExactInput`: want less input base as possible, so we set a upper bound of input base
  - `!isBaseToQuote` && `isExactInput`: want more output base as possible, so we set a lower bound of output base
  - `!isBaseToQuote` && `!isExactInput`: want less input quote as possible, so we set a upper bound of input quote
- deadline: the restriction on when this tx should be executed; otherwise, it fails
- referralCode: the referral code for partners

```
struct ClosePositionParams {
    address baseToken;
    uint160 sqrtPriceLimitX96;
    uint256 oppositeAmountBound;
    uint256 deadline;
    bytes32 referralCode;
}
```

There is a function with the name closePosition in ClearingHouse smart contract. Input is very similar to input for the openPosition function.

# Close Position

The first thing in the function is to check if the market is open. Then we call the swap function in the Exchange contract. The last thing is called a slippage contract. It is very similar to an open position, there is really nothing new.

```
_checkMarketOpen(params.baseToken);

    address trader = _msgSender();

    // must settle funding first
    _settleFunding(trader, params.baseToken);

    IExchange.SwapResponse memory response =
      _closePosition(
        InternalClosePositionParams({
          trader: trader,
          baseToken: params.baseToken,
          sqrtPriceLimitX96:
params.sqrtPriceLimitX96,
          isLiquidation: false
        })
      );

    // if exchangedPositionSize < 0, closing it is short,
B2Q; else, closing it is long, Q2B
    bool isBaseToQuote =
response.exchangedPositionSize < 0 ? true : false;
    uint256 oppositeAmountBound =
_getPartialOppositeAmount(params.oppositeAmount
Bound, response.isPartialClose);
```

```
    _checkSlippage(
      InternalCheckSlippageParams({
        isBaseToQuote: isBaseToQuote,
        isExactInput: isBaseToQuote,
        base: response.base,
        quote: response.quote,
        oppositeAmountBound:
oppositeAmountBound
      })
    );

    if (params.referralCode != 0) {
      emit
ReferredPositionChanged(params.referralCode);
    }
    return (response.base, response.quote);
```

# Add Liquidity

Makers can call addLiquidity to provide liquidity on the Uniswap V3 pool.
Tx will fail if adding base == 0 && quote == 0 / liquidity == 0

Function for adding liquidity can be found in ClearingHouse smart contract.
The input is AddLiquidityParams struct.

```
struct AddLiquidityParams {
    address baseToken;
    uint256 base;
    uint256 quote;
    int24 lowerTick;
    int24 upperTick;
    uint256 minBase;
    uint256 minQuote;
    bool useTakerBalance;
    uint256 deadline;
}
```

# Add Liquidity

**Parameters**

- `baseToken`: the base token address
- `base`: the amount of base token you want to provide
- `quote`: the amount of quote token you want to provide
- `lowerTick`: lower tick of liquidity range, same as UniSwap V3
- `upperTick`: upper tick of liquidity range, same as UniSwap V3
- `minBase`: the minimum amount of base token you'd like to provide
- `minQuote`: the minimum amount of quote token you'd like to provide
- `deadline`: a time after which the transaction can no longer be executed

**Returns**

- `base`: the amount of base token added to the pool
- `quote`: the amount of quote token added to the pool
- `fee`: the amount of fee collected, if there is any
- `liquidity`: the amount of liquidity added to the pool, derived from `base` & `quote`

# Add Liquidity

The first is to verify that the market is open.

```
struct AddLiquidityParams {
    address baseToken;
    uint256 base;
    uint256 quote;
    int24 lowerTick;
    int24 upperTick;
    uint256 minBase;
    uint256 minQuote;
    bool useTakerBalance;
    uint256 deadline;
}
```

```
    function registerBaseToken(address trader, address
baseToken) external override {
        _requireOnlyClearingHouse();
        address[] storage tokensStorage =
_baseTokensMap[trader];
        if (_hasBaseToken(tokensStorage, baseToken)) {
            return;
        }

        tokensStorage.push(baseToken);
        // AB_MNE: markets number exceeds
        require(tokensStorage.length <=
IClearingHouseConfig(_clearingHouseConfig).getMax
MarketsPerAccount(), "AB_MNE");
    }
```

As the owners we are able to close any market if we need. In that case users are not able to add liquidity to them.
If it is the first time for an user, we will register this base token to the maker. Function for registration of a token can be found in AccountBalance.registerBaseToken.

For adding liquidity itself is used function addLiquidity in OrderBook contract.
This function is first of all found in the correct pool in the MarketRegistry contract with getPool function. As input we set the base token.
Then we mint tokens we need in Uniswap V3.

# Remove Liquidity

Makers can call removeLiquidity to remove liquidity.

Remove liquidity will transfer maker impermanent position to taker position, if liquidity of RemoveLiquidityParams struct is zero, the action will collect fee from pool to maker.

The input is the RemoveLiquidityParams struct.

```
struct RemoveLiquidityParams {
    address baseToken;
    int24 lowerTick;
    int24 upperTick;
    uint128 liquidity;
    uint256 minBase;
    uint256 minQuote;
    uint256 deadline;
}
```

# Remove Liquidity

**Parameters**

- `baseToken`: the address of base token
- `lowerTick`: lower tick of liquidity range, same as UniSwap V3
- `upperTick`: upper tick of liquidity range, same as UniSwap V3
- `liquidity`: how much liquidity you want to remove, same as UniSwap V3
- `minBase`: the minimum amount of base token you want to remove
- `minQuote`: the minimum amount of quote token you want to remove
- `deadline`: a time after which the transaction can no longer be executed

**Returns**

- `base`: the amount of base token added to the pool
- `quote`: the amount of quote token added to the pool
- `fee`: the amount of fee collected, if there is any
- `liquidity`: the amount of liquidity added to the pool, derived from `base` & `quote`

Functionality is similar to Add Liquidity. First of all we check if the market is open. Then we call the remove liquidity function in OrderBook smart contract.

The function verifies via MarketRegistry if the pool with the token exists.Then needed tokens are burned via Uniswapv2

# Implementation

## Differences between Perp v2 and Deriverse

Perp v2 has as settlement token USDC, Deriverse should have ETH

### Solution 1

Perp v2 now supports settlement token USDC and collaterals WETH, ETH and FRAX. That means we basically do not need to change anything. If user deposit WETH, protocol mints 10x loan of vUSD backed by user's collateral. What is important is that in that case the user will have minted vUSD, not vETH.

### Solution 2

In this solution, we insist that the settlement token must be WETH. Settlement token is set within InsuranceFund initialized as a tokenArg parameter.

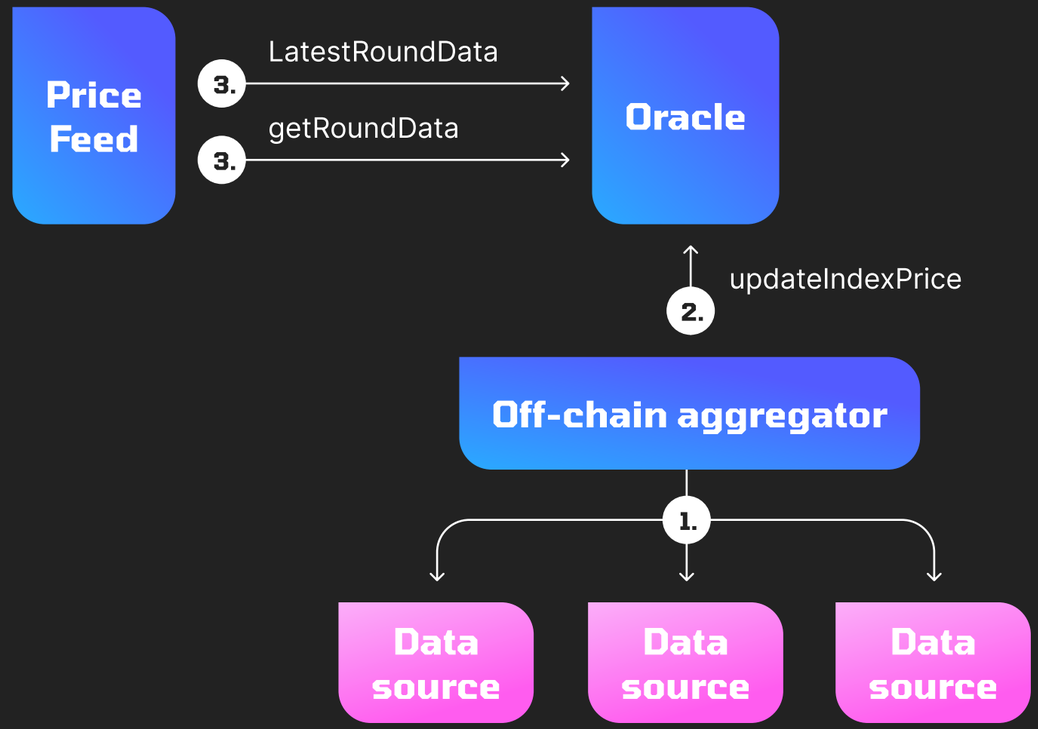
# Implementation

**Perp v2 trades fungible tokens, Deriverse trades non-fungible tokens.**

## Solution

Use existing Chainlink PriceFeed with our Chainlink aggregator which will fetch data from our backend. Backend can work as an aggregator with multiple floor price sources (MagicEden, Opensea, NFT Scoring, Upshot, ...), calculate the index price and push value to oracle smart contract. Then price feed can use our oracle smart contract as a data source for index prices.

ChainlinkPriceFeedV2 will be adjusted to be able to work with our aggregator.



# Admin part

Because of the fact that we will often want to add new collections, which means adding new base tokens, we propose to implement an admin page.

**Adding new base token means:**

- Deploy modified Aggregator smart contract
- Deploy modified ChainLinkPriceFeed smart contract
- Deploy modified BaseToken smart contract
- Create new pool for token
- Calling MarketRegistry.addPool

Admi part

# DERIVERSE

+

cleevio