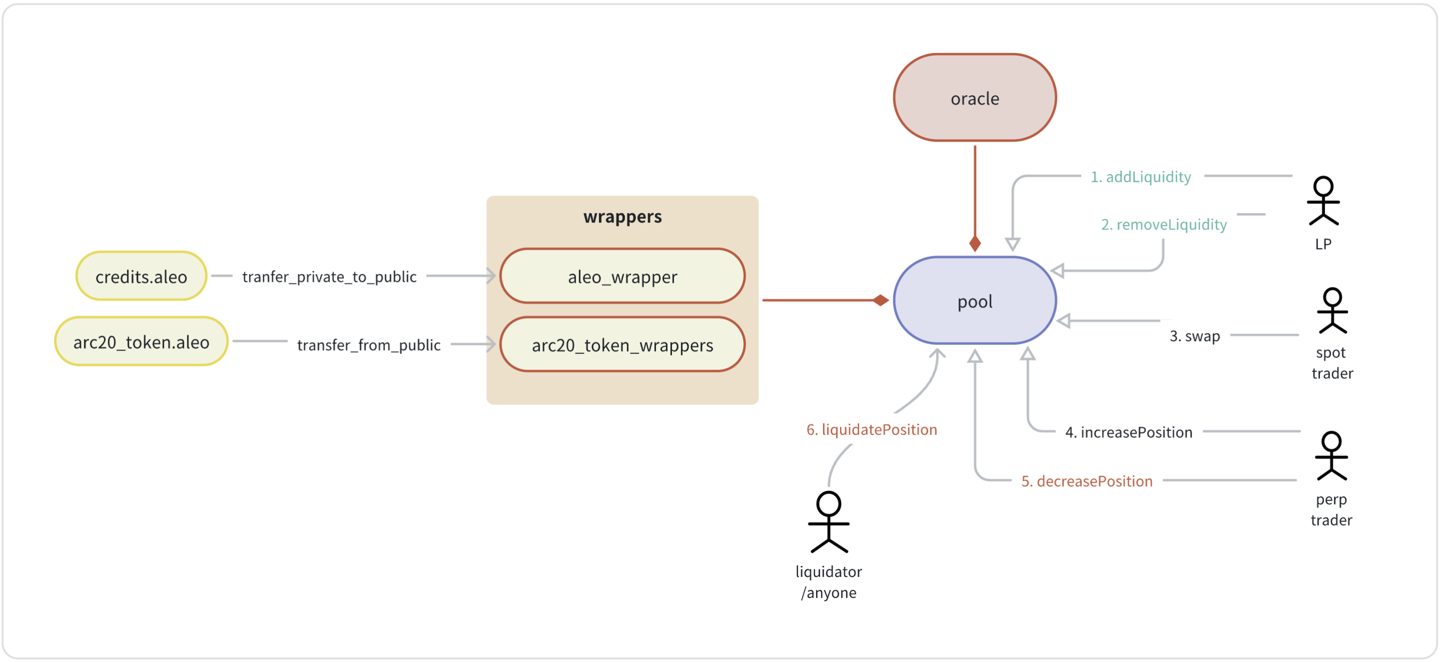
Aleo perpetual architecture design

1. Introduction

Aleo perpetual mainly contains the following modules:

- **`pool`** : master contract which provides a basket of assets and includes the following core functions.
    - `add_liquidity` : users can transfer assets into the pool to obtain the alp token which is created by project.
    - `remove_liquidity` : users can burn alp tokens to withdraw tokens supported by the pool.
    - `increase_position` : users can buy long or sell short to obtain the corresponding profits.
    - `decrease_position` : users can also adjust their position or collateral or leverage by decrease position to reduce risk.
    - `liquidate_position` : if users' collateral can not cover fees or exceed the max leverage, then its position will be liquidated.
    - `swap` : currently, pool can only support a few tokens, so we also provide the swap function.
- **`wrappers`** : this contract will be used to receive all arc20 tokens, and mint wrapped tokens in pool contract. Currently, only waleo, wusdc, wusdt, weth, and wbtc are supported.
- **`tp/sl`** **:** provides stop-loss and take-profit functions.
- **`oracle`** **:** provides feed price functions onchain.



2. ALP

When users execute `add_liqudity`, they will obtain alp tokens which are created by this project. While users execute `remove_liquidity`, alp tokens will be burned, arc20 tokens will be received.

3. wrappers

Wrappers contracts are used to receive all arc20 tokens, and mint wrapped tokens in the pool contract. Currently, only waleo, wusdc, wusdt, weth, and wbtc are supported.

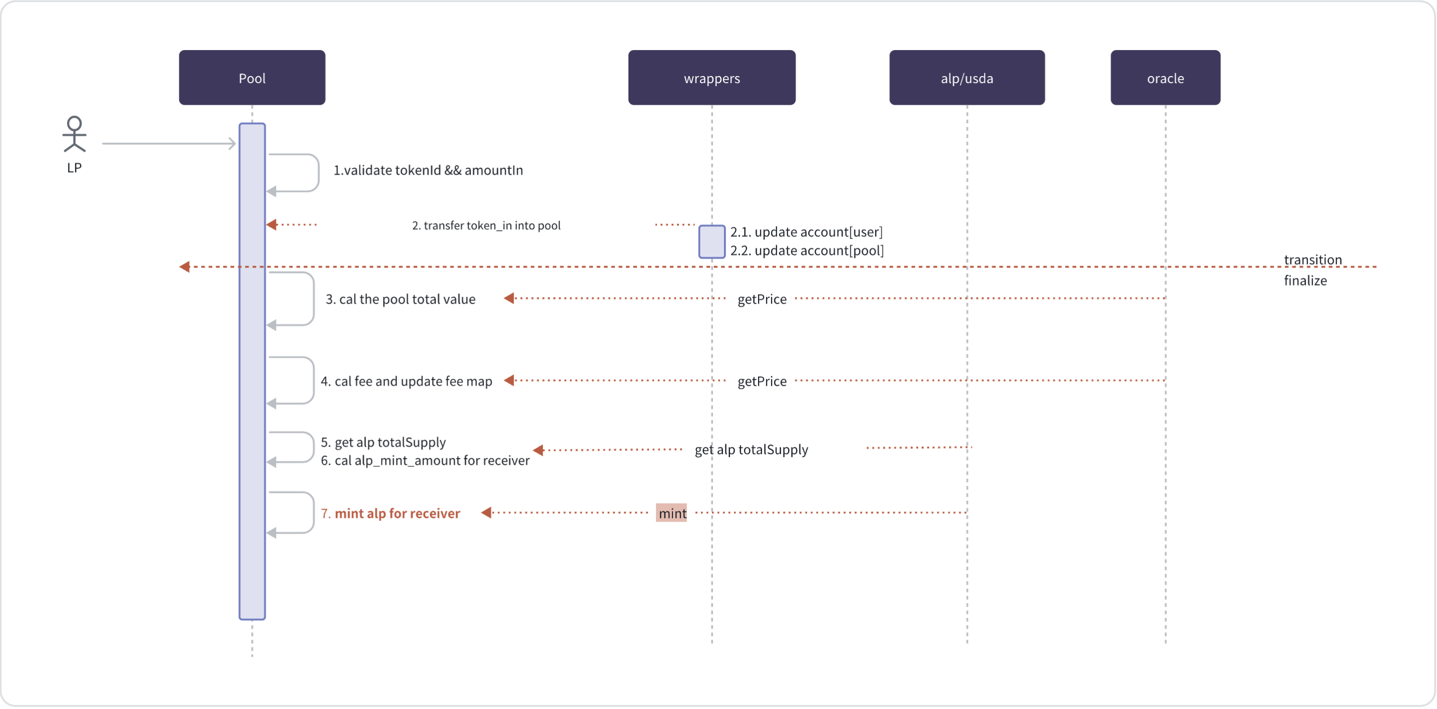
```
async transition wrap_public(
    public amount: u128,
    public receiver: address
) -> Future {}

async transition unwrap_public(
    public amount: u128,
    public receiver: address
) -> Future{}
```

4. pool

This contract provides a basket of assets and includes the following core interfaces.

4.1 add_liquidity

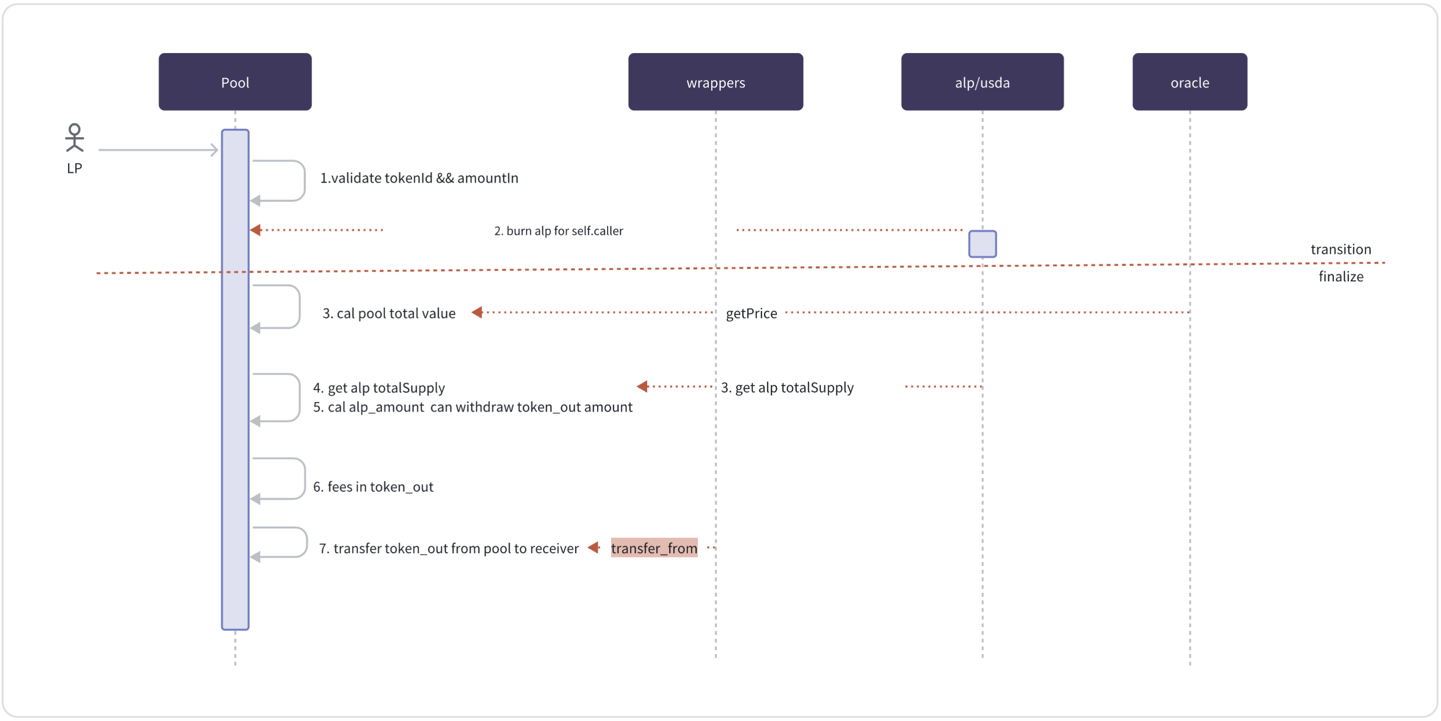
```
async transition add_liquidity(
        public token_in_id: field,
        public amount_in: u128,
        public min_usd: u128,
        public min_alp_amount: u128,
        public receiver: address
  ) -> Future {}
```



Pool
wrappers
alp/usda
oracle
LP
1.validate tokenId && amountIn
2. transfer token_in into pool
2.1. update account[user]
2.2. update account[pool]
transition
finalize
3. cal the pool total value
getPrice
4. cal fee and update fee map
getPrice
5. get alp totalSupply
6. cal alp_mint_amount for receiver
get alp totalSupply
7. mint alp for receiver
mint

4.2 remove_liquidity

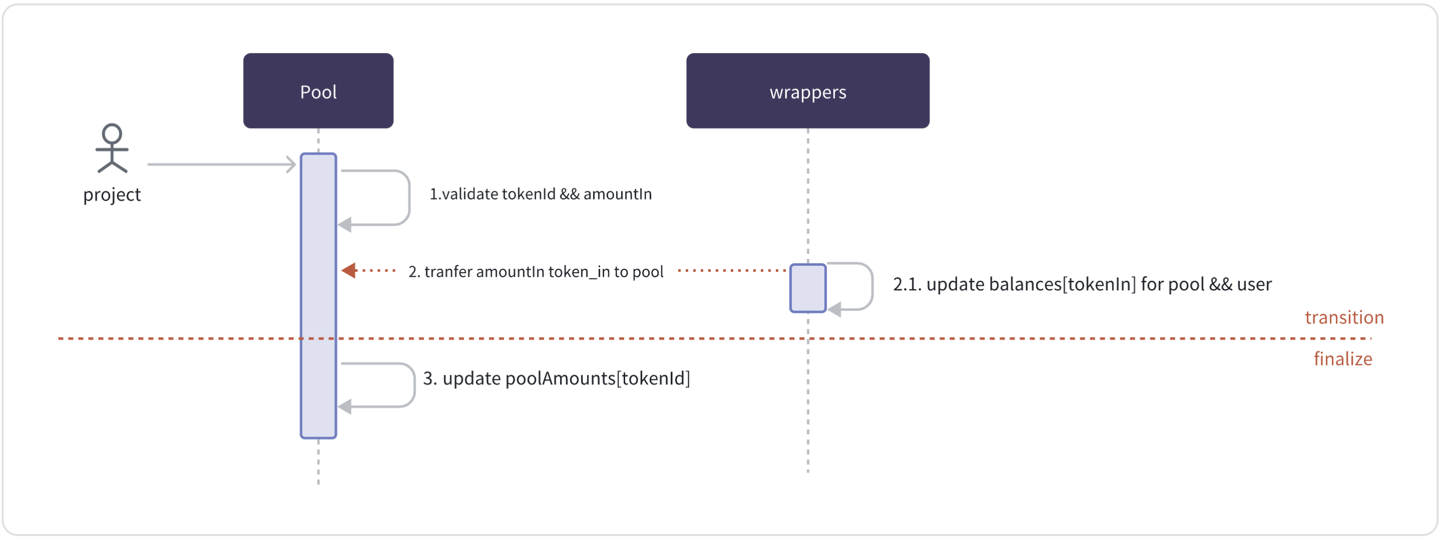
```
async transition remove_liquidity(
        public token_out_id: field,
        public alp_amount: u128,
        public min_out_amount: u128,
        public receiver: address
  ) -> Future {}
```



4.3 direct_pool_deposit

With this interface, anyone can deposit funds to the pool, then the pool's total value in USD will be increased. Which means a single alp token value will be increased.

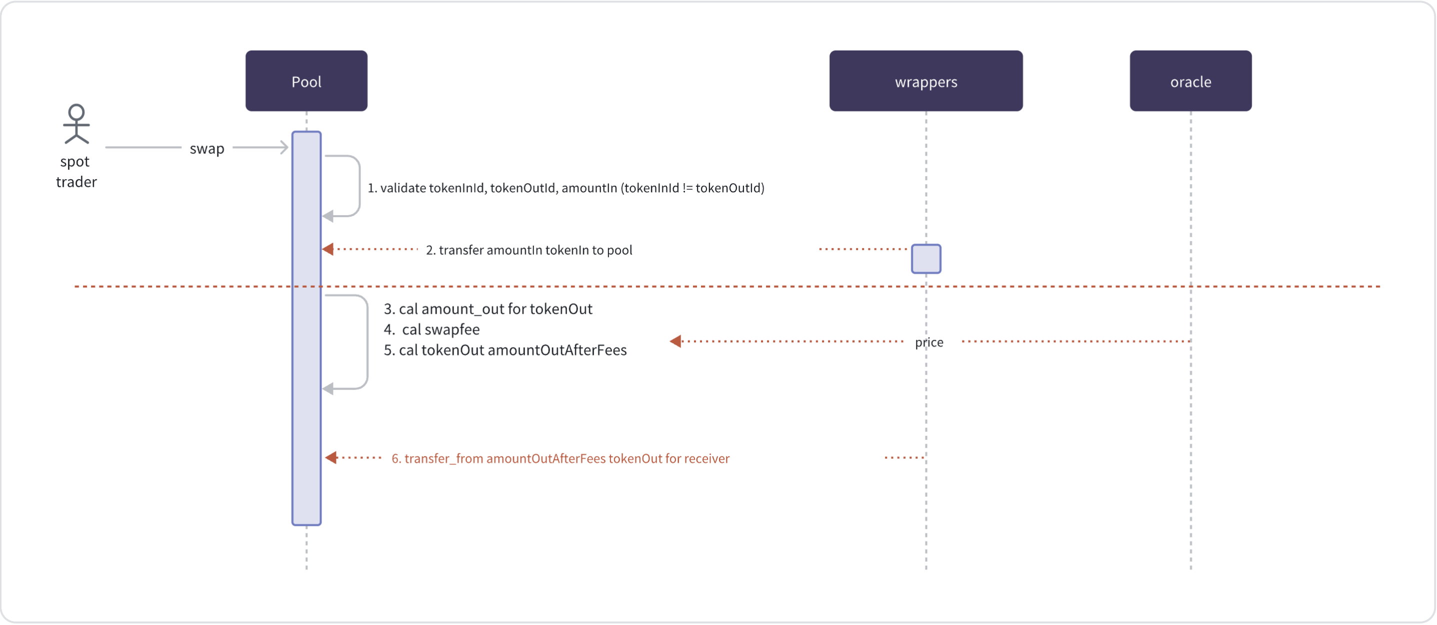
```
async transition direct_pool_deposit(
    public token_id: field,
    public amount_in: u128
 ) -> Future {}
```


Pool
wrappers
project
1.validate tokenId && amountIn
2. tranfer amountIn token_in to pool
2.1. update balances[tokenIn] for pool && user
transition
finalize
3. update poolAmounts[tokenId]

4.4 swap

Currently, the pool can only support a few tokens, such as "wbtc, weth, waleo, wusdc", so if users don't have these tokens, swap interface will provide convenience. At present, swap alp or usda is not supported.

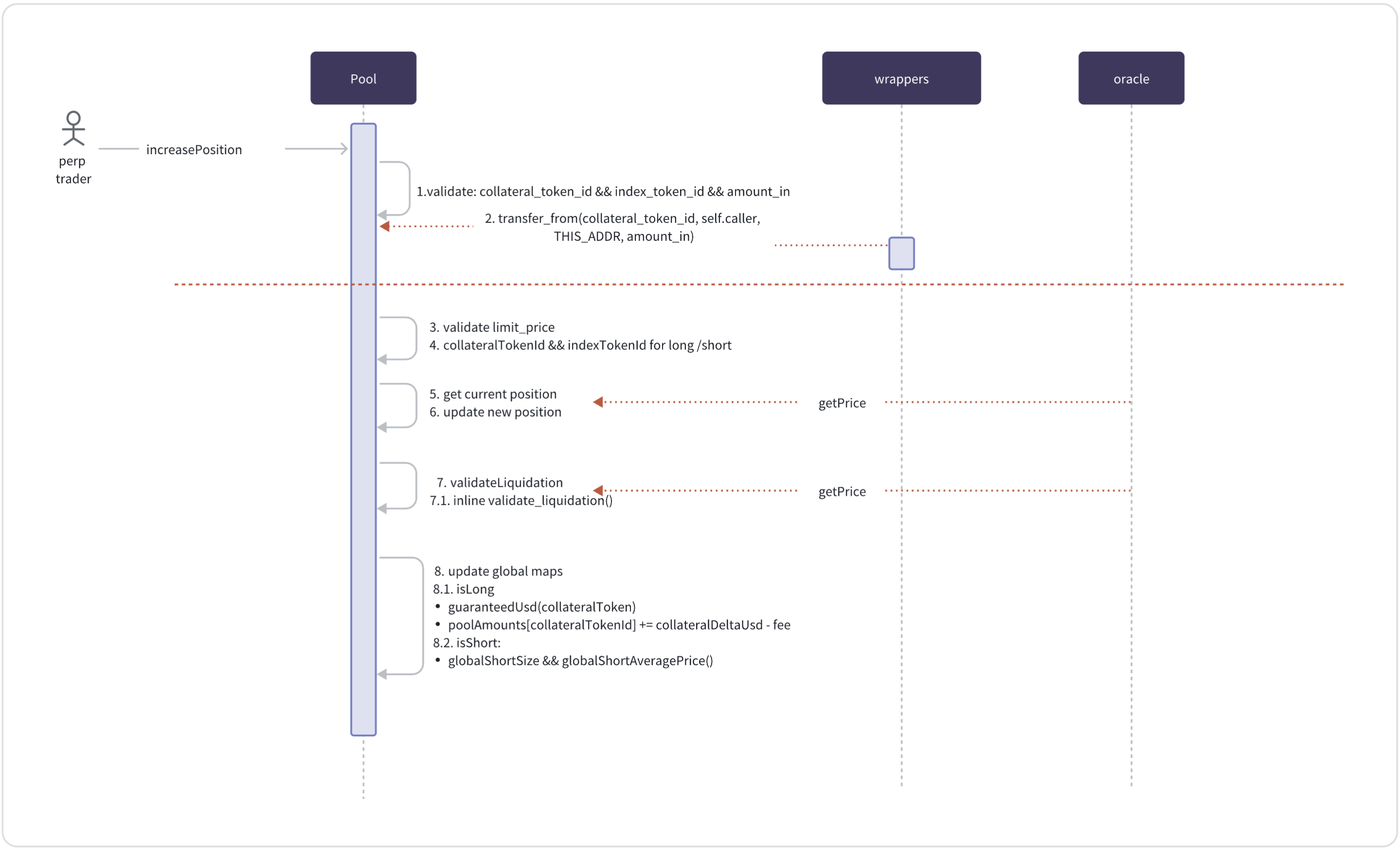
```
async transition swap(
        public token_in_id: field,
        public amount_in: u128,
        public token_out_id: field,
        public min_amount_out: u128,
        public receiver: address
    ) -> Future {}
```


Pool
wrappers
oracle
spot trader
swap
1. validate tokenInId, tokenOutId, amountIn (tokenInId != tokenOutId)
2. transfer amountIn tokenIn to pool
3. cal amount_out for tokenOut
4. cal swapfee
5. cal tokenOut amountOutAfterFees
price
6. transfer_from amountOutAfterFees tokenOut for receiver

4.5 increase_position

Open long or short positions. For a new user, initial collateral's value in USD must be higher than a small value, eg. 10USD. For long, collateral_token must equal index_token, and collateral must be non-stable token. For short, collateral_token_id must be a stable token, while index_token should be non-stable token and collateral must be stable token. Besides, for long limit_price must be higher than mark price, for short limit_price must be lower than mark price.

```
async transition increase_position(
        public user: address,
        public collateral_token_id: field,
        public amount_in: u128,
        public index_token_id: field,
        public size_delta: u128,
        public is_long: bool,
        public limit_price: u128
 ) -> Future {}
```

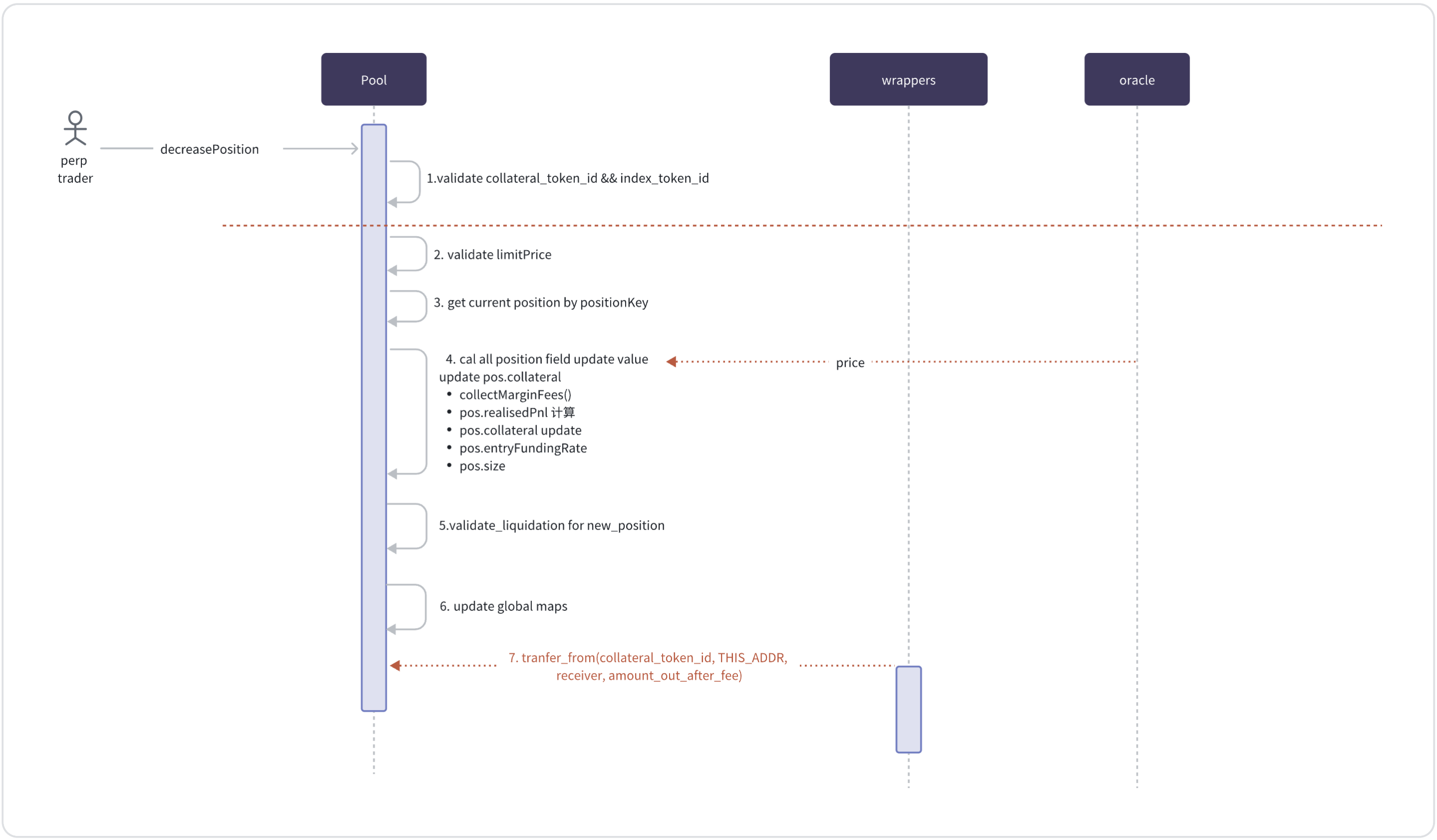

Pool
wrappers
oracle
perp trader
increasePosition
1.validate: collateral_token_id && index_token_id && amount_in
2. transfer_from(collateral_token_id, self.caller, THIS_ADDR, amount_in)
3. validate limit_price
4. collateralTokenId && indexTokenId for long /short
5. get current position
6. update new position
getPrice
7. validateLiquidation
7.1. inline validate_liquidation()
getPrice
8. update global maps
8.1. isLong
• guaranteedUsd(collateralToken)
• poolAmounts[collateralTokenId] += collateralDeltaUsd - fee
8.2. isShort:
• globalShortSize && globalShortAveragePrice()

4.6 decrease_position

Users can also adjust their position or collateral or leverage by decrease position.

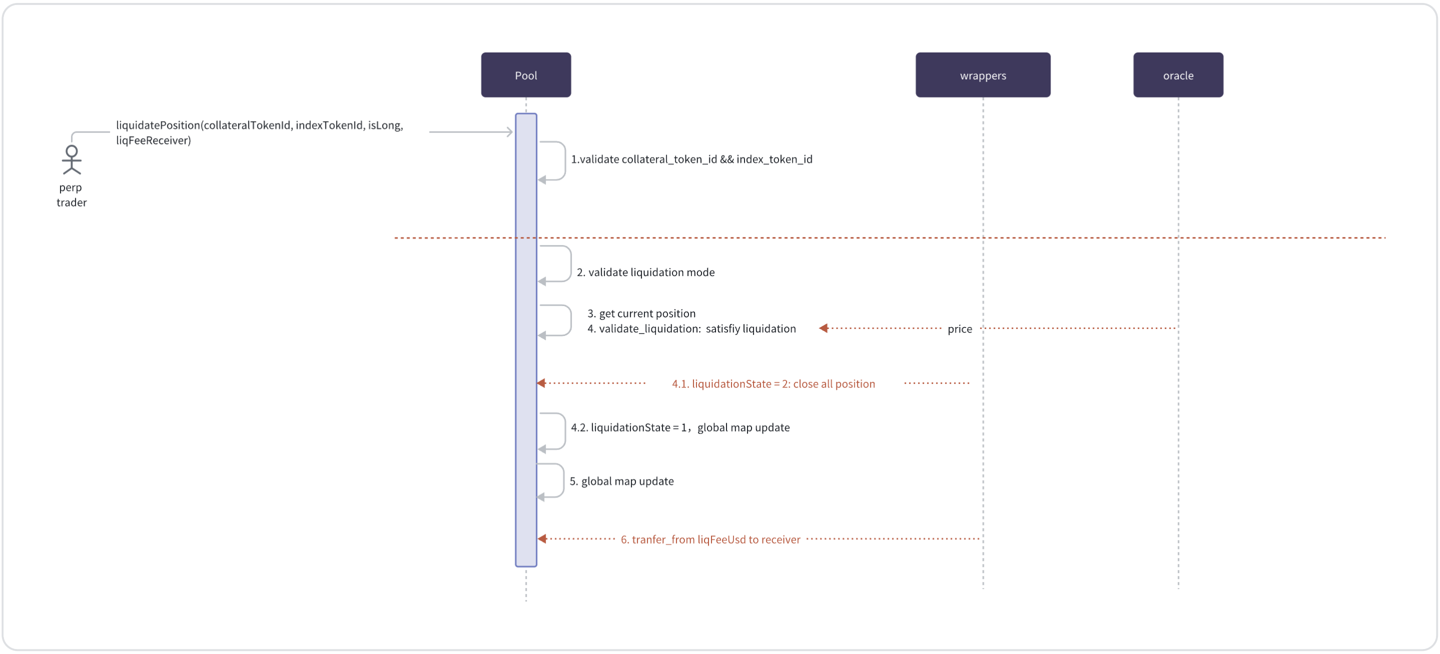
- collateral_delta is decreased collateral value.
- size_delta: is decreased position size.

```
async transition decrease_position(
        public user: address,
        public collateral_token_id: field,
        public index_token_id: field,
        public collateral_delta: u128,
        public size_delta: u128,
        public is_long: bool,
        public receiver: address,
        public limit_price: u128
 ) -> Future {}
```


Pool
wrappers
oracle
perp trader
decreasePosition
1.validate collateral_token_id && index_token_id
2. validate limitPrice
3. get current position by positionKey
4. cal all position field update value
update pos.collateral
• collectMarginFees()
• pos.realisedPnl 计算
• pos.collateral update
• pos.entryFundingRate
• pos.size
price
5.validate_liquidation for new_position
6. update global maps
7. tranfer_from(collateral_token_id, THIS_ADDR, receiver, amount_out_after_fee)

4.7 liquidate_position

```
async transition liquidate_position(
        public liq_user: address,
        public collateral_token_id: field,
        public index_token_id: field,
        public is_long: bool,
        public liq_fee_receiver: address
    ) -> Future {}
```


Pool
wrappers
oracle
liquidatePosition(collateralTokenId, indexTokenId, isLong, liqFeeReceiver)
perp trader
1.validate collateral_token_id && index_token_id
2. validate liquidation mode
3. get current position
4. validate_liquidation: satisfiy liquidation
price
4.1. liquidationState = 2: close all position
4.2. liquidationState = 1，global map update
5. global map update
6. tranfer_from liqFeeUsd to receiver

5. tp/sl

This contract provides stop-loss and take-profit interfaces. Users can create decreased position requests by executing `create_decrease_position`. These requests will be stored on-chain, and will be executed by `execute_decrease_position`.

```
async transition create_decrease_position(
        public collateral_token_id: field,
        public index_token_id: field,
        public collateral_delta: u128,
        public size_delta: u128,
        public is_long: bool,
        public receiver: address,
        public acceptable_price: u128,
        public execution_fee: u128
 ) -> Future {}

 async transition cancel_decrease_position(
        public key: field,
        public execution_fee: u128,
        public execution_fee_receiver: address
  ) -> Future {}

  async transition execute_decrease_position(
        public key: field,
        public execution_fee_receiver: address,
        public data: DecreasePositionRequest
    ) -> Future {}
```