Introduction

Akropolis' mission is to give everyone the tools to save, grow and provide for the future safely and without dependence on a central counterparty, or to fall prey to predatory financial practices of multiple intermediaries.

Our goal is to create yield-generating products that don't predominantly rely on inflationary emissions as the main source of yield and that generate returns regardless of the market conditions.

Our current products include:

- **Vortex** - An on-chain basis trading strategy that aims to generate long-term, sustainable and rewarding yields while remaining market-neutral.
- **yVaults** - Streamlined and simplified access to select Yearn vaults to optimize yields.

We believe Akropolis is the one-stop solution for any investor seeking access to passive, efficient and sustainable yield generation.

Roadmap

You can find the Akropolis roadmap here:

Notion – The all-in-one workspace for your notes, tasks, wikis, and databases.
Notion

Community channels

Telegram

Discord

Twitter

Medium

Governance forum

Using Akropolis

A step-by-step guide

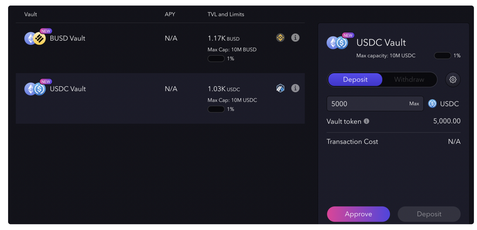
Connect your wallet

First, **Connect your wallet** using the button at the top right corner.

We support multiple types of wallets (the most popular choice is MetaMask). Make sure that your wallet is connected to the right network (Ethereum mainnet).

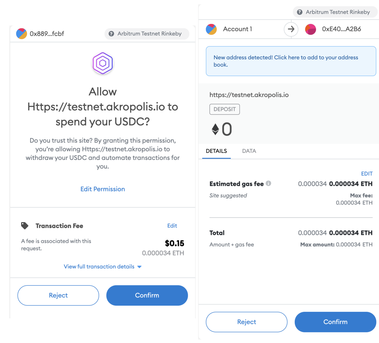
Deposit

- Select the vault that you would like to deposit into.
- Enter the amount of tokens you want to deposit into the vault.



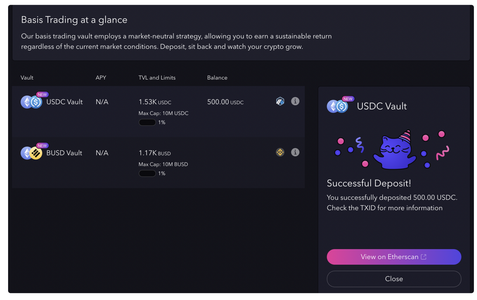
Make sure you have enough ETH/BNB/native network token to pay for transaction fees.

- Click on 'Approve' or 'Deposit' button. *Depending on previous approvals given, you might need to confirm 2 transactions (approval for spending & deposit).*
- Confirm the transaction in your wallet (you should see a popup). Transaction confirmation time depends on the gas fee & network congestion (usually it's as fast as 1-3 minutes).



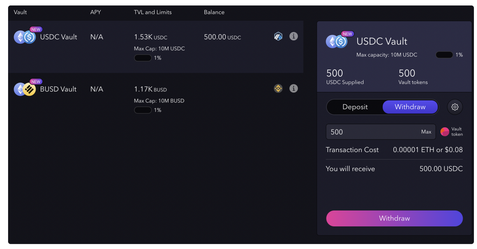
You can speed it up by paying a higher gas fee to the network. If your transaction gets stuck, see this guide on speeding up or cancelling the transaction.

- When your transaction succeeds, you will see your deposited balance in the interface.



Withdraw

- Select the vault that you would like to withdraw from.
- Enter how much you want to withdraw, or click 'Max" to withdraw the entire balance.



- Click 'Withdraw'.
- You will see a confirmation popup from your wallet. Click 'Confirm'.
- You will receive tokens in your wallet when your transaction succeeds.

What do I do if I don't have funds on the Vault's network?

This is quite common as sending funds across chains can be a stressful experience both in terms of security and user experience.

Bridging assets cross-chain is important, however, as you cannot initiate transactions without the chain's native asset to pay for gas fees.

Arbitrum

To transfer ETH over to Arbitrum, you can use this bridge:

Arbitrum Bridge

Here are some guides on using Arbitrum and transferring funds from L1 (Ethereum) to L2 (Arbitrum):

The Essential Guide to Arbitrum

Guide to Arbitrum and setting up Metamask for Arbitrum
Medium

Binance Smart chain

Binance Smart chain uses BNB (BEP-20 format) as its native asset for transaction fees.

To transfer assets cross-chain, you can use BSC bridge:

Binance DEX | Binance Smart Chain
Binance | Binance Smart Chain | Binance Bridge | Binance Swap | Binance.or

Here are some more resources on BSC and how to use it:

Binance | Binance Smart Chain | Binance Staking | Binance Swap | Binance.org
Binance | Binance Smart Chain | Binance Staking | Binance Swap | Binance.org

How to Get Started with Binance Smart Chain (BSC) | Binance Academy
Binance Academy

Overview

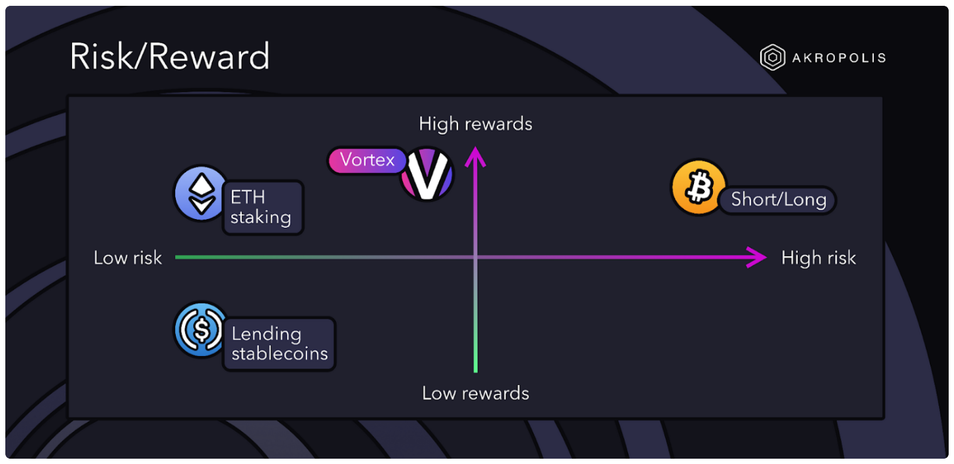
What is Vortex?

Vortex (v1) is an on-chain basis trading strategy that aims to generate long-term, sustainable and rewarding yields while remaining market-neutral.

Advantages of using Vortex

Advantages of using Vortex (v1) include:

- **Market Neutrality** - Vortex allows users to generate yield without being exposed to directional price risk. Regardless of whether it's a bull, bear or crab season, Vortex should generate sustainable yields.
- **Rewarding Yields** - Vortex's underlying strategy has proven to be profitable across all market conditions and has historically outperformed many other market-neutral strategies and higher-risk yield farms.
- **Single Asset** - Vortex only requires users to deposit a single asset - USDC. This makes Vortex an effective alternative to lending or farming with stablecoins.
- **Low Maintenance** - Vortex is a passive strategy for our users, but actively managed by our strategists for maintenance and risk management. The returns generated by Vortex are also periodically compounded, further enhancing yield.
- **Ecosystem Benefits** - Vortex provides liquidity that is crucial for decentralized derivative exchanges that offer perpetual contracts to function.



Risks of using Vortex

Risks involved with using Vortex (v1) include:

- **Smart Contract Risks** - Vortex is a smart contract that will connect to multiple external smart contracts, each with their own risks.
- **Negative Returns** - The use of Vortex may result in negative returns. This could be because of a range of unlikely events, including:
  - The funding rate on perpetual exchanges being consistently against Vortex
  - Vortex's position(s) being liquidated
  - Excessive slippage caused by a need to exit all Vortex positions quickly
- **Centralization** - The initial launch of Vortex will have elements of centralization, including control by a predetermined strategist who has the ability to open and close strategy positions.
- **Temporary Suspensions** - We may suspend deposits and/or withdrawals at any time for Vortex. We will only suspend when necessary and will notify via our social channels at the start and end of each suspension, providing reasons and time estimates where possible. Unless we say otherwise, funds will be safe.
- **Limits** - Deposit limits will be enforced for Vortex. These will scale quickly over time, but when the displayed limit is reached, further deposits will not be possible until the limit is increased.

> *Vortex has procedures in place to help mitigate these risks; see Risk Management*

Products - Previous
Vortex

Next
How it works

Last modified 1yr ago

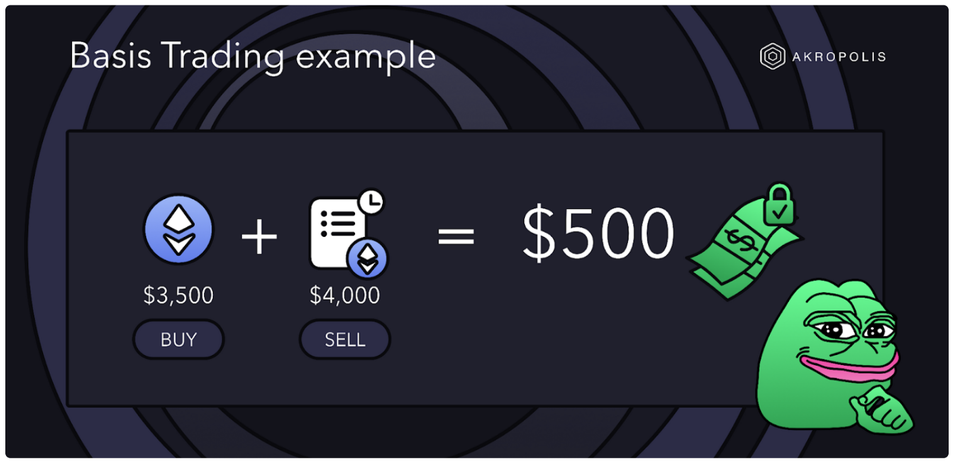
💻 How it works

High-level example

Here's a high-level example of how Vortex works in favorable conditions:

> ⓘ *Assuming 1 ETH = 3500 USDC*

1. User deposits 7000 USDC into Vortex, receiving a proportionate share of the pool as Vault Tokens.
2. Vortex's underlying strategy will then:
    1. Send 3500 USDC to a decentralized exchange to buy 1 'physical' ETH;
    2. Send 3500 USDC to a decentralized derivatives exchange and use it as collateral to short 1 ETH worth of *Perpetual Contracts*.
3. Vortex will automatically collect the *Funding Rate* and periodically compound and rebalance into both positions, increasing the value of the Vault Tokens.



> ✓ Vortex utilizes a **Basis Trading** strategy.

How Vortex applies Basis Trading

The crypto markets have historically been weighted towards longs as the majority of participants speculate that prices will go up.

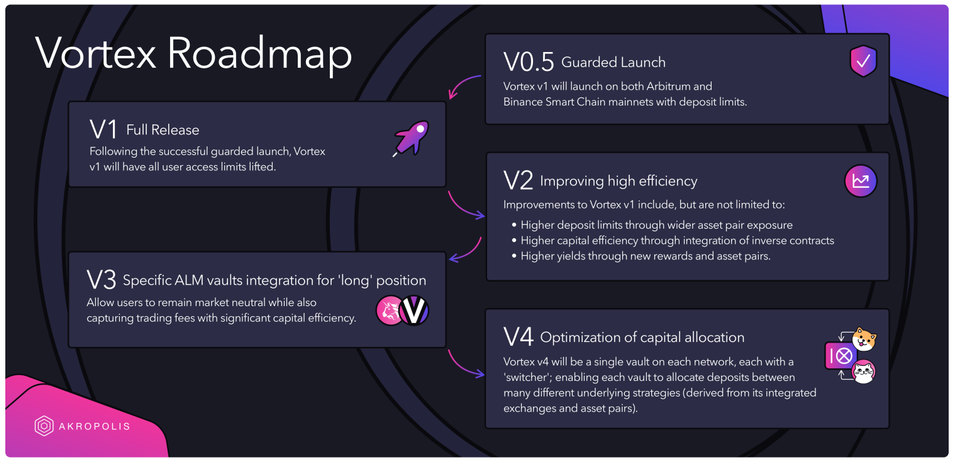
This trend has continued on decentralized derivatives exchanges that offer Perpetual Contracts, which means that Funding Rates have historically, on average, been positive. As a result, from a Funding Rate perspective, it has been profitable to open short positions - but then you may lose a lot more than your Funding Rate returns if prices suddenly moon.

Vortex fixes this by removing the directional price risk from the short position while maintaining the Funding Rate advantage, enabling users to generate market-neutral yields.

To learn more about Basis Trading and Perpetual contracts, please visit:


Basis Trading and Perpetual Contracts

Roadmap



Vortex v1 on Mainnet(s) - Guarded Launch

Vortex v1 will launch on both Arbitrum and Binance Smart Chain mainnets in a decentralized way. The launch of Vortex v1 will be determined by the results of the governance proposal and will be subject to additional restrictions as part of the guarded launch.

Vortex v1 on Mainnet(s)

Full Release Following the successful guarded launch, Vortex v1 will have all user access limits lifted. Note that deposit limits will still be in place due to the product's design.

Vortex v2

Improvements to Vortex v1 include, but are not limited to:

- Higher deposit limits through wider asset pair exposure
- Higher capital efficiency through integration of inverse contracts
- Higher yields through new rewards and asset pairs.

Vortex Multi-Chain

Vortex will expand beyond Arbitrum and BSC and will be released on all major networks. The expansion priority will be based on the deployment, liquidity and activity of the perpetual exchanges of each network.

Vortex v3

Vortex v3 will integrate specific ALM vaults for the "long" position, allowing users to remain market neutral while also capturing trading fees with significant capital efficiency. This will be applicable to all chains where both Vortex and Uniswap v3 are live.

Vortex v4

Vortex v4 will be a single vault on each network, each with a "switcher". This switcher will enable each vault to allocate deposits between many different underlying strategies (derived from its integrated exchanges and asset pairs). This format will simplify the user experience while also enabling the optimization of capital allocation and yield and the maximization of limits.

✍️ Competitor Comparison

Text	Vortex	Lemma	UXD
Format(s)	Vault Token	Vault Token Stablecoin	Stablecoin
Network(s)	Arbitrum Binance Smart Chain	Arbitrum	Solana
Exchange(s)	MCDEX	MCDEX	Mango Market
Performance Fees	25%	30%	-
Action during negative Funding Rate	Stablecoin Farming	Treasury Backstop	Insurance Fun
Resulting Risks	Slow position unwinding	Treasury depletion Slow position unwinding Bank run on leveraged positions	Insurance Fun depletion Governance Tok auction

AKRO staking

Currently, stake tab has only one pool - AKRO staking. It's a simple staking pool with easy mechanics - by sending AKRO to this pool you can earn AKRO itself. There is no lock-ups, you can withdraw any time.

How to stake?

- Open Akropolis app & connect your wallet (*don't forget to check that you have AKRO on it!*)
- Go to staking page
- Enter the amount of tokens you want to deposit & or click max to deposit all AKRO you have on your wallet.
- Click on 'Approve' or 'Deposit' button. *Depending on previous approvals given, you might need to confirm 2 transactions (approval for spending & deposit).*
- Confirm the transaction in your wallet (you should see a popup). Transaction confirmation time depends on the gas fee & network congestion (usually it's as fast as 1-3 minutes).
- When your transaction succeeds, you will see your deposited balance in the interface.

> Make sure you have enough ETH to pay for transaction fees.

> You can speed it up by paying a higher gas fee to the network. If your transaction gets stuck, see this guide on speeding up or cancelling the transaction.

> AKRO rewards are under vesting & are not auto compounded. To learn more, head here.

AKRO token

Token Details

Contract address - 0x8ab7404063ec4dbcfd4598215992dc3f8ec853d7

Symbol - AKRO

Decimals - 18

Total Token Supply - 5,000,000,000

AKRO Tokenomics

> **Tokenomics are evolving and subject to change**

Token Utility

AKRO is a governance token that grants holders the right to raise and participate in decisions that affect Akropolis. This protocol-level governance is tied to managing the suite of DeFi products built on Akropolis.

AKRO stakers also receive returns from protocol revenue and additional AKRO emissions.

Governance process

High-Level Summary

- Initial Governance will be done through the Akropolis Snapshot.page;
- Active AKRO stakers will be able to post proposals and vote on them.

What can I signal my vote on?

- Product features and integrations
- Product fees
- Strategy proposals

How do I vote?

- Go to the Akropolis space on Snapshot;
- Click on “Connect wallet” button in top right corner;
- Connect to the wallet which you use for staking of AKRO ;
- Click on the option you want to vote for;
- Sign the message via your wallet and done.
- Voting will be open for 3 days or 72 hours, so if it opens on Wed 1300CET, it will close on Friday 1300CET (approximately, as closing time is set by the block number).

Governance Workflow:

- All proposals should first be discussed over the governance forum and in Discord. That would give all governance participants an ability to understand all pros & cons before voting.
- The team will assign time values (i.e. urgent vs non-urgent, pending, etc) to titles based on our discretion.
- For feature requests please name your proposal AFR+n (i.e. third proposal is named AFR-003).
- For governance proposals AIP+n (i.e. third proposal is named AIP-003).
- After the initial discussion is completed & community (including team) agree that it should be officially voted on — team or community members can post it on Snapshot page for voting. And as it’s off-chain voting, it does not require any gas fees.
- After voting is completed & proposal reaches 10% quorum, we will add it to our development pipeline.

> If quorum is not reached, the proposal might be resubmitted for voting after 7 days cooldown.

Who can create or vote on proposals?

- Only AKRO stakers can create or vote for proposals. This way we ensure that only active users have a say in Akropolis governance.
- 1 vote equals 1 AKRO staked.
- There is a minimum of 25M AKRO to create a proposal on Snapshot.
- There is no minimum for voting.

Please check Snapshot docs for more information about creating proposals & voting on proposals.

High-Level View

Vortex actively maintains three core positions:

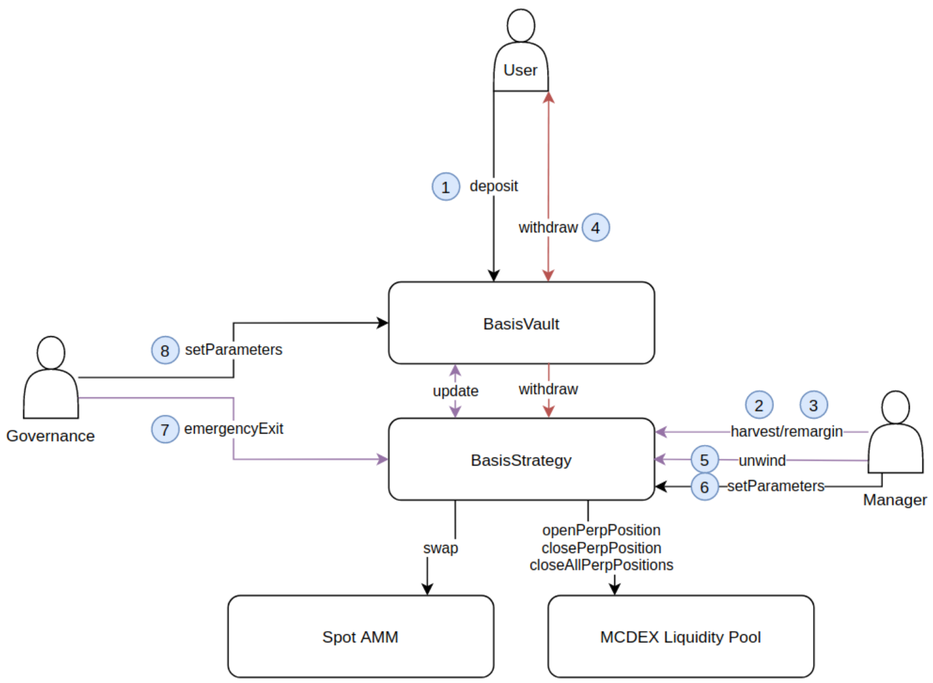
1. **ETH Holdings -** The *long* - ETH Holdings are sourced from a decentralized exchange and held idle in an address.
2. **ETH Short Contracts -** The *short* - ETH Short Contracts are collateralized in USDC and collect the Funding Rate from the decentralized derivatives exchange.
3. **Position Buffer -** The *safety* - The Position Buffer is sourced from user deposits and held idle to ensure the ETH Short Contracts are not liquidated.

> *For more information, see Position Buffer*

> Vortex v1 utilizes **MCDEX** as the decentralized derivatives exchange and has been developed to accommodate their unique **AMM Design**.

🍭 Contract Design

Overview of the vault and strategy implementation



1. **deposit** - A user deposits stablecoins into the vault and receives back tokenised shares of the vault representing their share of the vault.
2. **harvest** - *Only a Manager may call this function*. This is the function that puts funds to work, determines profit/loss of the strategy and distributes funds to the buffer position, short position and long position. It also handles vault updates and protocol fee collection.
3. **remargin** - *Only a Manager may call this function*. It first completes a *harvest*. Then runs some calculations to determine what assets must be swapped in order to return the buffer position to the correct amount and to return the leverage of the margin account to 1. This is a risk management function and keeps the strategy safe from liquidation risk.
4. **withdraw** - A user deposits their shares of the vault, which are then burnt and the user is returned stablecoins according to the value of the shares of the vault at the point of withdrawal. Withdrawals involve selling off strategy positions so that withdrawals from the vault can happen freely.
5. **unwind** - *Only a Manager may call this function*. It first completes a *harvest*. Then it will convert all longPositions to the stable asset, close all short positions and withdraw all funds from the margin account back to the strategy. This is a risk management function and keeps the strategy safe from prolonged negative funding rates and liquidation risk.
6. **setParameters (Manager)** - *Only a Manager may call these functions*. These functions set various parameters in the strategy.
7. **emergencyExit** - *Only a Manager may call this function*. It first completes an *unwind*. Then all funds are transferred to the governance multisig. This is a risk management function and protects funds from extreme conditions such as an exploit risk.
8. **setParameters(Governance)** - *Only Governance may call these functions*. These functions set various parameters in the vault and strategy.

🌾 Key Function - harvest

In-depth descriptions of key functions

harvest

```
    /**
     * @notice  harvest the strategy. This involves accruing profits from the strategy a
     *          user funds to the strategy. The funds are split into their constituents
     *          to their appropriate location.
     *          For the shortPosition a perpetual position is opened, for the long posi
     *          to the long asset. For the buffer position the funds are deposited to t
     * @dev     only callable by the owner
     */
    function harvest() public onlyOwner {
        uint256 shortPosition;
        uint256 longPosition;
        uint256 bufferPosition;
        isUnwind = false;

        mcLiquidityPool.forceToSyncState();
        // determine the profit since the last harvest and remove profits from the marg
        // account to be redistributed
        uint256 amount;
        bool loss;
        if (positions.unitAccumulativeFunding != 0) {
            (amount, loss) = _determineFee();
        }
        // update the vault with profits/losses accrued and receive deposits
        uint256 newFunds = vault.update(amount, loss);
        // combine the funds and check that they are larger than 0
        uint256 toActivate = IERC20(want).balanceOf(address(this));

        if (toActivate > 0) {
            // determine the split of the funds and trade for the spot position of long
            (shortPosition, longPosition, bufferPosition) = _calculateSplit(
                toActivate
            );
            // deposit the bufferPosition to the margin account
            _depositToMarginAccount(bufferPosition);
            // open a short perpetual position and store the number of perp contracts
            positions.perpContracts += _openPerpPosition(shortPosition, true);
        }
        // record incremented positions
        positions.margin = getMargin();
        positions.unitAccumulativeFunding = getUnitAccumulativeFunding();
        emit Harvest(
            positions.perpContracts,
            IERC20(long).balanceOf(address(this)),
            positions.margin
        );
    }
```

The harvest function is the strategy's "work" function. It is responsible for running the strategy.

```
    /**
     * @notice  determine the funding premiums that have been collected since the last e
     * @return  fee  the funding rate premium collected since the last epoch
     * @return  loss whether the funding rate was a loss or not
     */
    function _determineFee() internal returns (uint256 fee, bool loss) {
        int256 feeInt;

        // get the cash held in the margin cash, funding rates are saved as cash in the
        int256 newAccFunding = getUnitAccumulativeFunding();
        int256 prevAccFunding = positions.unitAccumulativeFunding;
        int256 livePositions = getMarginPositions();
        if (prevAccFunding >= newAccFunding) {
            // if the margin cash held has gone down then record a loss
            loss = true;
            feeInt = ((prevAccFunding - newAccFunding) * -livePositions) / 1e18;
            fee = uint256(feeInt / DECIMAL_SHIFT);
        } else {
            // if the margin cash held has gone up then record a profit and withdraw the
            feeInt = ((newAccFunding - prevAccFunding) * -livePositions) / 1e18;
            uint256 balanceBefore = IERC20(want).balanceOf(address(this));
            if (feeInt > 0) {
                mcLiquidityPool.withdraw(perpetualIndex, address(this), feeInt);
            }
            fee = IERC20(want).balanceOf(address(this)) - balanceBefore;
        }
    }
```

The harvest will begin by determining the profit/loss since the previous harvest. It will do this by getting the *unitAccumulativeFunding* and getting the difference from the last harvest's *unitAccumulativeFunding*. If the difference is positive then the profits will be withdrawn from the margin account. If the difference is negative then the loss will be determined.

```
    /**
     * @notice function to update the state of the strategy in the vault and pull any fu
     * @param  _amount change in the vault amount sent by the strategy
     * @param  _loss   whether the change is negative or not
     *                 be the sender
     * @return toDeposit the amount to be deposited in to the strategy on this update
     */
    function update(uint256 _amount, bool _loss)
        external
        onlyStrategy
        returns (uint256 toDeposit)
    {
        // if a loss was recorded then decrease the totalLent by the amount, otherwise i
        if (_loss) {
            totalLent -= _amount;
        } else {
            _determineProtocolFees(_amount);
            totalLent += _amount;
        }
        // increase the totallent by the amount of deposits that havent yet been sent to
        toDeposit = want.balanceOf(address(this));
        totalLent += toDeposit;
        lastUpdate = block.timestamp;
        emit StrategyUpdate(_amount, _loss, toDeposit);
        if (toDeposit > 0) {
            want.approve(strategy, toDeposit);
            want.safeTransfer(msg.sender, toDeposit);
        }
    }
```

Next, the harvest will update the vault of the profit/loss. The vault will update its *totalLent* which is the funds it has lent to the vault; if there is a profit then *totalLent* will increase and the protocol fees will be determined. If there is a loss then *totalLent* will decrease. Finally, funds that have been deposited after the previous harvest will be recorded and transferred to the strategy contract.

```
    /**
     * @notice  split an amount of assets into three:
     *          the short position which represents the short perpetual position
     *          the long position which represents the long spot position
     *          the buffer position which represents the funds to be left idle in the ma
     * @param   _amount the amount to be split in want
     * @return  shortPosition  the size of the short perpetual position in want
     * @return  longPosition   the size of the long spot position in long
     * @return  bufferPosition the size of the buffer position in want
     */
    function _calculateSplit(uint256 _amount)
        internal
        returns (
            uint256 shortPosition,
            uint256 longPosition,
            uint256 bufferPosition
        )
    {
        require(_amount > 0, "_calculateSplit: _amount is 0");
        // remove the buffer from the amount
        bufferPosition = (_amount * buffer) / MAX_BPS;
        // decrement the amount by buffer position
        _amount -= bufferPosition;
        // determine the longPosition in want then convert it to long
        uint256 longPositionWant = _amount / 2;
        longPosition = _swap(longPositionWant, want, long);
        // determine the short position
        shortPosition = _amount - longPositionWant;
    }
```

The harvest will then calculate the split of the gains (deposits and/or profits) and split them first into the buffer position, then the long position, then the short position. The long position is swapped from the deposit asset to the long asset using a Uniswap v2 or Uniswap v3 interfaced AMM.

```
    /**
     * @notice  deposit to the margin account without opening a perpetual position
     * @param   _amount the amount to deposit into the margin account
     */
    function _depositToMarginAccount(uint256 _amount) internal {
        IERC20(want).approve(address(mcLiquidityPool), _amount);
        mcLiquidityPool.deposit(
            perpetualIndex,
            address(this),
            int256(_amount) * DECIMAL_SHIFT
        );
        emit DepositToMarginAccount(_amount, perpetualIndex);
    }
```

The harvest will then deposit the short position and buffer position deposit asset into the strategy's MCDEX margin account.

```
    /**
     * @notice  open the perpetual short position on MCDEX
     * @param   _amount the collateral used to purchase the perpetual short position
     * @return  tradeAmount the amount of perpetual contracts opened
     */
    function _openPerpPosition(uint256 _amount, bool deposit)
        internal
        returns (int256 tradeAmount)
    {
        if (deposit) {
            // deposit funds to the margin account to enable trading
            _depositToMarginAccount(_amount);
        }

        // get the long asset mark price from the MCDEX oracle
        (int256 price, ) = oracle.priceTWAPLong();
        // calculate the number of contracts (*1e12 because USDC is 6 decimals)
        int256 contracts = ((int256(_amount) * DECIMAL_SHIFT) * 1e18) / price;
        int256 longBalInt = -int256(IERC20(long).balanceOf(address(this)));
        // check that the long and short positions will be equal after the deposit
        if (-contracts + getMarginPositions() >= longBalInt) {
            // open short position
            tradeAmount = mcLiquidityPool.trade(
                perpetualIndex,
                address(this),
                -contracts,
                price - slippageTolerance,
                block.timestamp,
                referrer,
                tradeMode
            );
        } else {
            tradeAmount = mcLiquidityPool.trade(
                perpetualIndex,
                address(this),
                -(getMarginPositions() - longBalInt),
                price - slippageTolerance,
                block.timestamp,
                referrer,
                tradeMode
            );
        }
        emit PerpPositionOpened(tradeAmount, perpetualIndex, _amount);
    }
```

The harvest will then open a short perpetual position on MCDEX using the short position collateral. The MCDEX internal oracle is used to determine the number of short contracts to open.

Finally, a few parameters are updated and a *harvest* event is emitted. *Fin.*

👨‍💻🧑💻 Integration guide

Guide for smart contract interactions with the vault

To interact with the system via smart contract there are only two functions you need to use, *deposit()* and *withdraw()*:

deposit(uint256 _amount, address _recipient)

```
    /**
     * @notice  deposit function - where users can join the vault and
     *          receive shares in the vault proportional to their ownership
     *          of the funds.
     * @param  _amount    amount of want to be deposited
     * @param  _recipient recipient of the shares as the recipient may not
     *                    be the sender
     * @return shares the amount of shares being minted to the recipient
     *                for their deposit
     */
    function deposit(uint256 _amount, address _recipient)
        external
        nonReentrant
        whenNotPaused
        returns (uint256 shares)
    {
        require(_amount > 0, "!_amount");
        require(_recipient != address(0), "!_recipient");
        require(totalAssets() + _amount <= depositLimit, "!depositLimit");

        shares = _issueShares(_amount, _recipient);
        // transfer want to the vault
        want.safeTransferFrom(msg.sender, address(this), _amount);

        emit Deposit(_recipient, _amount, shares);
    }
```

The recipient will receive vault shares that represent their share of funds in the vault at the time of deposit. Deposited funds are not deployed to the strategy immediately; they instead remain idle in the vault until a *harvest* is called where they are deployed to the strategy.

This function will revert if:

- The amount to be deposited is 0;
- The recipient is not a valid address;
- The deposit limit is reached;
- The caller has not approved funds to be used by the vault;
- The caller does not have *_amount* of the asset in their wallet.

withdraw(uint256 _shares, address _recipient)

```
/**
 * @notice  withdraw function - where users can exit their positions in a vault
 *          users provide an amount of shares that will be returned to a recipient.
 * @param  _shares    amount of shares to be redeemed
 * @param  _recipient recipient of the amount as the recipient may not
 *                    be the sender
 * @return amount the amount being withdrawn for the shares redeemed
 */
function withdraw(uint256 _shares, address _recipient)
    external
    nonReentrant
    whenNotPaused
    returns (uint256 amount)
{
    require(_shares > 0, "!_shares");
    require(_shares <= balanceOf(msg.sender), "insufficient balance");
    amount = _calcShareValue(_shares);
    uint256 vaultBalance = want.balanceOf(address(this));
    uint256 loss;

    // if the vault doesnt have free funds then funds should be taken from the strategy
    if (amount > vaultBalance) {
        uint256 needed = amount - vaultBalance;
        needed = Math.min(needed, totalLent);
        uint256 withdrawn;
        (loss, withdrawn) = IStrategy(strategy).withdraw(needed);
        vaultBalance = want.balanceOf(address(this));
        if (loss > 0) {
            amount = vaultBalance;
            totalLent -= loss;
            // all assets have been withdrawn so now the vault must deal with the loss t
            // _shares = _sharesForAmount(amount);
        }
        // reduce the totallent by the amount withdrawn, if the amount withdrawn is grea
        // then make it 0
        if (totalLent >= withdrawn) {
            totalLent -= withdrawn;
        } else {
            totalLent = 0;
        }
    }

    _burn(msg.sender, _shares);
    if (amount > vaultBalance) {
        amount = vaultBalance;
    }
    emit Withdraw(_recipient, amount, _shares);
    want.safeTransfer(_recipient, amount);
}
```

The withdraw function is slightly more complicated as funds are taken from the strategy if there are insufficient idle funds held in the vault contract.

The caller will have their share tokens burnt in exchange for the want value of their shares, this is determined by *calcShareValue(shares)*. If the vault doesn't have funds available immediately it will withdraw funds from the strategy. This involves closing the correct number of short positions, long positions and buffer positions which will output a withdrawal amount equivalent to the amount needed to fulfil the withdrawal. In the event of a loss, the amount returned will be reduced and this will be reflected in the share value and *totalLent*.

Position Buffer

Vortex v1 will remain at 1x leverage, but, as USDC is used as collateral, positions are at risk of liquidation. To manage this risk, a percentage of funds are held idle in an address as a **Position Buffer**. This is done to increase the available collateral in the margin account such that the margin account is always overcollateralised, dramatically reducing any liquidation risk.

This Buffer is a set size and will be maintained so long as active positions are open.

The Buffer can be changed at any time by the Vortex Manager; *remargin()* must be called after the buffer change.

> *For more information, on how this is managed during strategy operation see remargin*

remargin()

During operation of the vault it is possible for the leverage of the strategy's margin account to deviate away from 1x. If the strategy's leverage goes below 1x there is no financial risk, but the vault is not capital efficient as the non-profiting buffer represents a disproportionate amount of the strategy. More dangerously, however, is if the strategy's leverage goes above 1x, as then it is possible for the vault to fall below the margin ratio and for the margin account to get liquidated, which would result in a substantial loss.

The *remargin()* function solves this problem.

If the position's leverage is greater than 1x then a certain amount of the short and long positions are sold and added to the buffer. This brings the Position Buffer back to its correct value and resets the leverage back to 1x.

If the position's leverage is less than 1x then a certain amount of the buffer position is used to open short and long positions. This brings the Position Buffer back to its correct value and resets the leverage back to 1x.

> Position sizes will also increase based on returns accumulated by the strategy. These returns are *compounded* into positions during remargins.

> The *remargin()* function ensures that the **Position Buffer** is consistently maintained and that derivative positions remain at 1x leverage.

A description of the *remargin()* code and the maths behind it are shown below.

```
/**
     * @notice  remargin the strategy such that margin call risk is reduced
     * @dev     only callable by owner
     */
    function remargin() external onlyOwner {
        // harvest the funds so the positions are up to date
        harvest();
        // ratio of the short in the short and buffer
        int256 K = (((int256(MAX_BPS) - int256(buffer)) / 2) * 1e18) /
            (((int256(MAX_BPS) - int256(buffer)) / 2) + int256(buffer));
        // get the price of ETH
        (int256 price, ) = oracle.priceTWAPLong();
        // calculate amount to unwind
        int256 unwindAmount = (((price * -getMarginPositions()) -
            K *
            getMargin()) * 1e18) / ((1e18 + K) * price);
        require(unwindAmount != 0, "no changes to margin necessary");
        // check if leverage is to be reduced or increased then act accordingly
        if (unwindAmount > 0) {
            // swap unwindAmount long to want
            uint256 wantAmount = _swap(uint256(unwindAmount), long, want);
            // close unwindAmount short to margin account
            mcLiquidityPool.trade(
                perpetualIndex,
                address(this),
                unwindAmount,
                price + slippageTolerance,
                block.timestamp,
                referrer,
                tradeMode
            );
            // deposit long swapped collateral to margin account
            _depositToMarginAccount(wantAmount);
        } else if (unwindAmount < 0) {
            // the buffer is too high so reduce it to the correct size
            // open a perpetual short position using the unwindAmount
            mcLiquidityPool.trade(
                perpetualIndex,
                address(this),
                unwindAmount,
                price - slippageTolerance,
                block.timestamp,
                referrer,
                tradeMode
            );
            // withdraw funds from the margin account
            int256 withdrawAmount = (price * -unwindAmount) / 1e18;
            mcLiquidityPool.withdraw(
                perpetualIndex,
                address(this),
                withdrawAmount
            );
            // open a long position with the withdrawn funds
            _swap(uint256(withdrawAmount / DECIMAL_SHIFT), want, long);
        }
        positions.margin = getMargin();
        positions.unitAccumulativeFunding = getUnitAccumulativeFunding();
        positions.perpContracts = getMarginPositions();
        emit Remargined(unwindAmount);
    }
```

The *remargin()* begins by harvesting the strategy - this updates all positions.

Next the ratio of expected short:buffer, K, is calculated. This is calculated using the following equation:

$$K = \frac{(1 - buffer)/2}{((1 - buffer)/2) + buffer}$$

Then the *unwindAmount*, **Z**, is calculated. After *remargin()*, the value of long and short should be equal, thus we have the following equation - where **P** is the index price, **X** is the long or short size and **Y** is the margin size:

$$P(X - Z) = PZ + YK$$

Rearranging for **Z**:

$$Z = \frac{(PX - KY)}{2P}$$

The proof that this resets leverage is at the bottom of the page.

If **Z** is greater than 0, the vault is over-leveraged. Z positions are then closed from the short position and Z positions are closed on the long. This combined amount is then deposited to the margin account.

If **Z** is less than 0, the vault is under-leveraged. Z positions are then opened in the short position and Z positions are opened on the long - the funds to collateralise these positions are taken from the Position Buffer.

Proof that leverage, **L** is reset, after *remargin()*, the leverage is as follows:

$$L = \frac{P(X - Z)}{KY + PZ}$$

Replace **Z** with the remargined **Z** value:

$$L = \frac{P(X - \frac{PX - KY}{2P})}{KY + P\frac{PX - KY}{2P}}$$

$$L = \frac{PX - \frac{PX - KY}{2}}{KY + \frac{PX - KY}{2}}$$

$$L = \frac{PX + KY}{KY + PX}$$

$$L = 1$$

unwind()

In historic crypto-market conditions the funding rate has consistently been positive, which is an assumption that this strategy relies on. The Managers of Vortex are constantly monitoring the funding rate and ensuring that it remains positive; if it stays negative for a prolonged amount of time then the Manager can *unwind()* the funds to prevent any further losses. This function can also be used to prevent a liquidation. The *unwind()* function is described below:

```
/**
 * @notice  unwind the position in adverse funding rate scenarios, settle short pos
 *          and pull funds from the margin account. Then converts the long position
 *          to want.
 * @dev     only callable by the owner
 */
function unwind() public onlyAuthorised {
    require(!isUnwind, "unwound");
    isUnwind = true;
    mcLiquidityPool.forceToSyncState();
    // swap long asset back to want
    _swap(IERC20(long).balanceOf(address(this)), long, want);
    // check if the perpetual is in settlement, if it is then settle it
    // otherwise unwind the fund as normal.
    if (!_settle()) {
        // close the short position
        _closeAllPerpPositions();
        // withdraw all cash in the margin account
        mcLiquidityPool.withdraw(
            perpetualIndex,
            address(this),
            getMargin()
        );
    }
    // reset positions
    positions.perpContracts = 0;
    positions.margin = getMargin();
    positions.unitAccumulativeFunding = getUnitAccumulativeFunding();
    emit StrategyUnwind(IERC20(want).balanceOf(address(this)));
}
```

unwind() will close all long positions, then check if the MCDEX perpetual market has been settled. If it has been settled, then all funds will be withdrawn as all positions have already been closed by MCDEX. If the perpetual pool is not settled, then all short positions are closed and withdrawn from the margin account to the strategy contract.

emergencyExit()

In a situation where an immediate exit is required - such as if positions were to be liquidated or in the event of an exploit that may affect the system - a Manager can perform an *emergencyExit()*. This is where all positions are unwound and sent to the Vortex Governance multisig.

```
/**
 * @notice  emergency exit the entire strategy in extreme circumstances
 *          unwind the strategy and send the funds to governance
 * @dev     only callable by governance
 */
function emergencyExit() external onlyGovernance {
    // unwind strategy unless it is already unwound
    if (!isUnwind) {
        unwind();
    }
    uint256 wantBalance = IERC20(want).balanceOf(address(this));
    // send funds to governance
    IERC20(want).safeTransfer(governance, wantBalance);
    emit EmergencyExit(governance, wantBalance);
}
```

> This is a function of last resort, but is important to maintain security over funds.

Funding Rate Monitoring

To function correctly, the **Funding Rate** must be favorable to Vortex. If the Funding Rate is consistently against Vortex, positions will be managed/unwound accordingly. This funding rate is constantly managed by the Vortex Manager and, if is determined that the funding rate will remain unfavourable for a prolonged amount of time, then the strategy will be unwound to prevent further losses.

Amun Ra

What is Amun Ra?

Amun Ra is an AMM which is focused on providing low transaction costs, high throughput and best-in-class asset pricing. Amun Ra is built upon an Ethereum layer 2, StarkNet.

Advantages of using Amun Ra

Advantages of using Amun Ra include:

- **Low Cost** - Amun Ra has a low base trading fee and, thanks to StarkNet's infrastructure, a significantly lower transaction cost per trade.
- **High Throughput** - Amun Ra can facilitate more trades per second than its competitors on Ethereum layer 1, making it more suitable for time-sensitive or higher frequency trades.
- **Better Prices** - Amun Ra will offer the best prices for asset swaps on StarkNet due to its deep liquidity and, due to technical and product improvements, these will only get better in v2 onwards.
- **Yield Generation** - Amun Ra allows users to passively provide liquidity to capture a share of all trading fees accumulated by their selected pool.
- **Ecosystem Benefits** - Amun Ra will be able to facilitate the necessary liquidity for all new projects native to the StarkNet ecosystem.

Pensify

Pensify is a secure, non-custodial, no-loss and no-risk Pension Fund built on Ethereum blockchain.

- By using Robo-Advisor for Yield (RAY) from Staked.US, Fund constantly generates interest from different DeFi protocols
- Compound, Aave, dYdX, Fulcrum (latest is turned off atm), MCD, DSR.
- Members can also use Flash Loans to earn additional income via a browser bot for automatic arbitrage between Uniswap and Balancer pools.
- The fund is built using AkropolisOS framework, which allows automated liquidity provision enabled by the bonding curve, treasury management & automated yield rebalancing.

Github repo:

GitHub - AlexanderMazaletskiy/pensify: Pensify is a secure, non-custodial and risk-m...
GitHub

How it's made

We used AkropolisOS framework to build a basic architecture for Pensify. It is based on OpenZeppelin and allows automated liquidity provision enabled by the bonding curve, treasury management & automated yield rebalancing. To enable mobile support, we used Portis Wallet. It provides secure storage and access to Pensify from any device. We use Uniswap and Balancer protocols as a part of arbitrage strategies for fund members. They can earn additional income by utilizing Flash Loans and performing arbitrage between Uniswap and Balancer. We use Compound and Aave as an interest source through rebalancer - Robo Yield Advisor from Staked.Us. It allows us to accumulate interest on all Pensify funds.

C2FC

Commitments to Future Cashflows

A place to set up and trade Commitments to Future Cashflows (C2FC), a new financial primitive and a DeFi equivalent of cashflow financing.

Github repo:

GitHub - akropolisio/cashflowrelay: Cashflow Relay Mainnet (Live)
GitHub

C2FC bridges traditional finance and Web3 by providing DeFi and Web3 entrepreneurs with capital to grow.

Accelerate growth without giving up equity if you are an company or unlock new ways of funding your future if you are an independent.

- No equity, no fundraising, no dilution
- No warrants, no covenants, no bullshit
- Quick way to improve cashflow - get funds in faster
- Pay back as you grow
- Simple
- Acquire more customers

Built with MakerDAO + 0x + MetaMask

https://www.youtube.com/watch?v=-FPbc-ttMd4c
www.youtube.com

What is C2FC

The C2FC is a digital token, which acts as automatically executed right to claim a defined part of the future cashflow that arrives at issuer Ethereum address within a given time frame.

C2FC is materially different to the current implementations of ERC948 (EIP 1337) and EIP1620. The improvement is in the following features:

- Direct debit initialisation by the sender
- Ability to realize escrow logic for payments receiver
- Ability to transfer ownership to receive recurring payments: transferred to another person
- Tokens are composable and can be grouped into a portfolio according to pre-defined characteristics

What are the benefits

C2FC issuance allows to present future cashflows of any business or individual in as a single token, that can be easily exchanged, traded or used as collateral, thus materially simplifying workflows and opening up a potential for new DeFi instruments and interactions.

You can attract additional liquidity without using a 150% ensuring in digital assets by trading C2FC.

How does it work

An already operating business or an individual who has regular income payments, that needs additional liquidity at the moment can issue C2FC for defined future periods and trade them at marketplace. It looks like exchanging future cashflows for "money now" by selling discounted C2FC or borrowing funds, using C2FC as collateral.

Are there any risks involved in issuing or trading C2FCs?

C2FC issuance itself does not provide any risks until it is involved in the open market. When it is exchanged to other unit of value and changed ownership, there is a risk that future cashflow would be lower than estimated or there would no money arrived at all. This issue primarily is connected with the correct risk assessment of the C2FC issuer.

Considering the transparency of the C2FC itself and related indicators, the risk assessment would be no harder and in some cases can even be more reliable than in centralized finance.

Use cases

- 0x relayer or any node holder attracts additional capital to fund growth
- Spankchain operator forward-funds her coding bootcamp
- Gitcoin developers forward-fund their expenses
- Decentraland land owner receives funding against a leased asset
- Digital entrepreneurs forward-fund ad spend and attract customers

Previous
Pensify

Next
AkropolisOS

Last modified 1yr ago

AkropolisOS

AkropolisOS is A Solidity framework for building complex dApps and protocols (savings, pensions, loans, investments).

AkropolisOS is a framework for creating and managing distributed digital financial organisations. Anybody can use AkropolisOS to set up and collectively manage distributed capital pools with customisable user incentives, automated liquidity provision enabled by the bonding curve mechanism, and programmatic liquidity and treasury management. Designed as an upgradeable modular framework based on OpenZeppelin, AkropolisOS provides lego-like scalability without the loss of coherence and security.

Sparta is an undercollaterized credit pool based on AkropolisOS, where members of which can earn high-interest rates by providing undercollateralized loans to other members and by pooling and investing capital through various liquid DeFi instruments.

Delphi is a yield farming aggregator with dollar cost averaging tooling. Delphi allows users to gain yield on synthetic savings, farm tokens from integrated protocols/pools, and invest in volatile assets using an active “all in” approach or a passive dollar cost averaging strategy.

Github repo:

GitHub - akropolisio/akropolisOS: 💢 A Solidity framework for building complex dApp...
GitHub

Sparta

Sparta is an undercollaterized credit pool based on AkropolisOS, where members of which can earn high-interest rates by providing undercollateralized loans to other members and by pooling and investing capital through various liquid DeFi instruments.

Github repo:

GitHub - akropolisio/sparta: 💫 Access under-collateralised credit based on trust.
GitHub

Mainnet deployment

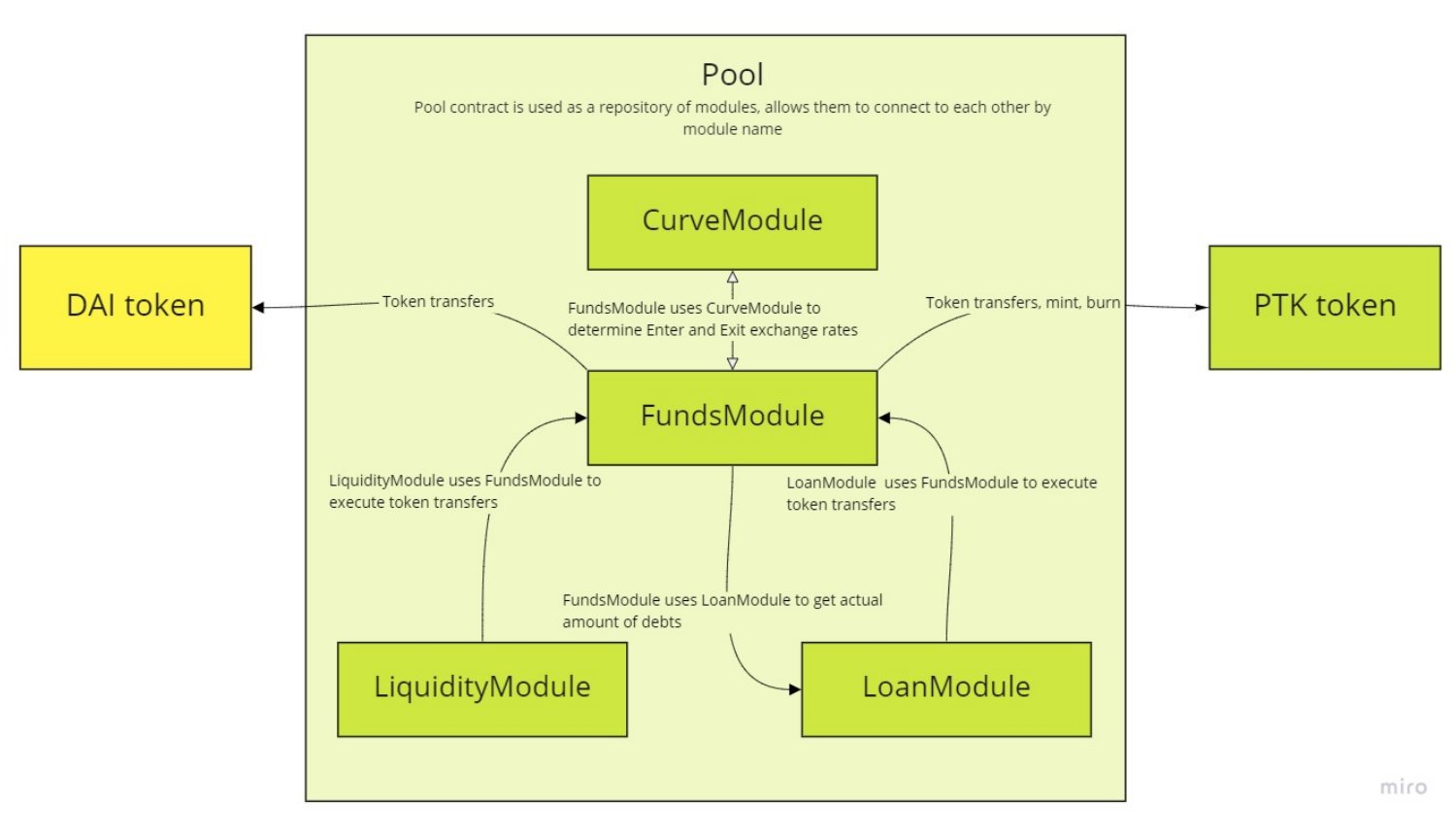
- DAI: `0x6b175474e89094c44da98b954eedeac495271d0f`
- Pool: `0x73067fdd366Cb678E9b539788F4C0f34C5700246`
- AccessModule: `0xfE7B0aeb84D134c5be6b217e51B2b040F5B7cB7B`
- PToken: `0xAA2edc0E5CDE4Da80628972c501e79326741dB17`
- CurveModule: `0xFb6b0103063CDf701b733db3Fa3F1c0686F19668`
- FundsModule: `0xc88F54A79CaE4C125D7A8c2Cf811daaE78b07D64`
- LiquidityModule: `0x543cBc6693f8cBCf0AE5f2cfd9922203cc13b10A`
- LoanLimitsModule: `0x42b41f636C9eBB150F859f65e3c0f938b0347f59`
- LoanProposalsModule: `0xd3bdEdA5e165E67985a4Dc7927E4651Bedd1950c`
- LoanModule: `0x42E24De51db5baf6E18F91619195375FBAe63b13`

Developer tools

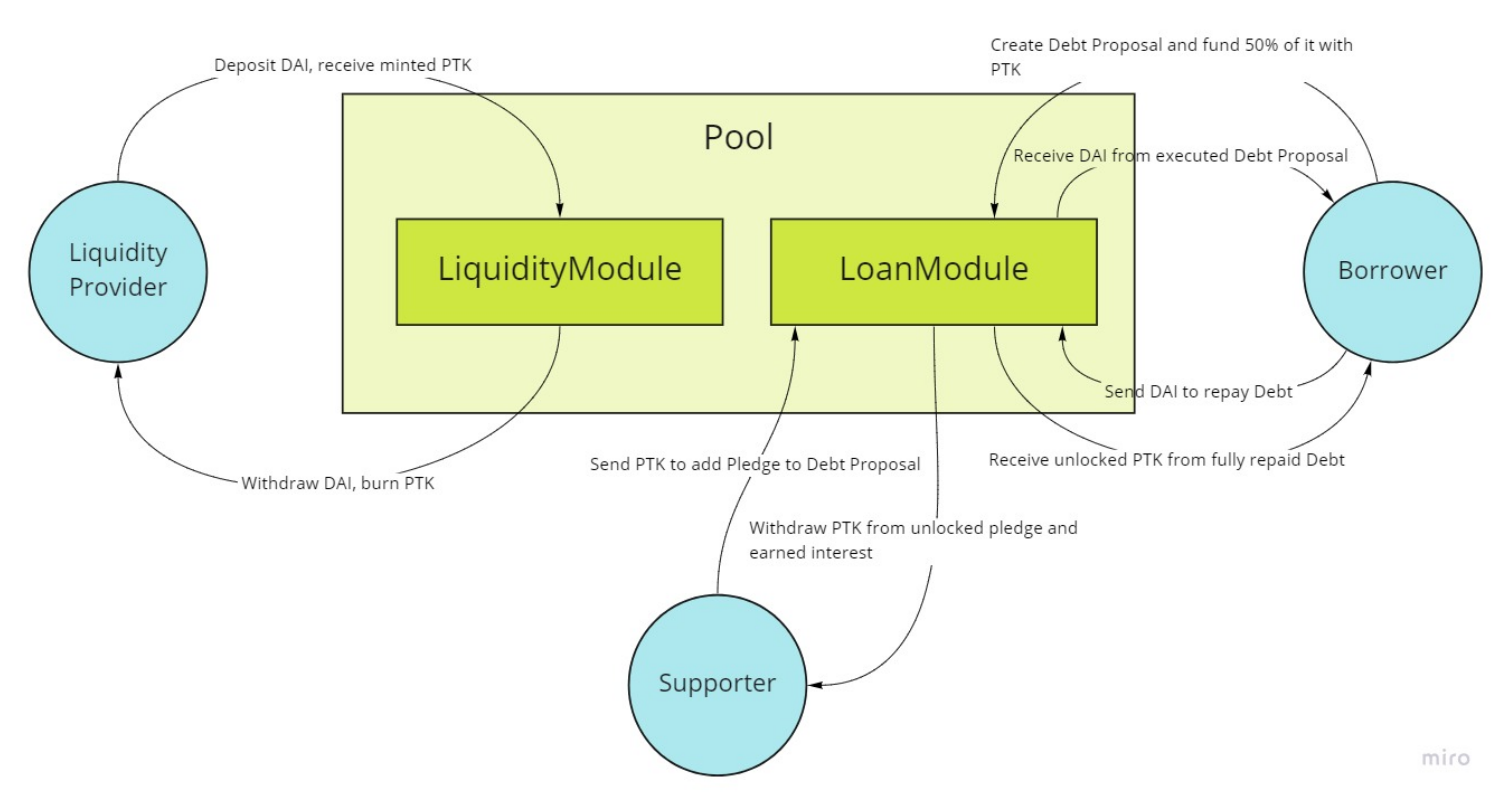
- Openzeppelin SDK
- Openzepplin Contracts
- Truffle

Diagrams

Modules



User Interactions



Deployment

Required data:

- Address of liquidity token (`LToken.address`)

Deployment sequence:

1. Pool
    1. Deploy proxy and contract instance
    2. Call `initialize()`
2. Liquidity token
    1. Register in pool: `Pool.set("ltoken", LToken.address)`
3. PToken
    1. Deploy proxy and contract instance
    2. Call `initialize(Pool.address)`
    3. Register in pool: `Pool.set("ptoken", PToken.address)`
4. CurveModule
    1. Deploy proxy and contract instance
    2. Call `initialize(Pool.address)`
    3. Register in pool: `Pool.set("curve", CurveModule.address)`
5. AccessModule
    1. Deploy proxy and contract instance
    2. Call `initialize(Pool.address)`
    3. Register in pool: `Pool.set("access", CurveModule.address)`
6. LiquidityModule
    1. Deploy proxy and contract instance
    2. Call `initialize(Pool.address)`
    3. Register in pool: `Pool.set("liquidity", LiquidityModule.address)`
7. LoanModule, LoanLimitsModule, LoanProposalsModule
    1. Deploy proxy and contract instance of LoanLimitsModule
    2. Call `LoanLimitsModule.initialize(Pool.address)`
    3. Register in pool: `Pool.set("loan_limits", LoanLimitsModule.address)`
    4. Deploy proxy and contract instance of LoanProposalsModule
    5. Call `LoanProposalsModule.initialize(Pool.address)`
    6. Register in pool: `Pool.set("loan_proposals", LoanProposalsModule.address)`
    7. Deploy proxy and contract instance of LoanModule
    8. Call `LoanModule.initialize(Pool.address)`
    9. Register in pool: `Pool.set("loan", LoanModule.address)`
8. FundsModule
    1. Deploy proxy and contract instance
    2. Call `initialize(Pool.address)`
    3. Register in pool: `Pool.set("funds", FundsModule.address)`
    4. Add LiquidityModule as FundsOperator: `FundsModule.addFundsOperator(LiquidityModule.address)`
    5. Add LoanModule as FundsOperator: `FundsModule.addFundsOperator(LoanModule.address)`
    6. Add FundsModule as a Minter for PToken: `PToken.addMinter(FundsModule.address)`

Liquidity

Deposit

Required data:

- `lAmount` : Deposit amount, DAI

Required conditions:

- All contracts are deployed

Workflow:

1. Call `FundsModule.calculatePoolEnter(lAmount)` to determine expected PTK amount (`pAmount`)
2. Determine minimum acceptable amount of PTK `pAmountMin <= pAmount`, which user expects to get when deposit `lAmount` of DAI. Zero value is allowed.
3. Call `LToken.approve(FundsModule.address, lAmount)` to allow exchange
4. Call `LiquidityModule.deposit(lAmount, pAmountMin)` to execute exchange

Withdraw

Required data:

- `pAmount` : Withdraw amount, PTK

Required conditions:

- Available liquidity `LToken.balanceOf(FundsModule.address)` is greater than expected amount of DAI
- User has enough PTK: `PToken.balanceOf(userAddress) >= pAmount`

Workflow:

1. Call `FundsModule.calculatePoolExitInverse(pAmount)` to determine expected amount of DAI (`lAmount`). The response has 3 values, use the second one.
2. Determine minimum acceptable amount `lAmountMin <= lAmount` of DAI , which user expects to get when deposit `pAmount` of PTK. Zero value is allowed.
3. Call `PToken.approve(FundsModule.address, pAmount)` to allow exchange
4. Call `LiquidityModule.withdraw(pAmount, lAmountMin)` to execute exchange

Credits

Create Loan Request

Required data:

- `debtLAmount` : Loan amount, DAI
- `interest` : Interest rate, percents
- `pAmountMax` : Maximal amount of PTK to use as borrower's own pledge
- `descriptionHash` : Hash of loan description stored in Swarm

Required conditions:

- User has enough PTK: `PToken.balanceOf(userAddress) >= pAmount`

Workflow:

1. Call `FundsModule.calculatePoolExitInverse(pAmount)` to determine expected pledge in DAI (`lAmount`). The response has 3 values, use the first one.
2. Determine minimum acceptable amount `lAmountMin <= lAmount` of DAI, which user expects to lock as a pledge, sending `pAmount` of PTK. Zero value is allowed.
3. Call `PToken.approve(FundsModule.address, pAmount)` to allow operation.
4. Call `LoanModule.createDebtProposal(debtLAmount, interest, pAmountMax, descriptionHash)` to create loan proposal.

Data required for future calls:

- Proposal index: `proposalIndex` from event `DebtProposalCreated` .

Add Pledge

Required data:

- Loan proposal identifiers:
    - `borrower` Address of borrower
    - `proposal` Proposal index
- `pAmount` Pledge amount, PTK

Required conditions:

- Loan proposal created
- Loan proposal not yet executed
- Loan proposal is not yet fully filled: `LoanModule.getRequiredPledge(borrower, proposal) > 0`
- User has enough PTK: `PToken.balanceOf(userAddress) >= pAmount`

Workflow:

1. Call `FundsModule.calculatePoolExitInverse(pAmount)` to determine expected pledge in DAI (`lAmount`). The response has 3 values, use the first one.
2. Determine minimum acceptable amount `lAmountMin <= lAmount` of DAI, which user expects to lock as a pledge, sending `pAmount` of PTK. Zero value is allowed.
3. Call `PToken.approve(FundsModule.address, pAmount)` to allow operation.
4. Call `LoanModule.addPledge(borrower, proposal, pAmount, lAmountMin)` to execute operation.

Withdraw Pledge

Required data:

- Loan proposal identifiers:
    - `borrower` Address of borrower
    - `proposal` Proposal index
- `pAmount` Amount to withdraw, PTK

Required conditions:

- Loan proposal created
- Loan proposal not yet executed
- User pledge amount >= `pAmount`

Workflow:

1. Call `LoanModule.withdrawPledge(borrower, proposal, pAmount)` to execute operation.

Loan issuance

Required data:

`proposal` Proposal index

Required conditions:

- Loan proposal created, user (transaction sender) is the `borrower`
- Loan proposal not yet executed
- Loan proposal is fully funded: `LoanModule.getRequiredPledge(borrower, proposal) == 0`
- Pool has enough liquidity

Workflow:

1. Call `LoanModule.executeDebtProposal(proposal)` to execute operation.

Data required for future calls:

- Loan index: `debtIdx` from event `DebtProposalExecuted` .

Loan repayment (partial or full)

Required data:

- `debt` Loan index
- `lAmount` Repayable amount, DAI

Required conditions:

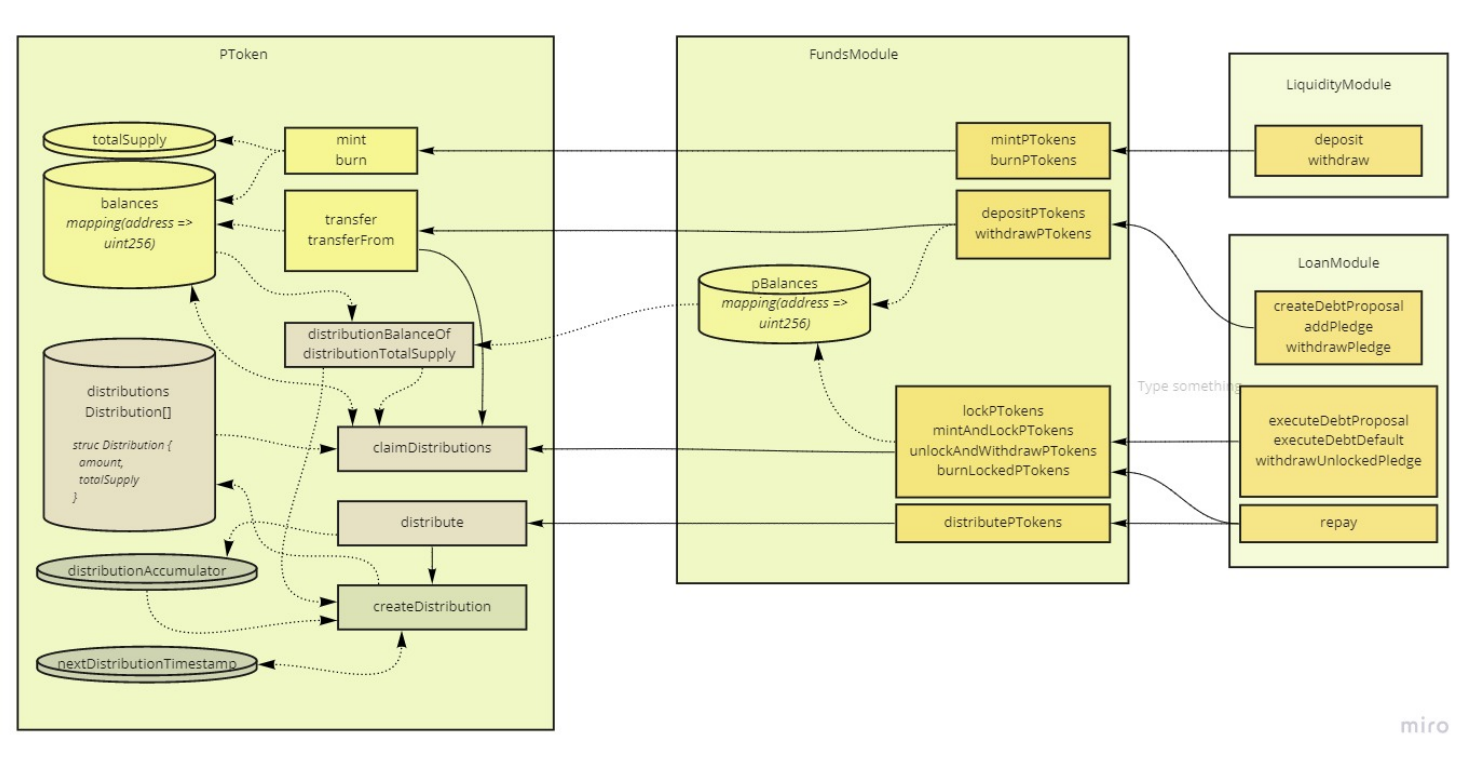
- User (transaction sender) is the borrower
- Loan is not yet fully repaid

Workflow:

1. Call `LToken.approve(FundsModule.address, lAmount)` to allow operation.
2. Call `LoanModule.repay(debt, lAmount)` to execute operation.

Distributions

When borrower repays some part of his loan, he uses some PTK (either from his balance or minted when he sends DAI to the pool). This PTKs are distributed to supporters, proportionally to the part of the loan they covered. The borrower himself also covered half of the loan, and his part is distributed over the whole pool. All users of the pool receive part of this distributions proportional to the amount of PTK they hold on their balance and in loan proposals, PTK locked as collateral for loans is not counted.



Distribution mechanics

When you need to distribute some amount of tokens over all token holders one's first straight-forward idea might be to iterate through all token holders, check their balance and increase it by their part of the distribution. Unfortunately, this approach can hardly be used in Ethereum blockchain. All operations in EVM cost some gas. If we have a lot of token holders, gas cost for iteration through all may be higher than a gas limit for transaction (which is currently equal to gas limit for block). Instead, during distribution we just store amount of PTK to be distributed and current amount of all PTK qualified for distribution. And user balance is only updated by separate request or when it is going to be changed by transfer, mint or burn. During this "lazy" update we go through all distributions occured between previous and current update. Now, one may ask what if there is too much distributions occurred in the pool between this updated and the gas usage to iterate through all of them is too high again? Obvious solution would be to allow split such transaction to several smaller ones, and we've implemented this approach. But we also decided to aggregate all distributions during a day. This way we can protect ourself from dust attacks, when somebody may do a lot of small repays which cause a lot of small distributions. When a distribution request is received by PToken we check if it's time to actually create new distribution. If it's not, we just add distribution amount to the accumulator. When time comes (and this condition is also checked by transfers, mints and burns), actual distribution is created using accumulated amount of PTK and total supply of qualified PTK.

Delphi

Delphi is a yield farming aggregator. Delphi allows users to gain yield on synthetic savings & farm tokens from integrated protocols/pools.

What is Delphi v1?

Delphi v1 is the first version of a DeFi yield-farming aggregator tool, build on AkropolisOS. Delphi v1 contains stablecoins savings and ADEL staking pools integrated.

Pools

Save

Savings is all about stablecoins pools. Currently, you can choose where to supply liquidity from selected Curve.fi and Compound pools. You can allocate in one or several pools in once click - just choose amounts & currencies, click on Allocate - your funds will be sent to the corresponding pools, earn interest & farm different tokens all at once.

Stake

Mechanics here are simple - you just send token to the chosen pool & farm additional tokens. No lock-up for the staked funds.

Github repo:

GitHub - akropolisio/delphi: 💥 Earn Rewards for Saving and Liquidity Provision (work ...
GitHub

Polkahub

Introduction

PolkaHub aims to create a managed container system, with one-click services for deploying and running different parachain nodes. The proposed system enables automatic updates and resource management for running nodes, additionally providing templates for launching parachains.

Github Repo:

GitHub - akropolisio/polkahub-monorepo: Fast, scalable cloud infrastructure for Subs...
GitHub

What is PolkaHub?

1. Polkahub is a fast, scalable blockchain infrastructure component for Substrate parachains.
2. Polkahub provides parachain developers with the ability to launch and manage network infrastructure using our command line utility and the ability to provide public node access.
3. Polkahub provides developers with a unified standard for packaging and deploying applications to cloud infrastructure.
4. Polkahub provides the functionality to track and control the parachain's versions. So in the event of critical bugs arising, developers can update or roll back the version of the parachain using simple commands (via running specific commands in a command line).
5. Polkahub supports Substrate node deployment to remote servers or cloud infrastructure via git.
6. Polkahub infrastructure is based on Docker Container Services such as Kubernetes.
7. Docker provides high-level interfaces for isolated environments within the node's execution. Easily scaled, managed and updated.
8. No need to rely on DevOps and System Administration for managing parachain's infrastructure - thanks to PolkaHub. You only need git, command line and a simple web-interface.

Polkahub

Quick Start

Install CLI

MacOS / Linux You can just use bash script:

```
$ bash <(curl http://get.polkahub.org/ -L)
```

This will install polkahub binary in your /usr/local/bin(MacOS) or /usr/bin(Linux) directory Then you can use it:

```
$ polkahub <action> [ARGS]
```

Or use docker image like this:

```
$ mkdir $HOME/.polkahub
$ docker run --rm -t1 -v $HOME/.polkahub_docker:/tmp/home -e POLKAHUB_HOME=/tmp/home rep
```

Usage CLI

Authentication

```
$ polkahub auth
Email: user@example.com
Password:

Login user with email user@example.com

done
```

Use email and password created via https://polkahub.org or create new email and password via CLI (see Registration section)

Registration

```
$ polkahub register
Email: user@example.com
Password:
Confirm Password:

Registration new user with email user@example.com

done
```

Create a new project

```
$ polkahub create akropolisos


done
https     -> "https://steadfast-surprise-6647-akropolisos-rpc.polkahub.tech"
ws        -> "wss://steadfast-surprise-6647-akropolisos.polkahub.tech"
remote  -> "https://git.polkahub.org/steadfast-surprise-6647-akropolisos.git"
```

Then you can add **remote** to your project, push it and it will automatically start CI build.

Find a project

```
$ polkahub find akropolisos

Looking for akropolisos project

steadfast-surprise-6647/akropolisos@0.8.2
steadfast-surprise-6647/akropolisos@v1
steadfast-surprise-6647/akropolisos@v2
```

Install a exists project

```
$ polkahub install steadfast-surprise-6647/akropolisos@0.8.2 -a alexander

Deploying akropolisos project with version 0.8.2

done
https  -> "https://frightened-brick-8071-alexander-rpc.polkahub.tech"
ws     -> "wss://frightened-brick-8071-alexander.polkahub.tech"
```

Deploying private Polkahub

Preliminary steps

1. Install Rust on host, where Polkahub components will be build. Manual documentation: https://www.rust-lang.org/learn/get-started
2. Buy a VPS, minimal requirements are 2CPU and 8GB RAM. The more CPU - the faster deployment of projects in CI
3. Install the Docker and the Docker Compose on host, where Polkahub components will run. Manual documentation: https://docs.docker.com/
4. Install the Docker Registry. You can use public or private one from the cloud provider of your choice. Also you can set up your own one. Manual documentation: https://docs.docker.com/
5. Install and configure the Kubernetes cluster. You can buy a cluster from the cloud provider of your choice or deploy your own one on your server. Manual documentation: https://kubernetes.io/docs/home/
6. Sign up on https://www.cloudflare.com/ and get the token, key_auth, zone_name and zone_id to create the DNS records via https://api.cloudflare.com/
7. Sign up on https://www.mailgun.com/ and get the api_key and domain_name to sending emails via https://api.mailgun.net/
8. Install packages for Debian-like Linux distributives: apt-get install -y libssl-dev ca-certificates git curl

Building the components

1. CLI. Building manual:https://github.com/akropolisio/polkahub-cli . To change URL REST API you need to edit constants INSTALL_URL, FIND_URL , POLKAHUB_URL in /src/parsing.rs
2. REST API. Building manual: https://github.com/akropolisio/polkahub-backend
3. Deployer. Building manual: https://github.com/akropolisio/polkahub-deployer
4. Cloudflare Manager. Building manual: https://github.com/akropolisio/cloudflare-manager
5. Mailgun Sender. Building manual: https://github.com/akropolisio/mailgun-sender

Running the components in the Kubernetes

1. Pack the Deployer in the Docker-image. Files and environment variables descriptions are here: https://github.com/akropolisio/cloudflare-manager Push to Docker Registry, run the Deployer in Kubernetes cluster and configure the access via HTTP to the Deployer for CI.
2. Pack the Cloud Manager in the Docker-image. Files and environment variables descriptions are here: https://github.com/akropolisio/cloudflare-manager. Secret Files content can be found in point 6 of Preliminary steps section. Push to Docker Registry, run the Cloudflare Manager in Kubernetes cluster and configure the access via HTTP to the Cloudflare Manager for CI.
3. Pack the Mailgun Sender in the Docker-image. Files and environment variables descriptions are here: https://github.com/akropolisio/mailgun-sender. Secret Files content can be found in point 6 of Preliminary steps section. Push to Docker Registry, run the Mailgun Sender in Kubernetes cluster and configure the access via HTTP to the Mailgun Sender for REST API.

Running the components on VPS

1. Install Jenkins. Manual documentation: https://jenkins.io/download/. Add the access token to REST API, configure 2 jobs: build-new-version and deploy-fixed-version. You may find examples of jobs at jenkins-test.polkahub.org. Configure the access to the Docker server and Docker registry for Jenkins.
2. Deploy the REST API. Environment variables descriptions are here: https://github.com/akropolisio/polkahub-backend Token and job (build-new-version) name can be found in the point 1 of "Running the components on VPS". Configure the access via HTTP to the REST API for CLI.
3. Deploy the Git server, configure the access via HTTP to the git repositories directories (e.g. via Nginx + fcgiwrap + git-http-backend)
4. Deploy the PostgreSQL, configure the access for REST API

Project Components

Name	Version	Purpose
cloudflare-manager	0.1.0	DNS management
mailgun-sender	0.1.0	Email sender via Mailgun
polkahub-backend	0.2.0	Manage user command execution
polkahub-cli	0.4.0	Process user input and trigger api calls
polkahub-deployer	0.2.0	Form docker manifest for Kubernetes API
polkahub-installer	0.4.0	Shell script to install latest Polkahub CLI build

Docker images built

All images available at registry.polkahub.org/`<name>`:`<tag>`

General

Name	Tag	Purpose
deployer	v2	Microservice to deploy the projects in Kubernetes
cloudflare-manager	v1	Microservice to create DNS-records in Cloudflare
mailgun-sender	v1	Email sender via Mailgun
substrate-builder	rust-1.41-v3	Docker-image with the environment for projects building in CI

Pre-Built for test environment

Name	Tag	Purpose
polkahub-backend	v3	API image for Polkahub test launch via Docker-compose
polkahub-git-backend	v1	Git server image for Polkahub test launch via Docker-compose
polkahub-jenkins	v1	Jenkins image for Polkahub test launch via Docker-compose
balancer	v3	Jenkins image for Polkahub test launch via Docker-compose
polkahub-cli	v3	CLI image interacting with the test environment
akropolis-akropolisos	0.8.2	AkropolisOS Node image
stimulating-plant-3173-alexander	v0.4	Polkadot-Alexander Node Image
decisive-picture-8329-kusama	v0.7	Kusama Node Image
polkahub-frontend	v1	Web portal

Ethereum <-> Substrate bridge

Ethereum <-> Parity Substrate Blockchain bridge for self transfers of ERC20 representation.

Github repo:

GitHub - akropolisio/erc20-substrate-bridge: Ethereum <-> Parity Substrate Blockch...
GitHub

You can try it out in our chain:

1. Make sure you have Ethereum and Substrate extensions. Typical choices would be:
   a. `Metamask` (or any other Ethereum extension) and switch it to `Kovan`
   b. `polkadot{.js}`
2. Go here
3. Connect with both extensions(two pop-up windows should appear)
4. You will see that your balances from extensions should appear on the page.
5. Transfer some Kovan test DAI to our Substrate-based chain.
6. Transfer some DAI from our chain to your Ethereum account.

```
cd ./frontend <br/>
```

```
npm i <br/>
npm run codegen <br/>
npm run dev <br/>
```

Web3 Wallet Kit

Introduction

This kit will help connect you dApp to different Ethereum wallets, e.g. Metamask. With web3-wallets-kit, you can create Web3WalletsManager and connect your dApp to the wallet of your choice using one of the supported wallet integrations.

Github repo:

GitHub - akropolisio/web3-wallets-kit: Package for connecting with Ethereum wallets...
GitHub

Wallet integrations:

Wallet	Integration Package	Size
Inpage (Extensions like Metamask or Web3 browsers like Cipher)	`@web3-wallets-kit/inpage-connector`	bundlephobia 429
WalletConnect	`@web3-wallets-kit/connect-wallet-connector`	bundlephobia 429
Bitsky	`@web3-wallets-kit/bitski-connector`	bundlephobia 429
Fortmatic	`@web3-wallets-kit/fortmatic-connector`	bundlephobia 429
Portis	`@web3-wallets-kit/portis-connector`	bundlephobia 429
Squarelink	`@web3-wallets-kit/squarelink-connector`	bundlephobia 429
Torus	`@web3-wallets-kit/torus-connector`	bundlephobia 429
Ledger	Coming soon	

Installation

`npm install --save @web3-wallets-kit/core`

`npm install --save @web3-wallets-kit/inpage-connector` or another integration

Creating and managing wallets

```
import Web3 from 'web3';
import { Web3WalletsManager } from '@web3-wallets-kit/core';
import { InpageConnector } from '@web3-wallets-kit/inpage-connector';

// Create Web3WalletsManager instance
const web3Manager = new Web3WalletsManager<Web3>({
  defaultProvider: {
    network: 'kovan',
    infuraAccessToken: 'INFURA_API_KEY',
  },
  makeWeb3: provider => new Web3(provider), // you can use web3.js, ethers.js or another
});

// Create connector
const connector = new InpageConnector();

// Connect to wallet
await web3Manager.connect(connector);

// Get address and Web3 for sending transaction
const myAddress = web3Manager.account.value;
const txWeb3 = web3Manager.txWeb3.value;

// Create contract
const daiContract = txWeb3.eth.Contract(DAI_ABI, '0x5592ec0cfb4dbc12d3ab100b257153436a1

// Send transaction
await daiContract.methods
  .transfer('0x0000000000000000000000000000000000000000', '1000000000000000000')
  .send({ from: myAddress });
```

Web3WalletsManager API

```
class Web3WalletsManager<W> {
    /** default Web3 instance for reading. Uses a provider created based on defaultProv
    web3: W;
    /** Web3 instance for sending transactions. An instance is created after connecting
    txWeb3: BehaviorSubject<W | null>;
    /** active account address */
    account: BehaviorSubject<string | null>;
    /** active network ID */
    chainId: BehaviorSubject<number | null>;
    /** status of the connection */
    status: BehaviorSubject<ConnectionStatus>;

    constructor(options: Options<W>);

    /** Connect to wallet; Returns account address and Web3 Instance for sending transac
    connect(connector: Connector): Promise<ConnectResult>;
    /** Disconnect wallet, close streams */
    disconnect(): Promise<void>;
}

interface Options<W> {
    defaultProvider: {
        httpRpcUrl?: string;
        wsRpcUrl?: string;
        infuraAccessToken?: string;
        /** default: 'mainnet' */
        network?: InfuraNetwork;
    };
    makeWeb3<W>(provider: Provider): W;
}
```

Substrate Staking portal

Introduction

Existing staking mechanism via https://polkadot.js.org/apps/ has developer-centric UX and is very complicated for ordinary users. To simplify UX we want to build Staking portal for AkropolisOSChain users.

Github repo:

GitHub - akropolisio/staking-portal: Akropolis Staking Portal
GitHub

Frontend:

Akropolis Staking Portal
akropolis-polkadot-chain-frontend

What is Polkadot Staking Portal?

A simple and intuitive interface and Akropolis browser extension will make the staking process accessible to a wide range of users - you only need an account on Polkadot and Polkadot-js for signing transactions.

What you can do with our staking portal:

- Check your overall balance and amount of all bonded tokens - as well as check each wallet connected
- Check the current validators set, their commission, how much is staked for them, etc. and decide whether you want to nominate for them or not.
- Check and edit stake conditions - add/withdraw funds, edit the list of nominees, stop nominating, redeem funds, etc.

For frontend (in frontend folder)

Install all dependencies

- `npm i` install frontend and contracts dependencies

To start locally

- `npm run dev` for development environment in watch mode
- `npm run prod` for production environment in watch mode

To build locally (see build folder)

- `npm run build:dev` for development environment without watch mode
- `npm run build:prod` for production environment without watch mode

To start bundle analyzer

- `npm run analyze:dev` for development environment
- `npm run analyze:prod` for production environment

To start test

- `npm test` or `npm t` for start test, before that you need start network (`npm run ganache-cli`)

We used:

- [x] polkadot.js/api for interacting with Akropolis Chain
- [x] Typescript
- [x] React
- [x] Redux
- [x] Redux-saga for side-effects
- [x] Material-UI

🔮 Audits

Vortex

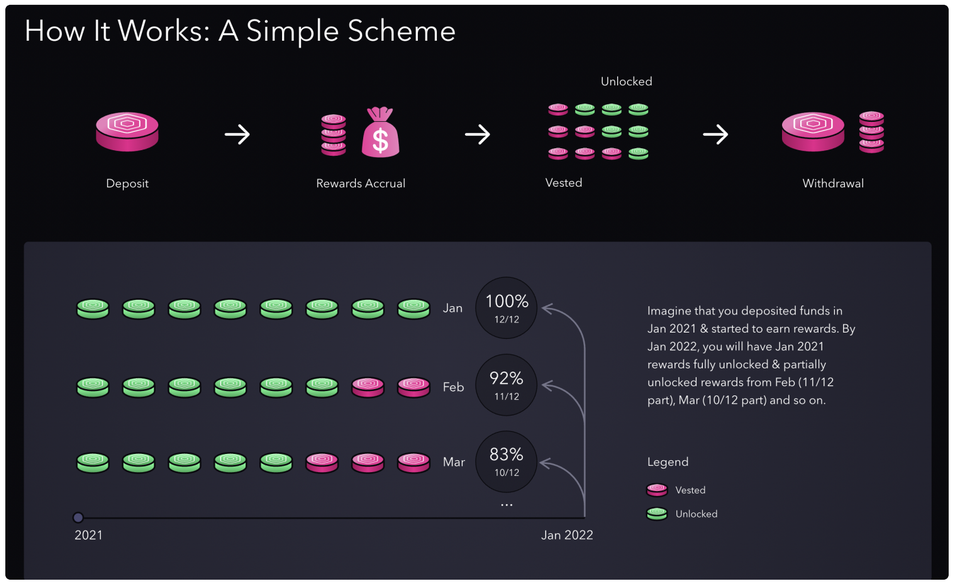
- Peckshield
- Xtropy

FAQ

What is rewards vesting?

Vesting is a lockup model for incentivizing long-term users and tokenholders. All AKRO rewards come with vesting - they gradually unlock throughout the year after you earn them. You can find the total vested amount you got and unlocked amount in your Summary tab.

> Unlocked amounts available for claiming are updated every two weeks. There is no timelimit for claiming. Read more on vesting mechanics **here**.



What's "Infinite unlock"?

Infinite unlock means that you preapprove the contract to be able to spend any amount of tokens when you interact with it. Enabling infinite unlock means that you approve spending tokens only once (thus reducing gas costs associated with signing “approve” transaction). Bear in mind - after enabling infinite unlock, all following transfers/deposit of the asset chosen won’t need approval - so please use it if you fully trust the contract.

General guide for interacting with pools:

Using Akropolis

To be continued...

Community Channels

Keep up-to-date with all Akropolis developments by following our **Twitter** and **Medium**, and join our community discussions in **Discord**, **Reddit** and **Telegram**.

Telegram

Discord

Twitter

Medium

Governance forum

Reddit

Basis Trading and Perpetual Contracts

This section will provide an overview of key concepts utilized by Vortex.

What is Basis Trading?

Basis Trading is an arbitrage strategy used in financial markets which takes advantage of the difference between the spot and future price of a commodity (the *basis*).

As an example of how basis trading works - and without going into too much detail - imagine a trader had the opportunity to purchase 1 ETH at $3500 and sell the equivalent of 1 ETH of futures contracts at $4000. The trader would then be able to lock in a profit from the basis, which in this case is $500.

Vortex works in a similar way, but instead of a centralized futures market, uses decentralized derivative exchanges that offer **Perpetual Contracts** to generate yield.

What are Perpetual Contracts?

Perpetual Contracts are similar to traditional futures, but, as their name suggests, have no expiration or settlement date.

This indefinite-until-closed nature also means that Perpetuals trade much closer to the current spot price than futures - but, being derivatives, they do still diverge. This price divergence from spot generally reflects the sentiment of traders on the exchange.

It is crucial that this divergence is controlled and the price of Perpetuals are frequently brought back to closely match spot prices.

The mechanism to achieve this control and incentivize spot/Perpetual price stability is known as the **Funding Rate**.

What is the Funding rate?

The **Funding Rate** is a fee periodically paid from the ‘more popular’ side of the market to the opposing ‘less popular’ side to incentivize contract purchases.

If the Perpetuals price is above the spot price, the Funding Rate will be positive and traders with open long contracts will pay the rate to traders with open short contracts.

Conversely, if the Perpetuals price is below the spot price, the Funding Rate will be negative and open shorts will pay open longs.