Angstrom v1 Whitepaper

July 2025

Karthik Srinivasan
karthik@sorellalabs.xyz

Ludwig Thouvenin
ludwig@sorellalabs.xyz

Ciamac Moallemi
ciamac@gsb.columbia.edu

Abstract

Angstrom v1 is a non-custodial decentralized exchange built for Ethereum as a Uniswap v4 hook. Angstrom improves upon existing decentralized exchange designs by holding high-frequency off-chain auctions in parallel to Ethereum block production, redistributing arbitrage value to liquidity providers and swappers while maintaining composability with the rest of the ecosystem.

1 Introduction

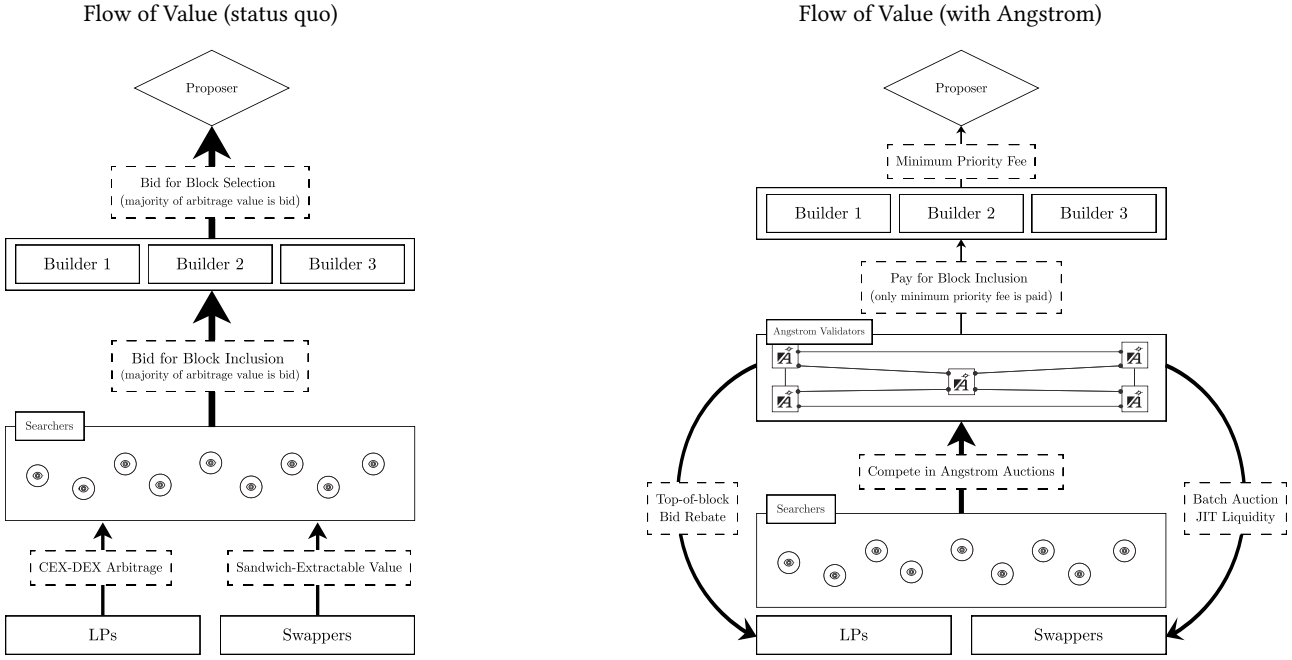
Decentralized exchange (DEX) protocols have evolved rapidly over the past several years, starting with constant product automated market makers (AMMs) like Uniswap v1, expanding to the more generalizable ERC-20 pairs of Uniswap v2 [1], and introducing the concept of concentrated liquidity in Uniswap v3 [2]. Most recently, Uniswap v4 [3] adds the notion of *hooks*, which supply an architectural framework for customizable pool behavior. However, two fundamental issues in decentralized exchange design remained unresolved:

(1) **Loss-Versus-Rebalancing:** Liquidity providers (LPs) in typical AMMs suffer losses when prices on more liquid external markets deviate from on-chain pool prices, as arbitrageurs execute swaps to pick off stale quotes that the AMM provides. As described in the LVR (Loss-Versus-Rebalancing) literature [4], such arbitrage imposes a systematic and quantifiable cost on on-chain LPs, diminishing the attractiveness and overall viability of providing liquidity.
(2) **Sandwich Attacks on Swappers:** Block proposers can reorder transactions for profit, incentivizing arbitrage from swappers in the form of sandwiching. This creates an uneven playing field for users executing trades, leading to billions of dollars in value extracted from unsuspecting swappers.

Angstrom is a new DEX protocol, built as a Uniswap v4 hook, that tackles these issues through a fundamentally different model of sequencing trades and redirecting arbitrage value. Specifically, it integrates two auctions for each pool that run in parallel to Ethereum block production:

- The **top-of-block auction** is a first-price auction for LVR value, awarding a single winning bidder the option to trade against the AMM at zero swap fees. The winning bid is then split between the AMM ticks that provided active liquidity in the swap such that the ticks receiving the bid trade at the same effective execution price. This distributes the arbitrage value arising from intra-block price deviations on external reference venues (e.g. Binance) back to on-chain liquidity providers.
- The **uniform-clearing batch auction** is executed afterwards, where all remaining limit orders in the off-chain order book are cleared at the same price along with the underlying AMM liquidity in the state resulting from executing the top-of-block swap against it. By batching user orders in a single clearing event, Angstrom obviates malicious reordering of transactions, thereby eliminating sandwich attacks on user swaps.

Angstrom's off-chain infrastructure orchestrates these auctions in a decentralized, censorship-resistant fashion. Orders in both auctions are signed EIP-712 meta-transactions and are considered for inclusion by Angstrom validators upon being received and



I. Competition in the MEV supply chain directs value from LPs and swappers to block proposers

II. Angstrom redirects arbitrage value back to LPs and swappers

Figure 1. MEV supply-chain flow of value with and without Angstrom. Arrow thickness indicates value magnitude.

validated. The aggregate set of valid orders, determined by the consensus protocol, is then used to derive a canonical Angstrom transaction for the given slot which is then submitted for on-chain settlement. The canonical transaction is an aggregation of all filled orders in both auctions over all Angstrom-supported pools. The exact clearing per pool is determined by selecting the winning top-of-block order, computing the resultant AMM state after the top-of-block swap has executed, then executing the uniform-clearing batch auction with all non-top-of-block orders on the computed AMM state. The resultant clearing event is compressed into one composite swap through the pool, a list of per-tick fee updates, and a list of filled orders to settle at their respective quantities. Liquidity itself is stored in specialized pools deployed on the Uniswap v4 Pool Manager contract, thereby inheriting the capital efficiency of Uniswap v3's concentrated liquidity design, while also leveraging the new extensibility and gas efficiencies inherited from Uniswap v4's singleton architecture.

With this model, Angstrom fundamentally reshapes the capture and distribution of arbitrage value in decentralized trading. Rather than allowing block producers to maximally profit from arbitrage at the expense of LPs and swappers, Angstrom channels market competition to an *application-controlled execution* layer. This competition enables value to flow back to adversely-selected participants within the system, shown explicitly in fig. 1.

Crucially, through determining order execution on the union of all valid meta-transactions received, the protocol enables cancellations before any on-chain interaction by including cancels as another order type. Since cancellations are handled entirely off-chain and bear no on-chain footprint, they incur zero gas. Additionally, should they be submitted prior to Round 1 of the consensus protocol, they are guaranteed to properly resolve and cancel the desired order in the given slot should there be no consensus faults committed by Angstrom validators. Market makers, therefore, can adjust or retract quotes in real time, responding to shifting reference prices, without needing to embed potential cancellation costs into their spreads.

In the sections that follow, we describe Angstrom's architecture and key components in detail. Section 2 introduces the protocol's consensus protocol for staked validators. Section 3 outlines how the off-chain dual-auction design coordinates and enforces the top-of-block and uniform-clearing batch auctions, while composable interactions remain possible through a post-batch pool-unlock mechanism. Section 4.1 examines capital-efficient mitigation strategies the protocol can employ against bidder optionality exploitation, and finally we culminate in a thorough explanation of our smart contract design in Section 5 along with providing gas benchmarks to quantify the on-chain execution savings.

2 Consensus

Angstrom utilizes a custom consensus protocol amongst a decentralized set of staked validators[1] to ensure the off-chain auctions are conducted fairly and robustly in parallel to Ethereum block production. The protocol is responsible for aggregating valid orders, selecting winning top-of-block swaps, executing the uniform-clearing batch auction on the post-top-of-block AMM state, and

[1]Economic security can be enforced by an Angstrom-specific staking contract or by opting into a restaking service, such as EigenLayer[5]

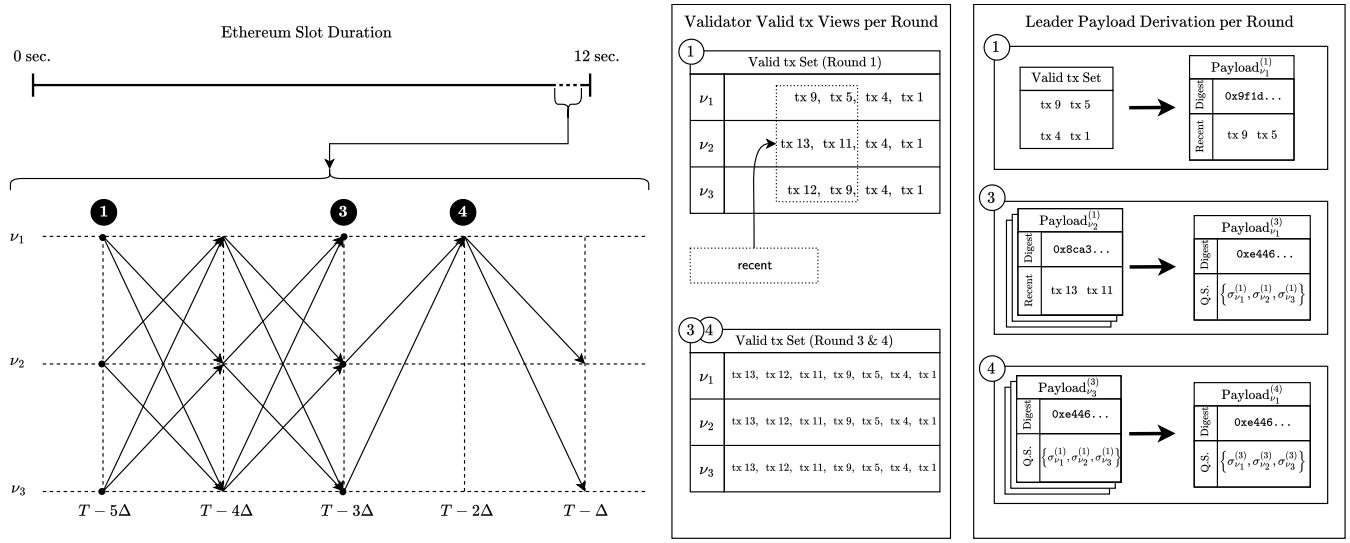


Figure 2. We show the consensus protocol proceeding in the happy path with leader ν_1. Each validator forms its Round-1 payload by computing its LocalDigest from its current valid transaction set and appending its set of recent orders seen within the last Δ-interval, shown in ❶. The next signed objects are emitted at ❸, where each validator computes its MergeDigest from its new valid transaction set and forwards the leader the signed object derived from the digest and the quorum signatures of other validators' Round-1 payloads that were used in the computation. Finally, at ❹, the leader aggregates the Round-3 signed objects, which all share the same MergeDigest, to produce the Round-4 signed object where MergeDigest* equals the common MergedDigest in the Round-3 signed objects. The leader then broadcasts this back to the other validators who can all deterministically derive the canonical transaction to submit for inclusion.

aggregating all the top-of-block swaps, batch orders, and LP fees across all supported pools into a single canonical transaction to be submitted for settlement in each slot. Unlike traditional consensus systems focused on liveness and partition tolerance, Angstrom's consensus protocol is tailored to the demands of high-frequency auctions, where latency, censorship resistance, and no-optionality properties are most important.

When one bidder in a common value auction can submit bids later than other participants, we have what is known as a "last look" problem—see, e.g., [6, 7]. This timing advantage can give an unfair informational advantage to that bidder, which undermines the fairness and efficiency of the auction. Angstrom's consensus mechanism is meant to limit the scope and impact of these free options.[2]

Each Angstrom slot corresponds to an Ethereum slot. In each slot, the protocol aims to create a transaction that Angstrom validators can submit for inclusion in that same block. For the purposes of this section, we assume that validators are able to get a transaction included in that block, i.e. there is no censorship from the Ethereum block proposer.

The protocol coordinates a set of $2f + 1$ Angstrom validators to jointly finalize and settle a bundle, determining the highest bid in the priority auctions as well as the set of orders to be cleared in the uniform-clearing batch auction. This section introduces the consensus protocol's design, its assumptions and guarantees, and potential attributable faults that can arise from potentially malfeasant validator behavior.

2.1 Assumptions and Setup

Notational Conventions.

- $\mathcal{V} = \{\nu_1, \ldots, \nu_{2f+1}\}$ denotes the validator set, with at most f Byzantine participants.
- The network is Δ-synchronous; every message sent by an honest validator at time t is received by every honest peer no later than $t + \Delta$.[3]
- $\text{sign}_\nu(\cdot)$ is a deterministic ECDSA signature under ν's public key.
- For every round $r \in \{1, 3, 4\}$ validators emit a single signed message as part of the consensus protocol.[4]

$$m_\nu^{(r)} = \big(\text{slot}, r, \text{Payload}_\nu^{(r)}\big), \qquad \sigma_\nu^{(r)} = \text{sign}_\nu\big(m_\nu^{(r)}\big),$$

  where the payload is specified below. The pair $(m_\nu^{(r)}, \sigma_\nu^{(r)})$ is a *signed object*; Appendix B.2 will treat two conflicting signed objects as evidence of an equivocation fault.
- $\mathcal{S}_\nu^{(r)}$ is the multiset of Round r signed objects received by validator ν from other validators (self-inclusive).

Recent–orders window. A given validator defines their pre-Round-1 recent-order window as

$$\text{recent} := \big\{\text{orders received in } [T - 6\Delta,\ T - 5\Delta)\big\}.$$

[2] We discuss the post-consensus option, another flavor of the last look problem that occurs outside the scope of the consensus protocol, in section 4.1 where we additionally detail a comprehensive mitigation strategy.

[3] For the small, co-located validator set upon launch, Δ can be on the order of tens of milliseconds, far less than Ethereum's slot time.

[4] Only the leader emits a signed message in Round 4

New orders received in this window may not reach all peers before the Round-1 cutoff, so they are explicitly forwarded inside the Round 1 payload.

Quorum Signature Aggregation For a validator v and $r \in \{1, 3\}$ define

$$\mathsf{QuorumSigs}_v^{(r)} = \left\{\sigma_\mu^{(r)} \mid m_\mu^{(r)} \in \mathcal{S}_v^{(r)}\right\}$$

as the aggregated set of signatures from all Round r signed objects that validator v received.

Opaque state commitments. Two deterministic, public algorithms are assumed to exist:

$$\begin{aligned}\mathsf{BuildDigest}(\mathcal{M}) &\longrightarrow \mathsf{LocalDigest} \in \{0,1\}^{256},\\ \mathsf{BuildMerge}(\mathcal{B}) &\longrightarrow \mathsf{MergeDigest} \in \{0,1\}^{256},\end{aligned}$$

where $\mathcal{M}$ is the multiset of transactions observed by the caller during Round 0 and $\mathcal{B} \in \{\mathcal{S}^{(1)}, \mathcal{S}^{(3)}\}$ is the multiset of either Round 1 signed objects received by the caller or Round 3 signed objects received by the leader. If two honest validators input the same $\mathcal{M}$ or $\mathcal{B}$ they obtain identical outputs. Both StateDigest and MergeDigest are aggregate commitments to the full order book state, where the former is a local commitment to the union of all orders seen during Round 0 and the latter is an aggregate commitment inclusive of all orders in the union of all signed objects received.

2.2 Protocol Description

The protocol proceeds as follows:

Round 0 (Gossip). Validators gossip new *place* and *cancel* transactions to their peers. At $t = T - 5\Delta$ every honest v deterministically computes

$$(\mathsf{LocalDigest}_v, \mathsf{Recent}_v) \;\coloneqq\; \big(\mathsf{BuildDigest}(\mathcal{M}_v),\ \mathsf{recent}\big),$$

where $\mathcal{M}_v$ is the multiset of messages received so far. No signed object is emitted in this round.

Round 1 (Local Build). Validator v assembles

$$\mathsf{Payload}_v^{(1)} = \big(\mathsf{LocalDigest}_v, \mathsf{Recent}_v\big)$$

and broadcasts the signed object $m_v^{(1)}$ with signature $\sigma_v^{(1)}$ during $[T - 5\Delta,\ T - 4\Delta)$.

Round 2 (View Relay). To thwart selective forwarding, every validator re-gossips *verbatim copies* of all Round-1 signed objects that arrived before $T - 4\Delta$. No fresh message is created, hence no new signature.

Round 3 (Merge). By $T - 3\Delta$ a validator v has the set $\mathcal{S}_v^{(1)}$ of Round-1 objects it has received from its peers. The validator sets

$$\mathsf{MergeDigest}_v \coloneqq \mathsf{BuildMerge}(\mathcal{S}_v^{(1)}).$$

It then broadcasts signed object $m_v^3 = (\mathsf{slot}, 3, \mathsf{Payload}_v^{(3)})$ with Payload

$$\mathsf{Payload}_v^{(3)} = \Big(\mathsf{MergeDigest}_v, \mathsf{QuorumSigs}_v^{(1)}\Big)$$

and signature $\sigma_v^{(3)} = \mathsf{sign}_v(m_v^{(3)})$ during the interval $[T - 3\Delta,\ T - 2\Delta)$.

Round 4 (Commit). Leader v_ℓ waits for at least f+1 Round-3 signed objects to compute $\mathsf{MergeDigest}^\star \coloneqq \mathsf{BuildMerge}\big(\mathcal{S}_{v_\ell}^{(3)}\big)$. The leader then appends the quorum signatures to this final digest to form the Round-4 payload:

$$\mathsf{Payload}_{v_\ell}^{(4)} = \Big(\mathsf{MergeDigest}^\star,\ \mathsf{QuorumSigs}_{v_\ell}^{(3)}\Big).$$

The leader then broadcasts the corresponding signed object to all other validators within the window $[T - 2\Delta,\ T - \Delta)$, as well as deriving the correct aggregation of meta-transactions to be cleared based on the final state that $\mathsf{MergeDigest}^\star$ commits to and sends this to the public mempool and all block builders.

Round 5 (Submit). The Angstrom smart contract accepts transactions sent by any Angstrom validator. This enables all other Angstrom validators to derive and submit *two transactions* from the $\mathsf{MergeDigest}^\star$ provided in the leader's signed object, which also must correspond with their own Round 3 signed object assuming there were no faults in the consensus procedure during $[T - \Delta,\ T)$.

The first transaction is the aggregate top-of-block and batch auction result for all pools based on the set of valid transactions implied by $\mathsf{MergeDigest}^\star$. The second transaction is *only the top-of-block auction results* for each pool, without the batch auction executed afterwards. Both of these transactions are sent by all other Angstrom validators to revert-protected endpoints of Ethereum block builders with the aggregate transaction and the transaction containing only top-of-block swaps sent with a priority fee marginally.

Since the leader submitted the transaction to the public mempool and all builder endpoints, requiring other validators to only submit to revert-protecting builders ensures that extra gas costs are not imposed on the aggregate system due to all validators submitting the Angstrom transaction for a given slot with all but one reverting upon block inclusion. The submission of the second transaction ensures that, in the event of a double-spending batch order—where a top-of-block bidder aims to exercise a post-consensus option described in section 4.1.2 by adding a malicious, double-spendable order from a Sybil address that is cleared in the batch auction—an Angstrom transaction containing each pool's top-of-block swap is still successfully executed. If the canonical transaction executes successfully without reverting, however, the second transaction is never included due to revert protection.

Round (r)	Window inside slot $[0, T)$	Purpose
0 Gossip	$[0,\ T - 5\Delta)$	Raw tx broadcast
1 Local Build	$[T - 5\Delta,\ T - 4\Delta)$	Compute and gossip LocalDigest
2 View Relay	$[T - 4\Delta,\ T - 3\Delta)$	Re-gossip all R1 payloads
3 Merge	$[T - 3\Delta,\ T - 2\Delta)$	Compute and gossip MergeDigest
4 Commit	$[T - 2\Delta,\ T - \Delta)$	Leader chooses canonical view
5 Submit	$[T - \Delta,\ T)$	All validators send tx to mempool

Upon completion of the consensus protocol for a given slot, the leader must gossip a *consensus trace* to all other validators within some pre-defined grace period to ensure no bid-shading via concealed equivocation occurred in the construction of the Round-4 signed object.[5] The formal specification for the consensus trace is deferred to appendix B.1.

[5]The grace period can be multiple orders of magnitude larger than Δ or even the slot time.

2.3 Faults

Angstrom's consensus protocol is *accountable*: every deviation is associated with a finite, verifiable proof and an on–chain penalty. Concretely, the protocol recognizes the following faults:

(1) **Equivocation.** A validator signs two different payloads for the same round in a single slot.
(2) **Absence.** A validator fails to distribute their Round 1 signed object to all other validators by the deadline $T-4\Delta$. A threshold of $f+1$ LateLocal attestations suffices to prove the absence or tardiness.
(3) **Final View Censorship.** The elected leader fails to distribute their Round 4 signed object to all other validators by the deadline $T-\Delta$. A threshold of $f+1$ LateFinal attestations suffices to penalize the withholding.
(4) **Trace Censorship/Data Witholding** The elected leader either fails to distribute a consensus trace for their Round-4 signed object to all other validators prior to a protocol-defined grace period, or distributes a malformed trace. A threshold of $f+1$ TraceFault attestations after the deadline suffices to penalize the fault.

Each fault admits a compact, signature-based certificate that can be assembled by the $f+1$ honest validators guaranteed by our model, while remaining unforgeable by any coalition of at most f Byzantine actors. The precise statement of each fault, the minimal proof objects, and possible protocol penalties are deferred to Appendix B.2. The formal censorship-resistance and no-free-optionality arguments for the consensus protocol are similarly deferred to Appendix B.3.

3 Order Execution

In the following subsections, we describe the sequential dual-auctions run by Angstrom's off-chain validators and detail the pool-unlock mechanism that, upon transaction execution, frees Angstrom liquidity for composable on-chain interactions. Each subsection first outlines the technical details of the execution mode it describes, which is then followed by an analysis of the economics for liquidity providers upon the completion of the execution mode. These analyses show that, in Angstrom, liquidity providers have positive P&L when the top-of-block auction resolves to common value and there is any residual order flow in either the batch auction or post-unlock state.

3.1 Top-of-Block Auction

At the start of every Ethereum block Angstrom runs a first-price auction for the *exclusive right* to perform one zero-fee swap in a pool. The winner specifies a tuple (B, P_f), pays a lump-sum bid B of asset Y to the pool contract, and executes a swap that moves the AMM price from its pre-auction value P_0 to a post-swap target $P_f > P_0$.[6] We represent concentrated liquidity by the set of tick-liquidity tuples $\{(i, L_i)\}$ and take p_i to represent the price at tick i. From [2] we have the amount of asset X that the AMM supplies between p_{start} and p_{end} inside a single tick $p \in [p_i, p_{i+1})$ with $P_0 \le p_i \le p_{\text{start}} \le p_{\text{end}} \le p_{i+1}$ as

$$\Delta X_i = L_i \left(\frac{1}{\sqrt{p_{\text{start}}}} - \frac{1}{\sqrt{p_{\text{end}}}} \right). \tag{3.1}$$

We want to solve for the budget in Y necessary to raise all utilized liquidity in a given swap up to the same effective execution price. Differentiating eq. (3.1) with respect to p_{end} and generalizing over all $p > P_0$, we get the marginal X-per-p density

$$\ell_X(p) = \sum_i \frac{L_i}{2p^{3/2}} \, \mathbf{1}_{p \in [p_i, p_{i+1})}, \qquad p > P_0. \tag{3.2}$$

Equation (3.2) allows us to define the *interval compensation cost* with $P_0 \le p_{\text{start}} < p_{\text{end}} \le p'$ as

$$\begin{aligned} \delta(p_{\text{start}}, p_{\text{end}}, p') &= \int_{p_{\text{start}}}^{p_{\text{end}}} \ell_X(q)\,(p'-q)\,dq \\ &= \sum_i \int_{a_i}^{b_i} \frac{L_i}{2q^{3/2}}\,(p'-q)\,dq \\ &= \sum_i L_i \left[p' \left(\frac{1}{\sqrt{a_i}} - \frac{1}{\sqrt{b_i}} \right) + \sqrt{a_i} - \sqrt{b_i} \right] \end{aligned} \tag{3.3}$$

where $a_i = \max\{p_i, p_{\text{start}}\}$ and $b_i = \min\{p_{i+1}, p_{\text{end}}\}$ and the sums are taken over all ticks$\{i \mid p_{\text{start}} < p_{i+1}$ and $p_i < p_{\text{end}}\}$, i.e. all ticks contributing liquidity in the swap range. Note that eq. (3.3) multiplies the liquidity density (in units of X-per-price) by the price compensator (in units of Y/X) and integrates across price, resulting in an aggregate compensation amount in units of Y.

$\delta(p_{\text{start}}, p_{\text{end}}, p')$ is strictly increasing and precisely measures the required budget to raise all AMM liquidity in $[p_{\text{start}}, p_{\text{end}}]$ up to the same effective execution price, p'.

Definition 3.1 (Cumulative Compensation to p). *For $p > P_0$, define the cumulative compensation to p as $C(p) := \delta(P_0, p, p)$. $C(p)$ is exactly the total payment required to raise all liquidity initially priced in the interval $[P_0, p]$ to the common execution price p, which is notably also the final pool price.*

Bid distribution. Let the winning bidder submit (B, P_f) with $P_f > P_0$. Define the *equalization frontier*

$$p_\star \;:=\; \min\big\{P_f,\ \inf\{p \ge P_0 : C(p) \ge B\}\big\},$$

which is the unique price where the spent budget first catches up with the cost curve. If $p_\star = P_f$, the bid suffices to reach all the way to P_f and we have a bid surplus

$$S \;:=\; B - C(P_f) \;\ge\; 0$$

that we want to distribute to the ticks that provided active liquidity in the swap. In this case, a new effective price $p_\star > P_f$ is determined that exhausts the whole bid without touching new ticks beyond P_f, while also ensuring that all active ticks trade at the same execution price $p_\star$. This price is given by

$$p_\star = P_f + \frac{S}{L_{\text{swap}}}, \qquad L_{\text{swap}} := \int_{P_0}^{P_f} \ell_X(q)\,dq,$$

i.e. we shift every utilized tick up by the same *price* increment $\frac{S}{L_{\text{swap}}}$, the surplus normalized by the integral of the liquidity density over the swap range. For each tick i with $p_i < p_\star$ we bid $b_i =$

[6] The $P_f < P_0$ case is symmetric.

$\delta(\max\{p_i, P_0\},\ \min\{p_{i+1}, p_\star\}, p_\star)$, distributing exactly the amount that would have been earned had the tick's entire liquidity executed at the common price $p_\star$.

Algorithm 1 Bid distribution

Require: Sorted ticks (p_i, L_i), start P_0, target P_f, bid B

$C_f \leftarrow C(P_f)$ ▷ cost to lift all crossed liquidity to P_f
if $B \le C_f$ **then** ▷ exact or under funded bid
  $p_\star \leftarrow \text{RootOf}(C(p) = B)$ ▷ $P_0 \le p_\star \le P_f$
  for all ticks i **do**
    $a \leftarrow \max\{p_i, P_0\}, \quad b \leftarrow \min\{p_{i+1}, p_\star\}$
    $b_i \leftarrow (p_i < p_\star)\ ?\ \delta(a, b, p_\star) : 0$ ▷ δ from (3.3)
  end for
else ▷ over funded bid with surplus
  $S \leftarrow B - C_f$ ▷ residual budget
  $L_{\text{swap}} \leftarrow 0$
  for all ticks i with $p_i < P_f$ **do**
    $a \leftarrow \max\{p_i, P_0\}, \quad b \leftarrow \min\{p_{i+1}, P_f\}$
    $L_{\text{swap}} \leftarrow L_{\text{swap}} + L_i \left(\frac{1}{\sqrt{a}} - \frac{1}{\sqrt{b}}\right)$
  end for
  $p_\star \leftarrow P_f + S/L_{\text{swap}}$ ▷ common post-comp. price
  for all ticks i **do**
    if $p_i < P_f$ **then**
      $a \leftarrow \max\{p_i, P_0\}, \quad b \leftarrow \min\{p_{i+1}, P_f\}$
      $b_i \leftarrow \delta(a, b, p_\star)$
    else
      $b_i \leftarrow 0$
    end if
  end for
end if
return $\{b_i\}$

It immediately follows that:

Proposition 3.1 (Monotone weighting). *The per-unit-liquidity reimbursement b_i/L_i is non-increasing in the tick distance from P_0. Hence liquidity closest to P_0 is compensated the most (cf. subfigure 3.4).*

Figure 3 provides intuition for how liquidity is compensated per algorithm 1 for a top-of-block swap with bid B, start price P_0, and target price P_f over the example liquidity distribution detailed in subfigure 3.1.

3.1.1 Analysis In this section, we will present an informal discussion of the economics of Angstrom from the perspective of liquidity providers. Our analysis assumes a perfectly competitive market of arbitrageurs in the style of Milionis et al. [4]. Here, time t, arbitrageurs can trade on an external venue (e.g., Binance) at the reference price S_t, while the on-chain spot pool price P_t may deviate from this price. We assume that arbitrageurs can trade frictionlessly (e.g., no fees, price impact, etc.) on the external market as well on Angstrom (e.g., no gas), and that all price discovery happens on the external market.

To begin, consider the state of the pool after the top-of-block auction occurs. Note that the top-of-block auction is a first-price auction in a fully common-value setting. In this setting, we can establish the following:

Theorem 3.2 (Informal). *After the top-of-block auction, the pool spot price P_t is equal to the reference price S_t, and the auction results in zero P&L for the each tick in the pool.*

Proof. A profit-maximizing arbitrageur will trade until there is zero marginal profit, that is, until the marginal price on the pool is equal to the external price P_t. Since there are no fees paid to the pool in the top-of-block auction, the marginal price on the pool is given by the spot price S_t, hence, after the arbitrageur's trade, we will have $S_t = P_t$. In a competitive, zero-profit equilibrium among arbitrageurs, the full profit of this trade will be bid in the top-of-block auction. As that bid is distributed to ticks according to the procedure described in Algorithm 1, the average trade price received by each tick is exactly P_t. □

Note that Theorem 3.2 relies on three critical assumptions: (1) the auction is perfectly competitive; (2) arbitrage trading is frictionless; and (3) bidders are unable to exploit a post-consensus option. If any of these assumptions are violated, the pool price may fail to converge to the external reference. For highly liquid assets, assumption (1) typically holds. The primary friction in (2)—on-chain gas cost—is amortized over the full depth of liquidity in the swap range and is thus negligible relative to the common value arising from expected intra-block reference price deviations. Assumption (3) is addressed in detail in section 4.1, which also outlines protocol-level mitigations that eliminate the viability of aiming to exercise post-consensus options.

3.2 Uniform-Clearing Batch Auction

Angstrom conducts its uniform-clearing batch auction upon the resultant AMM state after the top-of-block swap to protect against sandwich attacks and to ensure slippage is minimized via market makers providing just-in-time limit-order liquidity up to a fee-bound away from the external reference price.

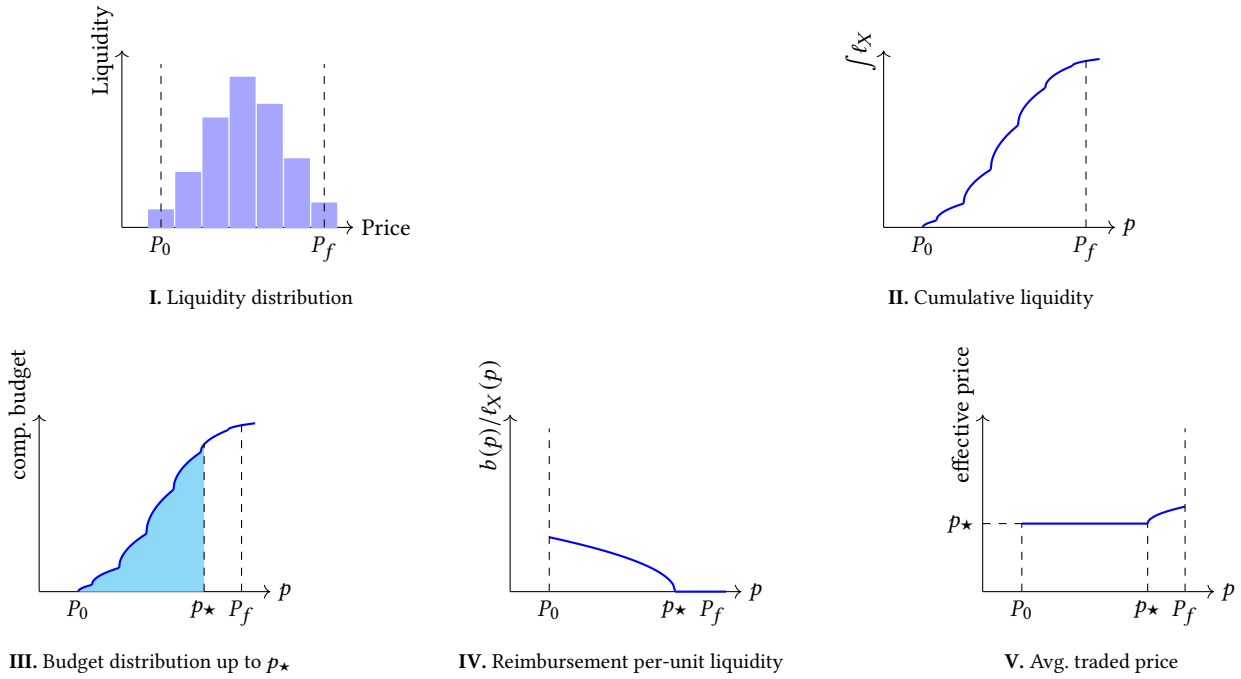
In their seminal work, Budish et al. [8] argue that high-frequency batch auctions in traditional limit order books lead to increased liquidity by reducing costs for market makers. Canidio and Fritsch [9] apply batch auctions to blockchains with a design that is able to clear all orders over a given asset pair in a block, alongside a constant product AMM, at a uniform clearing price.

Angstrom's batch auction builds upon this to determine the uniform clearing price across a diverse set of order types while also integrating the underlying *concentrated liquidity* AMM dynamics in the batch clearing. This section specifies the structure of order types, their contribution to the aggregate supply curve, the algorithm used to find the clearing price, and considers potential difficulties arising from incompatible or ambiguous order matching at that price.

Order Types and Liquidity Representation Every batch is reduced to a single net supply curve, $T_X(p')$. For any hypothetical clearing price p' — quoted in the canonical units of Y/X, i.e. *numeraire* Y (`token0`) per unit of *risky* asset X (`token1`) — the value $T_X(p')$ records the net amount of X that market participants are prepared to supply to ($T_X > 0$) or demand from ($T_X < 0$) the batch.

Two sources contribute to the net supply curve:

(1) the AMM's concentrated-liquidity ticks, whose reserves' ΔX is a deterministic function of P_0 and p'; and
(2) five off-chain order archetypes (exact-in ask, exact-out ask, exact-in bid, exact-out bid, and partially-fillable limits).



I. Liquidity distribution

II. Cumulative liquidity

III. Budget distribution up to $p_\star$

IV. Reimbursement per-unit liquidity

V. Avg. traded price

Figure 3. Five-panel intuition for bid redistribution. (I) Example liquidity distribution between P_0 and P_f. (II) Corresponding cumulative liquidity curve. (III) Bid B lifts ticks up to common price $p_\star$. (IV) With $b(p)$ as the marginal slice of the total bid amount, B, paid to price p, the reimbursement per unit liquidity is highest near P_0 and declines to zero at $p_\star$. (V) Effective execution price profile for a sample liquidity position from P_0 to P_f. Compensated liquidity in the range is executed at $p_\star$, then the effective execution price rises as $\sqrt{p\,P_f}$ for the prices greater than $p_\star$.

Each order is described by its *side* (+1 for **asks** that *supply* X, -1 for **bids** that *demand* X), its *execution mode* (exact-in, exact-out, or partial), a minimum fill amount k, an optional elastic tail e (for partial orders), and a *fee-adjusted limit price* p^{lim} in Y/X. A breakdown of how each order archetype, including the underlying pool liquidity, corresponds to the net supply curve is showcased in table 1.

Because the flow variable is now the risky asset X,

- an **ask** becomes active when it can sell at *or above* its limit $(p' \geq p^{\text{lim}})$;
- a **bid** becomes active when it can buy at *or below* its limit $(p' \leq p^{\text{lim}})$.

Under these sign–activation conventions the aggregate $T_X(p')$ is *monotone non-decreasing* in price, so it crosses the horizontal axis at most once, exactly as illustrated in subfigure 4.3. This preserves the uniqueness and guaranteed convergence of the bisection routine that locates the uniform clearing price.

The global supply curve is the sum over all active rows; bids enter with negative sign. For partial orders, the *kernel*, k, is treated as indivisible in the rigid amount, while any remainder up to e is elastically fillable and resolved pro-rata on all cleared partial orders once a uniform price p_e is fixed[7]. All fees are deducted in Y at the pool rate γ, hence limit prices are understood to be fee-adjusted as described in the following paragraph.

[7]The algorithm to deterministically find the clearing allocation at p_e is detailed in appendix A.2

Fees All trading fees are charged in the numeraire asset Y at the pool-specific rate $\gamma \in (0, 1)$ and are deducted *before* the uniform-clearing algorithm is run. A protocol split parameter $\phi_p \in [0, 0.5)$ and a validator split parameter $\phi_v \in [0, 0.5)$ determines how the batch's aggregate fee is shared between the protocol, the validators, and the underlying LPs.

- **Exact–in ask** (sell X for Y): the trader receives $p\,x_{\text{in}}$ units of Y at price p; the contract withholds γ of that amount, so only $(1-\gamma)\,p\,x_{\text{in}}$ enters the net-supply curve.
- **Exact–out bid** (buy X with Y): the trader transfers y_{in} units of Y; the pool retains $\gamma\,y_{\text{in}}$ as the fee, and the remaining $(1-\gamma)\,y_{\text{in}}$ is available for matching.

To keep every order's economic guarantee intact, the engine rewrites each user-supplied limit price p^{lim} as a *fee-adjusted limit*

$$\tilde{p} \;=\; \begin{cases} \dfrac{p^{\text{lim}}}{1-\gamma}, & \text{exact–in ask,} \\[2ex] p^{\text{lim}}\,(1-\gamma), & \text{exact–out bid,} \end{cases}$$

and the bisection routine operates on the multiset $\{\tilde{p}\}$. This guarantees that the uniform clearing price p_e remains acceptable once post-settlement fee transfers are applied.

Example. Take $\gamma = 30$ bps $= 0.003$ and an exact-in ask in which a trader sells $x_{\text{in}} = 50\,X$ at a minimum price $p^{\text{lim}} = 25\ Y/X$. The limit is internally lifted to

$$\tilde{p} \;=\; \frac{p^{\text{lim}}}{1-\gamma} \;=\; \frac{25}{0.997} \;\approx\; 25.075\ Y/X.$$

Archetype	**Intent**	**Activation domain for p'**	**Net X contribution $T_X(p')$**
AMM tick (i, L_i)	liquidity on $[p_i, p_{i+1})$	$\exists\, p \in [p_i, p_{i+1})$ s.t. $\begin{cases} p' < p < P_0, & p' < P_0 \\ P_0 < p < p', & p' > P_0 \end{cases}$	$L_i \begin{cases} \left(\frac{1}{\sqrt{\min\{p_{i+1}, P_0\}}} - \frac{1}{\sqrt{\max\{p_i, p'\}}}\right), & p' < P_0, \\ \left(\frac{1}{\sqrt{\max\{p_i, P_0\}}} - \frac{1}{\sqrt{\min\{p_{i+1}, p'\}}}\right), & p' > P_0 \end{cases}$
Exact-in ask	sell fixed x_{in}	$p' \geq \tilde{p}$	$+ x_{\text{in}}$
Exact-out ask	receive fixed y_{out}	$p' \geq \tilde{p}$	$+ \frac{y_{\text{out}}}{(1-\gamma)\, p'}$
Exact-in bid	spend fixed y_{in}	$p' \leq \tilde{p}$	$- \frac{(1-\gamma)\, y_{\text{in}}}{p'}$
Exact-out bid	acquire fixed x_{out}	$p' \leq \tilde{p}$	$- x_{\text{out}}$
Partial (min k, max $k+e$)	flexible fill, $\text{SIDE} \in \{+1, -1\}$	$\text{SIDE}\,(p' - \tilde{p}) \geq 0$	$\text{SIDE}\big(k + \eta(p')\big),\ \eta \in [0, e]$

Table 1. Order archetypes and their contribution to the net-supply curve $T_X(p')$. For price *above* P_0, the AMM *supplies* X (the piecewise term is positive); for prices *below* P_0, it *demands* X (the piecewise term is negative).

The order enters the batch only if $p_e \geq 25.075$. If the auction clears at $p_e = 25.10$, the pool owes the trader $p_e x_{\text{in}} = 1{,}255\, Y$; the fee is $0.003 \times 1{,}255 \approx 3.765\, Y$, so the trader receives $(1-\gamma) p_e x_{\text{in}} \approx 1{,}251.24\, Y$, just above the required $1{,}250\, Y$. The $3.765\, Y$ fee is split, with $(1 - \phi_p - \phi_v) \cdot 3.765\, Y$ distributed pro-rata to the LPs providing active liquidity in the swap range, $\phi_p \cdot 3.765\, Y$ retained by the protocol, and $\phi_v \cdot 3.765\, Y$ collected for distribution to validators.

3.2.1 Uniform Clearing Price Algorithm The uniform clearing price p_e is the (unique) price at which the aggregate net–supply curve for the risky asset, $T_X(p)$, crosses zero, i.e. total X offered equals total X demanded. Because $T_X(\cdot)$ is *monotone non-decreasing in price* (negative at low p where bids dominate, positive at high p where asks dominate), a simple bisection routine suffices:

Algorithm 2 Uniform Clearing Price via Net-Supply Bisection

Require: Net–supply curve $T_X(p)$ ($T_X > 0$: excess X supply, $T_X < 0$: excess demand)

Initialize $p_{\min} \leftarrow p_s$, $p_{\max} \leftarrow p_{\text{upper}}$ ▷ feasible price bounds
Choose tolerance $\epsilon \in (0, \text{lot size})$
while $p_{\max} - p_{\min} > \epsilon$ **do**
  $p_{\text{mid}} \leftarrow \dfrac{p_{\min} + p_{\max}}{2}$
  $e \leftarrow T_X(p_{\text{mid}})$ ▷ net excess at mid-price
  if $e > 0$ **then** ▷ excess supply — root lies lower
    $p_{\max} \leftarrow p_{\text{mid}}$
  else if $e < 0$ **then** ▷ excess demand — root lies higher
    $p_{\min} \leftarrow p_{\text{mid}}$
  else
    break ▷ exact balance found
  end if
end while
return $(p_{\min} + p_{\max})/2$

Convergence and complexity. Since T_X is monotone and continuous in the feasible interval $\left[p_s, p_{\text{upper}}\right]$ and satisfies $T_X(p_s) \leq 0 \leq T_X(p_{\text{upper}})$, the sign changes at most once, guaranteeing a single root. Each loop halves the bracket length, so after $\lceil \log_2\big((p_{\text{upper}} - p_s)/\epsilon\big) \rceil$ iterations the width falls below ϵ.

Evaluating $T_X(p)$ requires one pass over the n active AMM ticks and $\mathcal{N}$ off-chain orders, giving $O(n + \mathcal{N})$ time per iteration. Hence the total work is

$$O\Big((n + \mathcal{N})\ \log\big((p_{\text{upper}} - p_s)/\epsilon\big)\Big),$$

with constant auxiliary memory beyond the input data.

Feasible and Failed Auctions The bisection routine always yields a candidate uniform price p_e at which fee-adjusted supply equals demand; nonetheless, the subsequent allocation can still fail when the book contains indivisible ("rigid") orders whose quantities do not line up.

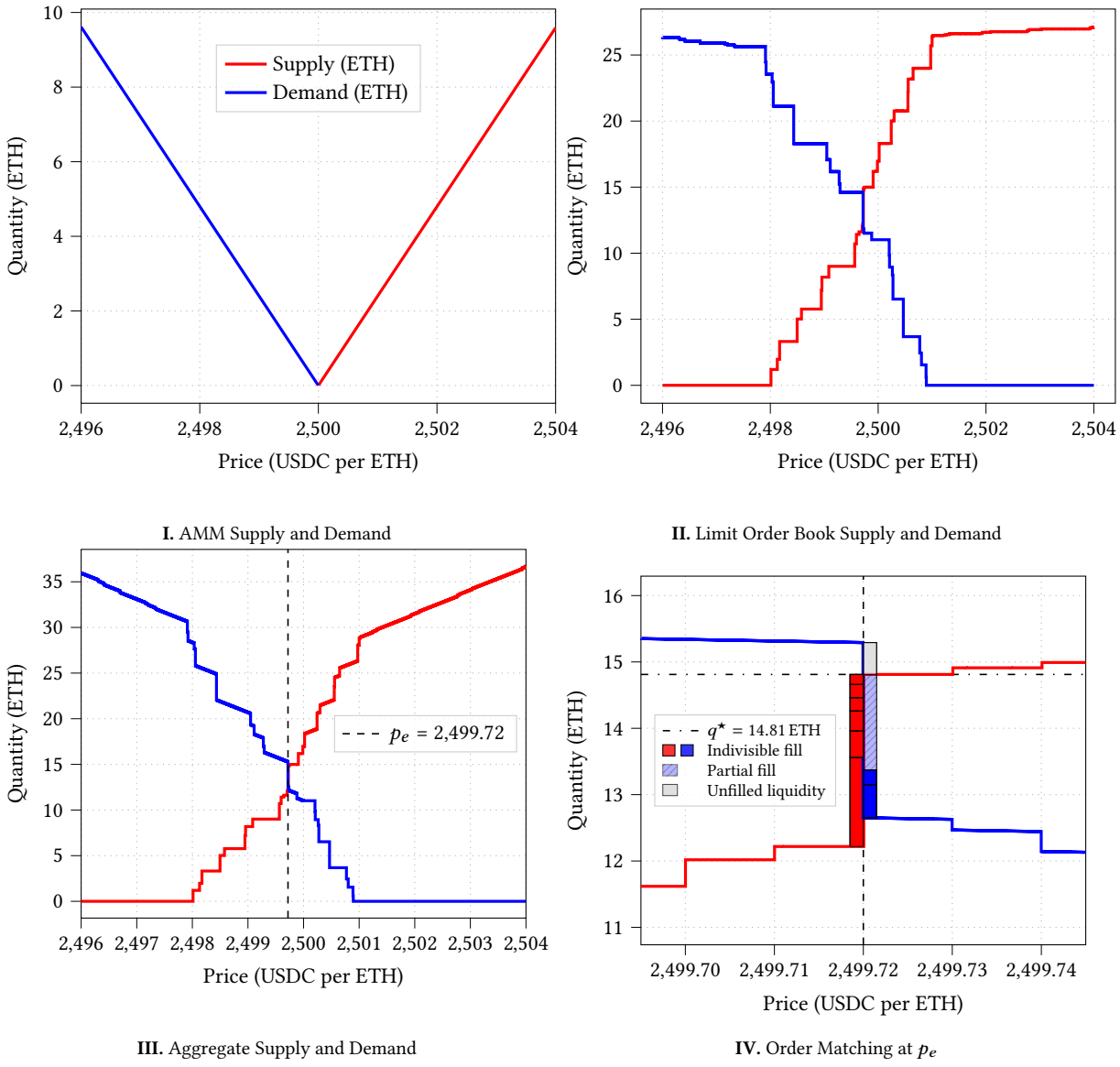
Infeasible example. Assume no pool liquidity and a limit book containing

- an *exact-out ask* that wishes to *receive* $1{,}200\, Y$ with $\tilde{p} = 1.00$ (hence it must deliver $1{,}200\, X$) and
- an *exact-in bid* willing to buy $1{,}000\, Y$ worth of X with $\tilde{p} = 1.00$.

At $p = 1.00$ the intersection point is $1{,}000\, Y$, so $p_e = 1.00$; yet the ask's rigid requirement of $1{,}200\, X$ overshoots available demand, leaving no feasible matching, thus the auction fails and neither order is cleared.

Feasible but ambiguous example. Now consider four asks $\{4, 4, 2, 10\}$ and four bids $\{5, 3, 2, 1\}$, all quoted at the same limit price of $1\ Y/X$. Two distinct allocations clear the same maximal volume 10 units of both assets: either (i) match the single 10-lot ask with bids $(5, 3, 2)$, or (ii) match asks $(4, 4, 2)$ with the same three bids. The execution algorithm must choose one of these equally valid matchings in a deterministic fashion.

Problem relaxation. Since every partially-fillable order declares a rigid kernel and an elastic tail, the search for a maximal allocation becomes a variant of the *balanced subset-sum* problem. The kernels on each side form two multisets; equality between the two subset sums may be offset by a slack no larger than the aggregate tails unlocked by those very same subsets. Appendix A.2 formalizes this relaxation and presents an efficient algorithm that always selects a volume-maximizing allocation. Notably, the algorithm ensures that orders with the largest elastic tail have queue priority at the post-bisection uniform clearing price. This tail-size incentivization has the positive second-order effect of minimizing failed auctions that result from quantity misalignment of indivisible orders.



I. AMM Supply and Demand

II. Limit Order Book Supply and Demand

III. Aggregate Supply and Demand

IV. Order Matching at p_e

Figure 4. We visualize how the AMM curve, off-chain order book, and resulting aggregate net-supply determine the uniform clearing price and matched quantity. Subfigure 4.4 shows how execution resolves at p_e when there is misalignment in supplied/demanded quantities, based on the specifications provided in appendix A.2.

3.2.2 Analysis In this section, we continue the analysis under the assumptions of Section 3.1.1. After the uniform-clearing batch auction, we can establish the following:

Theorem 3.3 (Informal). *The uniform-clearing batch auction results in a clearing price p_e that satisfies*

$$S_t\,(1-\gamma) \le p_e \le S_t/(1-\gamma).$$

Proof. Under our assumptions, an informed arbitrageur will be willing to buy at an all-inclusive price less then or equal to S_t or sell at an all-inclusive price greater than or equal to S_t. Given the fees charged by the auction, this translates to a fee-adjusted bid price of $S_t\,(1-\gamma)$ and a fee-adjusted ask price of $S_t/(1-\gamma)$. Then, the pool will clear at the price p_e that satisfies $S_t\,(1-\gamma) \le p_e \le S_t/(1-\gamma)$. □

3.3 Pool Unlock

A substantial share of on-chain swap volume involves multi-step, composable workflows, such as lending-protocol liquidations or multi-hop swap routes, that must pull in intermediate liquidity and use it atomically. Embedding trades with external-state dependencies directly into an Angstrom transaction would couple their revert semantics to the entire batch, jeopardizing aggregate settlement.

Furthermore, many of these actors require *self-execution*: their validation pipeline expects a fully formed user-signed transaction (for example, specialized routers with custom validation contracts), rather than an EIP-712 intent executed within the Angstrom batch. This model conflicts with Angstrom's intent-batch architecture, in which users' trades are submitted as intents and executed together in a single on-chain transaction.

To support both composability and self-execution, Angstrom flips a single `poolUnlocked` flag at the end of its canonical transaction. This unlocks the pool, enabling its on-chain liquidity to be available for composable interactions in the remainder of the block, but only after both auctions have been settled. In this unlocked state, the pool behaves like any other Uniswap v4 pool and can be accessed directly.

Because the flag is set inside the canonical Angstrom transaction, any follow-on swap must be placed after it. A higher-priority placement would simply revert against the still-locked pool, so a rational block builder will naturally sequence external calls *after* the Angstrom batch.

Once unlocked, the pool remains tradable for the rest of the block at an elevated fee, $\gamma_{\text{post}} > \gamma$.

3.3.1 Analysis In this section, we continue the analysis under the assumptions of Section 3.1.1. After the unlock, we can establish the following:

Theorem 3.4 (Informal). *After the unlock, swaps executed against the pool are, in aggregate, strictly profitable to the liquidity providers when valued at the contemporaneous reference price.*

Proof. Suppose there is more than one swap in the post-unlock period. We can construct an aggregate swap by netting input tokens (after fees have been collected) across all of the swaps. Such an aggregate swap will result in the same number of total output tokens from the perspective of each LP, since, after fees, trading on the pool is path independent. However, fewer fees will be paid by the aggregate trade if there is any netting. Hence, without loss of generality, we can restrict to the case of a single swap in the post-unlock period.

After the pool unlock, the fee-inclusive bid and ask are given by

$$p_{\text{bid}} = p_e\,(1-\gamma_{\text{post}}), \qquad p_{\text{ask}} = \frac{p_e}{1-\gamma_{\text{post}}}.$$

Recall that S_t denotes the contemporaneous reference price. From Theorem 3.3, we have that

$$S_t\,(1-\gamma) \le p_e \le S_t/(1-\gamma).$$

Then, since $\gamma_{\text{post}} > \gamma$, we have

$$p_{\text{bid}}^{\min} \triangleq S_t\,(1-\gamma)(1-\gamma_{\text{post}}) < p_{\text{bid}} < p_{\text{bid}}^{\max} \triangleq S_t\,\frac{1-\gamma_{\text{post}}}{1-\gamma},$$

$$p_{\text{ask}}^{\min} \triangleq S_t\,\frac{1-\gamma}{1-\gamma_{\text{post}}} < p_{\text{ask}} < p_{\text{ask}}^{\max} \triangleq \frac{S_t}{(1-\gamma_{\text{post}})(1-\gamma)}.$$

Then, it is clear that

$$p_{\text{bid}} \;<\; S_t \;<\; p_{\text{ask}},$$

cf. Figure 5. Since p_{bid} is the highest (fee-inclusive) price that any LP will buy at, and p_{ask} is the lowest (fee-inclusive) price that any LP will sell at, any aggregate trade must be profitable for all LPs. □

4 Discussion

4.1 Post-Consensus Last Mover and Optionality

We consider a single asset pair (x, y) with risky asset x and numeraire y. The external reference price process is a drift–free geometric Brownian motion

$$\frac{dS_t}{S_t} = \sigma\, dW_t, \qquad S_0 > 0,\ \sigma > 0, \tag{4.1}$$

where W_t is a standard Brownian motion. The on–chain pool is a full-range constant-product AMM with reserves (x_0, y_0), liquidity invariant $L := \sqrt{x_0 y_0}$, and initial pool price[8] $P_0 := y_0/x_0$.

Swap profit for a deterministic price move. If the pool price is displaced from P_0 to P while the oracle reads S, from eq. (3.3) we have the trader's cash profit (in units of y) as

$$\Pi_{\text{swap}}(P, S) = L\left[S\left(\tfrac{1}{\sqrt{P_0}} - \tfrac{1}{\sqrt{P}}\right) - \left(\sqrt{P} - \sqrt{P_0}\right)\right]. \tag{4.2}$$

4.1.1 Timing game.

(i) At time T the top-of-block auction allocates the exclusive right to be the first swapper through the given pool in the current slot. To win, an attacker must bid the intrinsic value of bringing the pool to the reference price[9], i.e. $B^* = \Pi_{\text{swap}}(S_T, S_T)$.
(ii) She chooses a *final pool price* P_f, executes the swap, and pays no pool fee.
(iii) During $(T, T+\tau]$ she may *censor or revert* the entire bundle at a flat on–chain fee $c > 0$.

Random cash flow at the censorship deadline. Conditioned on S_T, the incremental P&L against the pool is

$$\Delta\Pi_{T+\tau} = \Pi_{\text{swap}}(P_f, S_{T+\tau}) - B^* = \alpha(P_f) S_{T+\tau} + \beta(P_f)$$

with

$$\alpha(P_f) = L\left(\tfrac{1}{\sqrt{P_0}} - \tfrac{1}{\sqrt{P_f}}\right), \qquad \beta(P_f) = L\left(2\sqrt{S_T} - \sqrt{P_f} - \tfrac{S_T}{\sqrt{P_0}}\right). \tag{4.3}$$

Thus the attacker's *net* payoff is

$$\Pi(P_f, c) = \mathbb{E}\left[\max\{\alpha S_{T+\tau} + \beta,\ -c\} \;\middle|\; S_T\right]. \tag{4.4}$$

Option equivalence. Henceforth, we consider the case where the attacker chooses $P_f > P_0$, equivalent to them going long a call[10]. Write $K(P_f, c) := -\dfrac{\beta(P_f) + c}{|\alpha(P_f)|}$ and let $C(S, K)$ denote the Black–Scholes price of the call of maturity τ and strike K. Then

$$\Pi(P_f, c) = \alpha(P_f)\, C\big(S_T, K(P_f, c)\big) - c. \tag{4.5}$$

Because $K(\cdot, c)$ is affine in $P_f^{-1/2}$ and $C(S, \cdot)$ is convex in K, the map $P_f \mapsto \Pi(P_f, c)$ is strictly concave and is maximized by a unique P_f.

[8] A concentrated-liquidity generalization is algebraically identical but obscures the exposition.
[9] A competing bidder can bid this value and instantly hedge on the reference market at time T, ensuring that B^* is the floor bid of the auction
[10] The put case can be formulated symmetrically.

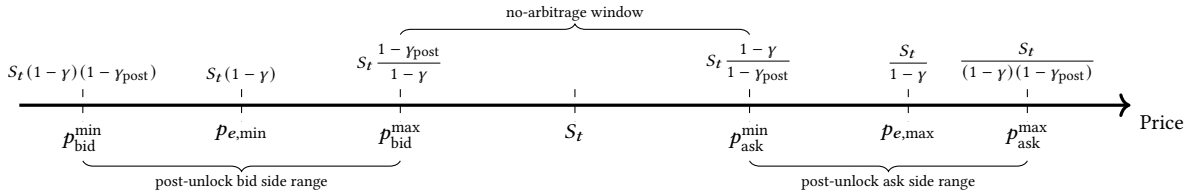


Figure 5. Post-unlock price landscape. Because the post-unlock fee satisfies $\gamma_{\text{post}} > \gamma$, the fee-inclusive bid–ask widens so that the best attainable bid $p^{\max}_{\text{bid}}$ remains below the reference price S_t and the best attainable ask $p^{\min}_{\text{ask}}$ remains above it; the upper brace highlights the resulting no-arbitrage window. Any taker swap is therefore prevented from preferentially taking the AMM liquidity marked to the instantaneous reference price.

First-order Condition Let $C_K := -\partial_K C(S,K)$. For $P_f > P_0$ we have $\alpha(P_f) > 0$, so differentiating (4.5) yields

$$\frac{\partial \Pi}{\partial P_f} = \alpha'(P_f)\, C\big(S_T, K(P_f,c)\big) - \alpha(P_f)\, C_K\big(S_T, K(P_f,c)\big)\, K'(P_f,c). \tag{4.6}$$

which simplifies to the implicit first-order condition

$$\alpha'(P_f^\star)\, C\big(S_T, K(P_f^\star,c)\big) = \alpha(P_f^\star)\, C_K\big(S_T, K(P_f^\star,c)\big)\, K'(P_f^\star,c) \tag{4.7}$$

which has a unique solution $P_f^\star = P_f^\star(S_T,\tau,\sigma,c)$ because the left–hand side is strictly decreasing and the right–hand side is strictly increasing in P_f.

4.1.2 Option Deterrence and Sybil Prevention The subfigures in Figure 6 show that, even for large amounts of underlying pool liquidity, only a small rollback cost is needed to deter bidders from exploiting the option; a desirable second order effect is that the rollback probability also trends to zero as this cost is borne. This cost can be modulated by the protocol to ensure specific thresholds for rollback percentage and option value are never exceeded.

Implementation-wise, since double spend is verifiable in the subsequent block, the contract only needs to hold the option-deterrent cost for each valid bidder in the core contract[11] for minimum-viable Angstrom-specific stake. Angstrom validators ensure that all incoming top-of-block orders originate from addresses with a stake amount exceeding the threshold. Should a bidder maliciously double spend the balance their order depended on to invalidate a bundle, the escrow amount is seized in the subsequent block and future bids are invalidated until they top up their internal stake balance. Bidders see immense capital efficiency gains from only escrowing the option-deterrent cost instead of needing to add their own internal balances to cover any valid top-of-block order.

The attentive reader will notice that a motivated attacker can avoid detection of their double spend attack by submitting a tiny order from a separate address that gets filled alongside the top-of-block trade. Should the post-consensus reference price $S_{T+\tau}$ result in an unprofitable top-of-block swap, the attacker can front-run the batch transaction on-chain to spend the token balance that their batch order depended on to revert the batch. Since this is not attributable to the top-of-block bidder's address, the attacker would be able to escape the double spend verification conducted by Angstrom nodes upon the block's finalization, preventing their Angstrom stake from being seized and resulting in an implied rollback cost that is far lower than the stake-modulated rollback cost, yielding a profitable option and resulting in the reversion of a significant percentage of Angstrom transactions. To protect against this, the Angstrom validator network sends a second transaction of only the top-of-block results for each pool to revert-protected builder endpoints as specified in the Round-5 description of section 2.2. This prevents the Sybil attack in its entirety. If a Sybil address attempts to grief the bundle the canonical Angstrom transaction, the top-of-block transaction will still execute. Any dobule spend that could block only the top-of-block transaction must come from a staked bidder equivocating, which will trigger slashing of their Angstrom stake.

[11] The stake can be held as internal balances in a validator-controlled address, where each validator stores an in-memory map of bidder addresses and their constituent stake quantities

5 Smart Contract Design

This section describes Angstrom's on-chain architecture, implemented entirely within a single Uniswap V4 hook contract. We proceed by detailing the pool-lock mechanism and bundle-execution flow in section 5.1. We then explain the transient accounting primitives in section 5.2, followed by the per-tick fee allocation and donation mechanics in section 5.3, and conclude with gas benchmarks demonstrating efficiency in section 5.4.

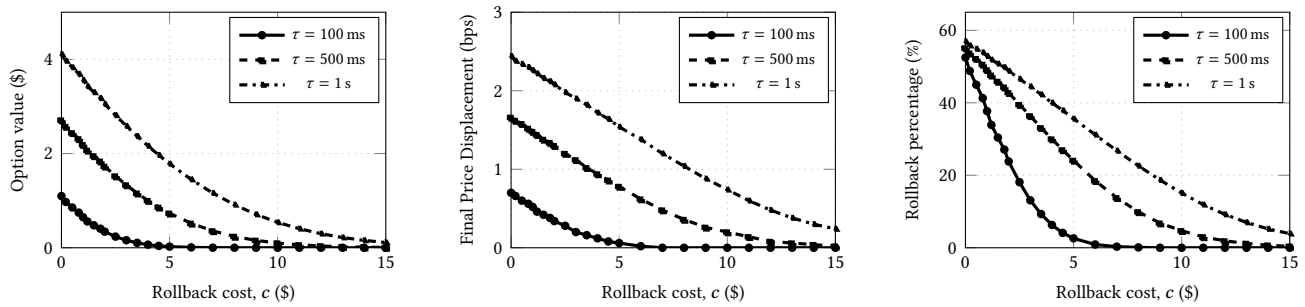
5.1 Pool Lock and Bundle Execution

Angstrom pools are deployed via the Uniswap v4 PoolManager, with each pool registering the settlement hook upon creation.

At the start of each block, the hook's `beforeSwap` callback locks the pool until a staked validator invokes the `execute(bytes bundle)`. This lock guarantees that no on-chain swap can bypass the auctions until both the top-of-block and batch auctions have been settled, after which the pool unlocks and becomes openly accessible as a standard Uniswap v4 pool (cf. section 3.3).

Bundle Execution Angstrom nodes run the top-of-block and batch auctions off-chain, derive the final uniform clearing price, and compute the execution payload: each pool's net AMM swap and multi-tick donations to distribute arbitrage proceeds and batch trading fees.

This representation is possible because successive AMM swaps can be compressed down to their net result; instead of executing two swaps sequentially, we can deterministically compute the aggregate swap off-chain and only execute one swap per pool instead of one swap per auction or one swap per order; the result is dramatic gas savings through amortization (cf. section 5.4).



I. Option value vs. rollback cost **II.** Optimal P_f deviation from S_T vs. rollback cost **III.** Rollback probability vs. rollback cost

Figure 6. Option economics under manipulation. (a) Option value, (b) attacker's optimal pool displacement, $P_f - S_T$, in basis points of S_T, and (c) probability of Angstrom transaction rollback all converge to 0 with rollback fee. Common parameters: $P_0 = 2500$, $S_T = 2501$, $L = 3{\times}10^6$, annualized volatility $\sigma = 150\%$.

These precomputed instructions are serialized using the Packed ABI Decodable Encoding (PADE) scheme (see appendix A) to minimize calldata size and decoding cost. Leveraging the precomputed execution payload, the on-chain execution can be confined to invariant verification—ensuring that for every token, total inflows match outflows—and state updates: applying the aggregated AMM swap, executing ERC-20 transfers for order settlement and donation crediting (cf. table 2).

Order Hooks Each batch order may include an optional composable hook, enabling arbitrary on-chain code to execute atomically alongside the swap. During execution, Angstrom first loans the swap output, crediting the recipient via either internal balances or a direct ERC-20 transfer—so that the hook contract immediately controls those tokens. It then invokes the hook, supplying the order sender's address and data. The hook must return the predefined 4-byte magic constant `EXPECTED_HOOK_RETURN_MAGIC`, or the entire transaction reverts. Finally, Angstrom debits the swap input from the sender—either by decrementing internal balances or calling `transferFrom`. This ordering implements a flash-swap pattern for the hook, supporting fully composable callbacks within the atomic settlement. To mitigate abuse and reduce bundle reverts, only a vetted subset of hooks is permitted by off-chain validators. Any whitelisted composable interactions that can execute atomically without risking aggregate reversion (i.e. token wrapping) can be added as a future protocol upgrade to the off-chain validation network without needing to upgrade the contracts themselves.

Internal Balances Users can leverage internal balances to save gas by depositing tokens directly into the hook contract. Internally, the contract tracks these with a simple mapping:

```
mapping(address asset ⇒ mapping(address owner ⇒ uint256 balance)) internal _balances;
```

When an order is settled with `use_internal = true`, the contract updates their internal balance instead of doing two ERC-20 transfers which saves substantial gas, especially when executing many orders.

5.2 Consolidated settlement accounting.

During execution the contract maintains an in-memory `DeltaTracker` map that records, for every ERC-20 token, the running difference between tokens taken in (positive) and sent out (negative). All bookkeeping occurs via transient storage (EIP-1153) which avoids costly SSTORE operations. At the end of the transaction the contract asserts the solvency invariant

$$\sum\nolimits_{\text{inflows}} \Delta_\alpha^{\text{in}} - \sum\nolimits_{\text{outflows}} \Delta_\alpha^{\text{out}} = 0$$

for every token α. When the invariant holds, the pool's state transition is identical to one where only the aggregate swap occurred, and all user balances (external or internal) have been credited.

Criteria	Invariant Checked
User limit prices	Executed price ≥ limit price for sells and ≤ limit price for buys.
`DeltaTracker` solvency	$\Delta_\alpha = 0$ for every ERC-20 after all bundle actions.
Maximum extra fee bound	Gas fee charged per order ≤ user-signed `maxExtraFee`; system-wide fee ≤ ∑ `maxExtraFee`.
Valid signature	Valid EIP-712 or EIP-1271 sig & unused nonce / order hash
Standing-order time validity	Block time ≤ `deadline`
Authorized sender	`msg.sender` is in the registry of authorized Angstrom nodes.

Table 2. Key criteria enforced by the core contract during `execute`.

5.3 Angstrom Fee Distribution Implementation

Angstrom settles one aggregate swap per block: the node nets the top-of-block order against the entire batch and sends only the resulting delta to the pool. Crucially, the ticks that receive the top-of-block donation are *not* necessarily the same ticks whose liquidity is consumed by the netted swap. To accurately reward LPs that supplied liquidity during the top-of-block auction, the contract must credit arbitrary initialized ticks—even when the aggregate per-pool swap path never touched them.

Uniswap's fee–growth primitive. Uniswap v4 tracks fees per unit liquidity with two accumulators for each fee token α:

- a pool-wide `feeGrowthGlobal`$_\alpha$;
- a per-tick `feeGrowthOutside`$_\alpha$ that stores the cumulative growth below the tick (if the tick is less than or equal to the current tick) or above the tick (if the tick is greater than the current tick).

By computing the cumulative fee growth within each tick range, we can determine how much liquidity has appreciated over that interval. Moreover, by storing two `feeGrowthInside` values per position, one for each token, we can accurately attribute each position's proportional share of accrued pool fees at any point in the future.

Mirroring the pattern for Angstrom rewards. Angstrom re-implements this pattern for its reward token (always `token0`). It tracks two additional global variables:

- `globalGrowth` — pool-level accumulator shared by all positions,
- `rewardGrowthOutside`$[i]$ — a sparse array keyed by initialized tick boundaries,

and one per-position variable, `lastGrowthInside`. During the bundle execution the node supplies a list of *(tick, amount)* pairs. The contract walks the bitmap from the current tick toward the side of the donation incrementally, until it reaches the active tick, then bumps `globalGrowth` by the grand total.

Whenever an LP adds or removes liquidity, the `beforeAddLiquidity` / `beforeRemoveLiquidity` hooks call back into the Angstrom core. The hook recomputes

$$\texttt{growthInside} = \texttt{globalGrowth} - \texttt{rewardGrowthOutside}[i_\ell] - \texttt{rewardGrowthOutside}[i_u],$$

pays out $(\texttt{growthInside} - \texttt{lastGrowthInside}) \times \texttt{liquidity}$, and updates `lastGrowthInside` to its current value.

Preventing just-in-time (JIT) abuse. A naive design would allow an attacker to monitor the Angstrom auctions for instances where the batch auction nets out in the direction opposing the top-of-block swap. The validated transaction will then reward ticks that the top-of-block swap reached, however the aggregation of both auctions into one swap through the AMM will result in pool state transition that does not take active pool liquidity up to the implied final price of the top-of-block swap. The attacker can then inject liquidity *only* to the rewarded ticks between the clearing price of the aggregate auctions and the implied final price for the top-of-block auction before the Angstrom transaction executes, thereby receiving a share of the top-of-block rewards without providing active liquidity for the trade. To close this attack the bundle encodes the following checksum over the tick range starting with current tick i_c:[12]

$$\text{checksum} = \texttt{keccak256}\big((i_c, L_c), (i_{c+1}, L_{c+1}), \dots\big),$$

the exact sequence of *(tick,liquidity)* pairs the validators observe while walking the tick bitmap on the AMM state observed after the last block. During execution, the contract recomputes the hash with live pool data; any JIT liquidity would change at least one pair's liquidity field and causes the transaction to revert. Thus, only liquidity that existed in the previous block can earn top-of-block donations, preventing front-running JIT liquidity provision attacks.

[12] This assumes the top-of-block swap is executing such that $P_f > P_0$. The $P_f < P_0$ case is symmetric.

Order Count	EFI (w/ AMM)	EFI (No AMM)	ESLn (w/ AMM)	ESLn (No AMM)
1	148.9k	70.4k	159.0k	80.7k
2	84.2k	44.9k	95.7k	56.5k
3	62.6k	36.4k	74.6k	48.4k
4	51.8k	32.2k	64.0k	44.4k
5	45.3k	29.6k	57.7k	42.0k
10	32.4k	24.5k	45.0k	37.2k
20	25.9k	22.0k	38.7k	34.8k
50	22.0k	20.4k	34.9k	33.3k

Table 3. Amortized per-order gas cost for batch settlements of size N, comparing Exact Flash Orders (EFI) using internal balances and Exact Standing Orders (ESL) using liquid tokens both with and without AMM swaps.

Operation	Gas Cost
Uni v4 direct pool swap (test router, no checks)	123,144
Uni v4 router `ExactInputSingle`	134,001

Table 4. Uniswap v4 swap gas cost.

5.4 Gas Benchmarking

Table 3 reports the amortized gas cost per order for two settlement modes—Exact Flash Orders (EFI, settling via internal balances) and Exact Standing Orders (ESLn, settling via on-chain token transfers)—across various batch sizes in a single-pool settlement scenario. Results are shown both with and without AMM swaps.

Users only incur the AMM swap gas when the batch triggers a swap against the pool and no arbitrageur executes a top-of-block trade; in all other cases the arbitrageur absorbs the AMM swap cost. Additionally these measurements assume one pool per batch. Spreading orders across multiple pools further dilutes both swap and encoding overhead, yielding additional per-order savings.

For reference, a bare Uniswap v4 pool swap (no router checks) consumes about 123k gas, while the standard router call costs approximately 134k gas (cf. table 4).

References

[1] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. Technical report, Uniswap, 2020.

[2] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. Technical report, Uniswap, 2021.

[3] Hayden Adams, Moody Salem, Noah Zinsmeister, Sara Reynolds, Austin Adams, Will Pote, Mark Toda, Alice Henshaw, Emily Williams, and Dan Robinson. Uniswap v4 core. Technical report, Uniswap, 2023.

[4] Jason Milionis, Ciamac C Moallemi, Tim Roughgarden, and Anthony Lee Zhang. Automated market making and loss-versus-rebalancing. *arXiv preprint arXiv:2208.06046*, 2022.

[5] EigenLayer Team. Eigenlayer: The restaking collective. Technical report, EigenLayer, 2023.

[6] Mallesh Pai and Max Resnick. Structural advantages for integrated builders in mev-boost. In *International Conference on Financial Cryptography and Data Security*, pages 128–132. Springer, 2024.

[7] Ciamac C Moallemi, Mallesh M Pai, and Dan Robinson. Latency advantages in common-value auctions. *arXiv preprint arXiv:2504.02077*, 2025.

[8] Eric Budish, 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):1547–1621, 2015. doi: 10.1093/qje/qjv027. URL https://academic.oup.com/qje/article/130/4/1547/1916146.

[9] Andrea Canidio and Robin Fritsch. Arbitrageurs' profits, lvr, and sandwich attacks: batch trading as an amm design response, 2025. URL https://arxiv.org/abs/2307.02074.

[10] Jae Kwon and Ethan Buchman. Cosmos: A network of distributed ledgers. Technical report, Cosmos, 2016.

A System Architecture

The off-chain Angstrom node network conducts the sequential auctions, validates the finalized bundle, and handles Ethereum synchronization in a single event-driven system. The protocol uses a stake-weighted round robin algorithm (see, e.g., [10]) to select the lead Angstrom proposer for a given slot. Upon completion of the Angstrom consensus protocol, all validators can encode the settled orders determined by the consensus end state with a custom encoding scheme to efficiently interface with the Angstrom contract. They all submit the resultant encoded transactions for both the aggregate result (inclusive of both auctions per pool) and only the top-of-block results for block inclusion per the specification detailed in section 2.2.

PADE Encoding To interface with the Angstrom smart contract, the node encodes the post-consensus bundle to a byte serialization via a *Packed ABI Decodable Encoding* (PADE) scheme. This format compresses array lengths, struct fields, and certain enumerated types in a more compact way than standard ABI encoding while remaining unambiguous to parse. Each segment of the payload is associated with a simple integer bitmap for variant fields or optional fields. Lists are prefixed with a three-byte length indicator, limiting the maximum array size but providing stable decoding in the EVM. PADE reduces the risk of extremely large calldata footprints, which is particularly beneficial for bundles with multiple orders and pool updates.

A.1 Order Validation

A single address may broadcast a set of signed EIP−712 or ERC-1271 meta-transactions ("orders") off-chain. Each individual order may be valid with respect to its signature, price, deadline, and gas parameters, yet the aggregate spend of all their signed orders can exceed the trader's actual spendable resources, namely their:

- on-chain ERC-20 balance for the given token,
- on-chain token allowance granted to the Angstrom contract, and
- internal protocol balance already held inside Angstrom.

Because nodes ingest orders asynchronously, two honest nodes may observe a user's aggregate order set in different sequences. The validation layer must therefore deterministically decide, independent of arrival order, which subset is executable and which orders must be parked or rejected.

High-level approach. Validation is split into three progressively more expensive stages:

(1) **Stateless checks**—Verify syntax: amount and price fields are set, gas limits are sane, max-gas is less than the minimum quantity, deadlines/blocks are in the future, and the EIP-712 or EIP-1271 signature matches the signer.
(2) **Account-state evaluation**—For each address, maintain an in-memory ledger of *pending* spend deltas sorted by a deterministic priority rule. When a new order arrives we:
   (a) read the latest on-chain balances, allowances, and Angstrom internal balances once,
   (b) subtract higher-priority orders to obtain the *live* spendable tuple for each of the fields values, and
   (c) decide whether order can be reserved without any component going negative. If not, the order is stored as *parked* together with a reason (insufficient balance, approval, etc.).
   (d) If an incoming order uses up the maximal balances for a given address, previous valid orders at lower priority states are updated to parked
(3) **EVM simulation**—For every order that passed stage 2 we locally call the Angstrom executor inside a REVM instance to check for runtime reverts[13].

Deterministic priority ordering. To guarantee identical results on every node, pending deltas are ordered by a strict total order:

(1) Top-of-block orders beat book orders.
(2) Higher top-of-block bids beat those with lower bids.
(3) Partial orders precede exact-quantity orders.
(4) Lower user-supplied nonce has precedence; if nonces tie, the lower meta-transaction hash breaks the tie.

This ordering is applied whenever the pending spend delta is updated, ensuring that two nodes processing the same multiset of orders end with identical reservation ledgers, even if their arrival orders differ.

A.2 Matching at p_e

The subset-sum algorithm used to determine the allocation at the post-bisection clearing price, p_e operates on integers. To convert between floating point representations of order quantities submitted by users and the integer representation for which the algorithm operates, a sufficiently granular protocol lot size is chosen; well-formed orders ascribe to this lot size, which is enforced as a check in the off-chain validation pipeline.

Let

$$S = \{(k_i, e_i)\}_{i=1}^{n_S}, \qquad D = \{(k_j, e_j)\}_{j=1}^{n_D}$$

denote asks (supply) and bids (demand), where each element consists of an integer kernel $k > 0$ and a non-negative tail elastic capacity e (exact orders have $e = 0$). Define

$$M \;=\; \min\Big(\sum_i k_i,\; \sum_j k_j\Big).$$

A pair of index sets $(\mathcal{S}, \mathcal{D})$ is feasible iff

$$\sum_{i\in\mathcal{S}} k_i - \sum_{j\in\mathcal{D}} k_j \;\le\; \sum_{j\in\mathcal{D}} e_j,$$
$$\sum_{j\in\mathcal{D}} k_j - \sum_{i\in\mathcal{S}} k_i \;\le\; \sum_{i\in\mathcal{S}} e_i.$$

The first inequality is the case that demand is short, while the second inequality is the case where supply is short. The goal is to maximize the cleared volume

$$V^\star = \max_{(\mathcal{S},\mathcal{D})\text{ feasible}} \Big(\sum_{i\in\mathcal{S}} k_i,\; \sum_{j\in\mathcal{D}} k_j\Big),$$

breaking ties so that allocations using the largest total tail capacity are preferred; any remaining ties are resolved by the smaller Angstrom order ID.

[13] The sender address can be on the blacklist for a certain Angstrom-supported ERC-20, in which case their order seems semantically valid without the EVM simulation. Upon simulation, a revert upon the `transferFrom` call is correctly exposed.

Dynamic-programming tables For every $v \in [0, M]$ keep three structures

$$\texttt{reachS}[v] \in \{0,1\},$$
$$\texttt{tailS}[v] \in \mathbb{N}_{\geq 0} \cup \{-1\},$$
$$\texttt{predS}[v] \in \{-1\} \cup (\texttt{tx\_id} \times [0, M]),$$

where

- $\texttt{reachS}[v] = 1$ iff v is attainable as a sum of kernels,
- $\texttt{tailS}[v]$ is the *maximum* unlocked tail capacity of any such subset (or -1 if unreachable),
- $\texttt{predS}[v] = (\texttt{tx\_id}, v_{\text{prev}})$ records the Angstrom order ID $\texttt{tx\_id}$ that last updated v and the predecessor volume v_{prev}; $\texttt{predS}[0] = -1$.

The forward pass in Algorithm 3 initializes $\texttt{reachS}[0] = 1$ and $\texttt{tailS}[0] = 0$ and then iterates through the orders. Whenever an order (k, e) with order ID $\texttt{tx\_id}$ extends a reachable state v to $v' = v + k$ (lines 6–18 of Algorithm 3) we apply the following tie-break rule:

(1) if $\texttt{reachS}[v'] = 0$, accept the update;
(2) else if the new tail budget t' is strictly larger than the stored $\texttt{tailS}[v']$, accept;
(3) else if $t' = \texttt{tailS}[v']$ but e is larger than the tail of the order already stored at v', replace it;
(4) else if $t' = \texttt{tailS}[v']$ and e equals the tail of the order already stored at v', and $\texttt{tx\_id}$ is smaller than the Angstrom ID of the stored order, replace it;
(5) otherwise keep the existing state.

This guarantees that, among all subsets realizing v', we retain the one that frees the most elastic capacity and, secondarily, the one that brings in the largest single-order tail, which directly maximizes the slack exploitable in the subsequent volume search.

An identical pass constructs $\texttt{reachD}, \texttt{tailD}, \texttt{predD}$ for the demand side.

Volume search The tables feed into the maximization

$$(V^\star, v_S^\star, v_D^\star) = \max_{\substack{v_S, v_D \leq M \\ \texttt{reachS}[v_S]=1 \\ \texttt{reachD}[v_D]=1}} f(v_S, v_D),$$

$$f(v_S, v_D) = \begin{cases} v_S, & v_S = v_D, \\ v_S, & v_S > v_D \ \wedge \ v_S - v_D \leq \texttt{tailD}[v_D], \\ v_D, & v_D > v_S \ \wedge \ v_D - v_S \leq \texttt{tailS}[v_S], \\ 0, & \text{otherwise.} \end{cases}$$

We scan all reachable pairs and return the tuple $(V^\star, v_S^\star, v_D^\star)$.

Back-tracking and tail allocation Algorithm 4 walks the predecessor maps from $v_S^\star$ back to 0, recovering the transaction IDs for the transactions that formed the kernel subsets $\mathcal{S}^\star$. The algorithm to find $\mathcal{D}^\star$ from $v_D^\star$ is analogous.

Only partial orders in these sets unlock their tails. If a gap remains ($v_S^\star \neq v_D^\star$), it is filled by the unlocked tails on the short side, resolved pro-rata on elastic volume over all partially-fillable orders that were cleared; any remaining tail inventory is left unfilled and does not carry over to future blocks: the order is exhausted.

Complexity Table construction is $O((n_S + n_D)M/64)$ bit operations, volume search $O(M^2/64)$ in theory but $O(M/64)$ in practice thanks to 64×64 word compression and early exit once the scan falls below the incumbent $V^\star$.[14]

Note that subset-sum, even with slack, is known to be NP-complete. Our algorithm is exponential in the worse case input-length N (as $M \leq 2^N$), but on realistic batches to be cleared at the final price, M is not impractically large.

Algorithm 3 Subset-Sum Forward Pass (for S)

1: **Input:** kernels $\{k_i\}$, tails $\{e_i\}$, capacity M
2: **Output:** boolean array reachS, tail budget tailS, predecessor map predS

3: reachS[0..M] ← 0; tailS[0..M] ← −1; predS[0..M] ← −1
4: reachS[0] ← 1; tailS[0] ← 0 ▷ base state

5: **for all** orders (k, e) in S **with ID** tx_id **do**
▷iterate backward so each order is used at most once
6: **for** $v = M - k$ **downto** 0 **do**
7: **if** reachS[v] = 1 **then**
▷ add current kernel and its tail capacity
8: $v' \leftarrow v + k$, $t' \leftarrow$ tailS[v] + e
9: **if** reachS[v'] = 0 **or** $t' >$ tailS[v'] **then**
10: reachS[v'] ← 1; tailS[v'] ← t'
▷ remember *who* (tx_id) and *from where* (v)
11: predS[v'] ← (tx_id, v)
12: **else if** t'=tailS[v'] **and** tx_id<predS[v'].tx_id **then**
13: predS[v'] ← (tx_id, v)
14: **else**
15: **continue**
16: **end if**
17: **end if**
18: **end for**
19: **end for**

Algorithm 4 Recover Supply Subset from predS

Require: target volume $v^\star$, predecessor map predS
1: $\mathcal{S} \leftarrow \emptyset$
2: **while** $v^\star > 0$ **do**
▷ order tx_id was last added
3: (tx_id, v_{prev}) ← predS[$v^\star$]
4: $\mathcal{S} \leftarrow \mathcal{S} \cup \{\texttt{tx\_id}\}$
5: $v^\star \leftarrow v_{\text{prev}}$ ▷ jump to predecessor state
6: **end while**
7: **return** $\mathcal{S}$ ▷ Set of Angstrom order IDs clearing $v^\star$

B Consensus

The following section assumes $|\mathcal{V}| = 2f + 1$ with at most f Byzantine validators and the Δ–synchrony model of section 2.1.

First, we define the consensus trace object, which is gossiped by the leader after the slot as the final step in the consensus procedure.

[14] For lot sizes in the millions, the dynamic program runs in under 1 m.s. in practice. The Angstrom validator network can impose lot size as part of order validation to optimize between order granularity and clearing performance.

The trace object is a necessary precursor for both the fault specifications defined in appendix B.2 and formal guarantees defined in appendix B.3.

B.1 Consensus Trace

The protocol needs a way to penalize any verifiable attempts of contradicting signed messages throughout the procedure to ensure no optionality can be exercised *for free* (see note in caption of figure fig. 8). To ensure this, the leader must gossip the consensus trace for their Round-4 signed object to all other validators before some grace period (tens of blocks) after the slot.

Intuitively, the consensus trace can be visualized as a tree which recurses through the quorum signatures of the Round-4 and Round-3 signed objects, ending with the aggregate Round-1 signed objects as the leaf layer. In practice, the footprint of this tree can be greatly reduced by storing pointers to duplicate Round-1 signed objects that appear in multiple Round-3 quorum signatures. This turns the conceptual tree into a DAG with at most one physical copy of each signed object. An example consensus trace is the list of (m, σ) tuples that form the tree pictured in fig. 7. From the consensus trace, any node is able to verify validity of the protocol for the slot to which the trace pertains or expose equivocation at any of the rounds.

Complexity Assume a validator set of n, a throughput parameter $k \propto \Delta$-interval txs, a signature size Σ (64 bytes for ECDSA), a digest size D (32 bytes), a transaction identifier size tx_id (< 32 bytes), and a pointer size $P \ll \Sigma$. The consensus trace contains one Round-4 signed object, up to n Round-3 signed objects, and up to n Round-1 signed objects. The Round-4 and Round-3 signed objects now hold n pointers (to Round-3 and Round-1 signed objects, respectively) instead of n full signatures. Thus, upon pointer compression, the total byte complexity becomes

$$O\Big(n(D+\Sigma) + n^2 \cdot P + \underbrace{nk \cdot \mathsf{tx\_id}}_{\text{tree leaves}}\Big).$$

In the initial setup with small n and large k (from active market making), the leaf layer dominates yielding size $O(nk)$. Should this be prohibitive, we detail a slight modification to the consensus protocol in appendix B.3.3 which distributes an abridged consensus trace without the leaf layer and achieves better no-optionality properties at the expensive of worse liveness. Additionally, adding the full consensus trace to the modified protocol raises the penalty for exercising an option significantly (from the absence fault defined in appendix B.2.2 to the trace censorship fault defined in appendix B.2.4). The protocol can easily move to either of these modified consensus specifications based on the frequency of exercised options as a future upgrade.

B.2 Fault Specifications

This appendix section specifies the provable misbehaviors that can arise during the consensus protocol of Section 2. For each fault type we (i) give a formal definition, (ii) describe the minimal evidence to be published on-chain as a proof of fault, and (iii) reason about viable protocol-level penalties[15].

[15] Slashing will not be live upon initial launch but will be added as a protocol upgrade in the future

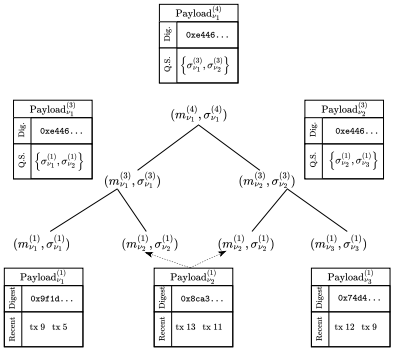


Figure 7. The valid trace of a more succinct signature aggregation (though equal in initial state and result) as the consensus path in fig. 2. Payloads are included in each message by construction. Given that the payload includes the quorum signature field which governs the tree structure, we also picture them adjacently to provide a more explicit construction.

B.2.1 Equivocation Fault

Specification. A validator equivocates in round r if it produces two distinct signed objects with the same slot–identifier but different payloads:

$$(m, \sigma) = \big(\mathsf{slot}, r, \mathsf{X}\big),\ (m', \sigma') = \big(\mathsf{slot}, r, \mathsf{Y}\big),\ \mathsf{X} \neq \mathsf{Y}.$$

Such behavior can only be malicious and thus incurs a full stake slashing along with immediate ejection from the active validator set.

Proof. Uploading $\big\{(m, \sigma), (m', \sigma')\big\}$ to the slashing contract is sufficient: verification succeeds iff (i) both signatures validate under v's public key, (ii) both messages have identical slot and round fields, and (iii) the payloads differ.

B.2.2 Absence Fault

Definition. Every validator v must multicast a Round 1 signed object by deadline $T - 5\Delta$, ensuring it gets to all honest validators by $T - 4\Delta$ by the network synchrony assumption. v incurs an absence fault if at least $f + 1$ validators attest that no valid Round 1 bundle from v was received by $T - 4\Delta$.

No-Bundle Attestation. Each validator ρ can issue at $T - 4\Delta$ a statement

$$c_\rho \;=\; \big(\mathsf{slot}, \mathsf{LateLocal}, v_{\mathrm{pk}}\big), \qquad \tau_\rho = \mathrm{sign}_\rho(c_\rho),$$

if and only if it has not received any signed object from v by the deadline.

Proof Format. An absence proof is the set

$$\mathcal{P}_{\mathrm{abs}} \;=\; \big\{(c_\rho, \tau_\rho)\big\}_{\rho \in H}, \quad |H| \geq f + 1.$$

The slashing contract checks that

(i) every τ_ρ verifies under ρ's key;
(ii) all c_ρ contain the same absentee public key v_{pk};
(iii) $|H| \geq f + 1$

Soundness. If v was honest and disseminated the bundle on time, every honest validator received and could locally verify it, so no honest validator would sign a LateLocal attestation and the threshold $f+1$ cannot be reached. If v withheld or delayed the bundle, all f+1 honest validators will sign LateLocal, guaranteeing that an absentee proof can be produced.

Penalty. Angstrom imposes graded slashing proportional to the evidence:

$$\text{slash fraction} \;=\; \Psi_{\text{abs}} \;\cdot\; \frac{|H| \;-\; f}{f},$$

where $\Psi_{\text{abs}} \in (0, 1]$ is the maximal amount of stake loss for an absence fault, which is taken when all $2f$ other validators validate the fault.

Because absence can be caused by network partition rather than malice, the protocol should set Ψ_{abs} based on the volatility of the reference price process and underlying pool liquidity parameter such that it is high enough to ensure exercising optionality (cf. fig. 8) is not incentive compatible, while not over-penalizing honest nodes who can potentially be randomly partitioned.

B.2.3 Final View Censorship Recall that the final view censorship fault occurs when the elected leader fails to distribute their Round 4 signed object to other validators by $T-\Delta$. The proof for this fault follows immediately from the absence fault, except LateFinal replaces LateLocal in the second field of the message tuple to be signed. The proof format, slashing contract criteria, and soundness arguments remain the same. Final view censorship is deemed more malicious than absence given the increased responsibilities and extractable value for the leader in a given round, but can still arise from a network partition; therefore, Angstrom imposes a similar grading slashing, proportional to the evidence, with the maximal slashing amount increasing to twice that of the absence fault.

B.2.4 Trace Censorship/Data Withholding Fault This fault occurs when the leader for a given slot fails to distribute a consensus trace for their Round-4 signed object to other validators prior to the expiry of a protocol-defined grace period, or if they distribute a malformed trace[16].

The attestation process for this fault also follows from the absence fault, except TraceFault replaces LateLocal in the second field of the message tuple to be signed. The proof format, slashing contract criteria, and soundness arguments remain the same. This fault is deemed more malicious than both absence and final view censorship, as the protocol defined grace period can be arbitrarily long such that even after recovering from a partition, the leader will be able to send other validators a valid consensus trace and avoid penalization; therefore, Angstrom imposes a grading slashing, proportional to the evidence, with the maximal slashing amount increasing to values on an order of magnitude higher than the absence fault's penalty.

[16] A Byzantine leader, attempting to conceal a bid-shade, can withhold messages from their trace to prevent any other validator from exposing their equivocation to the slashing contract

B.3 Formal Guarantees: Censorship Resistance, and No-Optionality

B.3.1 Censorship Resistance

Lemma B.1 (Inclusion of honest orders). *Let tx′ be any order broadcast by an honest validator during Round 0 of slot s. Then tx′ must be considered in the calculation of the final MergeDigest⋆ for that slot.*

Proof. Two cases:

(i) tx′ received before $T-6\Delta$. It is present in every honest validator's local multiset $\mathcal{M}_v$ and therefore in every LocalDigest_v. Because these digests are inputs to BuildMerge, tx′ is necessarily in every honest validator's merged multiset, and thus included in their Round-3 signed object. Since the leader must chose $f+1$ of these Round-3 signed objects to produce a canonical view, they must chose at least one multiset from an honest validator (as there are only f Byzantine validators). It follows that tx′ must be part of the union of orders involved in calculating MergeDigest⋆ for the slot.

(ii) tx′ received in $[T-6\Delta,\, T-5\Delta)$. It may reach only the originating honest validator before $T-5\Delta$, but that validator forwards tx′ inside its recent field in the payload for its Round 1 signed object. Since it's included in the validator's Round 1 signed object, it reaches every other honest validator before Round 3, so the order is included in each honest validator's Round-3 signed object by construction. Following the same line of reasoning as (i) ensures tx′ is included in any set of Round 3 signed orders which the leader uses to compute MergeDigest⋆. □

B.3.2 No-Optionality

Lemma B.2. *If there are no absence faults, every honest validator finalizes the same MergeDigest′ within slot s which corresponds with their Round 3 signed object's MergeDigest. If the leader is honest, this MergeDigest′ will be the MergeDigest⋆ for the slot.*

Proof. All Round-1 messages reach at least one honest validator by $T-4\Delta$ (due to the no-absence assumption) and, by Δ-synchrony, reach every honest peer before $T-3\Delta$. Consequently every honest validator possesses the identical set $\mathcal{S}^{(1)}_{\text{hon}}$ when Round 3 begins, and therefore computes a common MergeDigest⋆. Each honest validator includes this digest in its Round-3 signed object.

An honest leader will then aggregate the f Round-3 signed objects from other honest nodes that agree with their Round-3 MergeDigest′ to produce the Round-4 signed object which necessarily has MergeDigest⋆ = MergeDigest′. □

Theorem B.3 (No "free option"). *Denote tx_{opt} as the transaction which a Byzantine leader hopes to have an option on. The leader is unable to exploit the consensus protocol to create two sets of $f+1$ Round-3 signed objects where one set includes tx′ and the other does not without at least committing an absence fault.*

Proof. For the leader to avoid absence, they must send their bid view to at least one honest validator by the $T-4\Delta$ deadline. Since they want an option on tx_{opt}, their bid view must also include this transaction. By lemma B.2, all honest validators will possess the transaction in their set when constructing their Round-3 signed

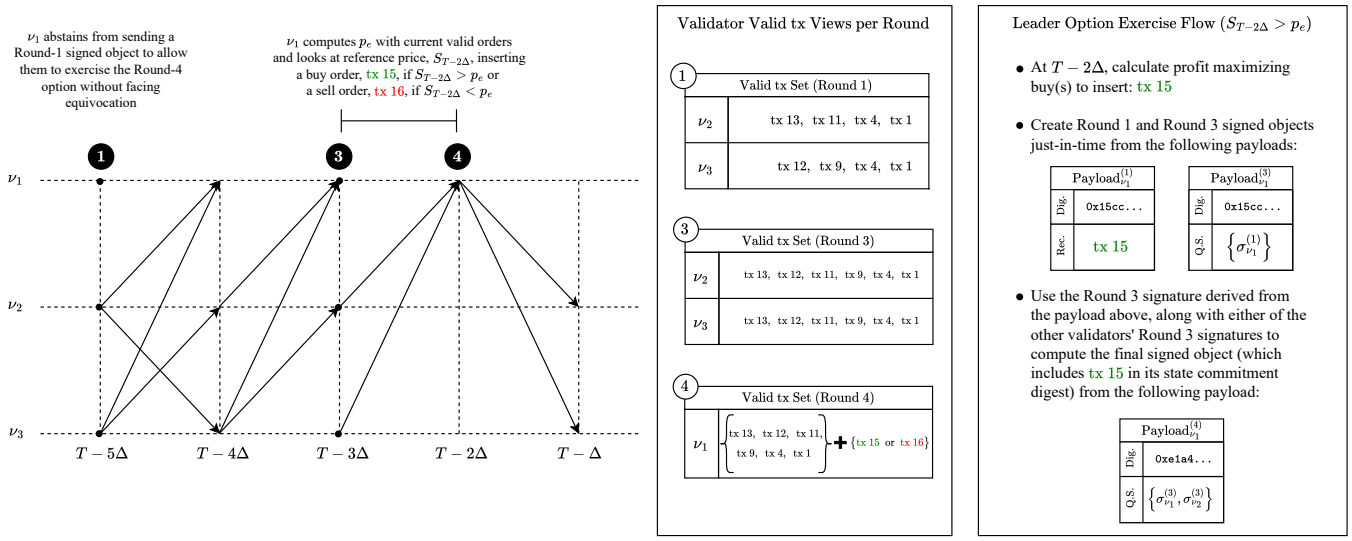


Figure 8. Byzantine leader ν_1 is able to hold the right to an option by committing an absence fault, not sending their Round-1 signed objects to other validators. At Round-4, they're able to compute the profit-maximizing transaction(s) to insert based on the external reference price by creating their Round-1 and Round-3 signed objects just-in-time (after their respective deadlines and immediately prior to the Round-4 deadline). Their resultant consensus trace appears valid to all observers. **Note:** Without the distribution of the consensus trace, the Byzantine leader would have a *free* option by sending their Round-1 view in its initial state to ν_2 and ν_3 prior to the $T-4\Delta$ deadline, but shading that view just before Round-4 to insert profitable transactions. Without the consensus trace, there is no way for either ν_2 or ν_3 to prove they equivocated, as they would not have the alternative Round-1 message or signature that was produced just-in-time to send to the slashing contract with the contradicting message they received prior to $T-4\Delta$ as proof of equivocation.

object. Since there are $f+1$ honest validators, $f+1$ Round-3 signed objects will include tx_{opt}. For the leader to create another set of $f+1$ signed objects that do not include tx' would require an honest node to equivocate, contradicting the honesty assumption. Thus, absence is the minimal fault required to hold the right to an option. □

It's important to note that the result of theorem B.3 implies that the leader must decide whether or not they want to hold the right to an option *ex-ante*, prior to market conditions upon the finalization of their Round-4 view being readily apparent. Without this property, Byzantine validators would be able to exercise a risk-free option by feigning absence whenever the value from exercising the option outweighs the absence penalty.

B.3.3 Protocol Modifications to Limit Optionality We can make a slight modification to the existing protocol to limit optionality and remove the need for distributing the Round-1 signed objects as part of the consensus trace upon a slot's finalization by simply requiring the $\geq f+1$ Round-3 signed objects that a leader includes to compute $\text{MergeDigest}^{\star}$ to all have equal values for their MergeDigest. All $\geq f+1$ Round-3 objects in this trace must share the same MergeDigest; validating this, along with verifying the signatures themselves, is enough to ensure no malfeasance.

Theorem B.4 (No "free option" (alternate protocol)). *In a slot with $2f+1$ validators and at most f Byzantine actors, no two distinct digests $\text{MergeDigest}^{\star} \neq \text{MergeDigest}^{\dagger}$ can each gather a valid Round-4 commit containing $f+1$ distinct signatures, without an attributable fault being committed.*

Proof. By Lemma B.2, if there is no absence, every honest validator signs $\text{MergeDigest}^{\star}$ in Round 3. Any quorum of $f+1$ signatures therefore contains at least one honest signature. Suppose, for contradiction, that a second digest $\text{MergeDigest}^{\dagger} \neq \text{MergeDigest}^{\star}$ also obtains $f+1$ signatures. Among them must be an honest signature, forcing an honest validator to have signed two different digests in the same round, committing an equivocation which contradicts the honest node assumption.

A Byzantine leader cannot fabricate MergeDigest commitments for the Round-3 signed objects of other validators, as the entire Round-3 signed object is distributed in the minimized consensus trace after the protocol's finalization, or the leader faces the far steeper trace censorship fault. □

The value of optionality for this protocol specification is also strictly less than that of the original protocol. This is because the modified protocol similarly requires that the Byzantine leader commits to an absence fault before the realization of potentially profitable external information, but it also necessitates that they commit to the *specific transactions* to include at time $T-3\Delta$ instead of being able to compute the profit maximizing transactions at $T-2\Delta$ as in the original protocol. The specific breakdown of the process a Byzantine leader must go through to hold an option is broken down in fig. 9.

Additionally, including the full consensus trace with the modified protocol is enough to ensure that the minimum fault to hold the right to an option is the costly trace censorship fault, as the process required to hold the right to an option requires equivocation in Round-1 of the protocol should the option need to be exercised (in

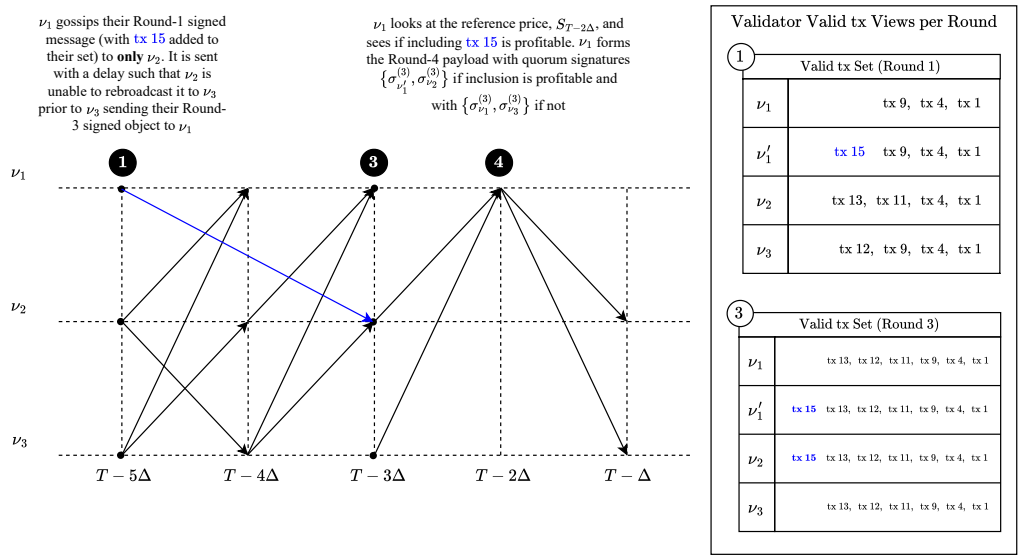


Figure 9. Byzantine leader ν_1 sends their Round 1 signed object that includes a late order to *one* honest node, ν_2, such that they receive it at $T - 3\Delta - \epsilon$. The honest node will include it in their merged view, allowing the Byzantine leader to produce two different, valid MergeDigest* based on whether they include the targeted honest node's MergeDigest or that of another honest node. In either case, the Byzantine consortium is able to maintain an option, but they must pay the cost of an absence fault and commit to their option ex-ante to hold that right. The absence fault follows from all $f + 1$ honest nodes attesting to not seeing the Byzantine node's Round 1 signed object prior to the $T - 4\Delta$ deadline. **Note:** If the leader backs out of the Round-1 view they submitted to ν_2 upon the inclusion of tx 15 being unprofitable based on $S_{T-2\Delta}$, the consensus trace still looks valid to ν_2 despite ν_1 equivocating by signing two different Round-1 messages. This is due to the fact that the abridged consensus trace does not include the Round-1 signed objects that ν_2 would need to provide to the slashing contract, along with the divergent Round-1 signed object they received from ν_1 at $T - 3\Delta - \epsilon$, to prove malfeasance.

addition to absence), which can be verified by requiring distribution of the unabridged consensus trace after the protocol finalizes.