SUPA CPMM: Solana Uniform-Price Auctions for CPMMs

Grant Stenger, Martin Lindsey, Edwin Suresh, Keith Warter, Agam Gambhir*

October 23, 2025

Abstract

Path-dependent execution of trades presents risks in financial markets. In the decentralized cryptocurrency setting, this risk appears as maximal extractable value (MEV), to which significant losses for traders are attributed, driving the development of various exchange structures in response. We introduce the Solana Uniform-Price Auction (SUPA) Constant-Product Market Maker (CPMM), a batch-clearing generalization of the standard CPMM architecture. This design prioritizes fairness and rewards unmanipulated market flow by neutralizing intra-slot MEV and reducing price impact via matching trade volume in opposite directions. We obtain the expression for the uniform clearing price in a CPMM and provide details of our Solana implementation.

Contents

1 Introduction **2**
- 1.1 Motivation and contributions . . . 2
- 1.2 Related literature . . . 3
- 1.3 Layout of this work . . . 4

2 Design overview **4**
- 2.1 Batch clearing cycle . . . 5
- 2.2 Uniform batch-pricing . . . 6
- 2.3 Fee policy . . . 7
- 2.4 Liquidity provision . . . 8

3 Implementation **8**

4 Analysis and discussion **10**
- 4.1 Trader economics . . . 10
- 4.2 Liquidity provider economics . . . 12
- 4.3 Simulated example . . . 12

*All authors are part of the Kinetic.xyz team.

4.4 Additional considerations 13
4.4.1 Practical tradeoffs and limitations 13
4.4.2 Further work and extensions 15

5 Conclusion **17**

A Core Program Objects **21**

B Glossary of Terms **23**

1 Introduction

1.1 Motivation and contributions

The design of modern financial markets, whether on-chain or off, creates two fundamental forms of execution risk. In traditional continuous markets, the risk comes from price-time prioritization, a race where the fastest participant at any given price level wins. On public blockchains, the risk comes from intra-slot[1] reordering, where adversaries can insert, remove, and rearrange pending transactions to their benefit (i.e., MEV). Though they manifest differently—as latency races in centralized order books and sandwich attacks on AMMs—both problems stem from a common root: the path-dependence of continuous execution.

Continuous limit order books dominate traditional equities markets and centralized cryptocurrency exchanges. In these markets, better-priced orders execute first, and within a price level, orders are executed in the order they arrive. This time priority turns into a latency arms race, where liquidity providers invest in speed to limit adverse selection and secure queue position. While that competition may improve price discovery and tighten quoted spreads, continuous matching also increases latency-arbitrage risk, raising adverse-selection costs and reducing the number of available orders when quotes are stale or market volatility is high. Latency-disadvantaged participants may benefit from more efficient mid-prices, yet face higher execution costs from adverse selection and thinner top-of-book liquidity. Order books lend themselves to centralization due to the complexity of coordinating between large numbers of market participants. They are notoriously challenging to implement on-chain, thus most decentralized exchange (DEX) volume runs through automated market makers (AMMs). AMMs do not have price–time queues; instead, the execution price is a deterministic function of the pool's current reserves, so each trade moves the price and outcomes depend on what executes just before it. Because pending transactions are public before inclusion, and block producers or searchers can influence which transactions are included and in what order, this path dependence turns into MEV. These mechanics create several well-known frictions, placing even greater emphasis on speed within AMMs. Because pending transactions are visible, adversaries can execute sandwich attacks: a user's trade is front-run and then back-run within the same block, extracting user surplus. MEV searchers pay priority fees or submit bundles to influence inclusion and ordering, allowing them to insert or bracket slower participants' trades, worsening execution prices and raising fees. When an AMM's price deviates from external markets, arbitrageurs rebalance the pool, generating the familiar loss-versus-rebalancing (LVR) borne by liquidity providers (LPs).

[1] On Solana, a slot is the base unit of time, lasting approximately 400 milliseconds, during which a validator has the opportunity to produce a block.

To address the root cause of path-dependence, we introduce the Solana Uniform-Price Auction (SUPA) framework to augment standard constant-function market maker (CFMM) architectures. This framework utilizes discretized time by clearing all trades within a single Solana slot at a uniform price, calculated to preserve the constant-function invariant. It serves as an example of how auction-based exchange mechanisms are naturally well-suited to decentralization. By consolidating transactions, SUPA remains fully on-chain and robust to sandwich MEV while preserving the familiar properties of a standard CFMM. Rather than emphasizing and rewarding speed, the design of SUPA emphasizes fairness and rewards non-exploitative trade flow.

This work formalizes, implements, and evaluates this model in the context of a CPMM, making the following contributions. First, we derive the uniform clearing price for a batch of CPMM transactions which preserves the constant-product invariant. Next, we show that this batching process generalizes the standard CPMM pricing mechanism to accommodate trade flow in both directions, and further demonstrate that it interpolates between the two scenarios in which each direction of aggregate trade flow is processed one at a time. We also present a basic numerical example to illustrate how collectively executing at the uniform clearing price approximates an average over all permutations of a given transaction sequence. Finally, we describe a concrete Solana program design that circumvents writable-account limits to support large batch sizes while balancing compute costs.

1.2 Related literature

Three strands of literature that motivate SUPA AMM are reviewed here: (i) analyses of frequent batch auctions (FBAs), particularly in comparison to continuous double auctions (CDAs); (ii) studies of call-market mechanisms already present in traditional equities; and (iii) recent innovations that incorporate forms of batch pricing into the decentralized finance setting. We highlight where SUPA extends or diverges from each line of work.

Frequent batch auctions in traditional finance. Budish, Cramton, and Shim [1] proposed FBAs to address the price-time arms race that fuels high-frequency trading in the CDA setting of traditional finance. Later studies [2, 3, 6] paint a more nuanced picture of FBAs, highlighting tradeoffs in price efficiency, available liquidity, and crash severity, though the benefits for traders that lack access to HFT infrastructure [4, 5] are widely corroborated. SUPA adapts the core idea of discrete clearing to the setting of Solana's $\sim$400 ms cadence and a constant-function AMM.

Call markets and closing auctions. Modern stock exchanges run call auctions at their open and close, where orders are aggregated and cleared at a single uniform price. Their share of total trading volume has risen over time. In the U.S., the closing auction's share rose from roughly 2–3% in 2010 to about 7–10% by 2018, and remained in that range through 2021, a pattern consistent with evidence that call auctions aid price discovery and liquidity [7–9]. In Europe, these auctions represent an even larger share of volume, with similar trends over the same time period. On Euronext Paris (CAC 40), closing auction share increased from 20–28% before 2016 to over 40% in 2019, reflecting a larger pattern across several major European markets [10, 11]. Notably, Euronext Paris exhibits tighter spreads [12] during its five-minute closing call, and NASDAQ observes reduced volatility and spreads [13] during its closing and opening call auctions. Academic work has shown that they can offer superior liquidity to CDAs, though potentially lead to less total trading volume [14] as well as theoretically less efficient price discovery [15] in comparison. SUPA generalizes this

design from two auctions per day to one per slot, implemented on-chain at the protocol level. In doing so, it aims to reproduce the relative improvements associated with call markets compared to CDAs, after which many current DeFi exchange mechanisms are modeled. We expect certain drawbacks noted in traditional settings to be reduced on-chain[2].

Batch pricing in DeFi. Several on-chain batch mechanisms have been implemented on the Ethereum blockchain. The CoW Protocol [21] makes use of a coincidence-of-wants (CoW) model to frequently group user swaps into a single uniform-price settlement, which is mediated by off-chain solvers that compete on gas fees for inclusion. It is, however, not without challenges: a deeper analysis [16] suggested that "CoW['s] occurrence on the CoW Protocol is negligible," likely motivating "different and more rigorous" [17] formalizations of CoW to be implemented on-chain. The dYdX v4 protocol [18] features batch auctions at regular intervals, aiming to neutralize latency advantage by way of a network of off-chain validators. Paradigm introduced the time-weighted average market-maker [19], using the concept of virtual orders that effectively execute "*in between* blocks" to reduce price impact and avoid sandwich attacks, but still clear along the price-time curve. For a further exploration of the contemporary landscape of batch mechanism designs in AMMs, see §1.2 of [20]; a notable detail pointed out by the authors thereof is that "batch clearing at uniform price does not guarantee incentive compatibility" on its own. In sum, these designs support batching and order-invariant execution, but most run at the router or off-chain solver layer. SUPA implements batching inside the AMM, producing a single on-chain clearing price and atomic state update each slot, removing intra-slot sequencing effects at the pool level.

1.3 Layout of this work

Section 2 overviews the SUPA mechanism and its application CPMMs. Section 3 specifies the on-chain architecture and details how batches are settled. Section 4 discusses how batching within a CPMM improves swap outcomes and mitigates MEV, as well as how LVR is unchanged in the batched-transaction setting. A simulated comparison between batched and sequenced transaction settings is also presented. The conclusion is followed by an appendix containing additional technical details.

2 Design overview

This section describes the core SUPA framework at several levels: the slot-based clearing cycle using permissionless settlement instructions, the CPMM uniform-price computation, fee collection, and end-of-batch liquidity adjustments.

We call the permissionless settlement instruction the *crank*. Each crank call is a public, idempotent invocation; it processes a subset of a past batch's trade orders or liquidity adjustments and distributes to users their allocation of token X, token Y, and/or LP tokens. Someone who calls the crank is referred to as a *keeper*. As the crank is permissionless, a keeper could be any individual, or even a bot in a 'keeper network.'

Due to limitations on the number of wallets to which a single transaction can transfer at once, a single crank call may not process all of a batch's trade orders or liquidity adjustments. Consequently,

[2] Higher auction frequency mitigates concerns about reduced trading volume or fewer trading opportunities, and tightly linked on-chain DEX & off-chain CEX prices alleviate price-discovery concerns.

the protocol stores received trade orders in *trade-order chunks* and received liquidity adjustments in *liquidity-adjustment chunks*. They are sized such that a single crank can execute the entire chunk.

2.1 Batch clearing cycle

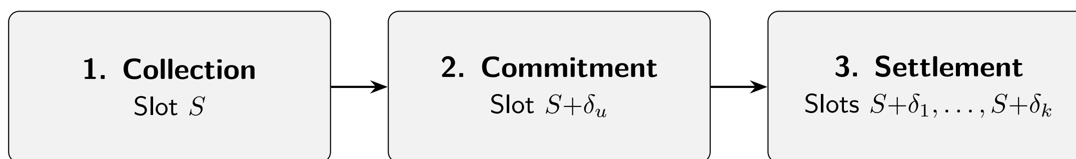

Figure 2.1: High-level cycle with timing. Orders enter in slot S (**Collection**); the first post-S order execution crank transaction or order submission transaction performs the **Commitment** at $S+\delta_u$ (committing the batch outcome and fixing parameters); permissionless **Settlement** cranks then process chunks over $S+\delta_1, \dots, S+\delta_k$. Here $\delta_u \geq 1$ is the gap from S to the first post-S transaction, and $1 \leq \delta_1 \leq \cdots \leq \delta_k$ are settlement lags (ideally $\delta_k=1$).

Collection:

During slot S, the program admits market sell orders and LP intents to add or remove liquidity. Inputs are escrowed into the pool's vaults, but the canonical reserve fields (X, Y)—the *effective reserves*—remain unchanged throughout S. Instead, the program maintains non-binding *indicative* figures (such as a provisional cumulative clearing price and post-batch reserves) that update as orders arrive; this information is held in *snapshot-batch* program-derived addresses (PDAs). These inform users and allow for cranks to later execute on a snapshot of pool reserve state for batch vault transfers without affecting previous or next batches. Orders are appended to per-slot *trade-order chunk* and *liquidity-adjustment chunk* PDAs, referenced by the snapshot batch accounts. All chunk PDAs are sized so a single crank can execute the entire chunk within Solana's per-instruction writable-account limits.

Commitment:

The first order execution crank transaction or order submission transaction at some slot $S+\delta_u$ performs the *Commitment*. Ideally the crank is very fast and happens at millisecond 1 of every new block. However, we chose to allow the first order to also commit the previous slot and carry forward the reserve data into the next slot when the state of the previous slot is what we refer to as "Uncommitted" when that first order arrives. This would happen when it arrives before the first crank. The reason is that accepting new orders always requires knowing the final commitment amounts from last batch to start calculating a cumulative price for this batch. Commitment is what writes the slot-S batch outcome to the effective reserves in top level pool state and fixes the parameters for settlement (e.g., batch price and committed post-trade reserve ratio). The Commitment is idempotent and does not itself distribute funds.

Settlement:

Settlement proceeds via permissionless crank calls during slots $S+\delta_k$, each consuming exactly one eligible chunk (i.e., a slot-S chunk with unsettled entries). Once the settlement fixes batch parameters, slot-S trade and liquidity-adjustment chunks may be processed in any order (and

in parallel across chunks as long as the two chunks do not rely on the same state). Cranks will distribute payouts at the fixed batch price using the established rounding policy. Liquidity chunks align escrowed deposits to the committed post-trade reserve ratio before minting LP and refund any excess. They also burn pool LP tokens and redeem X and Y tokens in the post-trade reserve ratio. The final crank that exhausts slot-S chunks resets batch metadata so these accounts can be used again in future slots.

2.2 Uniform batch-pricing

Normally, the order of a sequence of swaps affects the outcome of each transaction, since the spot price moves in response to each one. For a given swap, its outcome is better if it precedes a swap in the same direction or follows a swap in the opposite direction. Batching swaps together into a single transaction eliminates the notion of an ordered sequence within a slot and allows them all to be processed 'simultaneously.' We first show that the uniform batch price for any collection of swaps is equivalent to that of two aggregated swaps, and then derive the uniform batch price for a CPMM.

Problem formulation. Consider a constant-function AMM (CFMM) for exchanging tokens X and Y. Let m traders submit X→Y inputs $(\Delta X^i)_{i\in[m]}$ and n traders submit Y→X inputs $(\Delta Y^j)_{j\in[n]}$. Aggregate the directional inputs into

$$a = \sum_i \Delta X^i \qquad \text{and} \qquad b = \sum_j \Delta Y^j$$

to be exchanged through the CFMM such that the following properties hold:

- the constant-function condition within the pool is maintained
- a single price $p^\star$ is shared by all participants in the transaction, regardless of trade direction

The problem is to determine the value of $p^\star$, where throughout $p^\star$ is quoted in Y per X and we restrict to $p^\star > 0$. Denote the aggregate result from swapping (a, b) respectively as

$$\left(y = ap^\star,\, x = \frac{b}{p^\star}\right).$$

Since individual payouts are allocated proportionally to batch contribution, the problem reduces to finding a fair price for two simultaneous trades: one selling a of X, the other selling b of Y. Fees are neglected at first for simplicity.

CPMM case. Let a CPMM have token reserve state (X, Y) with $XY = k$. We show that the uniform batch price for the aggregate sold amounts (a, b) is

$$p^\star = \frac{Y + b}{X + a}.$$

Consider the pool state after a batched transaction—the X-reserves (resp. Y-reserves) will increase by a and decrease by x (increase by b and decrease by y). This leads to the equation

$$(X + a - x)(Y + b - y) = k$$

when applying the CPMM condition. Combining this with the uniform pricing conditions and the initial state of the pool yields the system

$$\begin{cases} XY = k \\ (X + a - x)(Y + b - y) = k \\ x = b/p^\star \\ y = ap^\star \end{cases} \tag{2.1}$$

which reduces to a single equation in $p^\star$:

$$(X + a - b/p^\star)(Y + b - ap^\star) = XY$$

and is solved using the quadratic formula. Its two roots are

$$p^\star \in \left\{ \frac{b}{a}, \frac{Y + b}{X + a} \right\} . \tag{2.2}$$

We identify the first root b/a as the trivial off-pool barter price. It ignores the AMM state, is degenerate when a or b is zero, and does not converge to Y/X in the limit of infinitesimal trade size, so we discard it.

The second root generalizes the standard CPMM transaction in a single trade direction. This is seen by considering a batch that contains only one direction of trade flow, in which case $p^\star$ reduces to the execution price of a standard CPMM. For example, when selling a of token X in a standard CPMM, y is determined by $(X + a)(Y - y) = XY$, and the execution price is

$$\frac{y}{a} = \frac{1}{a} \cdot \frac{aY}{X + a} = \frac{Y}{X + a} ,$$

which indeed coincides with $p^\star$ at $b = 0$. The analogous result is reached for an aggregate sold amount of $(0, b)$. We further demonstrate in §4 that transacting at $p^\star$ interpolates between the two possible outcomes where the aggregate sold amounts (a, b) are processed one-by-one in a standard CPMM. The spot price after batch execution is given by the updated reserve ratio

$$\frac{Y + b - ap^\star}{X + a - b/p^\star} = \frac{Y}{X} \left(\frac{1 + b/Y}{1 + a/X} \right)^2 .$$

This expression shows that $p^\star$ and the post-batch spot price coincide if and only if $a/X = b/Y$, in which case the two are also equal to the pre-batch spot price Y/X.

2.3 Fee policy

We apply a static fee rate $f \in [0, 1)$ to the outputs of swaps in a batch. Fees are split by governance-set shares

$$f = f_{\text{LP}} + f_{\text{crank}} + f_{\text{protocol}} ,$$

rewarding liquidity providers (LPs), settlement keepers, and the protocol, respectively. The LP share accrues into pool reserves to increase LP claims, and the remaining shares are paid out according to the settlement rules detailed in §3.

Trader payouts. Each order is filled at the batch price after applying the static fee:

$$\text{for X} \to \text{Y}: \quad y^i = (1-f)\,\Delta X^i\,p^\star\,, \qquad \text{for Y} \to \text{X}: \quad x^j = (1-f)\,\frac{\Delta Y^j}{p^\star}\,.$$

Per-batch accumulation. Aggregating across the batch, the total fees collected in each token are

$$(F_\text{X}, F_\text{Y}) = \left(\frac{f\,b}{p^\star},\ f\,a\,p^\star\right).$$

At settlement, these fees are realized deterministically: because they are assessed on outputs at the uniform price, $p^\star$ is computed on pre-fee flow (a, b) and fee accounting is order-invariant within the batch, with the LP share compounding into reserves and the keeper and protocol shares transferred to their respective vaults.

2.4 Liquidity provision

Liquidity providers add or remove liquidity by minting or burning LP tokens. As in a traditional AMM, each LP token represents a claim on the pool liquidity in proportion to its reserves. In SUPA, all LP adjustments in a slot are evaluated against the pool reserves at the beginning of the slot which are taken as a snapshot and held in the *snapshot batch account*. The crank mechanism handles token mints, burns, and transfers; implementation details in §3 show how the pool state is maintained such that LP adjustments are applied after the settlement of the slot of their submission and before the settlement of subsequent slots.

This sequencing prevents just-in-time liquidity adjustments. These adjustments could take the form of adding and removing liquidity immediately before and after a batch to extract fees, or withdrawing right before a batch clears to avoid the impact of large price changes. Such behavior is identified by some authors [27, 28] as a form of MEV attack, the executability of which can lead to less liquidity overall at an ecosystem level.

All LP adjustments within a slot maintain the same pool reserve ratio by definition, so they can be applied in any order. They all take place after the batch settlement in a slot and therefore must account for any resulting change in pool reserves. Liquidity additions must be submitted according to the reserve ratio before batch settlement, and then the mismatch is refunded after the batch clears via a dedicated refund crank.

Liquidity addition refund example. If the last post-batch effective reserve ratio is 2, but the batch execution moves the reserve ratio to 2.5, then a depositor who supplied 100 X and 200 Y needs only 80 X and 200 Y to match the post-trade ratio. The remaining 20 X is refunded.

3 Implementation

This section translates the mechanism overviewed in §2 into an example framework of Solana program objects and instructions. SUPA makes use of Solana's program-derived addresses (PDAs) to maintain a deterministic mapping between orders and metadata. It involves three levels of organization: *batches*, which are composed of *chunks*, each of which contains some number of *orders*. Batches are split into fixed-size order chunks that are processed by a crank (see "Crank mechanism" below).

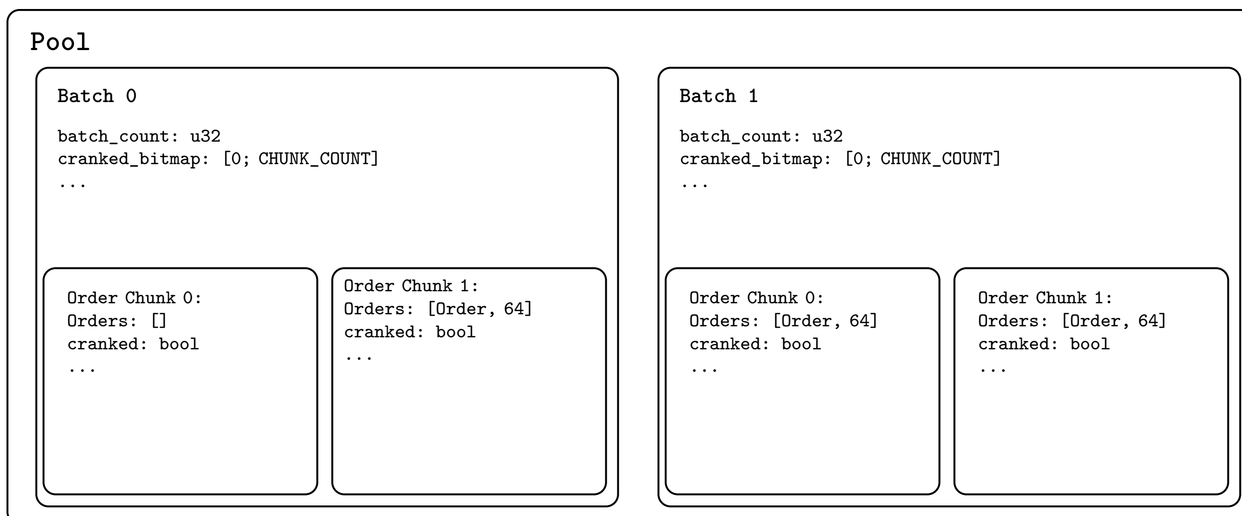

Figure 3.1: Schematic of the SUPA PDA structure. Program objects are further described in Appendix A. *Seeds* are the byte strings/indices used together with the program ID to deterministically derive each PDA; the shown seeds indicate how addresses are laid out (e.g., `"batch"` + BATCH_INDEX).

PDA design. A schematic of the PDA structure is shown in Fig. 3.1. At the topmost level, the pool PDA stores metadata and is the owner of the batch PDAs & order chunk PDAs. In the middle, a pool's batch PDAs are structured as a circular buffer to avoid extra compute costs associated with repeatedly opening and closing accounts. At the lowest level, the order chunk PDAs store up to a fixed number of orders, which is primarily set by the maximum number of writable addresses supported by a single Solana instruction. The size of an order chunk is ultimately a balance between the compute cost of settling a single chunk and the number of times the crank instruction must be called to clear an entire batch. Each batch is initialized with a fixed number of order chunks, which can be increased in response to volume.

Instruction set. Listed below are the instructions involved in the operation of a SUPA CPMM. At a high level, they support calling record and escrow intent, fixing batch parameters, calling payouts, marking entries as consumed, and minting/burning LP tokens.

- `submit_order:` Escrows the sold token from the user into the pool vault, records the order in the current slot's trade-order set, and refreshes indicative batch aggregates (e.g., provisional $p^\star$). If the previous slot has not been committed yet, this call first triggers the commitment for that slot. It also validates the properties of each order.
- `settle_order:` Processes a single eligible set of submitted orders within a slot. It sends payouts at the batch price $p^\star$ from the pool vaults, marks the orders consumed, and emits the keeper reward. If parameters were not yet committed, the first call to this instruction for a particular batch will commit the batch data and then settle the order chunk.
- `add_liquidity` / `remove_liquidity:` For adds: escrows user deposits into pool vaults in the pre-batch ratio and records intents. For removes: records intents which will result in LP token burns once the removal crank executes.

- `settle_additions`: Adjusts deposit size to the committed post-trade ratio, mints LP tokens, then marks entries consumed and emits the keeper reward.
- `settle_removals`: Burns LP tokens and redeems tokens at the post-trade ratio to the user, then marks entries consumed and emits the keeper reward.
- `settle_refunds`: Sends refund amounts from respective vaults to users if there are refunds to process, then marks entries consumed and emits the keeper reward.
- `claim_crank_reward`: Idempotent instruction that pays the keeper their share for a previously processed chunk (normally paid inline by the execute calls, but exposed in case of partial failure).
- `sweep_protocol_fees`: Transfers accumulated protocol fees to the protocol-treasury PDA; callable by the treasury authority; does nothing when balances are zero.

Fee realization. LP fees accrue into the pool reserves automatically, so that fee gains are realized when LP tokens are burned for a commensurate claim on the increased reserves. Keeper and protocol shares are realized via the `claim_crank_reward` and `sweep_protocol_fees` instructions, respectively.

Crank mechanism. Settlement is permissionless: any account may call the crank to process one eligible chunk and earn the keeper share f_{crank}. Each crank call processes a single order chunk via `settle_order`—if no chunk is eligible, then it does nothing. Because crank calls incur gas, the fraction f_{crank} of fees compensates[3] the caller. If outstanding batches remain unprocessed for many slots, the protocol prevents overwriting batch accounts that still have data and are not in the default state. This means the maximum time the mechanism can continue running without being cranked depends on the number of snapshot batch accounts allocated in the circular buffer.

To avoid races, the order chunk PDA maintains an `order_count` which enables parallel settlement across chunks while preserving chronological sequencing on the batch level. Keepers can asynchronously settle any chunk that contains submitted orders. Atomic mark-and-consume semantics prevent two keepers from settling the same chunk; a keeper may also bundle multiple valid crank calls into a single atomic transaction.

4 Analysis and discussion

The subsections below quantify some behavior of the uniform batch price $p^\star$, review a number of practical considerations, and identify opportunities to further develop the SUPA framework.

4.1 Trader economics

Eliminating intra-block ordering through uniform batch pricing improves outcomes for traders. Each individual transacts between their best- and worst-case sequential outcomes, while their trades also benefit from oppositely-directed flow. Furthermore, the pool's post-batch spot price lies between the lowest and highest possible spot prices resulting from sequential execution.

CPMM results. All prices in this subsection are quoted in token Y per token X. Let

$$r = \frac{Y}{X}$$

[3] An alternative is $f_{\text{crank}} = 0$ with an external keeper network.

be the pre-trade spot price of a CPMM with invariant $XY = k$, and define auxiliary variables

$$s = \frac{a}{X} \qquad \text{and} \qquad t = \frac{b}{Y}\,.$$

For a single X→Y swap of size a, the execution price, the post-trade spot price, and the updated reserves are

$$p \;=\; \frac{r}{1+s}\,, \qquad r' \;=\; \frac{r}{(1+s)^2}\,, \qquad \left(X(1+s),\, \frac{Y}{1+s}\right)\,.$$

For a single Y→X swap of size b, these quantities are

$$p \;=\; r\cdot(1+t)\,, \qquad r' \;=\; r\cdot(1+t)^2\,, \qquad \left(\frac{X}{1+t},\, Y(1+t)\right)\,.$$

The two spot prices and the execution price are all related by the equation $p = \sqrt{r \cdot r'}$.

Consider a standard CPMM and a SUPA CPMM with the same initial pool state (X, Y). Over one SUPA batch interval, both receive the same set of sell orders. We compare execution prices and spot prices across the two mechanisms. The lowest and highest prices after executing all orders in a standard CPMM respectively correspond to the sequences in which all X-sells precede the Y-sells (case A) and in which all Y-sells precede the X-sells (case B). These two extremes are equivalent to the two possible outcomes in a simplified scenario, wherein all trades in the CPMM are aggregated together into two sell orders: a of token X & b of token Y.

Execution price comparison. Since the standard CPMM processes the sell orders one at a time, it has two execution prices in each case. They are

$$p^{\mathrm{A}}_{\text{X-sell}} = r\cdot\frac{1}{1+s}\,, \qquad p^{\mathrm{A}}_{\text{Y-sell}} = r\cdot\frac{1+t-(1+s)^{-1}s}{1+s} \qquad\qquad (\text{case A})$$

and

$$p^{\mathrm{B}}_{\text{Y-sell}} = r\cdot(1+t)\,, \qquad p^{\mathrm{B}}_{\text{X-sell}} = r\cdot\frac{1+t}{1+s-(1+t)^{-1}t} \qquad\qquad (\text{case B})\,.$$

Writing the batch execution price as $p^\star = r\cdot\frac{1+t}{1+s}$, it follows by inspection that

$$p^{\mathrm{A}}_{\text{X-sell}} \le p^\star \le p^{\mathrm{B}}_{\text{X-sell}} \qquad \text{and} \qquad p^{\mathrm{A}}_{\text{Y-sell}} \le p^\star \le p^{\mathrm{B}}_{\text{Y-sell}}\,.$$

That is, for each trade direction, the value of $p^\star$ is always in between the respective execution prices of cases A & B.

Spot price comparison. For a standard CPMM, the final spot price in each case is

$$r^{\mathrm{A}} = r\cdot\left(\frac{1+t+st}{1+s}\right)^2 \qquad (\text{case A})$$

and

$$r^{\mathrm{B}} = r\cdot\left(\frac{1+t}{1+s+st}\right)^2 \qquad (\text{case B})\,.$$

Writing the SUPA CPMM spot price after the batched transaction as $r^\star = r\cdot\left(\frac{1+t}{1+s}\right)^2$, it follows by inspection that

$$r^{\mathrm{B}} \le r^\star \le r^{\mathrm{A}}\,.$$

Transacting at $p^\star$ therefore leaves the SUPA CPMM spot price between the post-transaction spot prices of cases A & B.

Averaging behavior. When trades are small compared to the reserves in the pool, the transaction outcome at $p^\star$ approximates the average outcome over all permutations of the order sequence within a batch. Compared to a given order execution sequence in a standard CPMM, the batched execution at $p^\star$ is better for traders who would have been disadvantaged by their position in the slot and worse for those who would have exploited it. This averaging property is a consequence of processing all the orders as a single update to the pool reserves. It also reduces net price impact when trades occur in both directions. We present a simple simulated example in §4.3 to illustrate this behavior.

Spot price update behavior. Batching changes the way the spot price updates under arbitrage conditions compared to a standard CPMM. Within a standard CPMM, the maximally-profitable arbitrage strategy brings the spot price precisely in line with some external price. In contrast, the maximally-profitable arbitrage strategy in a SUPA CPMM pushes $p^\star$ to match the external price, which produces an over-correction in the post-batch spot price.

For example, consider an idealized scenario with a fixed external price and perfect arbitrage execution. If the spot price is initially 10% greater than the external price, then arbitrage will cause the post-batch spot price to be 10% smaller than the external price. Then, in the next slot, since the spot price is 10% smaller, arbitrage will in turn cause the post-batch spot price to be 10% greater, and so on. This would amount to an oscillatory pattern of price discrepancies, with the amplitude of this oscillation remaining fixed over time.

However, since the perfect arbitrage strategy cannot be executed exactly[4] in practice, the value of $p^\star$ can never truly reach that of the external price, meaning the oscillation is damped compared to the idealized scenario. This would allow the spot price in a SUPA CPMM to track external prices under realistic arbitrage conditions, provided the oscillation decay is sufficiently fast compared to the external price volatility.

4.2 Liquidity provider economics

LVR and arbitrage. The standard description of LVR [24–26] within a constant-function AMM depends only on external price volatility and the form of the invariance equation. In this sense, the batching mechanism detailed here does not change the LVR analysis compared to that of a conventional CPMM. The static fee structure in this preliminary SUPA configuration is consistent with that of many conventional AMMs, but the protection against sandwich attacks is intended to attract greater volume—thereby offsetting LP losses more effectively—in comparison.

It is worth noting that if a successful arbitrage trade is made within a batch, then all participants in that batch transact at the arbitrage price. While this makes no difference to LPs, the batching mechanism effectively redistributes the arbitrage opportunity between all traders in a slot.

4.3 Simulated example

Direct calculation on an example set of transactions and initial pool state offers a way to compare the sequential- and batch-execution scenarios. For the example described in Figs 4.1 & 4.2, the outcomes of all 40,320 possible orderings on a set of eight randomly-generated transactions in a standard CPMM are compared to the batched outcome in a SUPA CPMM.

[4] It involves infinite trades of ever-decreasing size.

The parameters in this example are chosen to describe a simplified model of trading flow. Transaction inputs are drawn from a uniform distribution $\mathcal{U}(0, \Delta_{\max})$, while their directions are determined via fair coin. Values are chosen such that $N_{\text{txn}}\Delta_{\max} < \min\{X, Y\}$, where N_{txn} is the number of transactions, so that the trading volume within a slot does not exceed pool liquidity. Overall, the setup is thus scaled to some arbitrary initial reserve state ($X > 0, Y > 0$); these token reserves are randomly generated to be on the order of 10^3 here.

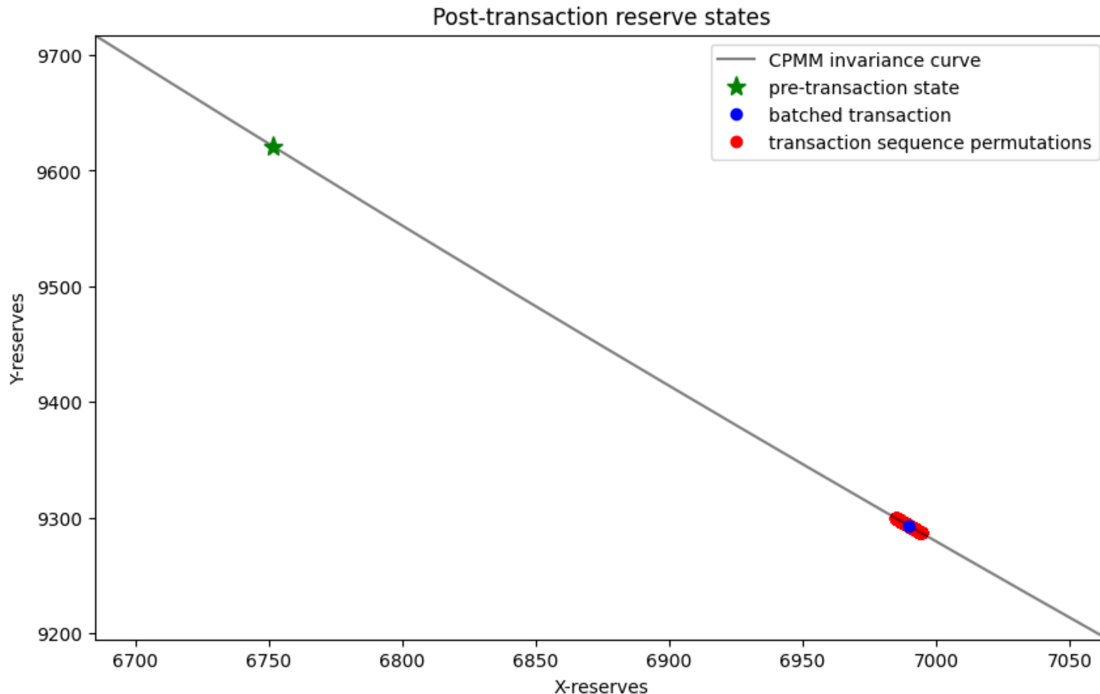

Figure 4.1: CPMM reserve state outcomes after the eight transactions occur. The set of all permutation outcomes (red dots) marks a segment of the $XY = k$ curve within which the batch-outcome (blue dot) is always found. The initial CPMM state is marked with a green star.

This example illustrates how the uniform pricing mechanism captures an average over all permutations of the transaction sequence. Under the reasonable assumption that the flow volume does not exceed any reserves in the pool, this averaging property becomes accurate.

4.4 Additional considerations

4.4.1 Practical tradeoffs and limitations

Quotes and price impact. The price impact on trades in a SUPA-CPMM follow a slightly different pattern compared to a standard CPMM. A simple approach would be to generate quotes directly using the pool price at the beginning of the slot. In this approach, all transactions in a batch encounter the same price impact, determined by the initial spot price and the final value of $p^{\star}$. Alternatively, if the indicative batch price is made available in the pool state during each slot, another approach would compare the indicative batch price at each individual order submission with the final value of $p^{\star}$. In that approach, price impact varies between orders depending on the information available when placing them. For the specification of SUPA described here, only simple

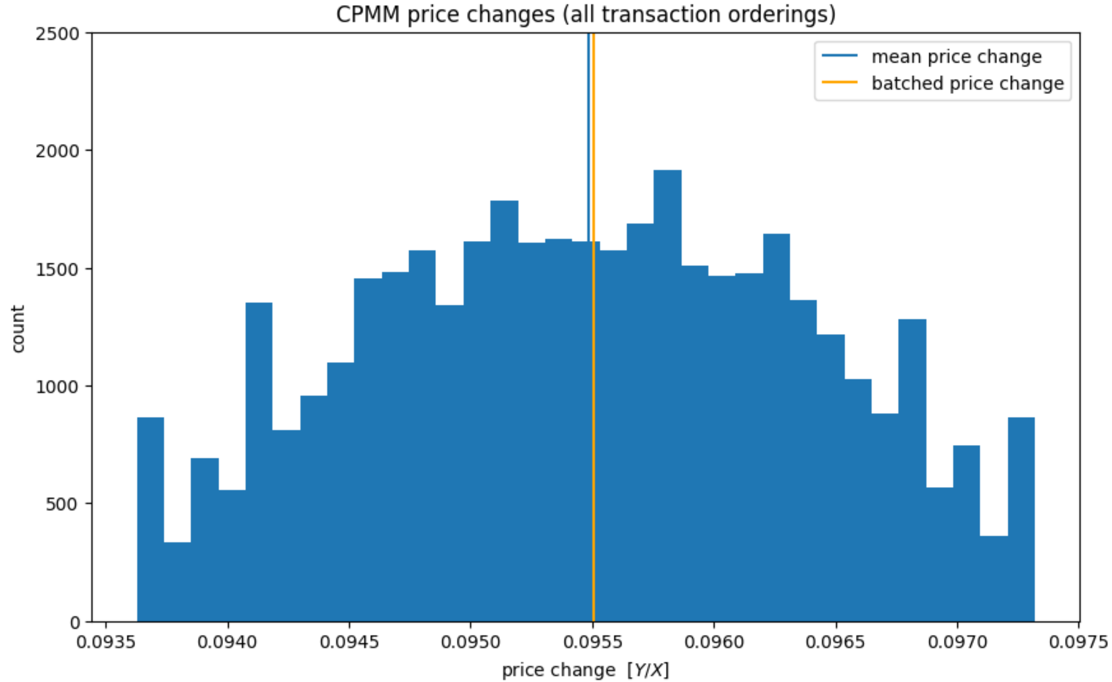

Figure 4.2: Histogram of CPMM price movement for all permutations of the transaction sequence. The blue vertical line marks the mean of the resulting price change distribution, while the orange vertical line marks the price change after batching all trades into a single transaction.

market orders without slippage tolerances are allowed—support for limit orders incurs additional complexity, which we identify as further work.

Pool state complexity. To overcome limitations on pool state storage, batch information is organized in a hierarchy of objects. This allows for a greater batch size in practice, but requires a detailed mechanism for permissionless order settlement, as described in §3.

In practice, user latency still imposes an upper limit on the number of orders that a single slot can accommodate. It is better for this upper limit to be determined by the Solana blockchain structure, rather than the pool architecture itself. Technically, a cap on the number of orders within a slot would allow for MEV in the form of submitting a large number of 'spam' orders to manipulate the result of the batch. This form of MEV can be mitigated by a sufficiently large order cap, or by imposing additional restrictions, e.g., on the number of orders that can be submitted[5] by a single user.

Solana ecosystem integration. Integration into pool aggregators and routers carries with it a number of implications. Absent this, the main utility would be in offering an exchange mechanism for high-volume token pairs that consolidates liquidity and is robust to sandwich MEV. One tradeoff to consider is that combination with standard DEX architectures would in theory re-expose traders to said MEV.

Since the SUPA CPMM contains the standard CPMM architecture, it would be able to switch

[5]This restriction would still exhibit some vulnerability to Sybil-style attacks, however.

between batched and standard operation over different time intervals. This hybrid execution approach would allow it to integrate into DEX routers in the same way as a standard CPMM during the corresponding intervals. It would also introduce price-correcting trade flow to mitigate the price update behavior mentioned in §4.1.

4.4.2 Further work and extensions

A generalization of the standard CPMM structure to batch-process bidirectional trading flow is presented in this work. It points the way toward a number of further extensions, e.g., to different types of transaction inputs, different AMMs, and mechanisms to compensate LPs more effectively.

Limit orders. While our baseline focuses on market orders, a natural extension is native support for limit orders that specify a price predicate on the batch-clearing price $p^\star$ (e.g., buy X with Y iff $p^\star \leq \bar{p}$; sell Y for X iff $p^\star \geq \underline{p}$). Because $p^\star$ is endogenous to the set of included orders, eligibility is coupled: settlement must select a self-consistent subset whose predicates all hold at the resulting $p^\star$. Operationally, this is a call auction selection problem—choose a mutually satisfiable set that maximizes matched volume (or a related objective) subject to the CPMM invariant and per-order constraints. A fully on-chain solver is possible but gas-intensive; a practical design escrows funds and predicates on-chain, applies deterministic tie-break rules, and uses a solver to propose an inclusion set and price together with succinct proofs of validity/maximality (see *Verifiable volume-maximizing clearing* below). As an incremental path, we can first support immediate-or-cancel (IOC) limit orders evaluated against the first eligible batch: if the predicate holds at commitment, the order fills at $p^\star$; otherwise it cancels and refunds. These extensions preserve uniform-price clearing and do not alter the computation of $p^\star$ in §2.2.

Scheduling and other order types. We propose a native *scheduling* primitive that lets traders precommit orders to a future batch. A scheduled order specifies (i) a target slot S_{target} or slot window $[S_{\text{start}}, S_{\text{end}}]$, (ii) an optional limit price (and slippage bound), and (iii) a time-in-force policy. Funds are escrowed at submission; the order becomes eligible only for the first batch whose slot lies inside the declared window. Outside that window the order is canceled or rolled according to its time-in-force. Because execution still occurs at the batch clearing price $p^\star$ (see §2.2), scheduling does not alter the clearing rule; it only filters eligibility by time. This enables users to concentrate execution in predictably liquid periods (e.g., end-of-day "closing" windows, top-of-hour rebalances, or event releases) while retaining SUPA's MEV robustness from uniform-price clearing.

With limit orders enabled, the same framework naturally expresses a richer family of auction-native order semantics familiar from centralized markets: *immediate-or-cancel* (IOC), *fill-or-kill* (FOK), *all-or-none* (AON), *good-till-slot/date* (GTS/GTD), and *imbalance-only* (IO) participation at the auction price. In our setting these are constraints evaluated at commitment: FOK/AON require that the batch clearing at $p^\star$ satisfies a minimum-fill predicate; IOC executes against the first eligible batch and cancels any remainder; GTS/GTD expire automatically at the stated slot or timestamp; and IO orders are included only if they reduce the signed notional imbalance at $p^\star$. These policies are implementable within the existing crank/settlement flow by escrowing inputs, recording per-order predicates and deadlines, and testing those predicates atomically when the batch is committed; orders failing their predicates are deterministically canceled and refunded. To mitigate pre-trade information leakage for large scheduled flow, intents can optionally use commit–reveal (hash-then-reveal) or threshold-encrypted submissions without changing the on-chain verifier.

By combining scheduling, time-in-force, and limit-price predicates with SUPA's uniform-price clearing, the protocol exposes sophisticated order types largely unavailable in current Solana AMMs, while preserving determinism, fairness, and compatibility with permissionless settlement.

Other AMM structures. The uniform-price batching framework is not limited to CPMMs. The core principle extends to any constant-function market maker (CFMM): for a given batch (a, b), solve for the single price $p^\star$ that satisfies both the AMM's invariant and the uniform-payout conditions $y = ap^\star$ and $x = b/p^\star$.

For a general invariant $\Phi(X, Y) = k$, this reduces to finding a root of

$$G(p) \ := \ \Phi(X + a - b/p,\ Y + b - ap) \ - \ k \ = \ 0,$$

with $p > 0$ sought on the feasible interval implied by reserves and batch sizes. As established in our bounds for sequential orderings, a solution $p^\star$ exists within the bracket defined by the two boundary paths (all $X \to Y$ first vs. all $Y \to X$ first).

Stable-swap (Curve-style)
Let $\Phi(x, y) = A(x + y)^2 + xy$ with amplification parameter $A > 0$. The form of $G(p)$ for this system is quadratic, with coefficients determined by (X, Y, a, b, A).

Constant-mean (Balancer-style)
Let $\Phi(x, y) = x^w y^{1-w}$ with weight $w \in (0, 1)$. The batch price solves

$$\left(X + a - b/p\right)^w \left(Y + b - ap\right)^{1-w} \ = \ X^w Y^{1-w},$$

which can for example be solved numerically.

Concentrated liquidity (CLMM)
With liquidity segmented across price ticks, a batch consumes ticks in order; the user's fill is the volume-weighted average of the per-tick uniform prices over the ticks touched.[6]

Verifiable volume-maximizing clearing. In a setting with limit orders, a solver-based call auction can be made verifiable on-chain without publishing a full order book. Let $\mathcal{P}$ be a discrete price grid (e.g., tick levels). From eligible limit orders (after eligibility filtering), define cumulative demand and supply on the grid as

$$B(p) = \text{buy volume willing to pay at least } p, \qquad A(p) = \text{sell volume willing to accept at most } p,$$

so B is non-increasing in p and A is non-decreasing. The matched volume at price p is $M(p) = \min\{B(p), A(p)\}$. Let

$$p^\dagger = \inf\{\, p \in \mathcal{P} : \ B(p) \le A(p) \,\}$$

be the first crossing of demand and supply on the grid (the volume-maximizing price[7] on $\mathcal{P}$).

A solver commits to the grid and cumulative curves via Merkle commitments (hash-tree roots) and proposes $p^\dagger$ together with two bracket prices $p^- < p^\dagger < p^+$ that are adjacent on the grid. The program verifies (i) adjacency of $p^\pm$, and (ii) Merkle proofs for the four cumulative values showing

$$B(p^-) \le A(p^-), \qquad B(p^+) \ge A(p^+).$$

[6] Full tick-walking details are implementation-specific and omitted here.

[7] By monotonicity of cumulative curves on a discrete grid, $M(p)$ is quasi-concave and is maximized at a crossing (or on a crossing plateau). If $B(p) = A(p)$ on an interval, any price in that plateau maximizes M; a deterministic tie-break (e.g., lowest such p) resolves it.

This constant-size on-chain check—the verifier does $O(1)$ arithmetic; witnesses are two Merkle proofs of length $O(\log |\mathcal{P}|)$—certifies that a crossing lies between p^- and p^+, hence $p^\dagger$ maximizes $M(p)$ on the grid. This verification concerns only limit-order auctions and is separate from (though compatible with) the SUPA CPMM batch price detailed in §2.2. Implementation within the SUPA framework would depend on the protocol supporting limit orders or private/commit–reveal[8] intents.

LVR recapture. The most effective LVR mitigation structures [22, 23, 29, 31] are those that respond to price volatility. One of the main approaches [30, 32] involves a dynamic overall trading fee based on volatility; these fees can furthermore be dynamically allocated between LPs, traders, and the platform. Imposing higher fee rates during high volatility can recapture LVR for LPs, while lower fees during low volatility incentivize trading activity in the pool.

Another main approach involves some form of auction mechanism [33, 34] that directly reallocates arbitrage losses to LPs. Larger price discrepancies during periods of high volatility encourage greater participation in auctions. Various approaches [37–42] have been proposed that aim to perform this recapture in different ways. The problem of designing an effective recapture strategy within the SUPA framework that responds to volatility is worthy of further investigation.

5 Conclusion

The problem of sandwich MEV is addressed here by batching order flow within each slot into a single, collective transaction. SUPA CPMM architecture directly generalizes the standard CPMM to support this type of discrete-time batching operation in a fully on-chain manner. It builds on the decentralized, discrete-time setting of the Solana blockchain, combining the benefits of auction mechanisms and AMM protocols in a natural way. According to a Solana MEV report [43] from Helius, a program responsible for "nearly half of all sandwich attacks on Solana" extracted a profit of approximately \$13 million over a 30-day period in late 2024—if, say, 3% of these victim transactions occurred in a sandwich-free exchange, they would have collectively saved over \$10k per day. There are significant gains to make available to DEX users by preventing sandwich MEV.

Under reasonable assumptions, uniform batch pricing approximates the average outcome over all possible transaction sequences within a slot, meaning all participants receive a fair trade. If only a single order is placed, this setup reduces to that of a standard CPMM. A simple implementation is detailed here that involves a permissionless crank mechanism to settle batched transactions chunk-by-chunk, as well as static fees to incentivize crank execution and partially mitigate LVR. Several refinements to this simple configuration are identified, such as enabling more complex order types, generalizing to other CFMM architectures, and fine-tuning incentive alignment between traders and LPs.

References

[1] Budish, Eric, Peter Cramton, and John Shim. "The high-frequency trading arms race: Frequent batch auctions as a market design response." The Quarterly Journal of Economics 130.4 (2015): 1547-1621. `doi:10.1093/qje/qjv027`.

[8] By *commit–reveal intents* we mean orders submitted as cryptographic commitments (or via threshold encryption) and revealed later to reduce pre-trade manipulation. We note also that studies of call markets in traditional finance have shown [35, 36] that a closed-book design leads to lower price volatility compared to an open-book design.

[2] Eibelshäuser, Steffen, and Fabian Smetak. "Frequent batch auctions and informed trading." (2022). doi:10.2139/ssrn.4065547.

[3] Haas, Marlene, and Marius A. Zoican. "Discrete or continuous trading? HFT competition and liquidity on batch auction markets." Université Paris-Dauphine (2016). doi:10.2139/ssrn.2738071.

[4] Wah, Elaine, Dylan Hurd, and Michael Wellman. "Strategic market choice: Frequent call markets vs. continuous double auctions for fast and slow traders." EAI Endorsed Transactions on Serious Games 3.10 (2016). doi:10.4108/eai.8-8-2015.2260356.

[5] Aldrich, Eric M., and Kristian López Vargas. "Experiments in high-frequency trading: comparing two market institutions." Experimental Economics 23.2 (2020): 322-352. doi:10.2139/ssrn.3154070.

[6] Jagannathan, Ravi. "On frequent batch auctions for stocks." Journal of Financial Econometrics 20.1 (2022): 1-17. doi:10.1093/jjfinec/nbz038.

[7] Bogousslavsky, Vincent, and Dmitriy Muravyev. "Who trades at the close? Implications for price discovery and liquidity." *Journal of Financial Markets* 66 (2023): 100852. doi:10.1016/j.finmar.2023.100852.

[8] Ozenbas, Deniz, and Robert Schwartz. "The Return of the Call Auction." Journal of Portfolio Management, Forthcoming (2022). doi:10.2139/ssrn.4115595.

[9] Jegadeesh, Narasimhan, and Yanbin Wu. "Closing auctions: Nasdaq versus NYSE." *Journal of Financial Economics* 143.3 (2022): 1120–1139. doi:10.1016/j.jfineco.2021.12.003.

[10] Raillon, Franck. "Growing importance of the closing auction in share trading volumes." Autorité des marchés financiers (Risk & Trend Mapping Note), October 2019. https://www.amf-france.org/sites/institutionnel/files/2020-02/growing-importance-of-the-closing-auction-in-share-trading-volumes.pdf.

[11] Basar, Shanny. "Closing Auction Volumes Rise in Europe." *Markets Media* (January 4, 2021). https://www.marketsmedia.com/closing-auction-volumes-rise-in-europe/.

[12] Pagano, Michael S., and Robert A. Schwartz. "A closing call's impact on market quality at Euronext Paris." Journal of Financial Economics 68.3 (2003): 439-484. doi:10.1016/S0304-405X(03)00073-4.

[13] Pagano, Michael S., Lin Peng, and Robert A. Schwartz. "A call auction's impact on price formation and order routing: Evidence from the NASDAQ stock market." Journal of Financial Markets 16.2 (2013): 331-361. doi:10.1016/j.finmar.2012.11.001.

[14] Friedman, Daniel. "How trading institutions affect financial market performance: some laboratory evidence." Economic inquiry 31.3 (1993): 410-435. doi:10.1111/j.1465-7295.1993.tb01302.x.

[15] Cason, Timothy N., and Daniel Friedman. "Price formation in single call markets." Econometrica: Journal of the Econometric Society (1997): 311-345. doi:10.2307/2171895.

[16] Vigan, Lladrovci. “Analysis and Evaluation of the CoW Protocol for MEV Mitigation on the Ethereum Blockchain.” 2023. Technical University of Munich, Master’s thesis. *Sebis Public Website*. https://wwwmatthes.in.tum.de/file/168hiqf8uay53/Sebis-Public-Website/-/Master-s-Thesis-Vigan-Lladrovci/Vigan%20Lladrovci%20Master%20Thesis.pdf

[17] Nag, Abhimanyu, Madhur Prabhakar, and Tanuj Behl. “A Coincidence of Wants Mechanism for Swap Trade Execution in Decentralized Exchanges.” arXiv:2507.10149 (2025).

[18] Tomar, Pari. “dYdX v4 Whitepaper — The Ultimate Technical Deep Dive.” 2025. https://medium.com/%40tomarpari90/dydx-v4-whitepaper-the-ultimate-technical-deep-dive-4c95f499d3bc. Accessed Sept. 2025.

[19] White, Dave, McCarthy, Dan, and Adams, Hayden. “TWAMM: Time-Weighted Average Market Maker (TWAMM)” 2021. https://www.paradigm.xyz/2021/07/twamm. Accessed Sept. 2025.

[20] Chan, T. H., Ke Wu, and Elaine Shi. “Mechanism Design for Automated Market Makers.” arXiv:2402.09357 (2024).

[21] CoW Protocol Team. CoW Protocol: A Decentralized Trading Protocol. White Paper, 2021. Available at https://cow.fi/.

[22] Milionis, Jason, Xin Wan, and Austin Adams. “Flair: A metric for liquidity provider competitiveness in automated market makers.” arXiv preprint, arXiv:2306.09421 (2023).

[23] Fritsch, Robin. “A note on optimal fees for constant function market makers.” Proceedings of the 2021 ACM CCS Workshop on Decentralized Finance and Security. 2021. doi:10.1145/3464967.3488589.

[24] Milionis, J., Moallemi, C., Roughgarden, T., and Zhang, A. (2024). “Automated Market Making and Loss-Versus-Rebalancing.” arXiv preprint, arxiv:2208.06046.

[25] Alexander, Abe, and Lars Fritz. “Impermanent loss and loss-vs-rebalancing I: some statistical properties.” arXiv preprint, arXiv:2410.00854 (2024).

[26] Dewey, Richard, and Craig Newbold. “The pricing and hedging of constant function market makers.” arXiv preprint, arXiv:2306.11580 (2023).

[27] Capponi, Agostino, Ruizhe Jia, and Brian Zhu. “The paradox of just-in-time liquidity in decentralized exchanges: More providers can sometimes mean less liquidity.” arXiv preprint, arXiv:2311.18164 (2023).

[28] Bayraktar, Erhan, Asaf Cohen, and April Nellis. “DEX specs: A mean field approach to DeFi currency exchanges.” arXiv preprint, arXiv:2404.09090 (2024).

[29] Alexander, Abe, and Lars Fritz. “Fees in AMMs: A quantitative study.” arXiv preprint, arXiv:2406.12417 (2024).

[30] Yan, C., Keol, S., Co, X., and Leung, N. (2025). Better market Maker Algorithm to Save Impermanent Loss with High Liquidity Retention. arXiv preprint, arXiv:2502.20001.

[31] Campbell, S., Bergault, P., Milionis, J., Nutz, M. (2025). “Optimal Fees for Liquidity Provision in Automated Market Makers.” arXiv preprint, arXiv:2508.08152.

[32] Lebedeva, I., Umnov, D., Yanovich, Y., Melnikov, I., Ovchinnikov, G. (2025). "Dynamic Fee for Reducing Impermanent Loss in Decentralized Exchanges." arXiv preprint, arXiv:2506.03001.

[33] Adams, Austin, et al. "am-amm: An auction-managed automated market maker." arXiv preprint, arXiv:2403.03367 (2024).

[34] McMenamin, Conor, Vanesa Daza, and Bruno Mazorra. "An automated market maker minimizing loss-versus-rebalancing." The International Conference on Mathematical Research for Blockchain Economy. Cham: Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-48731-6_6.

[35] Arifovic, Jasmina, and John Ledyard. "Call market book information and efficiency." Journal of Economic Dynamics and Control 31.6 (2007): 1971-2000. doi:10.1016/j.jedc.2007.01.006.

[36] Oehler, Andreas, and Matthias Unser. "Market transparency and call markets." (1998). doi:10.2139/ssrn.284957.

[37] Jepsen, Waylon, and Colin Roberts. "Analysis of the RMM-01 Market Maker." arXiv preprint, arXiv:2310.14320 (2023).

[38] Kositwattanarerk, Wittawat. "Dynamic exponent market maker: Personalized portfolio manager and one pool to trade them all." Blockchain: Research and Applications (2025): 100274. doi:10.1016/j.bcra.2025.100274.

[39] Jeong, Yeonwoo, et al. "Efficient liquidity providing via margin liquidity." 2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE, 2023. doi:10.1109/ICBC56567.2023.10174867.

[40] Nadkarni, Viraj, Sanjeev Kulkarni, and Pramod Viswanath. "Adaptive curves for optimally efficient market making." arXiv preprint, arXiv:2406.13794 (2024).

[41] Bergault, Philippe, et al. "Automated market makers: Mean-variance analysis of lps payoffs and design of pricing functions." Digital Finance 6.2 (2024): 225-247. doi:10.1007/s42521-023-00101-0.

[42] Nadkarni, Viraj, et al. "ZeroSwap: Data-driven Optimal Market Making in DeFi." arXiv preprint, arXiv:2310.09413 (2023). doi:10.1007/978-3-031-78676-1_12.

[43] Lostin. "Solana MEV Report: Trends, Insights, and Challenges" 2025. https://www.helius.dev/blog/solana-mev-report. Accessed October 2025.

A Core Program Objects

This appendix summarizes the core on-chain objects used by SUPA. It presents (i) a conceptual overview for quick comprehension and (ii) a minimal code-like listing for technical precision. Operational metadata (e.g., protocol fee vaults, refund ATAs, audit counters) is intentionally omitted here and can be found in the public IDL/repository.

Conceptual Digest

Table A.1: Conceptual overview of core on-chain objects.

Object	Key Fields & Purpose
`PoolState`	Canonical CPMM state (effective reserves (X, Y), invariant k), swap fee, and a pointer to the currently collecting batch.
`Batch`	One per slot. Aggregates total sell amounts (a, b), computes $p^\star$, and tracks which settlement chunks remain via bitmap.
`OrderChunk`	Fixed-size container of user `Order`s within a `Batch`, settled atomically by a single crank call.
`Order`	Market-sell only; records owner payout account, amount in, and trade direction. Payout is determined by $p^\star$.
`LPAdjustmentChunk`	Fixed-size container of `LPAdjustments` to be applied after trades, at the post-transaction spot price; LP tokens are minted for deposits (which are resized to the post-transaction spot price and excess refunded), or are burned for withdrawals.

Minimal Interface Listing

The following pseudo-code is structured to resemble the Rust language. It specifies the minimal fields needed to implement batch collection, uniform-price clearing, and liquidity update sequencing. Fixed-point numeric fields use a deterministic format (e.g., Q64.64).

Listing 1: Minimal on-chain accounts for SUPA

```
/// Canonical CPMM state. Matches paper notation: X, Y, k; fees on outputs (see Fee
    policy).
#[account]
pub struct PoolState {
    pub X: u128,              // effective reserves of token A
    pub Y: u128,              // effective reserves of token B
    pub k: u128,              // invariant X*Y checked/updated at settlement
    pub fee_bps: u16,         // swap fee (outputs only)
    pub current_batch_id: u64 // batch currently collecting orders
    // ... (mints, vaults, fee splits omitted)
}

/// One batch per slot; aggregates flow and fixes p* exactly once (Update in Fig. 1).
#[account]
pub struct Batch {
```

```
    pub slot: u64,               // slot index for this batch
    pub status: BatchStatus,     // {Collecting, ParametersFixed, Exhausted}
    // Aggregated market-sell inputs:
    pub a_total: u128,           // sum X->Y inputs (X)
    pub b_total: u128,           // sum Y->X inputs (Y)
    // Clearing parameters (fixed at Update):
    pub p_star: u128,            // uniform clearing price p* (fixed-point)
    // Work bitmaps for permissionless settlement:
    pub order_chunk_bitmap: u64,     // trade chunks remaining
    pub lp_adj_chunk_bitmap: u64     // LP adjustment chunks remaining
    // ... (aux bookkeeping omitted)
}

#[repr(u8)]
pub enum BatchStatus { Collecting=0, ParametersFixed=1, Exhausted=2 }

/// Bounded set of orders executed atomically by a single crank.
#[account]
pub struct OrderChunk {
    pub batch_id: u64,
    pub count: u8,
    pub orders: [Order; M],    // M sized to Solana account/compute limits
}

/// Market-sell order. No per-order limits or slippage; payouts computed at p*.
pub struct Order {
    pub owner_output_ata: Pubkey, // destination for payout
    pub amount_in: u64,           // input amount
    pub is_token_a: bool,         // true: sell A for B ; false: sell B for A
    // Optional recovery/refund fields are omitted in the white paper.
}

/// LP adjustments are executed AFTER trades at r_post = Y'/X' (see Liquidity provision).
#[account]
pub struct LPAdjustmentChunk {
    pub batch_id: u64,
    pub count: u8,
    pub items: [LPAdjustment; K], // processed after all OrderChunks
}

pub enum LPKind { Add, Remove }

pub struct LPAdjustment {
    pub kind: LPKind,
    // For Add: escrowed deposits; right-sized to r_post, excess refunded, LP shares
    minted.
    pub dep_a: u64,               // escrowed token A (0 if Remove)
    pub dep_b: u64,               // escrowed token B (0 if Remove)
    // For Remove: amount of LP shares to burn; redeem (A,B) at r_post.
    pub lp_shares: u64,           // 0 if Add
    // Payout/refund destinations:
    pub owner_token_a_ata: Pubkey,
    pub owner_token_b_ata: Pubkey
}
```

B Glossary of Terms

Batch Auction A market mechanism where orders are collected over a fixed interval (one Solana slot) and cleared together at a single, uniform price.

Chunk A component of a batch, sized to fit within a single Solana transaction. A batch is settled by sequentially processing its chunks via the crank instruction.

CPMM Constant-Product Market Maker; an AMM that maintains the invariant $X \times Y = k$, where X and Y are the reserves of two assets.

Crank A permissionless instruction that any user can call to process a chunk of orders from a previously settled batch, disbursing funds to traders and applying LP adjustments.

Effective reserves Canonical pool reserves (X, Y) that define state and change only when the Update commits a batch.

Indicative values Per-slot, non-state aggregates (e.g., tentative $p^{\star}$, projected reserves/LP supply) computed during collection and overwritten at Update.

LVR Loss-versus-Rebalancing; the loss liquidity providers incur due to arbitrage when the AMM's price lags the external market price. It represents the opportunity cost of providing liquidity compared to holding the assets.

MEV Maximal Extractable Value; profit extracted by block producers or searchers by leveraging their ability to reorder, insert, or censor transactions. Sandwich attacks are a common form of MEV.

PDA Program-Derived Address; a Solana account whose address is deterministically derived from a set of seeds and a program ID, allowing a program to have authority over it.

Seed A byte string (often with indices) used with a program ID to deterministically derive a PDA; different seed tuples define disjoint address namespaces the program controls.

SUPA Solana Uniform-Price Auction, the protocol described in this paper.

Uniform Clearing Price The single price ($p^{\star}$) at which all trades in a batch are executed, calculated as $p^{\star} = (Y + b)/(X + a)$ to preserve the CPMM invariant.