Maverick v2 - The Liquidity Operating System

Maverick Research Team

January 2024

1 Introduction

Maverick v2 introduces several improvements over Maverick v1:

- **Improved Swap Gas** - Swap gas cost in v2 is now less than 100k, making Maverick by far the lowest-gas concentrated-liquidity AMM in the market.
- **Programmable Pools** - Maverick v2 allows for protocols and pool creators to wrap pools in router contracts that implement specialty logic. This makes it straightforward for users to implement ideas like a KYC pool or a pool with dynamic swap fees, or even to build their own custom liquidity rebalancing mechanism. This logic-wrapping mechanism is both more flexible and more gas-efficient than hook-based paradigms.
- **Boosted Positions Incentive Directing** - With v1, Maverick proved the utility and efficiency of Boosted Positions (BPs) in shaping global liquidity distributions for objectives like holding a peg or creating buy/sell price walls. Maverick v2 BPs can now receive matching incentives on the basis of vote-escrowed token voting, adding more value for both token projects and LPs.
- **Any-Token Voting Escrow Factory** - Maverick v2 has a turn-key factory mechanism that lets any protocol create a ve token that can vote for Maverick BPs. The ve token creator can choose the base tokens and the parameters of the voting system specific to that token, thereby enabling a ready-made liquidity flywheel for new token projects that need to shape their global liquidity distribution.

2 AMM

Maverick v2 AMM is a dynamic distribution AMM where LPs can choose to have their liquidity automatically move to stay near the price. The outward-facing functionality of the v2 AMM is almost identical to v1, but the underlying mechanisms have changed significantly to lower gas costs and allow for more flexible liquidity movement options.

The following subsections define the implementation logic of the AMM.

2.1 Pool

Pools are created by calling the `create` function on the pool factory. Pools are defined by the following parameters:

- `fee` - Portion of each swap that is paid to the pool liquidity providers
- `tickSpacing` - Defines the tick width where tick width is $1.0001^{\texttt{tickSpacing}}$
- `lookback` - Lookback time that is used by the time-weighted average price component of the pool
- `tokenA` - Quote token in the pool
- `tokenB` - Base token in the pool
- `activeTick` - Starting tick of the pool that defined the initial pool price
- `tokenAScale` - Scale factor of quote token used to convert the token units from the decimal scale of the token to the internal 18-decimal pool accounting
- `tokenBScale` - Scale factor of base token used to convert the token units from the decimal scale of the token to the internal 18-decimal pool accounting
- `kinds` - Flag that indicates which bin kinds are available in the pool
- `accessor` - Address of pool accessor for programmable pools; permissionless pools do not have an accessor defined

The biggest difference from v1 is that in v2 a pool creator can choose to restrict the bin kinds (i.e., Static, Right, Left, Both) in a given pool to just a subset of the four bin kinds (Static is required). A pool creator may want to do this to further improve gas savings of swaps or to ensure that no LPs join a movement-mode bin that works against the objective of the pool creator.

A second notable difference is the concept of a programmable pool, where only one address can access the pool state-changing functions such as `addLiquidity`, `removeLiquidity`, and `swap`. By restricting access to a pool, a pool creator can implement creative mechanisms like dynamic swap fees, liquidity rebalancing, or any other strategy that can be executed on a smart contract. For more information, see Section 2.6 in this whitepaper.

The pool state is a structure on the pool that gets loaded during pool write operations. It is two storage slots wide and is composed of the following values:

- `reserveA` - Quote token `balanceOf` value at the end of the last operation
- `reserveB` - Base token `balanceOf` value at the end of the last operation
- `lastTwa` - Last value of the time-weighted average of $\log_{\sqrt{1.0001}} price$
- `lastLogPrice` - Last value of $\log_{\sqrt{1.0001}} price$
- `lastTimestamp` - Timestamp of last swap

- `activeTick` - Tick that contains the current price of the pool
- `isLocked` - Boolean indicator of whether the pool is locked or not
- `binCounter` - Number of bins that have ever been created in the pool
- `protocolFeeRatio` - Proportion of the swap fee that is kept aside by the pool as a protocol fee

In addition to these state values, there is another slot for state values that store the protocol fees: `protocolFeeA` and `protocolFeeB`. These two values are held aside and only loaded on swaps if `protocolFeeRatio` $\neq 0$. Finally, the pool has two mappings: one that maps `binId` to the state of that bin called `bins` and another that maps the tick position to the tick state object for that tick position called `ticks`.

2.2 Ticks

Maverick v2 introduces a new element of internal accounting called a “tick.” Each tick can contain up to four “bins,” one of each bin type: Static, Right, Left, and Both. Each tick also stores the total amount of reserves contained in the constituent bins as well as the “total tick supply” of the four bins. This new tick storage mapping allows the AMM to avoid having to read four storage slots on each swap to find the aggregate liquidity in a tick. Instead, the tick maintains this aggregate liquidity amount and the bin objects have claim to their pro rata share of the tick.

The `TickState` struct has the following elements:

- `reserveA` - Quantity of quote tokens in all bins in this tick
- `reserveB` - Quantity of base tokens in all bins in this tick
- `totalSupply` - Quantity of bin “shares” that the bins in this tick collectively possess
- `binIdsByTick` - Four-element array of the binIds in this tick.

The tick width is configurable at pool creation. We define the tick position by its two edges that correspond to the lower (p_l) and upper (p_u) price of the tick. The tick width can be as small as 1 basis point so that $p_u = 1.0001 p_l$ and as big as $p_u = 1.0001^{10000} p_l$ where $1.0001^{10000} \approx 2.71$.

The swap invariant for a tick is

$$L^2 = \left(B + \frac{L}{\sqrt{p_u}}\right)\left(A + L\sqrt{p_l}\right), \tag{1}$$

where B is the amount of base token in the bin, A is the amount of quote token in the bin, and L is the amount of liquidity in the tick. For each swap, the AMM contract uses this invariant to compute the output value of the swap by ensuring the invariant still holds after the swap.

2.3 Bins

Bins exist in ticks. There are four types of bin: Static, Right, Left, and Both. At any given time there can only be one of each type of active bin in a tick. Movement-mode bins (Right, Left, Both) can move from one tick to another depending on the price and configuration of the AMM. To track bin movements, bins are stored as a linked list where the parent bin of the list is said to be "active" and all of the children bins are said to be "merged." Only the active bin will exist in the tick's `binIdsByTick` mapping.

As movement-mode bins move, they may move to a tick that already has a bin of the same type. When this happens, the tick with the lower `binId` will remain active, and the bin with the higher `binId` will merge into this active bin. The detailed mechanics of this merge operation are described later in the whitepaper.

2.3.1 Bin State

Each bin has the following state elements:

- `mergeBinBalance` - Quantity of `mergeId` bin LP tokens this bin has a claim to; `mergeId` is only $\neq 0$ when bin has been merged
- `tickBalance` - Quantity of this tick's `totalSupply` that this bin possesses
- `totalSupply` - Quantity of LP tokens that have been minted for this bin
- `kind` - A value indicating the bin kind type in the set Static, Right, Left, or Both
- `tick` - Signed value of the tick position defined as $\texttt{lowerTick} = \texttt{tickSpacing}\ \log_{1.0001}(p_l)$
- `mergeId` - ID of bin that this bin has merged in to
- `balances` - LP token balance for each Position NFT ID

2.3.2 Adding Liquidity to a Bin

Users specify the amount of liquidity to add in terms of bin LP balance they desire and the contract calculates the amount of each token type required to get that much liquidity. The contract will call the `maverickV2AddLiquidityCallback` function on the calling contract to collect the required token amounts.

User liquidity tracking is stored in the `balances` mapping by both the `msg.sender`'s address and a "subaccount" that the user specifies. This allows a user to segment their liquidity holdings into different subaccounts. For instance, the `MaverickV2Position` ERC-721 contract uses subaccounts to store liquidity for different tokenIds for a given user.

The bin LP balance calculation takes one of two forms depending on whether the tick that contains the bin has liquidity already. If the tick has a non-zero reserve amount in either token, then the required token amounts are computed as

$$\Delta A = \frac{\texttt{deltaLPBalance}}{\max\{1, TS_{bin}\}} \frac{\max\{1, \texttt{binTickBalance}\}}{\max\{1, TS_{tick}\}} A_{\texttt{tick}}, \qquad (2)$$

$$\Delta B = \frac{\texttt{deltaLPBalance}}{\max\{1, TS_{bin}\}} \frac{\max\{1, \texttt{binTickBalance}\}}{\max\{1, TS_{tick}\}} B_{\texttt{tick}}, \tag{3}$$

where TS_{tick} is the total supply of the tick and TS_{bin} is the total supply of the bin. If the tick has no reserves, then the required token amounts are

$$\Delta A = \begin{cases} \texttt{deltaLPBalance}\sqrt{\texttt{p}_\texttt{l}} & \texttt{tick} < \texttt{activeTick} \\ 0 & \texttt{tick} \geq \texttt{activeTick} \end{cases} \tag{4}$$

$$\Delta B = \begin{cases} 0 & \texttt{tick} < \texttt{activeTick} \\ \texttt{deltaLPBalance} \Big/ \sqrt{\texttt{p}_\texttt{u}} & \texttt{tick} \geq \texttt{activeTick}. \end{cases} \tag{5}$$

These values are derived by defining

$$\texttt{deltaLPBalance} \triangleq \Delta\mathrm{L}\left(\sqrt{1.0001^{\texttt{tickspacing}}} - 1\right), \tag{6}$$

and realizing that an empty tick will always have only single-sided liquidity on the first add call. That is, for an all-A tick, using (1) we have the relationship

$$\Delta A = \Delta L\left(\sqrt{p_u} - \sqrt{p_l}\right), \tag{7}$$

and for an all-B bin, we have

$$\Delta B = \Delta L \frac{\sqrt{p_u} - \sqrt{p_l}}{\sqrt{p_u}\sqrt{p_l}}. \tag{8}$$

That is, substituting (6) into (4) and (5) results in the relationships (7) and (8). The scaling factor of $\sqrt{1.0001^{\texttt{tickspacing}}} - 1$ helps maintain consistent token-amount to LP balance ratios regardless of the tick width of a pool.

When adding to a bin, we also need to compute and update the delta balances for that bin in its tick. The `deltaTickBalance` is computed as

$$\texttt{deltaTickBalance} = \texttt{deltaLPBalance} \frac{\max\{1, \texttt{binTickBalance}\}}{\max\{1, \mathrm{TS}_{\texttt{bin}}\}}. \tag{9}$$

These computed delta values are used by the AMM to update the following state variables:

- `bin.totalSupply` - Incremented by `deltaLPBalance`
- `bin.balances[user][subaccount]` - Incremented by `deltaLPBalance`
- `bin.tickBalance` - Incremented by `deltaTickBalance`
- `tick.totalSupply` - Incremented by `deltaTickBalance`
- `tick.reserveA` - Incremented by ΔA
- `tick.reserveB` - Incremented by ΔB

The one caveat to this process is that the first `MINIMUM LIQUIDITY` amount of liquidity added to a bin is donated to the bin as permanent liquidity by the first user to add liquidity to the bin. This donated liquidity inflates the `totalSupply` of the bin, implicitly locking that amount of LP balance in the bin. But the user still receives exactly the amount of LP balance they requested, which is important to make it easy to interoperate with the pool.

2.3.3 Moving and Merging Bins

Movement-mode bins (i.e., bins with a `kind` property of right, left, both) may move right or left with price as traders swap with the pool. When this happens, the `tick` value of the bin changes. The new `tick` value is set to the new tick position of the bin.

The merging procedure is similar to the procedure in Maverick v1, with the extra complication that bins exist in ticks and ticks store the reserve balances for all bins in a given tick. Having ticks does not alter the concept of the merge; it just requires that the AMM smart contract do an extra level of accounting as part of the merge to update both the bin and tick state variables.

For the sake of example, assume that the movement condition has been triggered in the rightward direction and we are merging/moving mode right bins. The details of the movement condition are discussed in the TWAP section below. The process for moving a bin under these conditions is as follows:

1. Search left and right from the tick that contained the TWAP prior to the swap. In each of these three ticks (tick - 1, tick, tick + 1) determine if a mode-right bin exists by checking whether `binIdsByTick[1]` is non-zero. There will be at most three `binIds` that result from this search.

2. Search this 3-`binId` list to find the smallest `binId`, which we denote `firstBinId`. This will be the `binId` of the active bin in this tick at the end of the procedure. The other bins will be merged.

3. Merge the list of bins by

   (a) Computing the `reserveA` and `reserveB` amounts in each of the bins to be merged. These reserve values are computed with

   $$\Delta A = \texttt{tick.reserveA}\frac{\texttt{bin.tickBalance}}{\texttt{TS}_{\texttt{tick}}} \tag{10}$$

   $$\Delta B = \texttt{tick.reserveB}\frac{\texttt{bin.tickBalance}}{\texttt{TS}_{\texttt{tick}}}. \tag{11}$$

   (b) Computing the `mergeBinBalance` by computing the bin LP balance each of the merged bins has claim to in the new parent bin. First determine whether to use ΔA or ΔB in the calculation by determining whether the tick is all quote or all base. For this example, assume the bin is all quote token. The next step is to compute the amount of the quote reserves the new parent bin currently possesses with

   $$A_{firstBin} = \texttt{firstBinTick.reserveA}\frac{\texttt{firstBin.tickBalance}}{\texttt{TS}_{\texttt{firstBinTick}}}. \tag{12}$$

   Finally, each merged bin's LP balance in the `firstBin` is

   $$\texttt{mergeBinBalance} = TS_{firstBin}\frac{\Delta A}{A_{firstBin}}. \tag{13}$$

    (c) Setting the new parent bin's `mergeBinBalance` to be the sum of the `mergeBinBalances` from the one or two bins being merged.

    (d) Incrementing `tick.reserveA` or `tick.reserveB` by the sum of the ΔA or ΔB values

    (e) Incrementing the `firstBin`'s `tickBalance` to account for these new reserves

    (f) Setting the `mergeId` of each bin being merged to `firstBinId`

    (g) Removing the bin from the tick by decrementing `tick.totalSupply` by `bin.tickBalance`, decrementing `tick.reserveA` by ΔA, decrementing `tick.reserveB` by ΔB, and setting `tickState.binIdsByTick[1]` (1 corresponds to mode right) to zero.

4. Move the new parent bin (`firstBinId`) to its new tick if needed. It may be that the parent bin is already in the desired ending tick. If not, the movement operation is similar to the merge operation where the contract

    (a) Computes the delta reserve values in the moving bin

    (b) Adds these reserves to the new tick by modifying the reserves and `tick.totalSupply` of the old and new tick

    (c) Updates the bin's `tickBalance` value to be in units of the new tick

    (d) Updates the bin's `tick` value to be the new tick

The result of this process is that the merged bin now holds an LP token balance of `mergeBinBalance` in the active bin, but the LP `balances`, which track the LP balance of each user in each bin, have remained the same. The active bin now has a `totalSupply` of LP tokens equal to its original `balance` and the `mergeBinBalance` that the merged bin now holds in it.

Tracking the balances this way makes merges computationally tractable on chain, because it means that the contract does not have to iterate through all LPs' positions to update balances on a merge. Instead, as discussed in the next section, the LPs still claim their LP tokens on the original (now merged) bin, and the contract recursively traverses merged bins to get to the active bin where the reserves are all held.

2.3.4 Removing Liquidity

An LP can remove liquidity from a bin by specifying their subaccount, the bin's `binId`, and the `amount` of LP balance they want to remove from that bin.

For an `active` bin, the process is straightforward. The contract checks to make sure the Position has at least the amount of LP balance the user is trying to withdraw. If the balance is sufficient, the bin disburses a pro rata amount of reserve out back to the user:

$$A_{out} = \frac{\texttt{amount}}{TS_{bin}} \alpha \texttt{reserveA}, \tag{14}$$

$$B_{out} = \frac{\texttt{amount}}{TS_{bin}} \alpha \texttt{reserveB}, \tag{15}$$

where

$$\alpha = \frac{\texttt{bin.tickBalance}}{TS_{tick}}. \tag{16}$$

For a merged bin, the process is similar, but the calculation is recursive. Say an LP in the merged bin, binId = i_m, wants to remove amount of their LP balance from that now-merged bin. That merged bin no longer has any reserves directly associated with it. But it does possess an LP balance in the active bin with binId = i_a and those balances correspond to reserve amounts,

$$A_{i_a} = \alpha \frac{\texttt{mergeBinBalance}_\texttt{m}}{\texttt{totalSupply}_\texttt{a}} \texttt{reserveA}_\texttt{a}, \tag{17}$$

$$B_{i_a} = \alpha \frac{\texttt{mergeBinBalance}_\texttt{m}}{\texttt{totalSupply}_\texttt{a}} \texttt{reserveB}_\texttt{a}. \tag{18}$$

The LP is removing amount from bin binId = i_m. So the net amount the user will receive is

$$A_{out} = \alpha \frac{\texttt{amount}}{\texttt{totalSupply}_\texttt{m}} \frac{\texttt{mergeBinBalance}_\texttt{m}}{\texttt{totalSupply}_\texttt{a}} \texttt{reserveA}_\texttt{a}, \tag{19}$$

$$B_{out} = \alpha \frac{\texttt{amount}}{\texttt{totalSupply}_\texttt{m}} \frac{\texttt{mergeBinBalance}_\texttt{m}}{\texttt{totalSupply}_\texttt{a}} \texttt{reserveB}_\texttt{a}. \tag{20}$$

A user must first call migrateBinsUpStack for any binIds that the user wants to remove that are merged more than one level deep. The migrate function will move the bins up the linked list of merged bins until each migrated bin is pointing to an active bin. The requirements for this migration operation is that users should maintain the same pro-rata claim to the parent bin's LP balance before and after migration. As part of that function, mergeBinBalance of the bins being migrated will be updated along with the balances of the bin that was the head bin, i.e., balances[0] of the old head bin.

2.3.5 Migrate Bins Up Merge Stack

To migrate a bin up the merge stack, consider a merged bin with binId = i_1 and another merged bin with binId = i_2 that is merged into bin i_1. The active bin, with binId = i_a, has bin i_1 as a child so that the full list of bins in this example is $i_a \leftarrow i_1 \leftarrow i_2$. After the migration process, i_2 will move up the list so that both i_1 and i_2 are direct children of i_a.

Prior to migration, bin i_2 has the following implicit claim to the totalSupply in bin i_a:

$$claim_{i_a \leftarrow i_2} = \frac{\texttt{mergeBinBalance}_2}{\texttt{totalSupply}_1} \frac{\texttt{mergeBinBalance}_1}{\texttt{totalSupply}_\texttt{a}}. \tag{21}$$

Also,

$$\texttt{balance}_\texttt{a}[0] = \texttt{mergeBinBalance}_1. \tag{22}$$

Note that the mergeBinBalance of a given bin is in the unit of totalSupply or balance of the bin's parent. If the bin has no parent, then there is no mergeBinBalance because the bin has no balance ownership of another bin.

The migration process is:

1. Subtract $\texttt{mergeBinBalance}_2$ from the $\texttt{totalSupply}_1$ of bin i_1.

$$\texttt{totalSupply}'_1 = \texttt{totalSupply}_1 - \texttt{mergeBinBalance}_2. \tag{23}$$

2. Find the new $\texttt{mergeBinBalance}'_2$ of bin i_2 which is its balance in bin i_a. Note bin i_a and bin i_1 have different units of `totalSupply` which is why we need to transform the `mergeBinBalance` in the migration process.

$$\texttt{mergeBinBalance}'_2 = \frac{\texttt{mergeBinBalance}_2}{\texttt{totalSupply}_1}\texttt{mergeBinBalance}_1. \tag{24}$$

3. Subtract the $\texttt{mergeBinBalance}'_2$ from the $\texttt{mergeBinBalance}_1$ of bin i_1.

$$\texttt{mergeBinBalance}'_1 = \texttt{mergeBinBalance}_1 - \texttt{mergeBinBalance}'_2. \tag{25}$$

4. Update the `mergeId` in bin i_2 from i_1 to i_a.

After migration, bin i_2 now has a claim to the supply in i_a that is

$$claim_{i_a \leftarrow i_2} = \frac{\texttt{mergeBinBalance}'_2}{\texttt{totalSupply}_\texttt{a}}. \tag{26}$$

$$claim_{i_a \leftarrow i_2} = \frac{\texttt{mergeBinBalance}_2}{\texttt{totalSupply}_1}\frac{\texttt{mergeBinBalance}_1}{\texttt{totalSupply}_\texttt{a}}, \tag{27}$$

which is consistent with the initial condition in (21). Likewise, post migration, the `mergeBinBalance` in bin i_1 has been reduced by the amount of merged bin LP balance that got migrated away from this bin and up to its parent, i_a such that

$$\texttt{balance}_\texttt{a}[0] = \texttt{mergeBinBalance}'_1 + \texttt{mergeBinBalance}'_2, \tag{28}$$

which is consistent with (22). In this process, because of how the balance tracking is structured, there was no need to modify any of the state of bin i_a. This process is repeated recursively from bottom to active bin when the merge stack has more than three bins. As a note, once a bin is eligible for migration, it is not active any longer and will not have any more bins merge into it.

The AMM code contains rounding decisions for any quotient operations. As part of this migration, the AMM rounds the multiplication-division in (24) down, which has the effect of rounding up $\texttt{mergeBinBalance}'_1$ in (26). The result of this design decision is that when a bin has two or more merged children bins, the bins merged later get a one-unit rounding boost in their claim to the `mergeBinBalance`.

2.4 Swapping

2.4.1 Callback Mechanics

Swapping is callback-based, which allows users to “flash swap” such that they can collect the proceeds of their swap before they have to transmit what they owe to the contract. Users can swap by specifying either the exact amount they want to receive of a given token or the exact amount they want to swap in.

The contract will disburse the proceeds to the user. In calling a swap on the pool, the user had to provide a callback function. To pay the contract what it is owed for the swap, the contract will call the user-provided callback, which will need to transmit the user's tokens to the contract in order for the transaction to complete.

2.4.2 Swapping in a Bin

For any given position in price, there can be up to four active bins, but the changes in accounting in v2 mean that the contract does not need to pull reserves from four bins because the reserves in a tick are stored at the tick level. It also means that, post-swap, the amounts do not need to be written back to each of the four bins individually.

The process is as follows:

1. Pull the reserves A and B from the active tick.
2. Compute the aggregate liquidity by solving this quadratic for L

$$0=\left(\sqrt{\frac{p_l}{p_u}}-1\right)L^2+\left(\frac{A}{\sqrt{p_u}}+B\sqrt{p_l}\right)L+AB, \tag{29}$$

which leads to

$$L=\frac{\sqrt{p_u}}{\sqrt{p_u}-\sqrt{p_l}}\left(\frac{b}{2}+\sqrt{\frac{b^2}{4}+\frac{AB\left(\sqrt{p_u}-\sqrt{p_l}\right)}{\sqrt{p_u}}}\right), \tag{30}$$

where $b \triangleq A\big/\sqrt{p_u}+B\sqrt{p_l}$.

3. Compute sqrt price, $\sqrt{P}$

$$\sqrt{P}=\frac{A+L\sqrt{p_l}}{B+L/\sqrt{p_u}} \tag{31}$$

4. Extract fee from the token balance coming in and set that aside as either A_{fee} or B_{fee}
5. Extract the protocol fee from the total fee

$$A_{protocol}=A_{fee}\cdot\texttt{state.protocolFeeRatio} \tag{32}$$

$$B_{protocol}=B_{fee}\cdot\texttt{state.protocolFeeRatio} \tag{33}$$

6. Use the identities to compute ΔA and ΔB

$$\Delta\sqrt{P}=\frac{\Delta A}{L} \tag{34}$$

$$\Delta\frac{1}{\sqrt{P}}=\frac{\Delta B}{L} \tag{35}$$

7. Update the reserves of the tick

$$A_{new} = A + \left(\Delta A + A_{fee} - A_{protocol}\right) \tag{36}$$

$$B_{new} = B + \left(\Delta B + B_{fee} - B_{protocol}\right) \tag{37}$$

8. Update pool reserve values `state.reserveA` and `state.reserveB`
9. Update pool protocol fee values `state.protocolFeeA` and `state.protocolFeeB`
10. Update TWAP with the new ending price of the swap

2.4.3 Rounding

The contract is designed to round appropriately in order to remain solvent with respect to the pool's balance according to `tokenA` and `tokenB` ERC-20 contracts. In particular, the contract must ensure that the token balances according to the ERC-20 contracts are always greater than or equal to the sum of the tick's reserves.

$$\sum_i \texttt{reserveA}_i \le ERC_A - \texttt{state.protocolFeeA} \tag{38}$$

$$\sum_i \texttt{reserveB}_i \le ERC_B - \texttt{state.protocolFeeB} \tag{39}$$

2.4.4 Swapping Through Ticks

A swap may be large enough that it will swap an entire bin. If this happens, the swap-in-a-bin process described above will be repeated again for the next adjacent bin set with any remaining assets that are still to be swapped.

2.5 Moving Bins and TWAP

A pool's time-weighted average price (TWAP) is computed recursively using an autoregressive averaging process. "Price" in this context is actually the $\log_{1.0001} price$ value which is the price value in the tick domain. That is, we can think of TWAP being a moving average of the log price or of the fractional tick position of the pool. Each TWAP update changes the following three pool state values: `state.lastTwa`, `state.lastTimestamp`, `state.lastLogPrice`.

From those three state values and the current block timestamp, t, the current TWAP is calculated as the weighted average of `state.lastLogPrice` and `state.lastTwa` such that if zero time has passed since the last reading, the TWAP will be `state.lastTwa` while if more than `lookback` time has passed, TWAP will be weighted in the direction of `state.lastLogPrice`.

Mathematically, we compute TWAP by first precomputing two values, Δp, the deviation between `state.lastLogPrice` and `state.lastTwa`, and Δt, the deviation between `state.lastTimestamp` and the timestamp:

$$\overline{\Delta t} \triangleq \min\left\{\texttt{lookback},\ t - \texttt{lastTimestamp}\right\}, \tag{40}$$

$$\Delta t \triangleq \begin{cases} 0 & \overline{\Delta t} < \texttt{MIN INTERVAL} \\ \overline{\Delta t} & \overline{\Delta t} \geq \texttt{MIN INTERVAL}, \end{cases} \tag{41}$$

$$\overline{\Delta p} \triangleq \min\left\{1,\ |\texttt{lastLogPrice} - \texttt{lastTwa}|\right\}. \tag{42}$$

$$\Delta p \triangleq \begin{cases} -\overline{\Delta p} & \texttt{lastLogPrice} < \texttt{lastTwa} \\ \overline{\Delta p} & \texttt{lastLogPrice} \geq \texttt{lastTwa}, \end{cases} \tag{43}$$

From these equations, the following bounds hold:

$$-1 \leq \Delta p \leq 1 \tag{44}$$

and

$$0 \leq \Delta t \leq \texttt{lookback}. \tag{45}$$

These deviation values combine to compute the current TWAP via

$$TWAP = \Delta p \frac{\Delta t}{\texttt{lookback}} + \texttt{lastTwa}. \tag{46}$$

It is clear from the bounds that TWAP moves at most one tick in a given update and this max movement only occurs if `lookback` seconds have passed since the last swap and the new swap price is at least a tick away from the `lastTwa` value. Likewise, in the case where N swaps total happen at the minimum spacing of `MIN INTERVAL` seconds and all swaps are more than one tick away from the starting `lastTwa`, then the maximum movement is

$$TWAP = \texttt{lastTwa}_{starting} + \sum_{i=1}^{N} \frac{\texttt{MIN INTERVAL}}{\texttt{lookback}}. \tag{47}$$

From this equation, it is clear that the TWAP moves at most $\Delta t / \texttt{lookback}$ for any interval Δt, which means that the maximum movement is the same whether one swap happens at a large time spacing or many swaps happen on smaller time intervals.

The contract tracks the TWAP of the pool by registering the price of the pool at the end of the first swap in each block if `MIN INTERVAL` seconds has passed since the last swap. If less than `MIN INTERVAL` seconds has passed, then none of the TWAP state parameters are updated.

After a swap, the contract checks to see if any bins need to be moved. If so, then the move proceeds. Within a block, no time passes between operations, so the TWAP will also not change for the duration of the block. Because of this, no bins will move beyond the first swap in a block, as any subsequent checks for movement will find the bins already in line with the TWAP. These mechanisms mean that a swapper cannot move liquidity using a swap inside of a single block. This makes the movement robust to large inner-block flash swap operations that may significantly move the price.

For example, in the case of a large two-step flash swap that moved the price up and then back down inside the block, none of the dynamic liquidity bins would move in response and the TWAP would be unaffected by the large price excursion. For liquidity to move, a swapper would have to leave their capital on chain for at least one block period, which would leave that liquidity exposed to arbitrageurs, thereby discouraging any such toxic liquidity movement manipulations. Moverover, for a swapper to move the liquidity more than one tick requires that `lookback` seconds have passed per tick of movement. A common `lookback` value in Maverick `v1` was 3 hours, which is a considerable amount of time for a would-be pool manipulator to try a price manipulation attack.

Notation:

- t_c — `lowerTick` of the bin that contains the current price
- t_p — `lowerTick` of the bin that contains the previous price
- $t_{twap,c}$ — `lowerTick` of the bin that contains the current TWAP
- $t_{twap,p}$ — `lowerTick` of the bin that contains the previous TWAP
- $t_{bin,c}$ — `lowerTick` of the moving bin after the move
- $t_{bin,p}$ — `lowerTick` of the moving bin before the move

The movement conditions are the same as Maverick `v1`:

- If $t_c == t_p$ and $t_{twap,c} == t_{twap,p}$ no bins move
- Only bins within one tick of price or exactly the previous TWAP will move; other bins stay "stranded" until the price moves within one bin of their position
- The target right-most tick where `RIGHT` or `BOTH` bins will move to is $\text{target}_{right} = \min\{t_c - 1, t_{twap,c}\}$
- All `RIGHT` and `BOTH` bins from $\min\{t_p - 1, t_{twap,p}\}$ to $\text{target}_{right} - 1$ will be moved to target_{right}
- The target left-most tick where `LEFT` or `BOTH` bins will move to is $\text{target}_{left} = \max\{t_c + 1, t_{twap,c}\}$
- All `LEFT` and `BOTH` bins from $\max\{t_p + 1, t_{twap,p}\}$ down to $\text{target}_{left} + 1$ will be moved left to target_{left}.

2.6 Programmable Pools

When creating a pool, users have an option to specify that the pool be programmable so that only a single "accessor" address can call the pool's write functions. This allows for pools to be wrapped in manager contracts that provide custom modifications to a Maverick pool's operation.

For instance, a protocol may have a need for a KYC-based pool. This is simple with Maverick v2 pools. The process would be first to create a KYC wrapper contract that allows for only a whitelisted set of users access to the wrapper functions. Second, deploy a programmable Maverick v2 pool with that KYC wrapper contract as the `accessor`. Then, KYC users would interface with the wrapper contract, which would check their permissions and then pass on actions to the programmable pool if the KYC user has the proper credentials.

Programmable pools also have `setFee` function that lets the accessor address modify the fee of the pool. This is a useful feature for protocols wanting to build dynamic fee adjustment algorithms that optimize the LP fee performance.

Finally, other mechanisms like limit order, one-way LPing (only buying or selling), or even unforeseen novel mechanisms can be implemented as wrapper contracts to provide new user flexibility.

3 Liquidity Management and Staking

MaverickV2Pool contracts support minting and removing liquidity to `msg.sender`, to enable transferring liquidity positions and general management. Maverick v2 has two paradigms available for LPs. Users can choose to create non-fungible tokens that hold pool liquidity using the `MaverickV2Position` contract which has ERC-721 NFT interfaces. Alternatively, users can hold pool liquidity using the `MaverickV2BoostedPosition` contract which makes liquidity positions fungible through an ERC-20 interface where users can mint, transfer, and burn BP tokens that represent a pro rata share in the BP's aggregate liquidity.

By providing both ERC-20 and ERC-721 representations of pool liquidity, Maverick v2 gives protocol developers the flexibility to leverage Maverick positions as a fundamental liquidity building block.

3.1 Boosted Positions

A Boosted Position (BP) is a distribution of liquidity within a Maverick pool, consisting of one or more bins and with a liquidity mode selected. Any user can create a BP by cloning an existing liquidity distribution and adding liquidity. The key differences between a BP and a normal liquidity position are as follows:

- BP tokens can be staked in a reward contract; token incentives can be permissionlessly added to this contract and will be distributed over time to LPs in the BP
- Any user can join a BP by adding liquidity in the correct ratio to the current distribution; as an LP in the BP they will receive a share of any token incentives sent to the BP's reward contract
- An LP's share of a BP is represented using an ERC-20 token specific to that BP; this LP token must be staked into the BP's reward contract to receive token incentives

BPs represent a powerful tool for projects and other users to shape token liquidity by incentivizing specific distributions within a pool. Instead of overspending incentives to attract liquidity to every part of a pool, projects can use BPs to target narrow areas using custom distributions and Maverick AMM's movement modes.

While the primary use case for BP ERC-20 tokens is staking into the reward contract to receive token incentives, Maverick offers the option to leave these tokens unstaked in the anticipation of other use-cases being discovered for them as a liquidity primitive.

3.2 Boosted Positions Technical Details

`MaverickV2BoostedPosition` contracts are deployed from the `MaverickV2BoostedPositionFactory` contract and are defined by an array of pool `binIds` and bin LP balance `ratios` that determine the distribution of the liquidity in the BP. A BP is only composed of one kind of bin at a time. BPs made up of $kind = 0$ bins can have up to 24 `binIds`, while BPs of any of the movement modes are limited to a single `binId`. BPs have an ERC-20 interface and tokens are minted for LPs which represent their share of the BP. As swappers swap in the pool of the BP, fees accumulate in the BP's bin(s) and grow the amount of reserves that each BP LP is entitled to.

To join a BP, an LP must add liquidity to a pool with the BP contract address as the recipient such that the bin LP balance ratio of the pool liquidity matches the ratios in the BP's `getRatios()` function. At that point, the user needs to call `mint` on the BP, and the BP will compare the bin LP balance it has stored to the current BP LP balance. That comparison will show that the BP now has more bin LP balance and the contract will mint BP LP tokens to the user.

3.3 Reward Contract Technical Details

BPs can be permissionlessly incentivized through the `MaverickV2Reward` contract. As part of the contract deployment, the deploying user decides the BP `stakingToken` as well as up to five reward tokens.

Incentivizers can permissionlessly add reward tokens of any of the five defined kinds, which will then be disbursed to stakers. BP holders can stake their BP balance in the reward contract by calling the `stake()` function which will transfer the BP tokens from the user to the reward contract.

A rewards stake is not fungible and not transferable, but the reward contract does have an ERC-20 interface for accounting for stake balances through `balanceOfaddress` and `totalSupply()`. As time passes, each staker accumulates rewards in all of the incentivized reward tokens, which can be redeemed by calling `getReward()`.

3.3.1 Reward Accrual

Internally, the reward contract is tracking the following key quantities:

- `rewardPerTokenStaked` is the cumulative amount of reward token per unit of stake minted. This cumulative is tracked per token and is updated every time a user stakes or an incentivizer adds new incentives.

- `escrowedReward` is the amount of unclaimed reward available. This is reward that has been accrued to stakers but that they have yet to claim through `getReward()`.
- `rewardRate` is the amount of reward for a given token that is accrued to all stakers every second.
- `updatedAt` is a timestamp in seconds that is the minimum of the reward finish time and the last time the reward was updated by either a staker staking or an incentive add.
- `finishAt` is a timestamp in seconds when the current rewards accrual stops accruing.

At each update (either a stake or incentive add), the global reward state for each reward token is updated by

1. computing the amount of rewards per token that have been accrued since the last update
2. updating `rewardPerTokenStaked` by adding this newly accrued amount to it
3. updating `escrowedReward` by adding the product of the `totalSupply` and the newly accrued rate to it.

On stake, the tracking for the staker is also updated. In particular, two values are updated: 1) `rewards[account]` is updated with the amount of reward this staker has accrued since their last update, which is computed by `balanceOf(staker)` · (`rewardPerTokenStored` − `userRewardPerTokenPaid[account]`) and 2) `rewardPerTokenStaked` is checkpointed as `userRewardPerTokenPaid[account]`. At any time, this same process can be used to compute the amount of reward a staker has accrued.

3.3.2 Reward Incentive Boosting

In addition to specifying up to five reward tokens, the reward contract creator can also specify a voting escrow token to correspond with as many of the reward tokens as they choose. By doing this, the reward disbursement for that reward token changes in two ways: 1) the stakers have the option of staking their collected rewards into the voting escrow contract to receive a reward boost and 2) the stakers can get a further reward boost that is based on their voting power in the voting escrow contract.

The formulas for computing the output reward amount for the staker are

$$reward_{out} = reward_{max} \cdot boost_{staking} \cdot boost_{ve}, \tag{48}$$

where

$$boost_{staking} = 0.25 + \frac{stakeDuration}{fourYears} \cdot 0.75 \tag{49}$$

$$boost_{ve} = 0.75 + \min\left(1, \frac{proRata_{ve}}{proRata_{stake}}\right) \cdot 0.25 \tag{50}$$

and

$$proRata_{ve} = \frac{\texttt{balanceOf}_{ve}(staker)}{\texttt{totalSupply}_{ve}} \tag{51}$$

$$proRata_{staking} = \frac{\texttt{balanceOf}_{staking}(staker)}{\texttt{totalSupply}_{staking}}. \tag{52}$$

This boosting process is completely optional: anyone can permissionlessly create a reward contract and specify whether or not the reward tokens accrual will be subject to the boosting mechanism or not. The advantage of this mechanism is that, when coupled with the voting escrow factory, protocols can create their own voting escrow token flywheel to encourage sticky protocol engagement.

3.3.3 Unboosted Token Disbursement

When a staker does not maximize their boost, the remaining tokens are tracked in a variable called `unboostedAmount`. This amount will accumulate as time passes and is no longer subject to disbursement to reward stakers. Instead, this set-aside token amount will be pushed to the rewards mechanism in the ve token when any user permissionlessly calls `pushUnboostedToVe()` on the reward contract. The details of how this amount is disbursed to ve holder is detailed in Section 4.

3.3.4 Incentive Adding

Any user can permissionlessly add incentives to a reward contract in any of the reward tokens with which that reward contract was initialized. When adding incentives, the incentive adder chooses both the amount and duration of the incentive disbursement period. However, the reward rate for each reward token is fixed to a single number so there is no support for multiple disbursement periods to be running simultaneously.

Instead, when a user wishes to add incentives for a given reward token while that reward token is already actively accruing, there are two possibilities for how the duration and, thereby, the reward rate, are set. If the user is bringing more reward tokens than are currently awaiting disbursement, then the duration specified by the user is the new duration for all remaining rewards in the given reward token; that is, the reward rate set by some previous incentive adder is overruled and this new incentive adder is allowed to set a new rate.

If, instead, the new incentive adder brings fewer reward tokens than are awaiting disbursement, then the rate remains unchanged such that the new reward amount is “box-car” added to the end of the existing duration, thereby extending the reward period at the previous reward rate.

4 Voting Escrow Factory

Voting Escrow (ve) models are a common mechanism for 1) gathering community feedback and 2) distributing protocol proceeds. One challenge is that it is hard for new protocol teams to spin up the smart contract infrastructure and integrations required to successfully create a ve flywheel that directs liquidity. Instead, protocols are reduced to offering simple LP incentives to attract liquidity onto DEXs.

One advantage that protocols lose with simple incentives is that the incentives that get emitted have no stickiness and do not require any long-term investment on the part of the LP in the protocol. Conversely, ve staking aligns protocol and user incentives by creating a staking period.

Maverick v2 addresses these pain points by creating a factory contract that any user or protocol can permissionlessly use to create both 1) a ve contract for a given ERC-20 token and 2) BPs that emit the reward token, but that–importantly–also incentivize BP LPs to stake these emissions in the ve contract. This allows protocols to build an incentive-directing system where the stakers are given a boost if they stake their rewards in the ve contract of the protocol.

Let us illustrate the process using a hypothetical ABC protocol that has an `ABC` ERC-20 governance token. In Maverick v1, ABC protocol can choose to send `ABC` tokens to a `ABC-WETH` BP. This will result in new liquidity being LPed into that BP, providing liquidity for the token.

This process works fine and is effective, but Maverick v2 brings several new dimensions of flexibility through the ve factory and incentive directing system. In Maverick v2, ABC protocol can improve their flywheel as follows:

- Use the Maverick ve Factory to create a `veABC` token. This is a non-transferable ERC-20-Votes compatible token that holds staked `ABC` tokens. The voting power of a `veABC` holder depends on both 1) how long they have staked their `ABC` and 2) how much `ABC` they staked.

- ABC protocol can now create Maverick BP reward contracts that have two improvements over v1:

    1. ABC protocol can utilize the boost mechanisms described in Section 3.3.2 for the BP so that users with more `veABC` are entitled to more `ABC` emissions when they LP.

    2. ABC protocol can configure the BP to give a boost to `ABC` emissions that are immediately staked in as `veABC`.

- Instead of operating as an external user of Maverick BPs, ABC protocol now takes on the role of matcher in their incentive-directing system. ABC protocol can manage the matching and vote-matching budgets for each epoch in the ve cycle and choose whether to use veABC voting to add a further boost to their BPs. If the vote-matching budget is left empty, there will no boost to emissions based on veABC voting, essentially turning off this part of the flywheel. For a detailed explanation of the matching and vote-matching system, see Section 4.1.4 below.

- veABC can also be used in the ABC protocol's governance infrastructure to gather community feedback and direct rewards.

The result is that Maverick has democratized the liquidity directing flexibility that used to only be available to DEXs. In this sense, Maverick v2 acts as a sort of DeFi liquidity operating system and becomes a fundamental primitive that protocols on new chains can use to gather users and shape their liquidity.

4.1 Incentive Directing Through Matching

The MaverickV2IncentiveMatcher contract provides a mechanism for protocols to match and boost incentives in a reward contract based on ve-user voting. An incentive matcher contract can be permissionlessly deployed for each ve token created. It provides an opinionated framework that allows any protocol to easily provide matching and vote-based incentives to a given BP and its reward contract.

4.1.1 Types of Incentives

The incentive-directing system that the matcher contract implements incorporates three types of incentives:

- **External incentives**: token incentives added directly to a BP's reward contract at the start of an epoch, which form the basis for calculating matching with other incentives; to qualify, these incentives must be in the form of the ve system's base token

- **Matching incentives**: token incentives added to the BP's reward contract, intended as a straight match to external incentives received by that BP

- **Vote-matched incentives**: token incentives added to the BP's reward contract as a function of external incentives and ve vote received by that BP in this epoch

Matching and vote-matched incentives are collectively referred to as "emissions" in this whitepaper.

4.1.2 Users

Three parties that interact with the incentive matcher contract:

- **BP Incentive Adders**: a protocol or liquidity shaper who adds incentives to a reward contract

- **Matching Budget Adders**: a protocol who wishes to match the incentives that the BP Incentive Adders have added

- **Voters**: ve token holders who can vote to direct more of the matching budget to given BPs through their reward contracts

These users interact with the matching contract on an epoch-based cadence.

4.1.3 Epoch Cadence

Each epoch consists of three stages:

1. **External incentives and veMAV voting** (14 days): during this period, BP Incentive Adders can add MAV incentives to any BP to qualify for matching and vote-matched incentives from emissions; voters with a veMAV balance can vote for BPs to direct vote-matched incentives to them

2. **Veto period** (2 days): during this period, any wallet providing matching budget for this epoch can veto a BP from receiving that matcher's portion of the matching/vote-matched funds; this is designed to prevent bad actors from creating BPs using non-transferable tokens and rewarding themselves with emissions. The veto period starts directly after the voting period ends.

3. **Emissions period** (variable number of days): at the start of this period, matching and vote-matched funds are sent to the reward contract just like external incentives and are disbursed to the reward contract stakers as described in Section 3.3. This period is initiated by any user permissionlessly calling `distribute()` on the matcher contract.

Each new epoch begins as the first stage of the previous epoch ends; e.g., stage 1 of epoch 2 begins when stage 1 of epoch 1 ends.

4.1.4 Calculating Incentives

For each epoch, the matched contract tracks two budgets: a matching budget and a vote-matched budget. These will be used as the basis for calculating the respective types of incentives to be emitted to each BP.

The matching budget forms the basis of matching incentives. At the end of stage 1 of an epoch, the total external incentives received by any given BP is calculated, and used as the basis of computing its matching incentives. So long as the total external incentives received in the epoch does not exceed the total matching budget, the reward contract for each BP will receive a 1:1 match to external incentives from the matching budget.

For example, if a rewards contract received 100 MAV of external incentives in an epoch and the total matching budget of MAV on the contract was not exceeded, 100 MAV of matching incentives will be sent to that BP's reward contract at the start of stage 3 of the epoch. If the total external incentives received in the epoch exceeds the total matching budget, each BP with external incentives will receive matching incentives on a pro rata basis.

Mathematically, if the ith reward contract received $incentives_{raw,i}$, then the matching incentive amount is

$$incentive_{match,i} = \begin{cases} incentive_{raw,i} & \sum_k incetives_{raw,k} \leq budget_{match} \\ \frac{incentive_{raw,i}}{\sum_k incetives_{raw,k}} & \sum_k incetives_{raw,k} > budget_{match} \end{cases} . \quad (53)$$

The vote-matched budget forms the basis of vote-matched incentives. These incentives for the ith reward contract are calculated at the end of stage 1 of an epoch using the formula

$$incentive_{vote,i} = \frac{vote_i}{\sum_k vote_k}\frac{incentive_{raw,i}}{\sum_k incetives_{raw,k}} budget_{vote}. \tag{54}$$

Example Assume an epoch with a matching budget of 1,000 MAV and a vote-matched budget of 2,000 MAV. In this hypothetical epoch only 600 "raw" incentives have been added to two active reward contracts. The results is

- Reward contract A receives 100 MAV in external incentives, and 50% of the veMAV vote. The match amount is 100 MAV, and the vote match amount is $1/6 * 1/2 * 2000 = 166.66$.
- Reward contract B receives 500 MAV in external incentives, and 25% of the veMAV vote. The match amount is 500 MAV, and the vote match amount is $5/6 * 1/4 * 2000 = 416.67$.

4.1.5 Incentive Matcher Technical Details

Voting. Voting in each epoch is tracked by the voter's ve delegate balance at the start of the ve epoch. Specifically, the matcher contract calls `getPastVotes(voter)` on the ve contract and tracks this as the user's vote. A user can vote only once per epoch.

Vetoing. There is a two-day period after the voting period when incentive match budget providers can choose to veto a given reward contract. The vetoing user only has the power to veto their portion of the match budget.

Distributing Matched Rewards. After the veto period ends, any user can permissionlessly distribute the match incentives to one or more reward contracts. Through that process, the matcher contract adds the incentives to the reward contract as if it was any other incentive. As described in Section 3.3.4, when new incentives are added, the resulting duration depends on the amount of incentives yet to be disbursed and the new amount of incentives being added. The matcher contract adds incentives with a duration of 14 days specified, but, depending on the existing incentives, the actual disbursement time of the matched incentive will vary.

Rolling Over Excess Budget. At the end of an epoch, there may be unused match budget and it is likely that there will be unused vote-match budget. The matcher contract provides the `rolloverExcessBudget()` function that budget providers can call to move any unused budget from one epoch to another. This function should only be called after all matches from the epoch have been distributed. Any undistributed incentives from a given epoch will cause the rollover function to underestimate the rollover amount and will lock this excess amount in the contract with no mechanism for retrieval.

4.2 Voting Escrow Technical Details

The ve contract created by the ve factory has the following features:

- Exponential-Based Voting Power - Unlike the Curve or Velodrome ve systems where voting power decreases over time, the Maverick v2 ve vote tracking leaves the voting power unchanged for a given stake. Instead, an exponential function is used to give more recent and longer-duration voters more voting power. The result is a more robust voting system that is easier to integrate with other smart contracts.

  The voting power, v a given user get for staking x tokens is

  $$v = x \cdot 1.5^{\frac{d+t-\texttt{startingTimestamp}}{oneYear}}, \tag{55}$$

  where `startingTimestamp` is the timestamp when the ve contract was deployed, and the user stakes for duration d at time t.

- ERC20Votes-Based Delegation - Any ve holder can delegate their vote to any address. They can also unilaterally change delegation at any time to delegate back to themselves or another party.

- Past Voting-Power Tracking - The ve contract exposes `getPastVotes` and `getPastTotalSupply` functions, which make it compatible with smart-contract-based on-chain governance systems. The past timepoints are specified by timestamp instead of blocknumber, making this contract versatile across chains that may have variable block times.

- Past Balance Tracking - The ve contract has a `getPastBalanceOf` function that allows for historical retrieval of a user's balance as separate from their voting power. This allows for external contracts to reward users based on their balance at a given timestamp. ERC20Votes only track past voting power, which is impacted by vote delegation.

- Time-Point Based Reward Disbursement - Users can permissionlessly create an "incentive batch," which is a bounty of tokens paid to ve holders pro rata of their balance at a given timepoint. Incentive batch creators specify 1) the token of the incentive, 2) the amount, and 3) if the token is the base token of the ve contract, the duration that the token must be staked. This creates a very flexible system to reward ve holders in arbitrary tokens.

- Stake Management - Individual ve stakes can be managed separately, making it possible to have a ladder of stakes that end at different times. The ve contract also exposes `merge` and `extend` functions to give users a flexible set of tools to manage their stakes.

- Third-Party Management of Stakes - The ve contract has a mechanism through `approveExtender` to allow stakers to provide access to a third party who can manage their stake extensions.

- Backward Compatibility with Maverick v1 veMav - On chains where veMav is already deployed, the new v2 veMav contract will have a mechanism to `sync` the legacyVeMav balance with the new contract. Any caller can permissionlessly synchronize the voting power across the two contracts with a call to the `sync` function.