

1. Staking Contracts v2	2
1.1 IIP	3
1.2 Guides	5
1.2.1 Upgrading to v2	5
1.2.2 Migrating an Unlocked Deposit	5
1.2.3 Locking Flexible Deposits in v2	6
1.2.4 Staking and Claiming in v2	6
1.3 Technical	6
1.3.1 Architecture	6
1.3.2 Contracts	9
1.3.2.1 External Helper Contracts	9
1.3.2.1.1 UUPSUpgradeable	9
1.3.2.1.2 ReentrancyGuardUpgradeable	10
1.3.2.1.3 PausableUpgradeable	10
1.3.2.1.4 Initializable	10
1.3.2.1.5 Ownable	10
1.3.2.2 Illuvium Libraries	10
1.3.2.2.1 Stake	10
1.3.2.2.2 ErrorHandler	11
1.3.2.3 Base Contracts	11
1.3.2.3.1 FactoryControlled	11
1.3.2.3.2 VaultRecipient	12
1.3.2.3.3 Timestamp	12
1.3.2.3.4 CorePool	12
1.3.2.3.5 V2Migrator	17
1.3.2.4 Deployed Contracts	17
1.3.2.4.1 ILVPool	17
1.3.2.4.2 SushiLPPool	18
1.3.2.4.3 PoolFactory	19
1.3.2.4.4 FlashPool	20
1.3.2.4.5 Vault	22
1.4 References	24
1.4.1 Glossary	24
1.4.2 Equations	24

Staking Contracts v2

Staking Contracts v2 extend and refactor much of the functionality of the existing system. It focuses on bug fixes, workflow QOL improvements, gas efficiency, and many more features. It is a mandatory update for all those staking in the Illuvium System.

Main Open Zeppelin Contracts:

UUPSUpgradeable.sol

Inherited by core pool, flash pool and pool manager contracts, contains the functionality that follows EIP1822 and ERC1967 standards in order to allow safe upgradeability to smart contracts, and introduces initializer functions instead of constructor. Requires the use of open-zeppelin and hardhat upgradeability libs for deployments and upgrades following the standard.

ReentrancyGuardUpgradeable.sol

Includes variables and a modifier for protection of specific functions that calls third-party contracts against reentrancy attacks. Protects some important functions related to staking, unstaking, yield processing and claiming.

PausableUpgradeable.sol

Includes variables and a modifier that introduces pausing functionality for certain functions in the contracts. Stake and yield processing/claiming functions can be paused by the eDAO if necessary. Unstaking functions are never paused so users can always remove their unlocked funds.

Initializable.sol

Allows the creation of initializer functions instead of constructors without importing all the code from UUPSUpgradeable. Used by abstract base contracts that don't require all functionality from EIP1822 and ERC1977.

Ownable.sol

Adds an address (the `owner`) with special access to the contracts.

Libraries:

Stake.sol

Staking utility that stores each stake data packed in 1 storage slot and has a view function that receives the Data pointer for returning stake weight (i.e we don't need to store weights per stake).

Errors.sol

Adds support for Solidity 0.8.4 custom errors instead of reason strings and adds traceability for function signatures.

Base contracts:

FactoryControlled.sol

Links pools to the factory deployed instance, includes a modifier to protect functions that can only be called by the factory owner (eDAO).

VaultRecipient.sol

Inherited by core pools, stores vault variables and includes a modifier to protect functions that can only be called by the vault contract.

Timestamp.sol

Utility contract that delegates `block.timestamp` readings to an internal function, which allows easier mocking in unit tests.

CorePool.sol

Base smart contract for ILV and LP pool. Stores each pool user by mapping its address to the user struct. User struct stores v2 stakes, which fit in 1 storage slot each (by using the Stake lib), total weights, pending yield and revenue distributions, and v1 stake ids. ILV and LP stakes can be made through flexible stake mode, which only increments the flexible balance of a given user, or through locked staking. Locked staking creates a new `Stake` element fitting 1 storage slot with its value and lock duration. When calculating pending rewards, CorePool checks v1 locked stakes weights to increment in the calculations and stores pending yield and pending revenue distributions. Every time a stake or unstake related function is called, it updates pending values, but don't require instant claimings. Rewards claiming are executed in separate functions, and in the case of yield, it also requires the user checking whether ILV or sILV is wanted as the yield reward.

V2Migrator.sol

Inherits all CorePool.sol functionality and adds migration related functions. Adds functionality to allow minting v1 yield in v2 pools, allows a v1 user to bring v1 locked stakes to v2 (that aren't yield), and stores receipt hashes in order to keep track of v1 activity in v2 (minted yield, locked stakes already migrated).

Deployed Contracts:

ILVPool.sol (Deployed 1x)

ILV core pool, has extra functionality to allow routing claims to other pools registered in the protocol (so 1 transaction yield or vault claims are possible) and to receive compounding transactions from other external pool claims (such as LP pool or a flash pool).

SushiLPPool.sol (Deployed 1x)

Extends all functionality from V2Migrator base contract.

PoolRegistry.sol (Deployed 1x)

Registers deployed pools in the protocol, controls ILV and sILV minting (has the permission to mint from ILV and sILV contracts), emissions schedule and pool weights.

FlashPool.sol (1 deploy per flash pool)

Has a similar functionality to CorePool.sol base contract, but uses much cheaper and simpler data structures due to flash pool natures. Instead of dealing with weights and non fungible stakes, it stores token data in a single uint slot and executes yield calculations based on number of tokens instead of weight (since flash pools have no lock functionality).

Vault.sol (Deployed 1x)

Receives ETH from the `receive()` function and allows conversion to ILV by the address with the role `ROLE_VAULT_MANAGER` (0x0001_0000). This conversion can be done in multiple steps, which means it doesn't require converting all ETH balance in 1 function call. The vault is also responsible to be calling `receiveVaultRewards()` function in the core pools, which takes care of calculations of how much ILV should be sent to each pool as revenue distribution

IIP

Simple Summary

After collecting months of user feedback since the launch of the Illuvium Staking Contracts v1, the Illuvium team has designed a new set of smart contracts that implement additional features, gas optimizations, and fixes. The goal is to deploy a better version of staking with new mechanisms, smoother UX and prepare Illuvium protocol with a more flexible set of contracts for future features.

Abstract

The Illuvium DAO will finalise work on a new set of smart contracts, which will then be audited before deployment. Once complete we will implement the contracts and the upgrade process will begin. To gain yield, staking contracts v2 will be mandatory as the upgrade process requires the existing contracts to be deprecated. A detailed upgrade document will be created that will guide token holders and make the entire upgrade process as painless as possible.

It will include different user scenarios for different situations such as those with many deposits, low balance, flexible deposits, etc...

Overview

During a window of 2 weeks Staking v1 will be deprecated, and then v2 will begin generating yield. Users will be required to upgrade to v2, and in exchange receive a number of benefits and improvements.

i Please see Definitions prior to reading, to become familiar with the exact terminology.

Upgrade Process

Staking v1 users will have a two week window prior to v2 launch where they'll be able to execute the upgrade function. During this period yield will be halted in v1 and then when v2 begins yield, it will be given a bonus to cover the 2 week period of no yield from v1.

Users can still upgrade after the window closes, but they won't receive yield until they upgrade.

A Merkle Tree of each user's accumulated yield weight in v1 will be published and stored in IPFS for full transparency. Anyone will be able to verify the authenticity of the data.

A final claim of the remaining pending yield from v1 is optional at this point, but will be required if the user uses the migration function in the next step.

Detailed Process for Each type of Stake:

- Flexible Deposit in ILV Pool - Doesn't count to yield in v2 unless unstaked and locked for a minimum period of 1 month in v2.
- Flexible Deposit in LP Pool - Doesn't count to yield in v2 unless unstaked and locked for a minimum period of 1 month in v2.
- Locked Deposit in ILV Pool - Counts to yield rewards after upgrading to v2 at existing weight.
- Unlocked Deposit in ILV Pool - Counts to yield rewards after upgrading to v2 at existing weight. Additionally, can be migrated to v2 in order to save on gas fees from claiming yield.
- Locked Deposit in LP Pool - Counts to yield rewards after upgrading to v2 at existing weight.
- Unlocked Deposit in LP Pool - Counts to yield rewards after upgrading to v2 at existing weight. Additionally, can be migrated to v2 in order to save on gas fees from claiming yield.
- Locked ILV Yield - Counts to yield rewards after upgrading to v2, at existing weight.
- Unlocked ILV Yield - N/A (Has not been 12 months since launch but will continue to count to yield rewards even after unlocked)
- Seed and Team - Automatically migrated to v2. Does not count towards yield. Can be transferred to ILV pool on unlock.

Main Changes

Rewards Claiming

Users will no longer be required to claim on every stake or unstake function. This selection will only be required when claiming rewards, which can now be done for all pools in a single transaction. Users will also be able to unstake all unlocked tokens in a single transaction.

i A claim will still need to be made when migrating Unlocked Deposits to v2, however if the rewards are smaller than the gas fee, users will be able to forgo this yield.

Fixes and Gas Optimizations

The new architecture will come with multiple gas optimizations by reducing the number of storage slots used by the deposit data structure, using new yield processing flows, and optimizing the number of reads. This will make transactions much cheaper.

i Users that still have Deposits in v1 will pay slightly higher gas fees. As these are migrated to v2, gas fees will continue to decrease. Users are not required to migrate, and the yield will still be calculated from v1, so it will depend on how often you claim to decide if migration is worth it.

Transfer Positions

Since it's been frequently asked by the community, user can transfer their total positions in v2 to new addresses. This is useful in situations such as when a personal private key is leaked or anything that could motivate a user to move into a new Ethereum address.


i v1 locked positions are still unable to be moved to new addresses

Upgradeability

Staking v2 (Factory, Core Pools, and Flash Pools) implements the Universal Upgradeable Proxy Standard. This will be a big addition to Illuvium DAO governance since Council members are now able to vote on upgrades proposed to the contracts, opening a lot of possibilities for the future of Illuvium staking features and even for new products. It brings new powers to ILV token holders, and reinforces our Govern To Earn mantra. With upgradeability, there is a lot of flexibility for new IIPs and the evolution of Illuvium DAO as a whole.

Flexible Staking Removed

Flexible staking is being removed in Staking v2, which means v1 users that staked in flexible will be required to lock for a minimum period of 1 month in the new contracts. This will further reduce ILV token circulation and allow only longer term users to participate in revenue distributions. Weight calculations still follow the same 1 2 multiplier used in v1, which keeps the system fair and balanced for the long term.

 Maximum locked period is still 1 year (2.0 weight)

NFT Distributions

A separate IIP is in progress with more details on the NFT distribution integration, but Staking v2 will include all the features required to make this integration seamless.

Rationale

Staking v2 will be a big step for Illuvium DAO, and will bring many enhancements and fixes to the protocol which have been collected since v1 launch. With a lower gas cost, Illuvium DAO should have access to more users with a smaller barrier of entry and a simpler UX due to the flow changes and fixes. Contracts upgradeability will ensure that any future changes are swift, and will empower the community with even more effective governance.

Definitions

Locked Deposit - This is when a user has staked tokens with a lock, and the lock has not yet expired.

Unlocked Deposit - This is when a user has staked tokens with a lock, and the lock has expired.

Flexible Deposit - This is when a user has staked tokens with no lock.

Locked ILV Yield - Yield that has been claimed and locked for 12 months.

Unlocked ILV Yield - Yield that has been claimed more than 12 months ago, and hence unlocked.

Upgrade to v2 - The process of allowing the v2 contract to read the v1 state and including it into yield calculations in the v2 contract

Migrate - The process of taking an Unlocked Deposit in v1 and unstaking it, then restaking it v2 with the original weight.

Withdraw - Unstaking a Flexible or Unlocked Deposit (but not migrating it to v2)

Test Cases

N/A

Guides

Upgrading to v2

When To Do This?

This should be done after the shut down of v1 yield but before the initiation of v2 yield (if possible). This is require to start getting yield again and is the only mandatory step. This lets the v2 interface know to read your data and start yield.

1. Go to page
2. Connect wallet
3. Select staking
4. Click 'upgrade' (once per pool that you have deposits in)
5. Auth transaction
6. Done

Migrating an Unlocked Deposit

1. Go to website
2. Connect wallet
3. Go to staking

4. Select 'migrate' on an UNLOCKED deposit.
5. Confirm transaction
6. Done.

Locking Flexible Deposits in v2

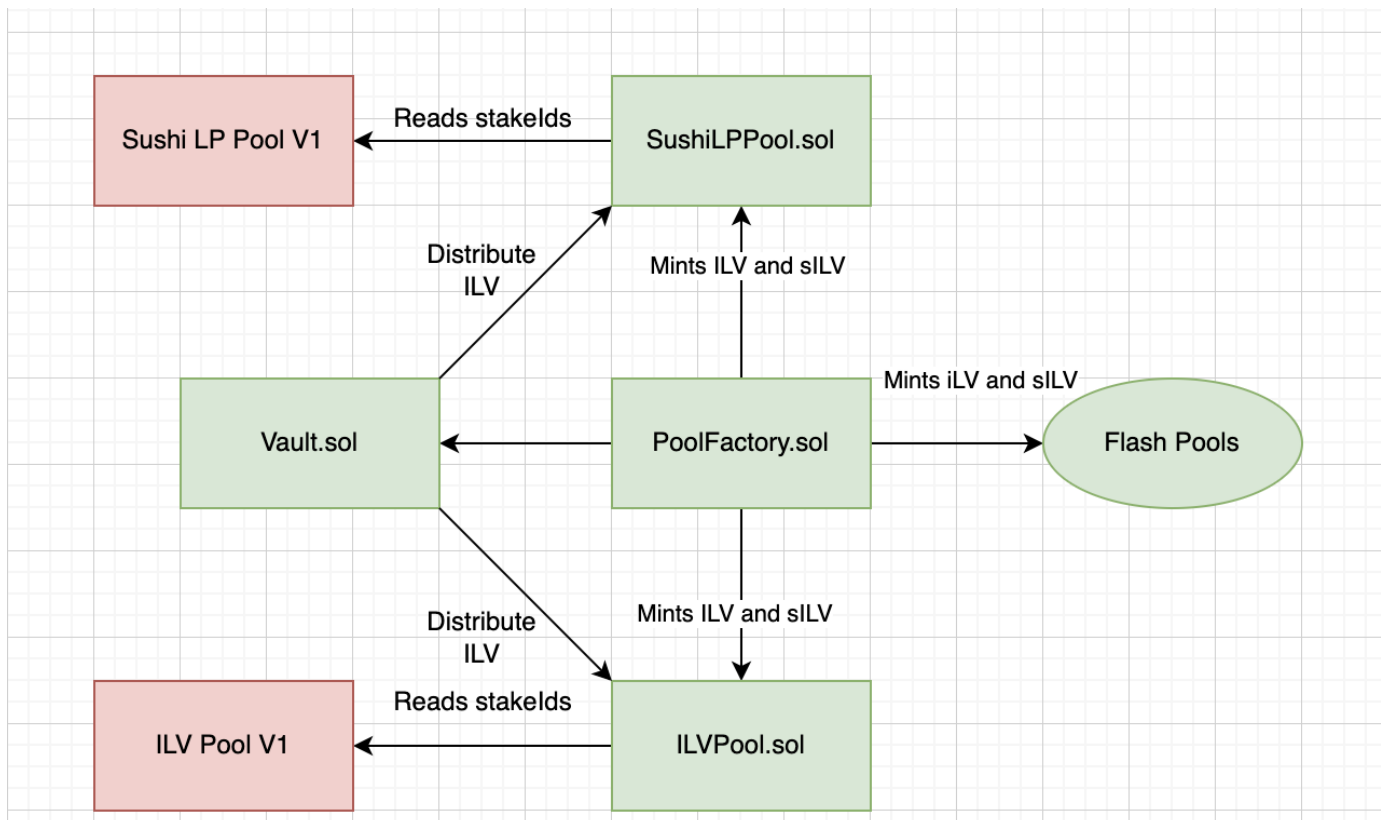
1. Go to site
2. Connect wallet
3. Go to staking
4. Select a flexible deposit
5. Select 'withdraw'
6. confirm
7. Go to 'Staking v2 tab'
8. Select 'make deposit'
9. Fill in details.
10. Done.

Staking and Claiming in v2

1. Go to website
2. Connect wallet
3. Go to staking v2
4. Select 'claim'
5. Choose to claim for a single batch, or for all rewards
6. Select ILV or sILV
7. Auth transaction
8. Done.

Technical

Architecture



As described in the image above, each v2 core pool instance deployed has its corresponding v1 core pool attached. Through the V2Migrator abstract contract, users are able to load their v1 stake ids and safely pass to the v2 pool contract which verifies the validity of the v1 stake and store its values.

A v1 stake may be locked, which means only the user that owns this locked stake in v1 is able to move it once it unlocks. However, with v2 pools yield calculations we need to not only take new stakes into account, but also previous v1 stakes to be added into yield calculations. The following diagram shows which function calls support reading v1 data and doing calculations correctly, both for new v2 users and users that came from v1 which might have new stakes in v2.



_useV1Weight

stakeAsPool (ILVPool only)

executeMigration (ILVPool only)

mintV1YieldMultiple (ILVPool only)

migrateUser

fillV1StakeId

_stake

unstakeLocked

unstakeLockedMultiple

_claimYieldRewards

_claimVaultRewards

migrateLockedStakes

The process that brings v1 users to the v2 staking pools is slightly different on the ILVPool contract compared to the Sushi LP Pool. In the ILVPool we call the `executeMigration` function, which allows v1 stake ids and the yield weight stored in a Merkle tree to be stored in the v2 pool. We don't have yield in LP tokens, hence we just call `migrateLockedStakes` in order to store v1 stake ids (which results in a cheaper transaction).

Yield Weights Merkle Tree



Yield Weights Tree

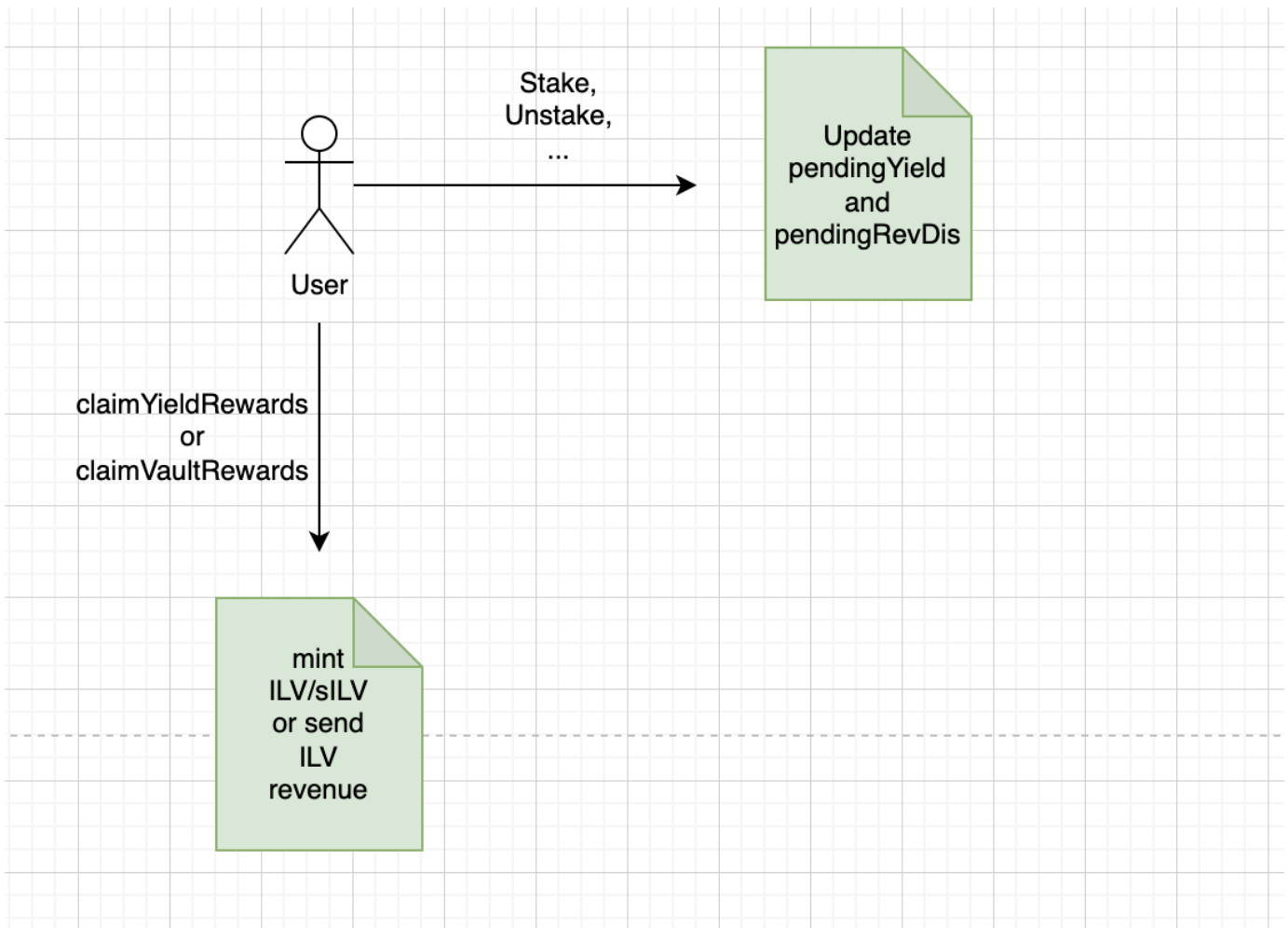
1. Bitmap index
2. Address
3. Total yield weight

Some of the most important components of the v2 contracts which will be explored during the launch, are the functions related to the yield weights migration and minting. Many v1 users have multiple yield stake ids which means it would be expensive and essentially unfeasible to migrate v1 yield stake ids the same way we store v1 stake ids in the v2 user struct. With that in mind, a Merkle proof verification has been designed in order to migrate the total v1 weight of a given user in the tree in one transaction, adding this value to `user.totalWeight`. This way, we only require 2 SSTORE operations; one for the user.totalWeight and the other to update the users yield migration bitmap, compared to multiple SSTOREs per stake id. Most v1 users have on average a low number of locked deposits, therefore the `executeMigration` function takes the merkle tree verification parameters (as displayed in the image below) and the v1 stake ids to completely bring the v1 user to v2. This is more efficient for the yield stake ids.

The tree will be published in the beginning of a 2 week window where users will be able to call `executeMigration` in the ILVPool and `migrateLockedStakes` in the Sushi LP pool, and uploaded to IPFS. Then, the Merkle root will be stored in the ILVPool contract, and users are then able to verify their data in the tree.

The v1 PoolFactory, which currently holds the rights to mint ILV, and the v1 pools holding the right to mint sILV are going to be disabled in the beginning of the migration window by the eDAO. Once v2 pools are setup and running, the v2 PoolFactory will have already received ILV and sILV minting rights and all v1 yield that unlocks in the v1 ILV pool, will be able to be minted through the v2 ILV pool. By routing the minting call from v2 and disabling it in v1, we are able to safely mint ILV yield while updating the necessary values as v2 yield is being generated.

Yield and Revenue Distribution Rewards in v2



Another important flow being redesigned in v2 is the way rewards are calculated, stored and claimed. Now, `claimYieldRewards` and `claimVaultRewards` in Flash Pools or in the Sushi LP Pool contract, after claiming the pending yield in the pool either with ILV or sILV as requested by the user, if the user chooses ILV (which needs to be compounded to the ILV pool through `stakeAsPool` function), stores accumulated ILV pool yield in the `user.pendingYield` variable. This is because of the new separation of `processRewards` and the claim functions. By processing the rewards we only store accumulated tokens (yield or revenue distribution), while in the claim we effectively execute the tokens distributions. This way, we have a cleaner process where we separate the yield or revenue distribution storage to the effective sILV /ILV minting or ILV compounding, having simpler transactions and improving UX so the users can more easily claim their yield. The ILVPool contract also acts as a "router" to the other pools by allowing yield and revenue distribution claiming calls to be aggregated in one function call through `claimYieldRewardsMultiple` and `claimVaultRewardsMultiple`, allowing users that stake in multiple pools to have a faster and cheaper claim process.

Regarding calculations, one of the main differences is that now we need to take v1 stake ids into account, and we can't rely 100% on v2 state for that. The approach decided to solve this problem, is to use the `_useV1Weight` internal function, which is responsible to check v1 contracts state, notice weight changes and do the necessary treatment of v1 stake ids to return the values which will be included in the yield calculation. This function also calls internally `_getSubRewardsValue`, which takes v1 parameters, compares to v2 stored parameters (weights) and recalculates the value which is updated in every `_processRewards` call to subtract when the user updates the `user.pendingYield` and `user.pendingRevDis` values. This way we are able to detect if the user has unstaked tokens from a v1 stake, and hence shouldn't receive rewards for these tokens since the last `_processRewards` call. So essentially, in this flow, users have their rewards calculated based on their stored `user.totalWeight` + their current v1 weight, with the possibility of having their subtract values updated depending on whether the user unstaked from a v1 stake id, but this won't be required for v2 stake ids since v2 contracts has all the required data stored and can ensure that `user.pendingYield` and `user.pendingRevDis` is always updated on unstake and other state mutating calls.

Check the **Contracts** sections for more details in each individual part of the protocol.

Contracts

External Helper Contracts

We use a number of external contracts from Open Zeppelin.

UUPSUpgradeable

Inherited by core pool, flash pool and pool manager contracts. Contains the functionality that follows EIP1822 and ERC1967 standards in order to allow safe upgradeability to smart contracts and introduces initializer functions instead of constructor. Requires the use of open-zeppelin and hardhat upgradeability libs for deployments and upgrades following the standard.

See: <https://docs.openzeppelin.com/contracts/4.x/api/proxy#UUPSUpgradeable>

ReentrancyGuardUpgradeable

Includes variables and a modifier for protection of specific functions that calls third-party contracts against reentrancy attacks. Protects some important functions related to staking, unstaking, yield processing and claiming.

PausableUpgradeable

Includes variables and a modifier that introduces pausing functionality for certain functions in the contracts. Stake and yield processing/claiming functions can be paused by the eDAO if necessary. Unstaking functions are never paused so users can always remove their unlocked funds.

Initializable

Allows the creation of initializer functions instead of constructors without importing all the code from UUPSUpgradeable. Used by abstract base contracts that don't require all functionality from EIP1822 and ERC1977.

Ownable

Adds an address (the `owner`) with special access to the contracts.

See: <https://docs.openzeppelin.com/contracts/4.x/api/access#Ownable>

Illuvium Libraries

We use a number of internal libraries that our contracts can interact with.

Stake

Stake library used by ILV pool and Sushi LP Pool.

Responsible for managing weight calculation and stores important constants related to stake period, base weight, and multipliers utilized.

Events

N/A

Modifiers

N/A

Initializer

N/A

Functions

`weight(struct Stake.Data _self) uint256 (internal)`

`weightToReward(uint256 _weight, uint256 _rewardPerWeight) uint256 (internal)`

Converts stake weight (not to be mixed with the pool weight) to ILV reward value, applying the 10^{12} division on weight

`rewardPerWeight(uint256 _reward, uint256 _globalWeight) uint256 (internal)`

Converts reward ILV value to stake weight (not to be mixed with the pool weight), applying the 10^{12} multiplication on the reward.

- OR -

Converts reward ILV value to reward/weight if stake weight is supplied as second function parameter instead of reward/weight.

Structs

Data

uint120 value

uint64 lockedFrom

uint64 lockedUntil

bool isYield

Notes

ErrorHandler

Introduces some very common input and state validation for smart contracts, such as non-zero input validation, general boolean expression validation, and access validation.

Throws pre-defined errors instead of string error messages to reduce gas costs.

Since the library handles only very common errors, concrete smart contracts may also introduce their own error types and handling.

Events

N/A

Modifiers

N/A

Initializer

N/A

Functions

```
verifyNonZeroInput(bytes4 fnSelector, uint256 value, uint8 paramIndex) (internal)
```

Verifies if an input is set (non-zero).

```
verifyInput(bytes4 fnSelector, bool expr, uint8 paramIndex) (internal)
```

Verifies if an input is correct.

```
verifyState(bytes4 fnSelector, bool expr, uint256 errorCode) (internal)
```

Verifies if the smart contract state is correct.

```
verifyAccess(bytes4 fnSelector, bool expr) (internal)
```

Verifies an access request to the function.

Notes

Base Contracts

These are the base contracts used by the Staking v2 system.

FactoryControlled

Abstract smart contract responsible to hold IFactory factory address. Stores PoolFactory address on initialization.

Events

N/A

Modifiers

N/A

Initializer

N/A

Functions

`__FactoryControlled_init(address _factory) (internal)`

Attaches PoolFactory address to the FactoryControlled contract.

`_requireIsFactoryController() (internal)`

checks if caller is factory admin (eDAO multisig address).

VaultRecipient

Inherited by core pools. Stores vault variables and includes a modifier to protect functions that can only called by the vault contract.

Events

`LogSetVault(address by, address previousVault, address newVault)`

Fired in `setVault()`.

Modifiers

N/A

Initializer

N/A

Functions

`setVault(address _vault) (external)`

Executed only by the factory owner to Set the vault.

`_requireIsVault() (internal)`

Utility function to check if caller is the Vault contract

Notes

Timestamp

Utility contract that delegates `block.timestamp` readings to an internal function, which allows easier mocking in unit tests.

Events

N/A

Modifiers

N/A

Initializer

N/A

Functions

`_now256`

Returns the current block timestamp

Notes

CorePool

An abstract contract containing common logic for ILV and ILV/ETH SLP pools.

Base smart contract for ILV and LP pool. Stores each pool user by mapping its address to the user struct. User struct stores v2 stakes, which fit in 1 storage slot each (by using the Stake lib), total weights, pending yield and revenue distributions, and v1 stake ids. ILV and LP stakes can be made through flexible stake mode, which only increments the flexible balance of a given user, or through locked staking. Locked staking creates a new Stake element fitting 1 storage slot with its value and lock duration. When calculating pending rewards, CorePool checks v1 locked stakes weights to increment in the calculations and stores pending yield and pending revenue distributions. Every time a stake or unstake related function is called, it updates pending values, but doesn't require instant claims. Rewards claiming is executed in separate functions, and in the case of yield, it also requires the user checking whether ILV or sILV is wanted as the yield reward.

Deployment and initialization

After proxy is deployed and attached to the implementation, it should be registered by the PoolFactory contract. Additionally, 3 token instance addresses must be defined on deployment:

- ILV token address
- sILV token address, used to mint sILV rewards
- pool token address, it can be ILV token address, ILV/ETH pair address, and others

Pool weight defines the fraction of the yield current pool receives among the other pools. Pool Factory is responsible for the weight synchronization between the pools. The weight is logically 20% for ILV pool and 80% for ILV/ETH pool initially (when there are no Flash Pools active). It can be changed through ICCPs and new flash pools added in the protocol. Since Solidity doesn't support fractions the weight is defined by the division of pool weight by total pools weight (sum of all registered pools within the factory)

For ILV Pool we use 200 as weight and for ILV/ETH SLP pool - 800.

Events

```
LogStake(address by, address from, uint256 stakeId, uint256 value, uint64 lockUntil)
```

Fired in `_stake()` and `stakeAsPool()` in ILVPool contract.

```
LogUpdateStakeLock(address from, uint256 stakeId, uint64 lockedFrom, uint64 lockedUntil)
```

Fired in `updateStakeLock()`.

```
LogUnstakeLocked(address to, uint256 stakeId, uint256 value, bool isYield)
```

Fired in `unstakeLocked()`.

```
LogUnstakeLockedMultiple(address to, uint256 totalValue, bool unstakingYield)
```

Fired in `unstakeLockedMultiple()`.

```
LogSync(address by, uint256 yieldRewardsPerWeight, uint64 lastYieldDistribution)
```

Fired in `_sync()`, `sync()` and dependent functions (stake, unstake, etc.).

```
LogClaimYieldRewards(address by, address from, bool sILV, uint256 stakeId, uint256 value)
```

Fired in `_claimYieldRewards()`.

```
LogClaimVaultRewards(address by, address from, uint256 value)
```

Fired in `_claimVaultRewards()`.

```
LogProcessRewards(address by, address from, uint256 yieldValue, uint256 revDisValue)
```

Fired in `_processRewards()`.

```
LogMigrateUser(address from, address to)
```

fired in `migrateUser()`.

```
LogReceiveVaultRewards(address by, uint256 value)
```

Fired in `receiveVaultRewards()`.

Modifiers

```
updatePool()
```

Used for functions that require syncing contract state before execution.

Initializer

```
__CorePool_init(address _ilv, address _silv, address _poolToken, address _corePoolV1, address _factory, uint64 _initTime, uint32 _weight) (internal)
```

Functions

```
pendingRewards(address _staker) uint256 pendingYield, uint256 pendingRevDis (external)
```

Calculates current yield rewards value available for address specified.

see `_pendingRewards()` for further details.

`external pendingRewards()` returns `pendingYield` and `pendingRevDis` accumulated with already stored `user.pendingYield` and `user.pendingRevDis`.

```
balanceOf(address _user) uint256 balance (external)
```

Returns total staked token balance for the given address.

loops through stakes and returns total balance.

```
getStake(address _user, uint256 _stakeId) struct Stake.Data (external)
```

Returns information on the given stake for the given address.

See `getStakesLength`.

```
getV1StakeId(address _user, uint256 _position) uint256 (external)
```

Returns a v1 stake id in the `user.v1StakesIds` array.

Get v1 stake id position through `getV1StakePosition()`.

```
getV1StakePosition(address _user, uint256 _desiredId) uint256 position (external)
```

Returns a v1 stake position in the `user.v1StakesIds` array.

helper function to call `getV1StakeId()`.

```
getStakesLength(address _user) uint256 (external)
```

Returns number of stakes for the given address. Allows iteration over stakes.

See `getStake()`.

```
stakePoolToken(uint256 _value, uint64 _lockDuration) (external)
```

Stakes specified value of tokens for the specified value of time, and pays pending yield rewards if any.

Requires value to stake and lock duration to be greater than zero.

```
migrateUser(address _to) (external)
```

Migrates `msg.sender` data to a new address.

V1 stakes are never migrated to the new address. We process all rewards, clean the previous user (`msg.sender`), add the previous user data to the desired address and update `subYieldRewards/subVaultRewards` values in order to make sure both addresses will have rewards cleaned.

```
fillV1StakeId(uint256 _v1StakeId, uint256 _stakeIdPosition) (external)
```

Allows an user that is currently in v1 with locked tokens, that have just been unlocked, to transfer to v2 and keep the same weight that was used in v1.

```
sync() (external)
```

Service function to synchronize pool state with current time. Can be executed by anyone at any time, but has an effect only when at least one second passes between synchronizations.

Executed internally when staking, unstaking, processing rewards in order for calculations to be correct and to reflect state progress of the contract.

When timing conditions are not met (executed too frequently, or after factory end time), function doesn't throw and exits silently.

```
claimYieldRewards(bool _useSILV) (external)
```

Pool state is updated before calling the internal function.

Calls internal `_claimYieldRewards()` passing `msg.sender` as `_staker`.

```
claimVaultRewards() (external)
```

Pool state is updated before calling the internal function.

Calls internal `_claimVaultRewards()` passing `msg.sender` as `_staker`.

```
receiveVaultRewards(uint256 _value) (external)
```

Executed by the vault to transfer vault rewards ILV from the vault into the pool.

This function is executed only for ILV core pools.

```
setWeight(uint32 _weight) (external)
```

Executed by the factory to modify pool weight; the factory is expected to keep track of the total pools weight when updating.

Set weight to zero to disable the pool.

```
_pendingRewards(address _staker, uint256 _totalV1Weight, uint256 _subYieldRewards, uint256 _subVaultRewards)
uint256 pendingYield, uint256 pendingRevDis (internal)
```

Similar to public `pendingYieldRewards`, but performs calculations based on current smart contract state only, not taking into account any additional time which might have passed.

It performs a check on `v1StakesIds` and calls the corresponding V1 core pool in order to add v1 weight into v2 yield calculations.

V1 weight is kept the same used in v1, as a bonus to V1 stakers.

Pending values returned are used by `_processRewards()` calls, which means we don't count `user.pendingYield` and `user.pendingRevDis` here.

```
_stake(address _staker, uint256 _value, uint64 _lockDuration) (internal)
```

Used internally, mostly by children implementations, see `stake()`.

```
unstakeLocked(uint256 _stakeId, uint256 _value) (external)
```

Unstakes a stake that has been previously locked, and is now in an unlocked state. If the stake has the `isYield` flag set to true, then the contract requests ILV to be minted by the PoolFactory. Otherwise it transfers ILV or LP from the contract balance.

```
unstakeLockedMultiple(struct CorePool.UnstakeParameter[] _stakes, bool _unstakingYield) (external)
```

Executes unstake on multiple `stakeIds`. See `unstakeLocked()`.

Optimizes gas by requiring all unstakes to be made either in yield stakes or in non yield stakes. That way we can transfer or mint tokens in one call.

```
_sync() (internal)
```

Used internally, mostly by children implementations, see `sync()`.

Updates smart contract state (`yieldRewardsPerWeight`, `lastYieldDistribution`),

Updates factory state via `updateILVPerSecond`

```
_processRewards(address _staker, uint256 _v1WeightToAdd, uint256 _subYieldRewards, uint256 _subVaultRewards)
uint256 pendingYield, uint256 pendingRevDis (internal)
```

Used internally, mostly by children implementations.

Executed before staking, unstaking and claiming the rewards.

Updates user.pendingYield and user.pendingRevDis.

When timing conditions are not met (executed too frequently, or after factory end block), function doesn't throw and exits silently.

```
_claimYieldRewards(address _staker, bool _useSILV) (internal)
```

sILV is minted straight away to _staker wallet, ILV is created as a new stake and locked for Stake.MAX_STAKE_PERIOD.

Claims all pendingYield from _staker using ILV or sILV.

```
_claimVaultRewards(address _staker) (internal)
```

Claims all pendingRevDis from _staker using ILV.

ILV is sent straight away to _staker address.

```
_useV1Weight(address _staker) uint256 totalV1Weight, uint256 subYieldRewards, uint256 subVaultRewards (internal)
```

If v1 weights have changed since last call, we use latest v1 weight for yield and revenue distribution rewards calculations, and recalculate user sub rewards values in order to have correct rewards estimations.

Calls CorePoolV1 contract, gets v1 stake ids weight and returns.

Used by _pendingRewards() to calculate yield and revenue distribution rewards taking v1 weights into account.

```
_getSubRewardsValue(uint256 _subRewardsStored, uint256 _totalWeightStored, uint256 _totalV1Weight, uint256 _previousTotalV1Weight) uint256 subRewards (internal)
```

Recalculates subYieldRewards or subVaultRewards using most recent _totalV1Weight, by getting previous yieldRewardsPerWeight used in last subYieldRewards or subVaultRewards update (through _previousTotalV1Weight) and returns equivalent value using most recent v1 weight.

This function is very important in order to keep calculations correct even after a user unstakes in v1.

If a user in v1 unstakes before claiming yield in v2, it will be considered as if the user has been accumulating yield and revenue distributions with most recent weight since the last user.subYieldRewards and user.subVaultRewards update. // We should warn the customer through the UI.

v1 stake token amount of a given stakeId can never increase in v1 contracts. This way we are safe from attacks that function by adding more tokens in v1 and gaining a higher accumulation of yield and revenue distributions.

```
_requireNotPaused() (internal)
```

Checks if pool is paused

```
_authorizeUpgrade(address) (internal)
```

See UUPSUpgradeable _authorizeUpgrade().

Just checks if msg.sender == factory.owner() i.e eDAO multisig address.

eDAO multisig is responsible by handling upgrades and executing other admin actions approved by the Council.

Structs

User

uint128 pendingYield

uint128 pendingRevDis

uint248 totalWeight

uint8 v1IdsLength

uint256 subYieldRewards

uint256 subVaultRewards


```
struct Stake.Data[] stakes
mapping(uint256 => uint256) v1StakeIds

UnstakeParameter

uint256 stakeId
uint256 value
```

Notes

V2Migrator

V2Migrator inherits all CorePool base contract functionality, and adds v1 to v2 migration related functions. This is a core smart contract of Sushi LP and ILV pools and manages users locked and yield weights coming from v1. Parameters need to be reviewed carefully before deployment for the migration process. Users will migrate their locked stakes, which are stored in the contract and v1 total yield weights by data stored in a Merkle tree using Merkle proofs.

Events

```
LogMigrateYieldWeight(address from, uint256 yieldWeightMigrated)

logs _migrateYieldWeights()

LogMigrateLockedStakes(address from, uint256 totalV1WeightAdded)

logs _migrateLockedStakes()
```

Modifiers

N/A

Initializer

```
__V2Migrator_init(address _ilv, address _silv, address _poolToken, address _corePoolV1, address _factory, uint64 _initTime, uint32 _weight, uint256 _v1StakeMaxPeriod) (internal)
```

V2Migrator initializer function.

Functions

```
migrateLockedStakes(uint256[] _stakeIds) (external)
```

External migrateLockedStakes call, used in Sushi LP pool.

```
_migrateLockedStakes(uint256[] _stakeIds, uint256 _v1WeightToAdd) (internal)
```

Reads v1 core pool locked stakes data (by looping through the _stakeIds array), checks if it's a valid v1 stake to migrate and save the id to v2 user struct.

Notes

The v2 migrator makes an external call to the corresponding v1 core pool, stores the stake id and the stake weight and executes checks to determine if the user has already been migrated or not.

Yield migration is handled solely by the ILVPool contract.

Deployed Contracts

Illuvium has a number of existing contracts deployed in Staking v1. These contracts are referenced and utilized in many of the Staking v2 contracts.

ILVPool

```
setMerkleRoot(bytes32 _merkleRoot) (external)
```

Sets the yield weight tree root.

hasMigratedYield(uint256 _index) bool (public)

Returns whether a user of a given _index in the bitmap has already migrated v1 yield weight stored in the Merkle tree or not.

stakeAsPool(address _staker, uint256 _value) (external)

Executed by other core pools and flash pools as part of yield rewards processing logic (_claimYieldRewards() function). Executed when _useSILV is false and pool is not an ILV pool - see CorePool._processRewards().

executeMigration(bytes32[] _proof, uint256 _index, uint248 _yieldWeight, uint256[] _stakeIds) (external)

Calls internal _migrateLockedStakes and _migrateYieldWeights functions for a complete migration of a v1 user to v2. See _migrateLockedStakes and _migrateYieldWeights.

claimYieldRewardsMultiple(address[] _pools, bool[] _useSILV) (external)

ILV pool works as a router for claiming multiple pools registered in the factory.

Calls multiple pools claimYieldRewardsFromRouter() in order to claim yield in 1 transaction.

claimVaultRewardsMultiple(address[] _pools) (external)

ILV pool works as a router for claiming multiple pools registered in the factory.

Calls multiple pools claimVaultRewardsFromRouter() in order to claim yield in 1 transaction.

mintV1YieldMultiple(uint256[] _stakeIds) (external)

Aggregates in one single mint call multiple yield stakeIds from v1.

Reads v1 ILV pool to execute checks, if everything is correct, it stores in memory total amount of yield to be minted and calls the PoolFactory to mint it to msg.sender.

Notes

This deploys an instance of the ILV token staking pool.

SushiLPPool

Extends all functionality from V2Migrator contract, there isn't a lot of additions compared to ILV pool. Sushi LP pool basically needs to be able to be called by ILV pool in batch calls where we claim rewards from multiple pools.

Events

N/A

Modifiers

N/A

Initializer

initialize(address _ilv, address _silv, address _poolToken, address _factory, uint64 _initTime, uint32 _weight, address _corePoolV1, uint256 _v1StakeMaxPeriod) (external)

Calls __V2Migrator_init().

Functions

claimYieldRewardsFromRouter(address _staker, bool _useSILV) (external)

This function can be called only by ILV core pool.

Uses ILV pool as a router by receiving the _staker address and executing the internal _claimYieldRewards().

Its usage allows claiming multiple pool contracts in one transaction.

claimVaultRewardsFromRouter(address _staker) (external)

This function can be called only by ILV core pool.

Uses ILV pool as a router by receiving the `_staker` address and executing the internal `_claimVaultRewards()`. Its usage allows claiming multiple pool contracts in one transaction.

`_requirePoolIsValid()` (internal)

Checks if caller is ILVPool.

Notes

This deploys an instance of the Sushi LP token staking pool.

PoolFactory

Registers deployed pools in the protocol. Controls ILV and sILV minting (has the permission to mint from ILV and sILV contracts), emissions schedule, and pool weights.

Pool Factory manages Illuvium staking pools. Provides a single public interface to access the pools. Provides an interface for the pools to mint yield rewards, access pool-related info, update weights, etc.

The factory is authorized (via its owner) to register new pools, change weights of the existing pools, and remove the pools (by changing their weights to zero).

The factory requires `ROLE_TOKEN_CREATOR` permission on the ILV and sILV tokens to mint yield (see `mintYieldTo` function).

Events

`LogRegisterPool(address by, address poolToken, address poolAddress, uint64 weight, bool isFlashPool)`

Fired in `registerPool()`

`LogChangePoolWeight(address by, address poolAddress, uint32 weight)`

Fired in `changePoolWeight()`.

`LogUpdateILVPerSecond(address by, uint256 newIlvPerSecond)`

Fired in `updateILVPerSecond()`.

`LogSetEndTime(address by, uint32 endTime)`

Fired in `setEndTime()`.

Modifiers

N/A

Initializer

`initialize(address _ilv, address _silv, uint192 _ilvPerSecond, uint32 _secondsPerUpdate, uint32 _initTime, uint32 _endTime)` (external)

Initializes a factory instance

Functions

`getPoolAddress(address poolToken) address` (external)

Given a pool token retrieves corresponding pool address.

A shortcut for `pools` mapping.

`getPoolData(address _poolToken) struct PoolFactory.PoolData` (public)

Reads pool information for the pool defined by its pool token address.

Designed to simplify integration with the front ends.

`shouldUpdateRatio() bool` (public)

Verifies if `secondsPerUpdate` has passed since last ILV/second ratio update and if ILV/second reward can be decreased by 3%.

```
registerPool(address pool) (public)
```

Registers an already deployed pool instance within the factory.

Can be executed by the pool factory owner only.

```
updateILVPerSecond() (external)
```

Decreases ILV/second reward by 3%, can be executed no more than once per `secondsPerUpdate` seconds.

```
mintYieldTo(address _to, uint256 _value, bool _useSILV) (external)
```

Mints ILV tokens; executed by ILV Pool only.

Requires factory to have `ROLE_TOKEN_CREATOR` permission on the ILV ERC20 token instance.

```
changePoolWeight(address pool, uint32 weight) (external)
```

Changes the weight of the pool.

Executed by the pool itself or by the factory owner.

```
setEndTime(uint32 _endTime) (external)
```

Updates yield generation ending timestamp.

Structs

PoolData

address poolToken

address poolAddress

uint32 weight

bool isFlashPool

Notes

This deploys an instance of the Pool Factory v2 contract.

FlashPool

A Flash Pool contract is a temporary pool with an arbitrary ERC20 token from a new Illuvium DAO partner voted by the council. Holders of this ERC20 token (which is stored at `poolToken`) are able to stake it and receive ILV yield rewards which can be claimed and vested in the ILV pool. Operations in Flash Pools are cheaper compared to Core Pools, since we don't lock tokens and we don't need to deal with mappings and arrays as much as we do in the ILV and Sushi LP pools.

Events

```
LogStake(address from, uint256 value)
```

Fired in `stake()`.

```
LogUnstake(address to, uint256 value)
```

Fired in `unstake()`.

```
LogSync(address by, uint256 yieldRewardsPerToken, uint64 lastYieldDistribution)
```

Fired in `_sync()`, `sync()` and dependent functions (`stake`, `unstake`, etc.).

```
LogClaimYieldRewards(address from, bool sILV, uint256 value)
```

Fired in `_claimYieldRewards()`.

```
LogProcessRewards(address from, uint256 value)
```

Fired in `_processRewards()`.

```
LogSetWeight(address by, uint32 fromVal, uint32 toVal)
```

Fired in `setWeight()`.

```
LogMigrateUser(address from, address to)
```

Fired in `migrateUser()`.

Modifiers

```
updatePool()
```

Used for functions that require syncing contract state before execution.

Initializer

```
initialize(address _ilv, address _silv, address _poolToken, address _factory, uint64 _initTime, uint64 _endTime, uint32 _weight) (external)
```

Initializes a new flash pool.

Functions

```
initialize(address _ilv, address _silv, address _poolToken, address _factory, uint64 _initTime, uint64 _endTime, uint32 _weight) (external)
```

Initializes a new flash pool.

```
pendingYieldRewards(address _staker) uint256 pending (external)
```

Calculates current yield rewards value available for address specified.

see `_pendingYieldRewards()` for further details.

```
balanceOf(address _user) uint256 balance (external)
```

Returns total staked token balance for the given address.

```
isPoolDisabled() bool (public)
```

Checks if flash pool has ended. Flash pool is considered "disabled" once time reaches its "end time".

```
stake(uint256 _value) (external)
```

Stakes `poolTokens` without lock.

```
migrateUser(address _to) (external)
```

Data is copied to memory so we can delete previous address data before we store it in new address.

Migrates `msg.sender` data to a new address.

```
sync() (external)
```

Service function to synchronize pool state with current time.

Can be executed by anyone at any time, but has an effect only when at least one second passes between synchronizations.

Executed internally when staking, unstaking, processing rewards in order for calculations to be correct and to reflect state progress of the contract.

When timing conditions are not met (executed too frequently, or after factory end time), the function doesn't throw and exits silently.

```
claimYieldRewards(bool _useSILV) (external)
```

Pool state is updated before calling the internal function.

Calls internal `_claimYieldRewards()` passing `msg.sender` as `_staker`.

```
claimYieldRewardsFromRouter(address _staker, bool _useSILV) (external)
```

This function can be called only by ILV core pool.

Uses ILV pool as a router by receiving the `_staker` address and executing the internal `_claimYieldRewards()`.

Its usage allows claiming multiple pool contracts in one transaction.

```
setWeight(uint32 _weight) (external)
```

Executed by the factory to modify pool weight; the factory is expected to keep track of the total pools weight when updating.

Set weight to zero if you wish to disable the pool.

```
setEndTime(uint64 _newEndTime) (external)
```

Updates flash pool ending timestamp.

```
_pendingYieldRewards(address _staker) uint256 pending (internal)
```

Similar to public `pendingYieldRewards`, but performs calculations based on current smart contract state only, not taking into account any additional time which might have passed.

```
unstake(uint256 _value) (external)
```

```
_sync() (internal)
```

Used internally, mostly by children implementations, see `sync()`.

Updates smart contract state (`yieldRewardsPerToken`, `lastYieldDistribution`), updates factory state via `updateILVPerSecond`.

```
_processRewards(address _staker) uint256 pendingYield (internal)
```

Used internally, mostly by children implementations.

Executed before staking, unstaking and claiming the rewards.

When timing conditions are not met (executed too frequently, or after factory end time), function doesn't throw and exits silently.

```
_claimYieldRewards(address _staker, bool _useSILV) (internal)
```

sILV is minted straight away to `_staker` wallet, ILV is created as a new stake and locked for `Stake.MAX_STAKE_PERIOD`.

Claims all `pendingYield` from `_staker` using ILV or sILV.

```
_tokensToReward(uint256 _value, uint256 __rewardPerToken) uint256 (internal)
```

Converts number of tokens staked to ILV reward value, applying the 10^{12} division on number of tokens (`_value`).

```
_rewardPerToken(uint256 _reward, uint256 _totalStaked) uint256 (internal)
```

Converts reward ILV value to reward/tokens

```
_authorizeUpgrade(address) (internal)
```

Function that should revert when `msg.sender` is not authorized to upgrade the contract.

Called by `{upgradeTo}` and `{upgradeToAndCall}`.

Normally this function will use an `xref:access.adoc[access control]` modifier such as `{Ownable-onlyOwner}`.

```
function _authorizeUpgrade(address) internal override onlyOwner {}
```

Notes

This deploys an instance of an ERC20 partner token temporary pool.

Vault

The Vault is responsible for gathering revenue from the protocol, swapping to ILV periodically, and distributing to core pool users from time to time.

The contract connects with Sushi's router in order to buy ILV from the ILV/ETH liquidity pool. Since we can change the vault address in the staking pools (see `VaultRecipient`), the Vault contract doesn't need to implement upgradeability.

It receives ETH from the `receive()` function and allows conversion to ILV by the address with the role `ROLE_VAULT_MANAGER` (0x0001_0000). This conversion can be done in multiple steps, which means it doesn't require converting all ETH balance in 1 function call. The vault is also responsible for calling `receiveVaultRewards()` function in the core pools, which takes care of calculations of how much ILV should be sent to each pool as revenue distribution.

Events

`LogSwapEthForILV(address by, uint256 ethSpent, uint256 ilvReceived)`

Fired in `_swapEthForIlv()` and `sendIlvRewards()` (via `swapEthForIlv`).

`LogSendILVRewards(address by, uint256 value)`

Fired in `sendIlvRewards()`.

`LogEthReceived(address by, uint256 value)`

Fired in default payable `receive()`.

`LogSetCorePools(address by, address ilvPoolV1, address pairPoolV1, address ilvPool, address pairPool, address lockedPoolV1, address lockedPoolV2)`

Fired in `setCorePools()`.

Modifiers

N/A

Initializer

`constructor(address _sushiRouter, address _ilv) (public)`

Creates (deploys) Vault linked to Sushi AMM Router and IlluviumERC20 token

Functions

`setCorePools(contract ICorePoolV1 _ilvPoolV1, contract ICorePoolV1 _pairPoolV1, contract ICorePool _ilvPool, contract ICorePool _pairPool, contract ICorePool _lockedPoolV1, contract ICorePool _lockedPoolV2) (external)`

Auxiliary function used as part of the contract setup process to setup core pools.

Executed by `owner()` after deployment.

`swapETHForILV(uint256 _ethIn, uint256 _ilvOut, uint256 _deadline) (external)`

Exchanges ETH balance present on the contract into ILV via Uniswap.

Logs operation via `EthIlvSwapped` event.

`sendILVRewards(uint256 _ethIn, uint256 _ilvOut, uint256 _deadline) (external)`

Converts an entire contract's ETH balance into ILV via Uniswap and sends the entire contract's ILV balance to the Illuvium Yield Pool.

Uses `swapEthForIlv` internally to exchange ETH -> ILV.

Logs operation via `RewardsDistributed` event.

Set `ilvOut` or `deadline` to zero to skip `swapEthForIlv` call.

`estimatePairPoolReserve(address _pairPool) uint256 ilvAmount (public)`

Auxiliary function used to estimate LP core pool share among 2 other core pools.

Since LP pool holds ILV in both paired and unpaired forms, this creates some complexity to properly estimate LP pool share among 2 other pools which contain ILV tokens only.

The function counts for ILV held in LP pool in unpaired form as is, for the paired ILV it estimates its amount based on the LP token share the pool has.

`receive()` (external)

Default payable function, allows to top up contract's ETH balance to be exchanged into ILV via Uniswap.

Logs operation via `LogEthReceived` event.

Structs

Pools

contract ICorePoolV1 ilvPoolV1

contract ICorePoolV1 pairPoolV1

contract ICorePool ilvPool

contract ICorePool pairPool

contract ICorePool lockedPoolV1

contract ICorePool lockedPoolV2

Notes

This deploys an instance of the Vault contract.

References

Glossary

Equations

Stake weight

$$weight = \left\{ \frac{[(lockedUntil - lockedFrom) * weightMultiplier]}{maxStakePeriod} + baseWeight \right\} * stakeValue$$

Weight to reward (yield or revenue distribution)

$$reward = \frac{weight * rewardPerWeight}{rewardPerWeightMultiplier}$$

Reward per weight

$$rewardPerWeight = \frac{ilvReward * rewardPerWeightMultiplier}{globalWeight}$$

