

Application-Defined Ordering Guarantees

Ritual

April 20, 2026

Abstract

In standard blockchain protocols, the proposer has unilateral authority over transaction ordering within a block. This authority is the root of ordering-dependent MEV: sandwich attacks, frontrunning, and backrunning all exploit the proposer’s (or builder’s) ability to choose the sequence in which transactions execute. Ordering guarantees make promises about the relative ordering of the subsequence of transactions targeting an application in a block. Verifying ordering conjuncts is cheap: the validator executes the block and checks function-call ordering dynamically. Block building is hard: the proposer must find a valid ordering across all constraints before execution, which can be shown NP-hard in the general case (cf. constrained job scheduling with precedence constraints [8]). For application-defined ordering, the finalized-state case, in which an ordering and its tiebreakers depend only on committed state, is resolved by adopting the Monotone Priority System [2], the unique contract-specific sequencing system under five natural axioms, whose global integer priorities replace general call-sequence policies. The non-finalized-state case is resolved by composing inclusion-guaranteed batch execution with a one-block-delay buffer, converting arbitrary state-dependent ordering into a single-transaction execution problem.

1 Introduction

Symphony [17] is an execution-aware SMR framework organized around a single principle: *protocol-enforced guarantees*.

The set of blocks that a proposer can build is constrained in order to provide guarantees on *inclusion*, *exclusion*, and *sequencing* to applications and users.

In standard blockchain protocols, the proposer has unilateral authority over transaction ordering within a block. This authority is the root of ordering-dependent MEV (maximal extractable value [5]): sandwich attacks, frontrunning, and backrunning all exploit the proposer’s (or builder’s) ability to choose the sequence in which transactions execute.

Application-controlled execution (ACE) has been deployed in practice. Hyperliquid enforces cancel prioritization at the L1 level: within each block, cancel orders must precede take orders, improving spreads by protecting market makers from stale-quote extraction. This is the most notable deployed instance of ACE and demonstrates that application-specific ordering yields measurable trading improvements [1].

Ordering guarantees transfer the proposer’s sequencing power to individual applications, allowing each application contract to declare its own ordering policy.

Ordering guarantees make promises about the relative ordering of the subsequence of transactions targeting an application in a block. A contract C declares an ordering policy $(\pi_C, \{\tau_i\})$: a finite ordered list of recognized call sequences over C ’s functions, with optional tiebreakers τ_i evaluated on finalized state for intra-priority ordering. Transactions whose call sequences match a recognized pattern must appear in the block in the order specified by π_C ; transactions with unrecognized call sequences are automatically deprioritized and placed below all recognized transactions. See Section 3 for a more in-depth treatment of ordering guarantees.

This paper makes the following contributions:

- Ordering constraints are conjuncts of the block validity function, checked by every validator (Section 3.2).
- Ordering guarantees constrain sequencing-dependent extraction on directly-accessed adopting contracts (Section 3.3).
- The design space for ordering guarantees has two tractable points and one open problem: (1) finalized-state ordering via the Monotone Priority System [2] with optional state-dependent tiebreakers (zero latency, any ordering computable from committed state), (2) non-finalized-state ordering via buffered lazy evaluation with inclusion guarantees (one-block latency, arbitrary ordering including current-block state), and (3) direct non-finalized-state ordering in the block construction path (zero latency, circular dependency, open) (Section 3.4).

Section 2 establishes the model: system model, extended state, extended block validity, and predictable validity. Section 3 instantiates ordering guarantees. Section 4 establishes liveness. Section 5 discusses limitations. Section 6 reviews related work. Section 7 concludes.

2 Model

2.1 System Model

We consider n stake-weighted replicas (validators) running a *state machine replication (SMR)* protocol [11] under partial synchrony [6]: after an unknown Global Stabilization Time (GST), messages between correct replicas are delivered within an unknown bound Δ . At most $f < n/3$ replicas are Byzantine; replica i holds stake $w_i > 0$, the total stake is $W = \sum_{i=1}^n w_i$, Byzantine replicas control less than $W/3$, and a set of replicas holding more than $2W/3$ is a *quorum*. SMR guarantees that all correct replicas deliver the same sequence of transactions and that the resulting state is uniquely determined by applying a deterministic state transition function $\text{STF} : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{S}$ to the initial state in that order, where $\mathcal{S}$ is the state space and $\mathcal{T}$ the set of transactions.

The proposer is untrusted. Outside the checks imposed by the block validity function (Section 2.3), a proposer has full power to determine the contents of the block, and a Byzantine proposer may include, exclude, and order transactions arbitrarily. Enforcement of a guarantee must therefore be Byzantine fault-tolerant: it holds because correct validators reject blocks that violate it.

2.2 Extended State and Guarantees

Symphony extends the standard SMR model [11] with additional state components:

Definition 2.1 (Symphony State). *The state S consists of:*

- *Execution state S_{exec} : account balances, contract storage (standard).*
- *Trigger pools: a set of active inclusion triggers $\mathcal{I} = \{(P_i, T_i, \text{TTL}_i)\}$ and exclusion triggers $\mathcal{E} = \{(Q_j, \text{TTL}_j)\}$ with predicates $Q_j : \mathcal{S} \times \mathcal{T} \rightarrow \{0, 1\}$, where each Q_j is scoped to transactions targeting the registering contract.*
- *Ordering policies: a map from contract addresses to ordering policies $\{(\pi_C, \{\tau_i\})\}_{C \in \mathcal{C}}$, where $\mathcal{C}$ is the set of contracts with a registered ordering policy (defined in Section 3.2).*

An inclusion or exclusion trigger is *active* from its registration until its TTL expires, and an ordering policy is active until it is deregistered; each active trigger or ordering policy is an *active guarantee*.

Definition 2.2. An inclusion trigger is a triple (P, T, TTL) registered in the chain state:

- $P : \mathcal{S} \rightarrow \{0, 1\}$ is a state predicate.
- T is a transaction.
- $TTL \in \mathbb{N}$ is the time-to-live in blocks.

2.3 Extended Block Validity

Definition 2.3 (Extended Block Validity). A block B proposed for state S is valid if and only if all of the following conjuncts hold:

$$\text{Valid}(B, S) = \text{Structural}(B) \tag{1}$$

$$\wedge \text{Execution}(B, S) \tag{2}$$

$$\wedge \text{InclusionComplete}(B, S) \tag{3}$$

$$\wedge \text{ExclusionCompliant}(B, S) \tag{4}$$

$$\wedge \text{OrderingCompliant}(B, S) \tag{5}$$

The first two conjuncts are standard in any BFT protocol:

- $\text{Structural}(B)$: the block is well-formed, all transaction signatures are valid, the parent hash is correct, and the proposer is the designated leader for this round.
- $\text{Execution}(B, S)$: the proposed post-state S' is the correct result of applying the block's transactions to S . For replicated transactions, every correct validator re-executes and checks.

The final three conjuncts are Symphony's extensions:

- $\text{InclusionComplete}(B, S)$: for every active inclusion trigger $(P_i, T_i, TTL_i) \in \mathcal{I}$ such that $P_i(S) = 1$, the transaction T_i appears in B . The semantics are defined in [16].
- $\text{ExclusionCompliant}(B, S)$: for every active exclusion trigger $(Q_j, TTL_j) \in \mathcal{E}$, no transaction $T \in B$ satisfies $Q_j(S, T) = 1$. The semantics are defined in [16].
- $\text{OrderingCompliant}(B, S)$: for every contract C with a registered ordering policy $(\pi_C, \{\tau_i\})$, the subsequence of transactions in B that target C appears in the order specified by π_C and the tiebreakers τ_i evaluated on S (Definition 3.2). Section 3 defines the semantics.

A block that omits a triggered transaction is invalid. Every correct validator rejects it.

All three guarantees follow the same structure. Each is implemented as a conjunct in Definition 2.3:

1. A beneficiary (user, application, or the protocol itself) registers a guarantee by publishing a predicate on-chain. This registers a trigger in $\mathcal{I}$, $\mathcal{E}$, or $\{(\pi_C, \{\tau_i\})\}$.
2. The predicate becomes a conjunct of the block validity function.
3. Every block proposed by a correct proposer must satisfy all active conjuncts.
4. Every correct validator independently evaluates all active conjuncts and rejects blocks that violate any. This deterministic evaluation is the default and the basis for Symphony's BFT safety argument.
5. The proposer retains only the *residual* discretion: the freedom to act that is not constrained by active guarantees.

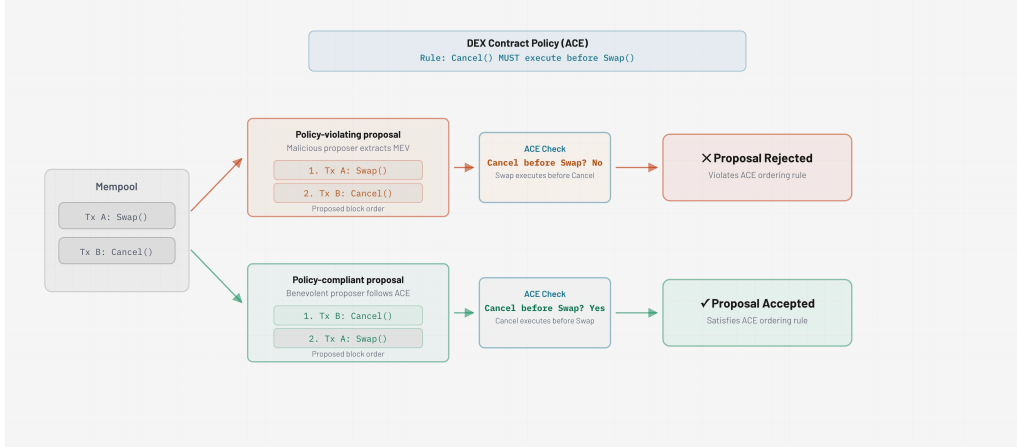


Figure 1: Application-defined ordering; in the figure, ACE (application-controlled execution) denotes the ordering-policy check. A smart contract declares an ordering policy over recognized call sequences. A malicious proposer submitting transactions in a policy-violating order is deterministically rejected; an honest proposer complying with the policy is accepted.

2.4 Predictable Validity

Extended external validity requires that the block proposer and all correct validators agree on the state against which validity predicates are evaluated.

Definition 2.4 (Predictable Validity). *An SMR protocol provides predictable validity if, whenever a correct proposer constructs a block B , there is a unique state S (the state the proposer is building on), derived from the committed prefix of the ledger in the proposer’s view, such that the proposer can compute S from its local information, every correct validator that receives B can verify that S is the correct pre-state and that B is valid when applied to S , and every predicate $p : S \rightarrow \{0, 1\}$ yields the same value $p(S)$ for the proposer and for all correct validators.*

Proposition 2.5 (Predictable Validity Requirement [15]). *If an SMR protocol does not provide predictable validity because a correct proposer and a correct validator can derive different states from their respective local information, then there exist a state predicate p used in the block validity function and executions in which a correct proposer constructs a block that it considers valid while a correct validator considers the block invalid, or vice versa.*

The proof [15] exhibits a predicate that distinguishes the states derived from the proposer’s and the validator’s local information. Single-leader protocols such as CometBFT [3] and Hot-Stuff [14] provide predictable validity.

3 Application-Defined Ordering Guarantees

3.1 Transaction Ordering as Application-Layer Authority

Ordering guarantees can be implemented at multiple levels [1]: as an app-chain (Hyperliquid), via on-chain batching with delayed execution (AMQs on Solana), via off-chain sequencing delegated to specialized parties (Sorella), through voluntary proposer commitments (PEPC), or enshrined in L1 consensus with mandatory validator participation. Symphony implements the last of these: ordering constraints are conjuncts of the block validity function, checked by every validator (Figure 1).

Symphony’s ordering guarantees differ from prior work on ordering-fairness in three respects:

1. **Per-contract, not global.** Each contract declares its own ordering policy. An AMM contract may order transactions by price impact. A lending contract may order liquidations by collateral ratio.
2. **Semantic, not temporal.** Ordering policies are functions of transaction content and chain state, not arrival times. This enables application-specific ordering logic that temporal rules cannot express.
3. **Simultaneous heterogeneous policies.** Multiple contracts with different ordering policies coexist in the same block. Transactions targeting contract A are ordered by A 's policy; transactions targeting contract B are ordered by B 's policy; transactions targeting neither are ordered at the proposer's discretion.

3.2 Formal Definition

Let $\mathcal{F}_C = \{f_1, \dots, f_{n_C}\}$ be the set of external function signatures of contract C , and let $\mathcal{F}_C^*$ denote the set of all finite sequences over $\mathcal{F}_C$ (including repetitions). A single transaction targeting C may invoke any number of functions on C , any number of times, in any order. The *call sequence* $\sigma(T)$ of a transaction T on contract C is the ordered list of function invocations on C observed in T 's execution trace. A transaction *targets* C if its call sequence $\sigma(T)$ on C is non-empty.

Definition 3.1 (Ordering Policy). *An ordering policy for contract C consists of:*

- *A finite ordered list of recognized call sequences:*

$$\pi_C = [s_1, s_2, \dots, s_k] \quad \text{where each } s_i \in \mathcal{F}_C^*$$

The position i is the sequence's priority (lower index = higher priority). The policy has finite support: only finitely many call sequences are recognized.

- *An optional tiebreaker function per recognized sequence: $\tau_i : \mathcal{S} \times \mathcal{T} \times \mathcal{T} \rightarrow \{-1, 0, 1\}$, evaluated on finalized state S , that orders transactions matching the same s_i ; $\tau_i(S, T, T') = -1$ means that T precedes T' , 1 that T' precedes T , and 0 that the proposer may order them either way, and τ_i must induce a strict weak order on the transactions it compares. If no τ_i is declared, the proposer retains discretion within that priority level.*

All other sequences in $\mathcal{F}_C^*$ are *unrecognized* and are automatically assigned the lowest priority: every transaction with a recognized call sequence must precede every transaction with an unrecognized call sequence in the block. Among transactions with an unrecognized call sequence, the proposer retains full ordering discretion. This avoids requiring the contract to enumerate all possible call sequences: the contract specifies only the patterns it cares about, and everything else is deprioritized by default. This choice gives extra confidence to application designers and users who know that any searcher trying to use the contract in unexpected ways will always be sequenced after the “expected” user transactions.

The tiebreaker τ_i is evaluated on the finalized state (the pre-state of the current block), so the intra-priority ordering is deterministic and known before block construction begins. This is how state-dependent ordering (e.g., sorting liquidations by collateral ratio) is expressed within the static call-sequence framework.

Definition 3.2 (Enforcement of ordering guarantees). *A block B proposed for state S satisfies the **OrderingCompliant** conjunct if and only if: for every contract C with a registered ordering policy $(\pi_C, \{\tau_i\})$ and for every pair of transactions $T_i, T_j \in B$ targeting C :*

- *If both $\sigma(T_i) = s_a$ and $\sigma(T_j) = s_b$ are recognized with $a < b$, then T_i appears before T_j in B .*

- If $\sigma(T_i)$ is recognized and $\sigma(T_j)$ is unrecognized, then T_i appears before T_j in B .
- If both $\sigma(T_i) = \sigma(T_j) = s_a$ (same recognized sequence) and τ_a is declared, then T_i and T_j must appear in the order specified by $\tau_a(S, T_i, T_j)$.
- If both $\sigma(T_i)$ and $\sigma(T_j)$ are unrecognized, no ordering constraint applies.

Recognized transactions are ordered by their priority in π_C and always precede unrecognized transactions. Within the same priority level, the tiebreaker τ_i (if declared) determines ordering based on finalized state. Among unrecognized transactions, the proposer retains discretion.

Note that, within a priority level, the tiebreaker semantics are expressive enough to implement any order over the transactions matching that level's call sequence. The benefit of having a finite, pre-specified list of call sequences π_C is that we expect application designers to have a good idea of the common usage patterns of their contracts, without wanting to specify semantics over transaction ordering outside of that set (other than the priority of the common patterns).

Call sequence determination. $\sigma(T)$ is determined by execution: it is the projection of T 's execution trace onto calls to contract C , in execution order. For a given transaction and pre-state, $\sigma(T)$ is deterministic, so all correct validators agree. The block builder may need to simulate execution to determine $\sigma(T)$ before ordering; for the lazy evaluation path (Section 3.4.1), this is moot since the contract's batch-processing function handles ordering internally. For direct execution, transaction metadata (pre-committed call sequences, analogous to access lists) mitigates: transactions that diverge from their declared sequence revert.

Policy examples.

- **Cancel prioritization (Hyperliquid):** $\pi_C = [(\text{Cancel}), (\text{PostOnly}), (\text{GTC}), (\text{IOC})]$. Single-function call sequences. All pure cancel transactions precede all post-only transactions, which precede all GTC transactions, etc. A transaction calling $(\text{Cancel}, \text{GTC})$ (a multi-call sequence) is unrecognized, hence placed after all recognized transactions, unless the contract explicitly adds that pattern to π_C . Deployed in production [1].
- **Oracle freshness:** $\pi_C = [(\text{Update}), (\text{Read})]$. Updates precede reads. A transaction that does $(\text{Update}, \text{Read})$ in a single transaction could be recognized at a specific priority or left unrecognized (placed after all recognized transactions), at the contract's discretion.
- **Lending with state-dependent tiebreaker:** A lending contract recognizes:

$$\pi_C = [(\text{Liquidate}), (\text{Liquidate}, \text{Repay}), (\text{Borrow})]$$

Pure liquidations first, then liquidate-and-repay combos, then pure borrows. The contract declares a tiebreaker τ_1 on the (Liquidate) priority level: $\tau_1(S, T_a, T_b)$ orders liquidation transactions by the collateral ratio of their target position in finalized state S , lowest ratio first. The call-sequence priority is static; the intra-priority ordering is state-dependent.

- **Opt-out:** A contract with $\pi_C = []$ (empty list) imposes no ordering constraints. All transactions are unrecognized and the proposer has full discretion.

3.3 MEV Surface Analysis

Ordering guarantees constrain sequence-based MEV for contracts that adopt it and are accessed via direct (non-wrapped) transactions. The residual MEV surface includes:

Intra-application ordering MEV: constrained. Sandwich attacks, frontrunning, and back-running require the attacker to control the relative ordering of their transactions with respect to the victim’s within the same contract. When the ordering is determined by π_C , the proposer cannot insert attacker transactions at strategic positions in the per-contract subsequence. However, the proposer retains discretion over inter-application ordering (where the C -targeted subsequence sits within the block), inclusion (which transactions targeting C enter the block), and interleaving (how transactions targeting different contracts are interspersed).

Exclusion-based MEV: residual. The proposer can selectively exclude transactions (subject to inclusion constraints). Ordering constraints apply to the included subset; the proposer chooses what subset to include. A proposer who dislikes the ordering constraint specified by π_C can selectively exclude specific transactions that aren’t subject to the inclusion constraints to attempt to obtain a more favorable sequence from the remaining included transactions. For example, consider a borrow/lend protocol that has an ordering constraint requiring that oracle updates precede liquidations. If there is no inclusion constraint on the oracle update, a proposer can side-step the ordering constraint by simply only including the liquidation transactions.

Cross-contract MEV: residual and open. When an MEV opportunity spans multiple contracts with different ordering policies, the attacker exploits the interaction between policies rather than the ordering within a single policy. Cross-contract MEV is not addressed by ordering guarantees and is an open problem (Section 3.4).

Ordering guarantees constrain sequencing-dependent extraction on directly-accessed adopting contracts. MEV does not disappear; it migrates from the ordering surface to the inclusion and composability surfaces [5].

3.4 Block Building Challenges and Mitigations

Single- vs. multi-contract ordering. For transactions that each target a single contract with a registered ordering policy, the ordering is determined by π_C . Block validation is straightforward: the validator checks during execution that the per-contract subsequence matches π_C , rejecting the block on any violation. When transactions interact with multiple contracts, each with its own ordering policy, the block building problem becomes combinatorially harder. Bahrani and Neuder [1] provide worked examples with diagrams illustrating each of the following challenges.

Mutual exclusion from heterogeneous policies. Consider two contracts, each with a cancel-prioritization policy (cancels before takes). A transaction T_1 performs **Cancel**(A) + **Swap**(B), and a transaction T_2 performs **Cancel**(B) + **Swap**(A). Placing T_1 before T_2 satisfies A ’s ordering (**Cancel** before **Swap**) but violates B ’s (**Swap** before **Cancel**). The reverse violates A ’s policy. Neither ordering is valid; the transactions are mutually exclusive despite arising from natural user behavior (Alice and Bob each removing stale LP positions and arbitraging the other pool) [1].

The same pattern arises across contract types. An Oracle requiring **Update** before **Read** and a DEX requiring **Cancel** before **Swap**: a transaction that does **Update** + **Swap** and a transaction that does **Read** + **Cancel** cannot coexist in the same block. With two transactions, the resolution is simple: exclude one. With N transactions touching M contracts with heterogeneous policies, determining which subset can coexist and in what order is a combinatorial search over the inclusion and ordering space.

State-dependent transaction behavior. The block building problem is harder when transaction behavior depends on state. If a transaction calls **Remove** or **Swap** depending on the current

pool price, the block builder cannot determine which function the transaction will call without executing it. Static analysis of ordering constraints is insufficient; the builder must simulate execution at each candidate position, and the result may change depending on which other transactions have already executed.

Building vs. validation asymmetry. Verifying ordering conjuncts is cheap: the validator executes the block and checks function-call ordering dynamically. Block building is hard: the proposer must find a valid ordering across all constraints before execution, which can be shown NP-hard in the general case (cf. constrained job scheduling with precedence constraints [8]). This asymmetry means ordering guarantees do not meaningfully increase validation cost but do increase block building complexity.

Bypass via wrapping. If targeting is instead determined by the outermost call target (a cheaper heuristic than the call sequence of Definition 3.2), the policy is trivially bypassable: an attacker wraps their transaction in an arbitrary attacker-controlled outermost contract (with no ordering policy), which then calls the victim contract internally. The ordering constraint of the victim contract does not apply because the outermost target is the attacker’s wrapper. This limits the enforceability of ordering guarantees to direct, non-composable interactions. Applications requiring ordering guarantees must restrict their external entry points (e.g., reverting calls from contracts without registered ordering policies).

Transaction metadata as mitigation. Block building complexity can be reduced if transactions pre-commit to the functions they will call (analogous to Ethereum’s access lists for storage slots). Transactions that call undeclared functions revert but still pay gas. This gives the block builder enough information to order transactions without executing them, at the cost of restricting transaction expressiveness: state-dependent routing (e.g., an on-chain aggregator probing multiple DEXes) cannot pre-commit to its function calls. The tradeoff between expressiveness and expected time-to-inclusion is an open design question [1].

General case. A general solution to multi-contract ordering composability would require either (a) a global constraint solver in the block construction path, (b) cooperative registration between contracts, or (c) a weaker consistency model where per-contract ordering is best-effort for cross-contract calls.

3.4.1 Finalized-State and Non-Finalized-State Ordering

The challenges above concern multi-contract composability: whether transactions touching multiple contracts can coexist under heterogeneous policies. An orthogonal question is whether ordering policies can depend on chain state. Two cases are relevant:

1. *Finalized-state ordering.* π_C and its tiebreakers τ_i depend on finalized (committed) state. Under instant deterministic finality, this is the pre-state of the current block — known before block construction begins and agreed upon by all validators. The ordering is computable deterministically; for policies expressed as Monotone Priority System priorities (below), block building is tractable, while general policies of Definition 3.1 can remain hard even when they depend only on finalized state (Section 3.4). This subsumes the fully static case (where π_C and τ_i do not read state at all) as a degenerate instance.
2. *Non-finalized-state ordering.* The ordering depends on state produced during or after the current block’s execution. This introduces a circular dependency: the ordering determines which transactions execute in what sequence, which determines the state, which determines the ordering. Resolving this circularity in the general case requires either

fixed-point computation (expensive) or the lazy evaluation pattern described below, which defers execution to the next block and converts non-finalized state into finalized state at the cost of one block of latency. Despite the complexity, this feature could be a meaningful improvement for applications. For example, a borrow/lend protocol could ensure that positions can only be liquidated if the conditions for the liquidation are sustained beyond the instantaneous state (e.g., to protect against flash-loan style malicious liquidations).

Finalized-state ordering: the Monotone Priority System. For cross-contract composability under finalized-state ordering, Durvasula [2] shows that the *Monotone Priority System*, in which each function call is assigned a global integer priority subject to the constraint that a call’s priority cannot exceed that of any call it references, is the unique system satisfying five natural axioms (existence of valid blocks, baseline per-contract priority, propagation through references, reducibility, and independence of irrelevant calls). For policies expressed in this model (global integer priorities that propagate through references, a different policy model from Definition 3.1), block building reduces to sorting by priority ($O(n \log n)$), avoiding the NP-hardness of the general case. The propagation-through-references axiom addresses the wrapping bypass identified above: ordering constraints propagate through the call graph, preventing circumvention via wrapper contracts. Because the MPS operates on finalized state, it accommodates both static priorities and state-dependent tiebreakers — any ordering computable from the committed pre-state is compatible.

Non-finalized-state ordering: lazy evaluation via inclusion guarantees. When the ordering depends on state produced during the current block’s execution, the lazy evaluation pattern breaks the circularity. User transactions targeting contract C do not execute application logic directly; they append to a buffer stored in C ’s state. An inclusion trigger [16] fires at the end of each block with predicate $P(S) = (\text{buffer}(C) \neq \emptyset)$, invoking C ’s batch-processing function as a single scheduled transaction. This scheduled transaction reads the entire buffer and the current chain state, then executes the buffered operations in any application-defined order.

This eliminates the multi-transaction ordering problem for the block builder: user transactions are trivial append operations with no meaningful ordering discretion, and the scheduled transaction is a single transaction whose internal execution order is determined by the contract, not by the proposer. The wrapping bypass does not apply (the buffer is internal contract state). NP-hardness does not arise (there is nothing to schedule across contracts; each contract processes its own buffer independently).

The tradeoff is one block of latency. User transactions are not immediately executed; they are buffered and processed in the next cycle. This is the same pattern as asynchronous message queues [1], but implemented natively through Symphony’s inclusion mechanism rather than requiring a separate batching layer. Cross-contract atomicity is also weakened: a user who wants to atomically interact with two buffered contracts cannot do so within a single block, since each contract’s buffer is processed by its own scheduled transaction.

Design space summary. The design space for ordering guarantees has two tractable points and one open problem: (1) finalized-state ordering via the Monotone Priority System [2] with optional state-dependent tiebreakers (zero latency, any ordering computable from committed state), (2) non-finalized-state ordering via buffered lazy evaluation with inclusion guarantees (one-block latency, arbitrary ordering including current-block state), and (3) direct non-finalized-state ordering in the block construction path (zero latency, circular dependency, open). Points (1) and (2) cover the majority of practical use cases.

4 Liveness

Liveness means that every correct transaction eventually makes progress.

Application-defined ordering adds a constraint to block construction. If each transaction in the block targets at most one contract with an ordering policy, it does not prevent block production: the proposer can always order transactions according to π_C (deterministic function of the transaction set and state). If transactions call several contracts with ordering policies, the proposer may have to exclude one transaction of a mutually exclusive pair (Section 3.4), and no valid block exists if inclusion triggers mandate both. The cost is $O(|T_C| \log |T_C|)$ per contract C with an ordering policy, where $|T_C|$ is the number of transactions targeting C in the block.

5 Discussion

5.1 Limitations

Multi-contract ordering composability. Symphony’s ordering policies are well-defined for single-contract transactions. The finalized-state case (orderings computable from the committed pre-state, including state-dependent tiebreakers) is resolved by adopting the Monotone Priority System [2], whose global integer priorities replace general call-sequence policies. The non-finalized-state case is resolved via buffered lazy evaluation with inclusion-triggered batch processing at the cost of one block of latency and reduced cross-contract atomicity (Section 3.4.1). Direct non-finalized-state ordering in the block construction path (zero latency, full composability) remains open.

6 Related Work

MEV (maximal extractable value) was identified by Daian et al. [5] and has driven significant protocol innovation. Proposer-Builder Separation (PBS) [4] addresses the centralization risk by separating block building from proposing, but transfers the *entire* block content discretion to the builder rather than disaggregating it. SUAVE [13] and Espresso [7] propose cross-chain MEV solutions.

Order-fairness [9, 10] constrains the sequencing rights with a single global rule. Encrypted mempools (Shutter [12]) blind the proposer’s information advantage.

Symphony’s guarantees [15] unify these as instances of a general framework. PBS transfers the full power to a single actor. Inclusion lists constrain exclusion via forward-looking, next-block validity enforcement. Aequitas [9] constrains sequencing with a single global rule. Symphony disaggregates inclusion, exclusion, and sequencing, and transfers each to structurally appropriate holders via same-block in-protocol validity enforcement.

7 Conclusion

Ordering guarantees transfer the proposer’s sequencing power to individual applications: each contract declares an ordering policy over recognized call sequences, enforced as a conjunct of the block validity function and checked by every validator. Ordering guarantees constrain sequencing-dependent extraction on directly-accessed adopting contracts; MEV does not disappear but migrates from the ordering surface to the inclusion and composability surfaces. Verifying ordering conjuncts is cheap, while block building across heterogeneous policies can be shown NP-hard in the general case. Finalized-state ordering, including state-dependent tiebreakers evaluated on committed state, is resolved by adopting the Monotone Priority System, whose

global integer priorities replace general call-sequence policies and reduce block building to sorting by priority. Non-finalized-state ordering is resolved by composing inclusion-guaranteed batch execution with a one-block-delay buffer, at the cost of one block of latency. Direct non-finalized-state ordering in the block construction path remains open.

References

- [1] Ethereum Research. Application-controlled execution: A case study on cancel prioritization. <https://ethresear.ch/t/application-controlled-execution-a-case-study-on-cancel-prioritization/23977>.
- [2] N. Durvasula. The Monotone Priority System: Foundations of contract-specific sequencing. arXiv:2601.20783, 2026.
- [3] E. Buchman, J. Kwon, Z. Milosevic. The latest gossip on BFT consensus. arXiv:1807.04938, 2018.
- [4] V. Buterin. Two-slot proposer-builder separation. Ethereum Research, <https://ethresear.ch/t/two-slot-proposer-builder-separation/10980>, 2021.
- [5] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, A. Juels. Flash Boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *Proc. IEEE Symp. Security and Privacy*, pp. 910–927, 2020.
- [6] C. Dwork, N. Lynch, L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM* 35(2):288–323, 1988.
- [7] Espresso Systems. Espresso Sequencer. 2023.
- [8] M. R. Garey, D. S. Johnson. *Computers and Intractability*. 1979.
- [9] M. Kelkar, F. Zhang, S. Goldfeder, A. Juels. Order-fairness for Byzantine consensus. In *Proc. CRYPTO*, pp. 451–480, 2020.
- [10] M. Kelkar, S. Deb, S. Long, A. Juels, S. Kannan. Themis: Fast, strong order-fairness in Byzantine consensus. IACR ePrint 2021/1465, 2021.
- [11] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22(4):299–319, 1990.
- [12] Shutter Network. 2022.
- [13] Flashbots. SUAVE. 2023.
- [14] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proc. 2019 ACM Symp. Principles of Distributed Computing (PODC)*, pp. 347–356, 2019.
- [15] Ritual. Extended External Validity. 2026.
- [16] Ritual. Inclusion and Exclusion Guarantees. 2026.
- [17] Ritual. Symphony: Execution-Aware Consensus and Proposer Disaggregation. 2026.