Superposition of Execution Models

Ritual

April 20, 2026

Abstract

Blockchain consensus protocols optimize performance (latency and throughput) under the implicit assumption that transaction execution is computationally inexpensive. This assumption, however, fails for workloads that are resource-intensive, heterogeneous, and randomized. For randomized computations under independent private randomness, replicated state machine execution fails as a correctness mechanism: honest replicas evaluating the same randomized function with independent random coins produce divergent outputs. A single state machine simultaneously supports replicated execution (for deterministic, lightweight workloads) and delegated execution (for randomized or resource-intensive workloads) over shared state.

1 Introduction

Symphony [22] is an execution-aware SMR framework organized around a single principle: *protocol-enforced guarantees*.

Blockchain state machine replication (SMR) protocols have converged on optimizing performance, namely latency and throughput. Recent work has driven finality times down and block production up: Solana's Alpenglow [11], MonadBFT [14], Simplex [12], and Minimmit [13] reduce latency; DAG-based protocols (Narwhal [6], Bullshark [16]) decouple dissemination from ordering to achieve higher throughput. The implicit assumption underlying this trajectory is that transaction execution is cheap relative to SMR.

This assumption is breaking as increasingly off-chain compute comes on-chain. Workloads that are resource-intensive (requiring GPUs, hundreds of gigabytes of memory, minutes to hours of compute), heterogeneous (different transactions requiring fundamentally different hardware), and randomized (producing different outputs on each execution) do not fit the replicated execution model on which blockchain SMR was built.

When SMR was the bottleneck, execution was an afterthought: all replicas could afford to execute all transactions because transactions were lightweight (token transfers, storage updates, simple contract calls), and the cost of reaching agreement dominated. For the workload class that motivates Symphony, this relationship inverts. Neural network inference with billions of parameters requires tens to hundreds of gigabytes of GPU memory and seconds to minutes of compute per forward pass; Monte Carlo simulations, probabilistic optimization, and other resource-intensive computations have similar profiles. Execution latency exceeds SMR latency by three to six orders of magnitude. Replicated execution of these workloads is not merely expensive: it requires every validator to possess specialized hardware, undermining the decentralization properties that replicated execution was designed to preserve.

A blockchain state machine typically assumes a single execution model: all replicas execute all transactions deterministically. Symphony replaces this assumption with a *superposition* of two execution models operating over shared state within a single block. This is not lane separation (isolated execution environments with separate state) but genuine superposition: both models read and write the same state, and their interactions are mediated by the verification lattice.

This paper makes the following contributions:

- A single state machine simultaneously supports replicated execution (for deterministic, lightweight workloads) and delegated execution (for randomized or resource-intensive workloads) over shared state (Section 3.3).
- Delegated execution follows a saga pattern [7] with no-rollback semantics and economic safety (Section 3.4).
- Liveness guarantee: every computation that is committed will eventually either settle (correct result with proofs) or expire and be rescheduled, provided at least one willing and capable executor exists in the network (Section 4).

Section 3 describes the superposition of replicated and delegated execution. Section 4 establishes safety and liveness. Section 5 discusses limitations and future work. Section 7 concludes.

2 Randomized Computation and the Limits of Replication

A *workload* is a triple $W = (\Theta, F, D)$ where:

- $\Theta \subseteq \mathbb{R}^d$ is a parameter space (finite-dimensional, possibly very large),
- $\mathcal{X}$ is the input space and $\mathcal{Y}$ is the output space,
- $F : \Theta \times \mathcal{X} \to \mathcal{D}(\mathcal{Y})$ maps parameters and input to a probability distribution over outputs,
- $D = (c_{\text{compute}}, c_{\text{memory}}, c_{\text{time}})$ is a resource descriptor.

Replicated execution assumes deterministic state transitions: given the same input and pre-state, all correct replicas compute the same post-state [15]. Two independent sources of randomness violate this assumption for the target workload class.

2.1 Hardware Non-Reproducibility

In practice, re-execution often does not yield identical results across machines. Floating-point accumulation in parallel GPU threads is non-associative: the order of reduction depends on thread scheduling and hardware-specific reduction trees [10]. Two correct replicas executing the same neural network forward pass with identical weights, input, and random seed on different GPU architectures can produce bit-different outputs—a consequence of IEEE 754 arithmetic, not a software defect. Neural network inference on GPUs exhibits this for any model with parallel reductions (attention, layer normalization). Deterministic GPU modes exist but impose severe performance penalties and are not universally available. No protocol-level coordination restores bit-exact agreement across heterogeneous validators; replication fails because the hardware cannot produce identical outputs. This necessitates delegated execution with external verification.

2.2 Algorithmic Randomness

Independently of hardware, many target workloads are intentionally randomized: they sample from distributions rather than computing deterministic functions. Neural network inference with temperature sampling, Monte Carlo simulation, and randomized optimization all map inputs to distributions over outputs. When replicas sample independently, they produce different outputs—not because any replica is faulty, but because the function is inherently randomized. Any comparison scheme that requires identical outputs across replicas will fail. This motivates delegated execution: correctness requires proving that a given output is consistent with the computation, not comparing outputs across replicas.

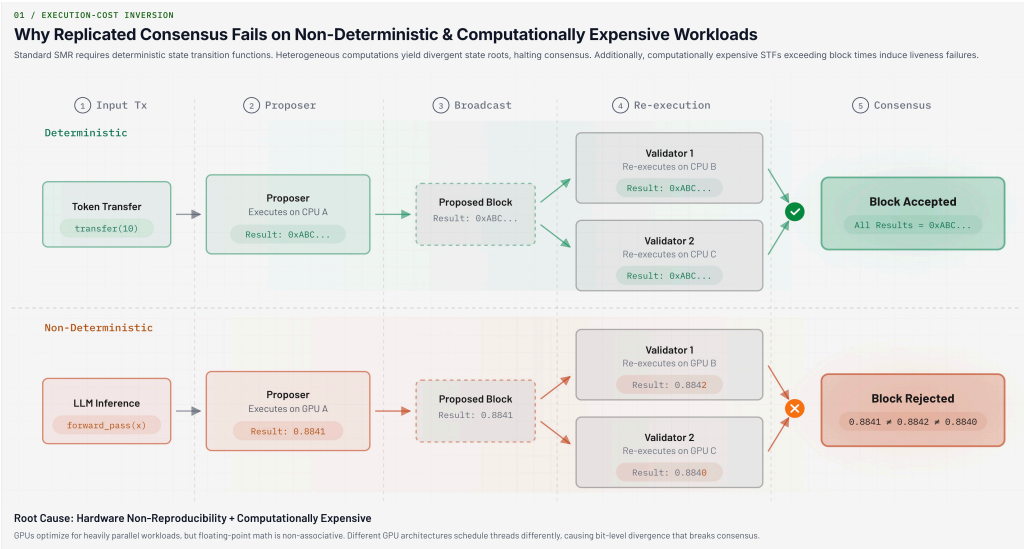

Figure 1: Execution-cost inversion. Standard SMR assumes deterministic, lightweight state transitions (top). Heterogeneous GPU workloads produce divergent state roots across replicas, breaking the replication assumption (bottom).

Definition 2.1 (Deterministic Workload)**.** *A workload W is deterministic if for all $\theta \in \Theta$ and $x \in \mathcal{X}$, the distribution $F_\theta(x)$ is a point mass: there exists a unique $y \in \mathcal{Y}$ such that* $\Pr[F_\theta(x) = y] = 1$.

Definition 2.2 (Algorithmically Randomized Workload)**.** *A workload W is algorithmically randomized if there exist $\theta \in \Theta$ and $x \in \mathcal{X}$ such that $F_\theta(x)$ has positive Shannon entropy: $H(F_\theta(x)) > 0$.*

A shared seed derived from a beacon or VRF [9] makes the computation deterministic given the seed, addressing algorithmic randomness. But this re-exposes the workload to hardware non-reproducibility (Section 2.1) and makes the output predictable from the public seed.

2.3 Combined Effect

Either source of randomness alone breaks replication. Hardware non-reproducibility is the more fundamental barrier: it is intrinsic to the execution substrate and cannot be eliminated by protocol design. Algorithmic randomness is addressable by shared seeds but at the cost of predictability and with the residual problem that the "deterministic" replay is still not bit-reproducible across hardware.

For workloads exhibiting either or both forms of randomness, correctness requires proving that a given output y is consistent with $F_\theta(x)$ rather than comparing outputs across replicas. This is the domain of verifiable computation [8, 17].

3 Superposition of Execution Models

This section describes the two execution modes, how they compose, and the saga-based mechanism for long-lived delegated execution.

3.1 Replicated Execution (Deterministic Path)

Deterministic state transition functions that fit within the block's computational budget are executed by all correct replicas in the standard SMR manner [15]. This is identical to how Ethereum, Solana, and other replicated-execution blockchains process transactions today.

Formally, a replicated transaction tx applies a deterministic function $g : \mathcal{S} \to \mathcal{S}$ to the current state:

$$S' = g(S, \text{tx})$$

Every correct replica computes g independently and arrives at the same S'. Safety follows from the determinism of g and the agreement property of the SMR protocol (Lemma 4.1).

Replicated execution handles the vast majority of blockchain transactions: token transfers, AMM swaps, governance votes, contract deployments. These are deterministic, lightweight, and well-served by the standard model. Symphony does not modify or replace this path.

3.2 Delegated Execution (Randomized / Resource-Intensive Path)

Workloads that are randomized (Definition 2.2) or that exceed the per-block computational budget are executed by a single staked executor. The remaining replicas verify proofs of the execution rather than re-executing.

A delegated transaction for workload $W = (\Theta, F, D)$ with input x proceeds as follows:

1. **Executor selection.** A staked executor E with sufficient resources (matching the resource descriptor D) is selected. Selection may be deterministic (round-robin among qualified executors) or market-based, via a two-sided matching mechanism that accounts for heterogeneous user valuations and heterogeneous node costs [2].

2. **Execution.** E evaluates $y \sim F_\theta(x)$ inside the TEE, producing an output y and an initial proof (the hardware attestation π_1). Proofs for slower proof systems $(\pi_2, \ldots, \pi_m)$ are generated asynchronously and may arrive much later.

3. **Settlement.** E submits a settlement transaction containing (y, π_1). The result enters the verification lattice at position $(1, 0, \ldots, 0)$. Subsequent proofs π_i are submitted in separate transactions as they become available, each advancing the corresponding lattice dimension.

4. **Verification.** Each proof π_i is verified independently by its assigned committee [18]. Upon committee quorum for V_i, the lattice dimension v_i advances from 0 to 1. Proofs arrive and are verified on their own timescales: hardware attestation within milliseconds of execution, ZKPs minutes to hours later.

Other replicas never execute $F_\theta(x)$. They verify proofs, which is computationally cheaper, both for TEE and ZK proofs.

The Fabric Execute-Order-Validate (EOV) architecture [1] is the closest architectural ancestor. Fabric's endorsement policies play the role of proof systems, but endorsement policies are interactive (endorsing peers must be online and responsive) and based on identity rather than cryptographic soundness. Symphony replaces endorsement policies with non-interactive proof systems [19], adds the product lattice for heterogeneous verification [19], and enshrines intra-block guarantees via external validity [20].

3.3 Superposition over Shared State

Both execution modes operate on the same state S within a single block. A block's state transition is a composition:

$$\text{STF}_{\text{block}}(S) = \text{STF}_{\text{replicated}}(\text{STF}_{\text{delegated}}(S))$$

The ordering shown (delegated settlements processed before replicated transactions) is a simplification. In practice, the block applies transactions in their sequential order within the block,

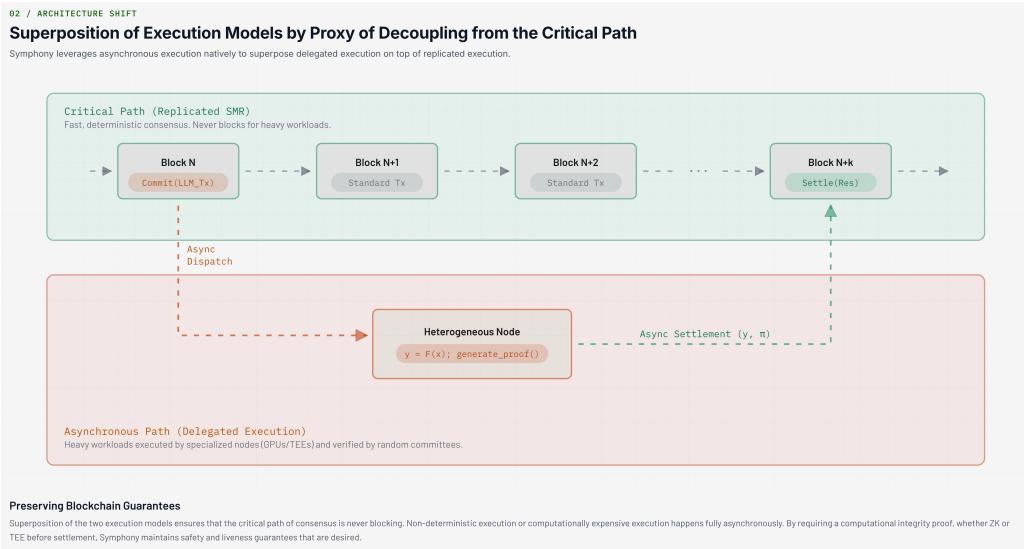

Figure 2: Superposition architecture. The critical path handles lightweight deterministic transactions via replicated execution. Resource-intensive or randomized workloads are delegated to a single executor and verified asynchronously by committees, merging back into shared state upon settlement.

regardless of type. Each transaction's type (replicated vs. delegated settlement/commit) determines which execution path processes it, but the ordering is the block's transaction sequence, not a type-based partition. The composition is uniquely determined by the block contents and the transaction sequence, which is agreed upon by SMR.

The interactions between modes are explicit and mediated by the verification lattice:

Replicated reads delegated results. A replicated transaction can read the output of a settled delegated computation, subject to the constraint that the result's lattice position meets or exceeds the application's declared safety requirement [19]. If the result has not yet reached the required lattice position, the read returns a sentinel value indicating "not yet verified at required level," and the replicated transaction can branch on this.

Replicated triggers delegated execution. A replicated transaction can register an async commit for a delegated computation. For example, a smart contract receiving a price update (replicated, deterministic) can trigger an ML-based risk assessment (delegated, resource-intensive). The trigger is mediated by the async result registry [20].

Delegated settlement triggers replicated actions. When a delegated computation settles and its lattice position reaches a specified threshold, an inclusion trigger [21] can fire a replicated transaction that consumes the result. This is the mechanism for "chaining" delegated and replicated computations.

Shared state, not shared execution. The two modes share state but not execution. Replicated transactions are executed by all correct replicas. Delegated transactions are executed by one executor and verified by committees. The state is the integration point; the execution paths are distinct. This is an unbundling of SMR from execution that is *within* a single protocol, not across protocols (as in the Celestia/rollup modular architecture [5]).

3.4 Two-Phase Delegated Execution via Saga Semantics

Delegated execution is inherently long-lived: proof generation for large workloads takes minutes to hours. The chain cannot block during this period. Symphony uses a two-phase pattern based on the saga model [7]:

Phase 1: Commit. An async commit transaction anchors the computation parameters on-chain:

- Computation identifier id.
- Parameters θ (or a hash commitment to parameters stored off-chain).
- Input x (or a hash commitment).
- Designated executor E (or an open assignment for market-based selection).
- Time-to-live TTL: maximum blocks until settlement.
- Proof system requirements: which dimensions of the lattice must be populated.

The result registry is updated: $(\mathrm{id}, \theta, x) \mapsto \mathsf{Pending}$. The executor's bond for this computation is locked.

Phase 2: Settlement. In a future block (possibly many blocks later), the executor submits a settlement transaction:

- Output y.
- Proofs $\{\pi_1, \ldots, \pi_m\}$ (some may be submitted later if proof generation is ongoing).
- Auxiliary data for verification (e.g., intermediate commitments for distributed verification).

The result status transitions to $\mathsf{Settled}$, and verification begins. The lattice position starts at $\perp = (0, \ldots, 0)$ and advances as proof systems verify.

Compensating action on failure. If the TTL expires without settlement:

1. The result status transitions to $\mathsf{Expired}$. (This status is formally defined in the extended state machine, [20], alongside Pending, Settled, and Disputed.)
2. The executor's bond is partially slashed ($\beta \cdot b$ for slashing parameter $\beta \in (0, 1]$).
3. The computation becomes eligible for rescheduling: any qualified executor can claim it.
4. Applications that had registered downstream triggers on this computation's result are notified via a sentinel state, enabling compensating actions (e.g., falling back to a cached result, reverting to a safe default, or refunding the user).

No locks. Between commit and settlement, the chain advances freely. No state is locked. Other transactions can read, write, and modify any state except the result registry entry for the pending computation. Pending results are not readable as settled values; any transaction that attempts to read a pending result receives the sentinel value.

This differs from two-phase commit (2PC) [3] in a critical way: 2PC locks resources until the coordinator decides. Sagas release resources immediately and use compensating transactions on failure. For a blockchain where state is globally visible and blocks must be produced continuously, sagas are the appropriate model.

Stale-precondition risk (TOCTOU—Time-Of-Check to Time-Of-Use). Between the async commit at block N and the settlement at block $N + k$, other transactions may alter the state assumptions on which the delegated computation relies. The executor computed $y = F_\theta(x)$ against state S_N, but settlement occurs against state S_{N+k}, which may invalidate the relevance of y. Symphony does not enforce read-set consistency between commit and settlement at the protocol level; this is a deliberate design choice that shifts atomicity responsibility to the application layer. Applications requiring state consistency across the async gap should (a) declare the relevant state slots at commit time, (b) check for state drift in the settlement transaction, and (c) trigger compensating actions if drift exceeds tolerance. The default behavior—settling without read-set verification—is safe for computations whose correctness is independent of intermediate state changes (e.g., proving properties of the model weights, which do not change during the async gap), but is a "foot-gun" for computations that depend on volatile state (e.g., computing a liquidation amount based on prices at commit time).

Downstream chaining. Computations can be chained: the output of one delegated computation serves as input to the next, or triggers a replicated transaction. Chaining is mediated by inclusion triggers: "when result R reaches lattice position $\sigma_{\min}$, include transaction T." This allows pipelines of mixed replicated and delegated execution without blocking the chain at any stage.

4 Safety and Liveness

4.1 Safety

Safety means that no correct replica accepts an invalid state transition. We decompose safety by execution mode.

4.1.1 Replicated Execution

Lemma 4.1 (Replicated Execution Safety)**.** *For deterministic state transition functions executed in the replicated path, Symphony inherits the safety of the underlying BFT SMR protocol: if fewer than $f < n/3$ replicas are Byzantine, no two correct replicas commit different state transitions for the same block height.*

Proof. Replicated execution follows standard SMR [15]. Each correct replica independently evaluates the deterministic state transition function. By the quorum intersection property ($2 \cdot 2n/3 > n$), any two quorums share at least one correct replica. A correct replica votes for a unique state transition at each height. The proof is identical to Castro and Liskov [4] (for PBFT) or the relevant proof for the underlying SMR protocol. □

4.1.2 Delegated Execution: Single Proof System

Lemma 4.2 (Single-System Delegated Safety under Full-Network Verification)**.** *For a delegated computation verified by a single non-interactive proof system V_i with soundness parameter s_i, where every correct validator independently runs* $\mathsf{Verify}(x, \pi_i)$*, the probability that an incorrect result is accepted is at most s_i.*

Proof. An incorrect result y^* is accepted only if Verify accepts an invalid proof. By the soundness definition [19], this occurs with probability $\leq s_i$. Since every correct validator independently verifies, no trust delegation is involved: the safety guarantee is purely the proof system's soundness parameter. □

For the proof systems relevant to Symphony, full-network verification is practical. TEE attestation verification is a signature check (milliseconds). SNARK verification is $O(1)$ in proof size. Neither requires specialized hardware. Every validator can and should verify every top-level proof.

4.2 Liveness

Liveness means that every correct transaction eventually makes progress.

4.2.1 Executor Liveness

A delegated computation may stall if the executor crashes or refuses to produce a result.

Definition 4.3 (Executor TTL)**.** *Each async commit includes a time-to-live TTL: the number of blocks within which the executor must submit a settlement transaction.*

If the TTL expires without settlement:

1. The computation status transitions from Pending to Expired.
2. The executor's bond for that computation is partially slashed (to compensate for wasted block space and user delay).
3. The computation is eligible for rescheduling: a new executor can claim it via a standard transaction.

Liveness guarantee: every computation that is committed will eventually either settle (correct result with proofs) or expire and be rescheduled, provided at least one willing and capable executor exists in the network.

TTL and partial synchrony. The TTL mechanism creates a tension with the partial synchrony model. Before GST, messages can be delayed arbitrarily. An honest executor who computes a correct result and submits a valid settlement transaction may have the transaction delayed past the TTL by a network partition—resulting in slashing of an honest executor. This is a synchrony assumption hidden in the TTL mechanism: the executor must be able to deliver the settlement transaction within TTL blocks of committed chain time.

Mitigation. The TTL should be set conservatively (much larger than the expected computation + proof generation time) to account for network delays. Additionally, the protocol can tie TTL expiry to committed state rather than wall time: the TTL counts committed blocks, not elapsed real time, so the executor has TTL committed blocks to submit. Since committed blocks require SMR (which resumes after GST), the TTL clock effectively pauses during partitions that prevent block production. However, if blocks continue to be produced by a partition that excludes the executor, the executor can still be slashed despite acting honestly. This is an inherent limitation of timeout-based mechanisms in partially synchronous systems.

5 Discussion

5.1 Limitations

Stale preconditions in delegated execution. The saga model (Section 3.4) does not enforce read-set consistency between async commit and settlement. State may change between commit (block N) and settlement (block $N + k$), invalidating the assumptions under which the delegated computation was performed. The protocol provides compensating actions but does

not prevent settlement based on stale preconditions by default. Applications must implement their own read-set validation in the settlement transaction. This is a deliberate design choice (avoiding locks on shared state) with a known footgun for applications that depend on volatile state during the async gap.

6 Related Work

Replicated execution (Ethereum, Solana) is the dominant model: all validators execute all transactions. Fabric's EOV [1] breaks this pattern by separating execution from ordering and validation, but uses interactive endorsement policies rather than cryptographic proofs.

Parallel execution (Sealevel/Solana, Block-STM/Aptos) optimizes throughput for deterministic workloads by executing non-conflicting transactions concurrently. This is orthogonal to Symphony: parallel execution can be applied within the replicated path of the superposition model.

Modular execution (Celestia [5] with rollups) separates data availability from execution across different protocols. Symphony's execution unbundling is *within* a single protocol: the replicated and delegated paths share state, not just data availability, and the verification lattice mediates their interaction [19].

The saga model [7] for long-lived transactions uses compensating actions instead of rollback. Two-phase commit (2PC) [3] locks resources until commit. Symphony uses sagas for delegated execution (Section 3.4): the chain does not lock state between commit and settlement, and compensating actions handle TTL expiry.

7 Conclusion

Replicated state machine execution fails for the workload class that motivates this paper. Hardware non-reproducibility across GPU architectures makes bit-exact agreement impossible for parallel floating-point computations, regardless of shared randomness. Algorithmic randomness in randomized inference compounds the problem. The transition from SMR-performance optimization to execution-aware SMR is architectural.

Delegated execution operates in superposition with replicated execution over shared state.

Sagas release resources immediately and use compensating transactions on failure. For a blockchain where state is globally visible and blocks must be produced continuously, sagas are the appropriate model.

References

[1] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, J. Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proc. 13th EuroSys Conf.*, pp. 1–15, 2018.

[2] M. Bahrani, N. Durvasula. Resonance: Transaction fees for heterogeneous computation. arXiv:2411.11789, 2024.

[3] P. A. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] M. Castro, B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd Symp. Operating Systems Design and Implementation (OSDI)*, pp. 173–186, 1999.

[5] Celestia Labs. Celestia. 2022.

[6] G. Danezis, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman. Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus. In *Proc. 17th EuroSys Conf.*, pp. 34–50, 2022.

[7] H. Garcia-Molina, K. Salem. Sagas. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pp. 249–259, 1987.

[8] R. Gennaro, C. Gentry, B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proc. CRYPTO*, pp. 465–482, 2010.

[9] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proc. 26th ACM Symp. Operating Systems Principles (SOSP)*, pp. 51–68, 2017.

[10] H. Jooybar, W. W. L. Fung, M. O'Connor, J. Devietti, T. M. Aamodt. GPUDet: A deterministic GPU architecture. In *Proc. ASPLOS*, pp. 1–12, 2013.

[11] Anza. Alpenglow: Solana consensus upgrade. 2025.

[12] B. Y. Chan, R. Pass. Simplex consensus: A simple and fast consensus protocol. IACR ePrint 2023/463, 2023.

[13] B. K. Chou, A. Lewis-Pye, P. O'Grady. Minimmit: Fast finality with even faster blocks. arXiv:2508.10862, 2025.

[14] M. M. Jalalzai, K. Babel, J. Komatovic, T. Klenze, S. Das, F. Elsheimy, M. Setrin, J. Bergschneider, B. Gilkalaye. MonadBFT: Fast, responsive, fork-resistant streamlined consensus. arXiv:2502.20692, 2025.

[15] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22(4):299–319, 1990.

[16] A. Spiegelman, N. Giridharan, A. Sonnino, L. Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *Proc. ACM SIGSAC Conf. Computer and Communications Security*, pp. 2705–2718, 2022.

[17] M. Walfish, A. J. Blumberg. Verifying computations without reexecuting them. *CACM*, 58(2), 2015.

[18] Ritual. Verifiable Computation with Distributed Proving and Verification. 2026.

[19] Ritual. Multi-System Verification Model. 2026.

[20] Ritual. Extended External Validity. 2026.

[21] Ritual. Inclusion and Exclusion Guarantees. 2026.

[22] Ritual. Symphony: Execution-Aware Consensus and Proposer Disaggregation. 2026.