An Overview of Uniswap v4 for Researchers*

Brad Bachu† Joel Hasbrouck‡ Fahad Saleh§ Xin Wan¶

Uniswap Labs *NYU Stern* *University of Florida* *Uniswap Labs*

Abstract

This article provides an overview of the Uniswap v4 protocol, highlighting its key innovations. Each v4 liquidity pool is defined by five components: two assets, a fee level, tick spacing, and a new feature called a hook. While the first four components are inherited from previous versions, the hook introduces novel customization capabilities. We explore the technical definition of hooks and how they enable tailored liquidity pool designs, such as dynamic trading fees, novel pricing rules, and new order types. Importantly, while hooks expand the flexibility of liquidity pool design, they also come with constraints. We discuss the boundaries of hook implementation and their implications for liquidity management.

Keywords: *Uniswap v4, Hooks, Liquidity Pools, Automated Market Makers, AMMs, Decentralized Exchanges, DEXs.*

JEL Classification: *G10, G23, G28*

*We thank Sara, Alice, Haardick and Sauce for valuable discussions.

†email: brad.bachu@uniswap.org

‡email: jh4@stern.nyu.edu

§email: fahad.saleh@ufl.edu

¶email: xin.wan@uniswap.org

This article examines the structure of Uniswap v4, highlighting its evolution from earlier versions and clarifying how the design space for specifying market mechanisms within v4. In Section 1, we introduce the core components that define a v4 liquidity pool, focusing on how v4 differs from v2 and v3. Section 2 then discusses hooks, a new feature within v4 that enables customization of liquidity pools. Among other points, in Section 2, we discuss the ability of hooks to modify trading fees, to implement alternative pricing rules, and to facilitate new order types. Section 3 then discusses the dynamic behavior of v4 liquidity pools and how they respond to market activity.

Formally, v4 defines a liquidity pool as a data structure known as *PoolKey*. In turn, this data structure consists of five objects: two assets, a fee level, tick spacing, and a hook. Although the first four components are carried over from v3, the hook concept is unique to v4 and serves as its most salient feature. Section 1 introduces each of the five objects defining a pool and traces their origins in earlier versions of Uniswap (v2 and v3), explaining how each iteration of Uniswap expanded the liquidity pool design space from the prior iteration.

To provide some high-level context, a v4 liquidity pool can be conceptualized as two sub-liquidity pools: one managed by a master smart contract, *PoolManager*, and the other managed by the hook specified in PoolKey. The sub-pool managed by PoolManager enforces identical pricing and liquidity provision rules as a v3 liquidity pool (Hasbrouck, Rivera, and Saleh 2025). Consequently, unless the hook specifies otherwise, the v4 pool operates identically to a v3 liquidity pool. Importantly, the PoolManager code logic is fixed at deployment and cannot be modified thereafter. In turn, creating a pool with customized pricing or customized liquidity provision rules requires employing a hook which encodes the desired customization. Nonetheless, the v4 protocol entails limits to this customization. The hook must conform to a particular interface, specified in `IHooks.sol`. Moreover, if the hook requires liquidity (trading capital, in this context) this must be obtained separately by the hook: the hook does not have access to the regular v3 liquidity residing in the PoolManager. We discuss these possibilities and their implications in depth in Section 2.

Finally, while the five objects comprising PoolKey define a pool, a pool is necessarily dynamic, owing to the fact that trading and liquidity provision alter quantities of economic relevance such as the composition of assets inventory at the pool and the implied trading costs. Section 3 explains how v4 tracks these changes through a data structure called *Pool.State*. This data structure records essential information, such as the exchange rate between the two assets defining the pool, the positions of liquidity providers, and accumulated fees. Moreover, economic activity causes transitions in Pool.State which then affects the terms of future economic activity.

At various points in the paper we will refer to three use cases, each exemplifying behavior

that is feasible in v4 (through appropriately designed hooks), but not available in Uniswap v2 or v3:

- **Dynamic Fees**
  As we will discuss in Section 2.1, Uniswap v4 enables the implementation of dynamic fees. Notably, this customization does not require the hook contract to divert liquidity away from the PoolManager contract. Rather, since Uniswap v3's pricing can applied to an arbitrary fee level, a Uniswap v4 hook contract can dynamically adjust the fee level before each trade and then execute the v3 pricing logic in the PoolManager where the trading would be intermediated with liquidity held at the PoolManager contract. A dynamic fee implementation is discussed in further detail within Sections 2.1.3 and 2.1.4.

- **Custom Pricing Rule**
  A pricing rule distinct from the v3 Constant Product Automated Market Maker (CPAMM) rule can be implemented in Uniswap v4. However, such a customization is more challenging than the dynamic fee customization because the PoolManager v3 pricing logic is fixed as CPAMM and thus the sub-liquidity-pool managed by the PoolManager cannot intermediate trades with pricing logic distinct from CPAMM. Rather, to implement this customization for a v4 liquidity pool, the desired pricing code logic must be included in the hook contract and sufficient liquidity must be diverted to the hook contract to intermediate the trading volumes implied by the encoded pricing logic. The implementation of this customization is discussed in Sections 2.1.3 and 2.2.2.

- **State-Contingent Orders**
  Within Uniswap v4, a liquidity pool may be designed to take state-contingent orders so long as the relevant contingent event is recorded on the blockchain. For example, a v4 liquidity pool could offer to execute trades subject to realized volatility being within a particular range. This would require the hook contract to store the realized volatility which could be updated automatically after each trade. The hook contract would also need to store the state-contingent orders and implement code logic to check if the conditions for any orders are satisfied and thereafter execute any orders for which the relevant conditions are satisfied.

1 Economic Objects Defining A Liquidity Pool

From a conceptual perspective, Uniswap v4 builds upon v3 which builds upon v2. As discussed, within v4, a pool is defined by five objects. Four of those objects exist also within

the v3 pool definition. Moreover, three of the objects within the v3 pool definition also exist within the v2 pool definition. In the remainder of this section, we build up to the five objects by first introducing the three objects from the v2 pool definition then the incremental fourth object in the v3 pool definition and then finally introducing the novel fifth object within the v4 pool definition.

1.1 v2 Pool Definition

In the v2 protocol, a liquidity pool specifies a market involving spot swaps of exactly one asset pair (in either direction). The names of the two assets comprising the pair are two (of the three) objects that define the pool. For example, within v2, there exists a unique pool for swapping ETH for USDC or USDC for ETH. Moreover, in that case, ETH and USDC are directly encoded into the pool definition.

The third object is the trading fee. The fee is paid by customers when they trade against the pool (that is, with the pool as the counterparty). The pool's investors share in the trading fees (pro rata, in porportion to their contribution to the pool's liquidity). Economically, the fees are necessary incentives for pool investors because they offset costs due to adverse selection and from the opportunity cost of capital (see Capponi and Jia 2021, Lehar and Parlour 2024 and Hasbrouck, Rivera, and Saleh 2024).

Within v2, the trading fee is hard-coded at 30 bps for all liquidity pools. As such, liquidity pools are unique upto the asset pairs that define them. For example, there exists only one v2 liquidity pool that allows for directly exchanging USDC for ETH or vice versa. Moreover, within this pool, traders are charged a fee of 30 bps.

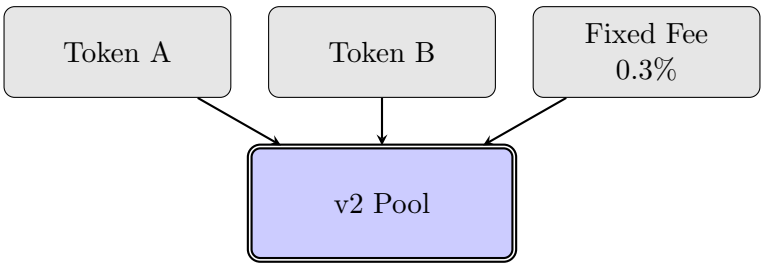

Figure 1: Uniswap v2's Pool Architecture: Each pool is defined by a token pair and a fixed 0.3% fee, resulting in a single possible pool configuration per token pair.

1.2 v3 Pool Definition

Uniswap v3 adopts the three objects from the v2 pool definition and adds one more: a tick spacing factor. The tick spacing is closely connected to the pool operations. We elaborate

within the remainder of this section.

In a traditional securities market, a dealer's bid expresses a willingness to buy a given quantity at a given price. The dealer's bid thus provides liquidity to other market participants, an option for them to sell. In this context, we might say that the dealer's liquidity (on the buy side) is supplied at a single price: others can sell at the dealer's bid, but not above, and only up to the quantity (size) of the bid.

In a v2 liquidity pool, customer purchases and sales occur at prices that vary depending on direction (buy or sell) and quantity traded. Large customer purchases drive the price to ∞ (and sales, to zero). The capital contributed by the investors is essentially available over the full range of non-negative prices, a phenomenon termed "uniform liquidity provision." This entails a substantial commitment of capital, which is expensive for the pool investors. Moreover, such a distribution of liquidity provision may not be efficient: the pool is capitalized to trade at prices that are remote from the current market price and unlikely to be realized in practice.

v3 aims to address the aforementioned concern within v2. In particular, within v3, liquidity providers are given discretion to select specific price intervals within which their liquidity can be used for intermediating trades. More explicitly, v3 pools are defined by an exogenous partition of all possible price levels. In turn, when a liquidity provider provides liquidity, they must also specify the particular intervals to which their liquidity will be allocated. As a practical matter, this structure generally leads liquidity provision to concentrate in price intervals near the current price level because trading fees are passed only to those investors providing liquidity at traded prices (see Hasbrouck et al. 2025). To recap, traditional liquidity is fixed (at a given price); v2 liquidity is uniform (over all nonnegative prices), and v3 liquidity is concentrated (within the investor's chosen interval).

With regard to v3 pool definition, it is important to understand that the exogenous partition of all possible price levels within a v3 pool is determined by a single object, the aforementioned tick spacing. More explicitly, the price partition is a sequence of intervals, $\{[\Psi_n \Psi_{n+1}]\}_{n=-\infty}^{\infty}$ where the lower endpoint of the nth interval, equivalently the upper endpoint of the $(n-1)$st interval, is given explicitly as follows:

$$\Psi_n = (1+\Delta)^n$$

and $\Delta > 0$ is the tick spacing which is part of the pool definition. Within v3, there are currently only four possible values for the tick spacing of a pool: 1 bp, 10 bps, 60 bps and 200 bps. As a related point, v3 relaxes v2's restriction of a 30 bp fee level for all pools. Instead, v3 specifies four possible fee levels: 1 bp, 5 bps, 30 bps and 100 bps. Furthermore,

within v3, the selection of the fee level and tick spacing are related as follows:

Fee Level (bps)	Tick Spacing (bps)
1	1
5	10
30	60
100	200

Table 1: Uniswap v3 Fee Level and Tick Spacing Dependence

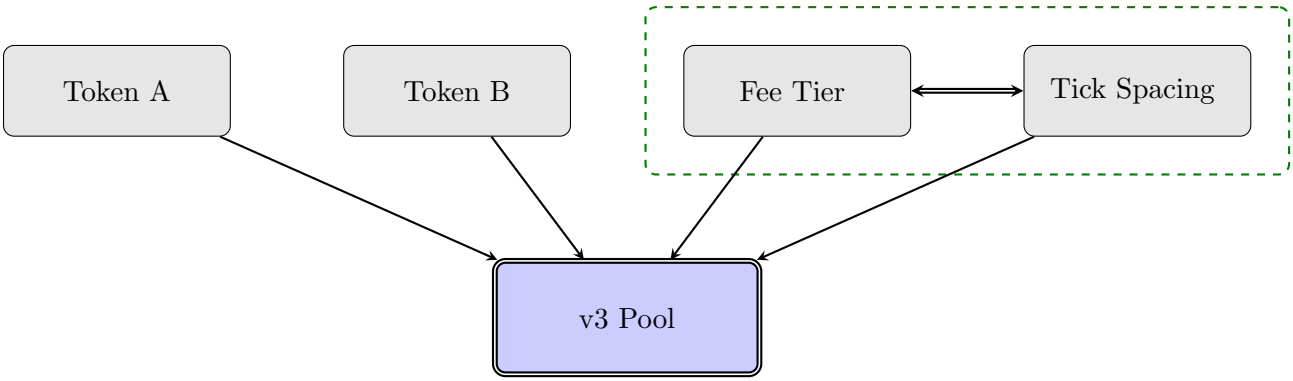

Figure 2: Uniswap v3's Pool Architecture. v3 introduced multiple fee tiers with corresponding tick spacings, allowing different pools for the same token pair.

That is, selecting the feel level determines the tick spacing for a v3 pool (and vice versa) as per Table 1. In turn, since there are exactly four possible fee levels (equivalently, four possible tick spacings) within v3, there can be as many as four pools for any asset pair within v4. For example, there are currently three v3 liquidity pools for ETH-USDC with those pools being differentiated by fee level and tick spacing. Moreover, one further ETH-USDC v3 liquidity pool can be created as long as a pool with its fee level and tick spacing is not already deployed within Uniswap v3, and the fee level-tick spacing combination appears in Table 1.

1.3 v4 Pool Definition

As noted, v4 builds conceptually upon v3, defining a pool by five objects where four of those objects are the objects that define pools in v3. More concretely, a v4 pool is partly defined by two assets, a fee level and a tick spacing level where those four of these objects fully define a v3 pool. Notably, v4 then adds one further object: a hook. We discuss hooks in detail in Section 2, so we do not provide concrete detail here. Rather, in this section, we provide some high-level context and explain how hooks affect pool design relative to v3.

Conceptually, hooks are intended to enable a wide set of customizations for a liquidity pool. The three use cases considered above are illustrative, but more complex augmentations might also be welfare-enhancing. Notably, whereas v3 augmented v2 on a specific dimension (i.e., increasing the action space for liquidity providers), v4 allows developers deploying pools discretion as to how their pools should vary from v3 pools. As we will discuss, the discretion given to developers in v4 is not fully arbitrary but it is vast relative to the discretion given in v3 relative to v2. As a related point, it is feasible for developers to specify an empty hook. In this case, a v4 pool behaves exactly as a v3 pool except that the discretion around setting the fee level and tick spacing is fully arbitrary and not restricted by Table 1.

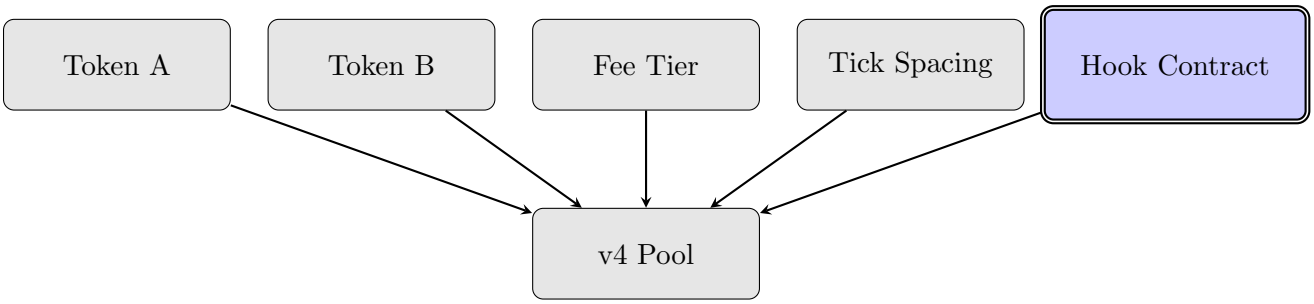

Figure 3: Uniswap v4's Hook-Based Architecture. v4 enables customization of liquidity pools through hooks.

Feature	**v2**	**v3**	**v4**
Token Pairs	✓	✓	✓
Fee Level	Fixed (0.3%)	Multiple Preset (1, 5, 30, 100 bps)	Flexible (Any Value)
Tick Spacing	✗	Preset Options (1, 10, 60, 200)	Flexible (Any Value)
Hooks	✗	✗	✓ (Custom Logic)

Table 2: Evolution of Uniswap Features.

2 Hooks

Formally, a hook is a smart contract with a smart contract being computer code deployed on a blockchain (see John, Kogan, and Saleh 2023). Notably, a hook is not fully arbitrary computer code. Rather, a hook must implement a particular standardization such that each hook must specify 10 particular functions and also must specify 14 specific true/ false flags. We detail the functions in Section 2.1, and we detail the flags in Section 2.2.

2.1 Hook Functions

Conceptually, the purpose of Hook functions is to "hook" in before or after routine operations of Uniswap. More concretely, there are various operations necessary for the regular functioning of a decentralized exchange (e.g., swapping, adjusting liquidity, etc...), and a hook enables customized logic before and after these operations. The particular customized logic is specified as the code body of the functions that a hook must entail. In more detail, the v4 code makes a function call to a particular hook function before each Uniswap operation and a function call to another particular hook function after each Uniswap operation. In turn, there are two hook functions for each Uniswap operation. As we will discuss, there are five operations, and thus there are $2 \times 5 = 10$ functions that each hook entails.

To provide more detail, the five Uniswap operations are (1) creating a pool, (2) trading with a pool, (3) adding liquidity to a pool, (4) removing liquidity from a pool and (5) donating to a pool. Notably, the first four aforementioned operations exist in v3 and the code logic for those operations in v4 is identical to the code logic for those operations in v3. In turn, as referenced earlier, when hook function code bodies are left empty, v4 then operates as v3 would. To that end, in the remainder of this section, we occasionally reference the code logic of these operations without explaining that code logic. This is purposeful because our focus is on refinements from v3 to v4 and since the code logic for the core operations is identical from v3 to v4, we direct the reader to references on v3 to understand the core Uniswap operations (see, e.g., Hasbrouck et al. 2025). Within the remainder of this section, we briefly discuss the specification and design space of each before and after function for each of the five aforementioned operations.

2.1.1 beforeInitialize: Hook Function Preceding Pool Creation

The hook function before pool creation is named *beforeInitialize*, reflecting that the pool creation step is generally referred to as the pool initialization. Notably, since pool creation occurs only once, the beforeInitialize function also executes only at most once. Moreover, by construction, the output of this function is not stored, implying that this function does not affect the initial state of the pool.

One use case for the beforeInitialize function is to restrict the instances in which the associated hook can be attached to a pool. More explicitly, the beforeInitialize function can be used to revert the entire pool creation based on conditional logic thereby restricting instances of a pool being created with the particular hook. A special case of this would be where the hook creator would like the hook to serve as a signal regarding the pool creation following a particular process. In this case, the hook creator can include an auxiliary function

in the hook code and follow the specific desired steps of pool creation within that function. The beforeInitialize function can then be used to revert any attempts to create a pool with the hook attached where the pool creation is not triggered by the hook auxilary function. An example of this type of hook is the dumpnofun hook which employs an auxiliary hook function to launch a new token with fixed supply and then creates a pool for this token against ether where the pool implements a 30 day lock up period on the full token supply. The beforeInitialize function reverts any attempts to attach this hook to a pool that does not execute through the referenced auxiliary hook function and thus the hook being attached to a pool serves as proof that the underlying token is a fixed supply token with a 30 day lock up period. Intuitively, the dumpnofun hook provide resistant to rug pulls in the sense that no pool with the dumpnofun hook can execute a rug pull.[1]

2.1.2 afterInitialize: Hook Function Following Pool Creation

The hook function after pool creation is named *afterInitialize*, also based on the convention that pool creation is referred to as initialization. As with the beforeInitialize function, the output for the afterInitialize function is not stored. Nonetheless, since the pool has been created when the afterInitialize function executes, the afterInitialize function can nonetheless modify state variables of the pool immediately after the pool has been created so long as those state variables are accessible by the hook. Moreover, as a separate point, a practical use of the afterInitialize function could be to take actions that would integrate the newly created pool with other aspects of the cryptoeconomic ecosystem. For example, the afterInitialize function could add the newly created pool to an open registry such as one that might be used by an aggregator to source liquidity.

2.1.3 beforeSwap: Hook Function Preceding a Trade

The hook function before a trade is named *beforeSwap* with the name being because trades are mainly executed through a function called *swap*. Notably, the beforeSwap function is able adjust the pool trading fee, and it is also able to conduct a portion of the trade ahead of the main trading function, swap. To provide detail on the former point, fees can be adjusted before each trade through the beforeSwap function, and this is the way in which a liquidity pool may implement dynamics fees. To provide detail on the latter point, part of a trade can be conducted through the beforeSwap function with the balance of the trading volume then being traded through the swap function. As an aside, to implement any trading through

[1]The dumpnofun hook is deployed on Sepolia: https://sepolia.etherscan.io/address/0xe9f3ccd1844e0f9116f1800b087a1398eb23e000 with the code available on GitHub: https://github.com/Jun1on/dumpnotfun-hook. A frontend is available at https://dumpnofun-ui.vercel.app/.

the beforeSwap function as referenced, the liquidity for such trading must be held outside of PoolManager which is the previously mentioned master smart contract where liquidity is normally held. One frequent design choice is to store the liquidity directly in the hook contract. As we discuss in Section 2.1.5 and 2.1.6, storing liquidity in the hook contract can be implemented via other hook functions, beforeAddLiquidity and afterAddLiquidity.

2.1.4 afterSwap: Hook Function Following a Trade

The hook function following a trade is named *afterSwap*. This function allows for state variables to be updated after the trade based on details that would become known only after the trade has been executed. For example, a hook could implement dynamic fees as a function of observed gas prices. Then, the afterSwap function could be used to update a state variable corresponding to the moving average of gas prices. This functionality would work in tandem with the beforeSwap function which would update the pool fee based on the most recent value for this state variable. Concretely, the gas price state variable updates after each trade (via afterSwap) and then the fee adjusts thereafter before the subsequent trade (via beforeSwap).

2.1.5 beforeAddLiquidity: Hook Function Preceding Adding Liquidity

The hook function preceding additions of liquidity to a pool is named *beforeAddLiquidity*. Notably, this function can be used to modify how liquidity is stored in a way that is crucial for particular hook implementations. For example, as discussed in Section 2.1.3, the beforeSwap can be used to intermediate a portion of a trade, but such intermediation requires that the associated liquidity must be stored outside of the general Uniswap v4 contract. In turn, one method of implementing trading through the beforeSwap function is to use beforeAddLiquidity to store all liquidity directly in the hook contract and then to use the liquidity in the hook contract for trading through the beforeSwap function. This could particularly be implemented by having beforeAddLiquidity revert in all cases so that liquidity cannot be provided through the typical Uniswap function for adding liquidity. Then, to allow for liquidity to be provided directly to the hook contract, the hook can include an auxiliary function that directly deposits liquidity into the hook contract.

2.1.6 afterAddLiquidity: Hook Function Following Adding Liquidity

The hook function following additions of liquidity to a pool is named *afterAddLiquidity*. As with beforeAddLiquidity, this function can be used to modify how liquidity is stored. Notably, the implementation discussed in Section 2.1.5 entails entirely short-circuiting the

code logic of liquidity addition in the absence of hooks and storing all new liquidity directly in the hook. As a contrast, afterAddLiquidity can be used to maintain the usual liquidity provision method for some liquidity while diverting the remainder of the liquidity into the hook contract. In particular, after the typical liquidity addition process has occurred, afterAddLiquidity could accept further liquidity from the investor and store this within the hook contract. Thereafter, prior to a subsequent trade, the beforeSwap function could intermediate a portion of the trade size with the hook contract liquidity and reserve the balance of the trade for the typical Uniswap v3 trading process. As discussed earlier, the typical Uniswap v3 trading process would operate through the swap function and would rely on liquidity in PoolManager with the associated liquidity having been deposited through the v3 add liquidity logic which precedes afterAddLiquidity.

2.1.7 beforeRemoveLiquidity: Hook Function Preceding Liquidity Removal

The hook function preceding removals of liquidity from a pool is named *beforeRemoveLiquidity*. To provide context on how this hook function could be used, a simple application of it would be to impose a minimum lock-up period on liquidity providers. More concretely, a minimum lock-up period could be implemented by having the hook function revert if the liquidity provider seeking to remove liquidity has not had their liquidity stored within the pool for sufficiently long. In this way, the liquidity removal operation would succeed only when the liquidity provider has had their liquidity stored for longer than the lock-up period.

2.1.8 afterRemoveLiquidity: Hook Function Following Liquidity Removal

The hook function following removals of liquidity from a pool is named *afterRemoveLiquidity*. Notably, this hook function can be used to allocate revenues from the liquidity pool across liquidity providers in a customized way. In more detail, the afterRemoveLiquidity function can deposit the proceeds from the liquidity removal into the hook contract rather than returning it fully to a liquidity provider seeking to remove liquidity. In turn, the hook contract can then implement a customized allocation of the liquidity pool revenues across liquidity providers by distributing funds stored in the hook to liquidity providers in a customized way.

2.1.9 beforeDonate and afterDonate

A novel feature of Uniswap v4 relative to Uniswap v3 is a new operation that enables a user to donate assets to a pool. To provide more concrete context, this new operation enables users to provide assets to a liquidity pool. The assets are specifically sent to the PoolManager contract where they are added to the fees associated with the liquidity pool to which they

are being donated. Then, as part of the fee balance for the liquidity pool, they can add to the pay-offs for liquidity providers for that liquidity pool. Notably, as with the other four Uniswap operations, the donation operation entails hook functions preceding and following it. These hook functions are named *beforeDonate* and *afterDonate* respectively.

2.2 Hook Flags

Uniswap v4 entails two sets of true/ false hook flags. The first set corresponds to the hook functions, whereas the second set corresponds to whether particular hook functions entail value transfers to or from the hook contract. We refer to the former as hook function flags and the latter as ReturnDelta flags. There are ten hook function flags and four ReturnDelta flags where these flags are summarized in Table 3. We first explain the first set of flags and then we detail each flag in the second set separately. As an aside, we note that the value of each true/ false flag is static in that it cannot be changed after a hook is deployed.

Hook Function	**Hook Function Flag**	**ReturnDelta Flag**
beforeInitialize	✓	
afterInitialize	✓	
beforeAddLiquidity	✓	
afterAddLiquidity	✓	✓
beforeRemoveLiquidity	✓	
afterRemoveLiquidity	✓	✓
beforeSwap	✓	✓
afterSwap	✓	✓
beforeDonate	✓	
afterDonate	✓	

Table 3: Hook flags: Each hook function must be implemented in a hook contract, with corresponding flags determining whether the function is called. Some functions have an additional flag (ReturnDelta) that allows them to transfer economic value.

2.2.1 Hook Function Flags

For each hook function discussed in Section 2.1, there exists a hook flag. This is because the envisioned purpose of a hook need not require all ten functions. Nonetheless, since the technical implementation of a hook requires that all ten hook functions always be specified, there is a true/ false flag for each function to enable particular function(s) to be ignored. In more detail, the code logic within Uniswap v4 specifies that a particular hook function is executed only if the associated true/ false flag is true. In turn, if a hook developer would

like to not use a particular hook function, that hook developer may specify the code body of the associated function arbitrarily and then set the hook flag for the function as false. Then, the function being specified would satisfy the technical requirement of the implementation but the function would never execute and thus would not affect functionality of a liquidity pool using the hook.

2.2.2 ReturnDelta Flags

As we have discussed in Section 2.1, some hook functions may entail transferring economic value to and/ or from the hook contract. For example, as per Section 2.1.3, the beforeSwap function may intermediate a portion of a user trade and the liquidity for the trade would come directly from the hook contract. Moreover, as discussed in Sections 2.1.6 and 2.1.8, the afterAddLiquidity and afterRemoveLiquidity functions can be used to store liquidity in a hook contract. Although not discussed, the afterSwap function could also be used to transfer economic value to and/ or from the hook. Notably, economic value transfers to and/ or from hooks entail risks for users because the incremental value being transferred to the hook contract may come from a user's wallet.[2] In turn, to ensure that the potential for such value transfers are transparent, Uniswap v4 entails true/ false flags for each possible value transfer to and/ or from the hook. Moreover, the Uniswap v4 code specifically checks the flag associated with a potential value transfers and then allows particular value transfers only if the relevant flag is set to true. In the remainder of this section, we discuss each of these flags briefly:

- **beforeSwapReturnDelta**

  There exists a true/ false flag named *beforeSwapReturnDelta* which determines whether the beforeSwap function can intermediate a portion of the user trade directly with the hook contract. If the beforeSwapReturnDelta flag is true, then the user's trade size is reduced by the volume intermediated in the beforeSwap function and only the remainder of the trading volume is intermediated by Uniswap's routine trading function, swap. However, if the beforeSwapReturnDelta flag is false, then the entire user trading volume is intermediated through the swap function irrespective of the code logic within the beforeSwap function.

- **afterSwapReturnDelta**

[2]It is not possible to extract value from a user beyond the level that a user has approved due to the usual cryptographic methods. Nonetheless, users frequently approve the potential for larger transfers than needed because market movements render the exact level of a transfer needed as uncertain. Importantly, a hook contract that enables transfers to the hook contract could extract value from a user by exploiting this user tendency for excessive allowances.

There exists a true/ false flag named *afterSwapReturnDelta* which determines whether the hook contract can require a transfer of assets from the user to the hook contract through the afterSwap hook function. If this afterSwapReturnDelta flag is false, then any such value transfer in the afterSwap function would be ignored. As a technical aside, we reiterate that no smart contract function can extract value from users without user approval, but it is nonetheless sometimes practically necessary for users to allow for value extraction beyond their specific trading demand (see Footnote 2). In turn, this flag is useful because, when it is set as false, then there is no risk that liquidity pools with this hook will extract user value in the afterSwap function.

As an additional technical aside, Uniswap trades are specified so that each user conducting a trade must identify the exact volume of one leg of their trade. Notably, irrespective of the hook flag values and the hook function code, the trading cannot violate the user's stipulation for trading volume specified. As a consequence, when afterSwapReturnDelta is true, then the concern is specifically that afterSwapReturn function could extract value in terms of the unspecified leg of the trade. For example, if a user wishes to buy ETH with USDC, then the user could specify the exact amount of USDC to be spent. However, in that case, if afterSwapReturnDelta is true, then it is possible that the user recives no ETH because the ETH received from the PoolManager (through the swap function) is deposited into the hook contract through the afterSwap function.

- **afterAddLiquidityReturnDelta**

  There exists a true/ false flag named *afterAddLiquidityReturnDelta* which determines whether the hook contract can require a transfer of assets from the user to the hook contract through the afterAddLiquidity hook function. If this afterAddLiquidityReturnDelta flag is false, then any such value transfer in the afterAddLiquidity function would be ignored. As before, we emphasize that value cannot be extracted from a user without the user's approval (see Footnote 2).

- **afterRemoveLiquidityReturnDelta**

  There exists a true/ false flag named *afterRemoveLiquidityReturnDelta* which determines whether the hook contract can require a transfer of assets from the user to the hook contract through the afterRemoveLiquidity hook function. If this afterRemoveLiquidityReturnDelta flag is false, then any such value transfer in the afterRemoveLiquidity function would be ignored. We again emphasize that value cannot be extracted from a user without the user's approval (see Footnote 2).

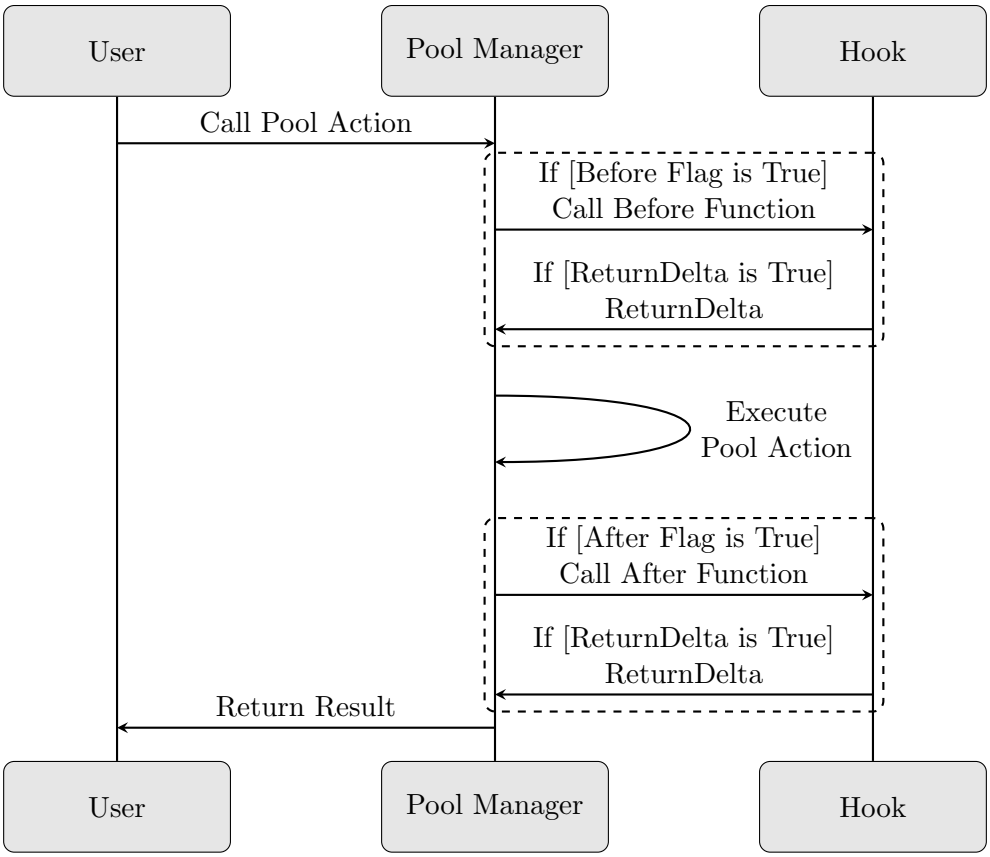

Figure 4: Hook Execution Flow in Uniswap v4: When a standard operation is called, if enabled, a beforeHook function can modify the operation parameters. The standard operation is then executed on the pool state, followed by an optional afterHook that can process the results.

3 Pool Dynamics

The objects discussed within Section 1 are defining characteristics of a liquidity pool that do not vary over time. Nonetheless, there are other characteristics which are time-varying that are referred to as the state of the pool. Those time-varying characteristics are specifically stored in a data structure named *Pool.State* and they evolve through the operations discussed in Section 2.1. On this point, it is important to understand that the pool state variables have restricted access and therefore cannot be changed by an arbitrary user action. In fact, most state variables can be changed only by core Uniswap v4 contracts such as PoolManager. Notably, even hook contracts cannot alter most state variables except indirectly by calling functions in PoolManager. As an example, a hook contract can change the price at a liquidity pool by executing a trade which would execute swap in PoolManager; however, a hook contract cannot arbitrarily change the price at a liquidity pool.

In the remainder of this section, we provide further context on the components of

Pool.State:

3.1 slot0: Pricing and Fee Levels

Pool.State contains an object *slot0* which stores the pricing and fee levels for the liquidity pool. More specifically, slot0 contains the exchange rate (i.e., price) across the two assets in the pool, the liquidity provision fee and also the protocol fee. The liquidity provision fee corresponds to the fee referenced in Section 1 and this fee is paid to liquidity providers; in contrast, the protocol fee is paid directly to the liquidity pool and stored within the PoolManager.

We emphasize that there is information related to the liquidity provision fee in both PoolKey (see Section 1) and Pool.State. More explicitly, when the liquidity pool is defined to have a fixed static fee, then this fee is set within PoolKey and the liquidity provision fee in Pool.State is ignored. In contrast, if the liquidity pool is defined as a dynamic fee liquidity pool, then the fee in PoolKey is set to a specific value, 0x800000, which serves as a flag for the liquidity pool featuring dynamic fees. In that case, the liquidity provision fee level in Pool.State is used for the liquidity pool's liquidity provision fee, and that fee can be updated over time through changes to Pool.State.

3.2 Realized Fees

In addition to fee levels, Pool.State also stores realized fees, namely fees that have already been accrued to liquidity providers from traders. These fees are stored in the PoolManager and can be withdrawn by liquidity providers subject to constraints imposed by the afterRemoveLiquidity hook function (see Section 2.1.8).

3.3 Tick Data

Pool.State stores data associated with each tick where each tick corresponds to a pool price. The specific data stored includes the liquidity at each tick and also the fees associated with each tick. There is some technical detail associated with the exact quantities being stored, and we abstract from that detail for simplicity. Note that this tick data changes with trading activity because trading activity affects both pricing and accrued fees.

3.4 Liquidity Provider Position Data

Pool.State stores the position of each liquidity provider. More specifically, Pool.State stores an id for each position, its liquidity level and the fees accrued to the position. The id of the

position subsumes the range over which liquidity is provided, the liquidity provider's address and a further unique identifier known as the salt.

4 Conclusion

The Uniswap v4 protocol expands the economic design space for liquidity pools beyond that from prior Uniswap deployments (e.g., v2 and v3). This article clarifies the specific manner in which the economic design space has been expanded, also highlighting the extent to which the design space remains limited. The use cases considered in the article illustrate basic extensions of pool capabilities: dynamic fees, alternative pricing rules, and contingent trades. More complex augmentations might lead to mechanisms to capture and rebate losses that would otherwise be experienced by traders (see Maximal Extractable Value discussion within Harvey, Hasbrouck, and Saleh 2024). We are hopeful that future work will provide formal economic analysis of the augmented design space, clarifying welfare implications.

References

Capponi, A., Jia, R., 2021. The adoption of blockchain-based decentralized exchanges. Columbia University Working Paper .

Harvey, C. R., Hasbrouck, J., Saleh, F., 2024. The evolution of decentralized exchange: Risks, benefits, and oversight. Wharton Initiative on Financial Policy and Regulation Working Paper .

Hasbrouck, J., Rivera, T. J., Saleh, F., 2024. The need for fees at a DEX: How increases in fees can increase DEX trading volume https://ssrn.com/abstract=4192925.

Hasbrouck, J., Rivera, T. J., Saleh, F., 2025. An economic model of a decentralized exchange with concentrated liquidity. Management Science .

John, K., Kogan, L., Saleh, F., 2023. Smart contracts and decentralized finance. Annual Review of Financial Economics 15, 523–542.

Lehar, A., Parlour, C., 2024. Decentralized exchange: The uniswap automated market maker. The Journal of Finance .