Remora: Scale-out Deterministic Execution for Smart Contracts

Zhengqing Liu[1], Alberto Sonnino[2,3], Igor Zablotchi[2], Eleftherios Kokoris Kogias[2], Marios Kogias[1]
[1]Imperial College London, [2]Mysten Labs, [3]University College London

Abstract

Modern blockchains rely on a modular architecture that decouples consensus from execution. Recent advances in consensus algorithms have shifted the bottleneck to the execution layer, which must deterministically follow the consensus order and handle increasingly complex, compute-intensive smart contracts. We identify that single-node validators cannot keep up, motivating the need for a scale-out design.

We design Remora, a scale-out smart contract execution engine. Remora adopts an efficient asymmetric architecture with centralized transaction dispatching and distributed execution, and depends on an object versioning scheme with a strict ownership model to guarantee deterministic scale-out execution. Remora achieves up to 3× throughput improvement compared to state-of-the-art deterministic execution schemes, scales up to 250k TPS matching modern consensus performance, and reduces latency by up to 5*ms*. We also show that Remora elastically adapts to bursty workloads and dynamic access patterns using real-world traces. Remora's main performance benefits come from a novel stateful/stateless separation during smart contract execution, which overlaps the execution of state-independent tasks with consensus, and a new locality-aware and load-balanced scheduling scheme.

PVLDB Reference Format:
Zhengqing Liu[1], Alberto Sonnino[2,3], Igor Zablotchi[2], Eleftherios Kokoris Kogias[2], Marios Kogias[1] . Remora: Scale-out Deterministic Execution for Smart Contracts. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/remora-sys.

1 Introduction

Modern blockchains are operated by *validators*, which adopt a layered architecture (Figure 1): the *consensus* layer establishes a globally ordered sequence of transactions, and the *execution* layer performs transaction execution. While enormous progress has been made in scaling consensus, with production systems today sustaining 200k–300k transactions per second (TPS) [7, 8], the execution layer has failed to keep pace. The need for faster execution is amplified by the rise of smart contracts, which drive new demands from decentralized applications spanning finance, gaming, and identity [35, 36, 77, 81]. Thus, execution, not consensus, now defines the scalability frontier for blockchains.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1
doi:XX.XX/XXX.XX

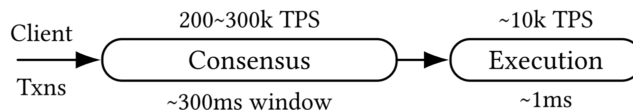

Figure 1: Validator architecture in modern blockchains.

Despite advances in exploiting multi-core parallelism on smart contract VMs [34, 65, 84, 97], *scale-up* execution alone cannot meet the computational demands of modern smart contracts. As smart contract logic and cryptographic authentication grow increasingly expensive, due to techniques like zero-knowledge proofs and post-quantum cryptography [3, 55, 94], further scaling-up becomes impractical as computational needs exceed the capacity of single-node validators. Addressing this bottleneck requires a shift toward ***scale-out*** designs that distribute validator's execution across multiple machines.

A core requirement of blockchain execution that makes the problem more challenging compared to prior work on distributed transaction processing [46, 60, 111], is ***strict determinism*** [20], i.e., preserving the total order established by the consensus layer during execution even in the presence of parallel or distributed executors. In blockchains, replicas are mutually untrusted, and transaction order often carries financial significance in applications like auctions and flash loans. Enforcing the consensus-established order is essential because it guarantees fairness, preserves transparency, and ensures that execution remains verifiable and tamper-proof [68, 87].

In this work, we ask: *how to efficiently scale-out smart contract execution while preserving strict determinism?* Revisiting prior work on deterministic transactional systems, we identify two fundamental design pitfalls. First, all prior deterministic databases adopt a *symmetric* architecture, that duplicates the full control stack, i.e., sequencing and scheduling all transactions, on every node, while only performing execution for a fraction of them [76, 78, 83, 89, 95]. This redundancy inflates the coordination overhead and leads to limited scalability. Second, there is currently no distributed transaction-scheduling scheme that jointly satisfies strict determinism, locality awareness, and load balance, properties that are necessary for efficient scale-out smart contract execution. Load-driven schemes trigger excessive state migration and coordination for distributed transactions, while locality-driven ones concentrate records on a single node [48, 66]. Recent work attempts to balance both [67, 89], but often breaks strict determinism via reordering, which is unacceptable to blockchain systems.

Beyond identifying design pitfalls in prior work, we further observe two new domain-specific opportunities by leveraging blockchain characteristics. First, smart contract execution naturally comprises two steps: a *stateless* one, performing verification and authentication, which is compute-intensive but independent of shared state; and a *stateful* one, performing business logic of smart contracts, which thus requires access to the blockchain state but tends to be lightweight. Accessing shared state introduces dependencies and ordering constraints across transactions. However, we argue

that the stateless parts have no such dependencies and can be executed independently on any available compute resources, outside the critical dependency boundaries. Second, the stability of existing consensus algorithms introduces a *window* of opportunity between transaction proposal and commitment with high predictability in proposal outcomes, which allows for speculative off-path execution to perform useful work ahead of time [18]. We pinpoint the parts of the validator execution that can be efficiently performed during the consensus window, without incurring excessive state transfers and cascading aborts in a scale-out setting.

Driven by these findings, we design Remora, a scale-out execution engine for blockchain systems that preserves strict determinism. Remora adopts an *asymmetric* architecture, with a single Coordinator managing a pool of Workers to decouple scheduling from execution. The Coordinator receives the globally ordered transaction stream from consensus and dispatches each transaction to a Worker for execution. To enforce strict determinism, Remora introduces *per-object versioning* and a *lease-based ownership* model: each versioned object is exclusively owned by either the Coordinator or a single Worker at any point in time. Each transaction is annotated with an explicit read/write set, and the Coordinator assigns object versions in consensus order. When a transaction requires objects owned by a different Worker, Remora transfers leases and object state accordingly. This design simplifies the enforcement of determinism and, by allowing dynamic flow of object ownership, enables scheduling optimizations that collocate related transactions and improve locality.

Remora decouples the execution of stateless and stateful parts, proposing *pre-consensus stateless execution and stateful scheduling* to gain both latency and throughput benefits. The separation allows compute-heavy stateless parts to harness available resources within the cluster. To handle stateful parts under contended and skewed workloads, Remora introduces a *subgraph-first scheduling (SFS)* scheme that focuses on scheduling disconnected subgraphs within the transaction dependency graph, jointly considering transaction ordering, load distribution, and data locality. To tolerate worker failures, Remora further employs periodic snapshots to persist batched updates. Remora also implements a Coordinator-driven elasticity scheme that adds or removes nodes to the cluster and depends on lazy state transfers to the new nodes.

We evaluate Remora using transactional benchmarks and real-world smart-contract traces, showing up to 3× throughput improvement compared to state-of-the-art deterministic execution scheme [67], while effectively masking the latency of both stateless execution and stateful scheduling in consensus window, thus reducing end-to-end latency by up to 5ms. Remora's asymmetric architecture also demonstrates higher efficiency than traditional deterministic databases [76, 95] as system scales, thus reducing deployment cost. Our implementation sustains over 250k TPS even in the presence of split execution and the computationally heavy SFS scheduling, matching the modern consensus throughput.

This paper makes the following contributions:

- Two blockchain-specific design opportunities (§3), i.e., decoupling stateless and stateful execution, and enabling pre-consensus stateless execution and stateful scheduling to improve performance in a scale-out setting (§4.3, §4.4).

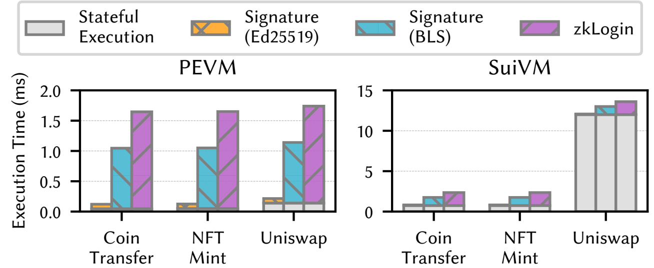

Figure 2: Execution breakdown of common smart contracts.

- An object-versioning and lease-based ownership model to preserve strict determinism in a distributed setting and simplify deterministic parallel execution in each Worker (§4.2).
- SFS, a subgraph-first scheduling scheme that jointly considers load, locality, and dependency ordering for efficient execution of smart contract workloads (§4.5).
- Remora, a scale-out execution engine for smart contracts that adopts an asymmetric architecture, improving efficiency over prior deterministic distributed systems and addressing blockchain-specific requirements (§4.1).

2 Background and Motivation

In this section, we describe the high-level architecture of modern blockchains, identify the scalability bottleneck, and motivate the need for scale-out. We then outline the unique requirements of blockchain infrastructure and explain why existing (non-) deterministic distributed transaction systems fall short of meeting them.

2.1 Blockchain Architecture and Bottlenecks

Modern blockchains typically adopt modular architectures that decouple consensus from execution [53, 57, 58]. Validators first organize transactions in blocks and propose them to the network. These blocks are then ordered by a consensus protocol before being executed deterministically by the validators' execution engine. This decoupling enables independent evolution of the two layers and allows diverse combinations of technologies. For example, both Aptos [52] and Cronos [54] adopt BlockSTM [34] for execution, despite relying on different consensus protocols.

This architectural decoupling also means that either component, consensus or execution, can independently become a bottleneck. Although early consensus protocols only scaled to a few hundreds or thousands of TPS [14], modern designs now sustain over 300k TPS by allowing validators to propose transactions in parallel and leveraging state-of-the-art BFT protocols [7, 8]. As a result, *the bottleneck has now shifted to execution.* Despite extensive parallelization efforts [34, 37, 38, 56, 69], end-to-end throughput in practice remains at tens of kTPS due to the smart contract complexity and the overhead of the underlying VMs [11].

To better understand the execution bottleneck, Figure 2 shows the average execution time of several smart contracts on two virtual machines [57, 84]. Depending on the contract logic and the verification method, the execution can take up to several milliseconds of compute time. Even under ideal assumptions with a modest execution time of 1 ms per transaction, uniform load with limited inter-transaction dependencies, Little's Law implies that sustaining 300k TPS (the throughput of state-of-the-art consensus protocols)

would require roughly 300 cores dedicated to execution. This far exceeds the capacity of typical validator hardware [90–92].

A broad line of work tackles such bottlenecks via *blockchain sharding*, where the blockchain state (and hence execution) is partitioned across shards maintained by different validator committees in parallel [5, 24, 39, 42, 50, 73, 110]. However, sharding introduces recurring challenges. First, it relies on subcommittee formation through random sampling which needs stronger adversarial assumptions to keep every shard secure with high probability [24, 50, 110]. Second, transactions that span shards incur substantial overhead due to Byzantine-resilient cross-shard coordination (often akin to atomic commit), which can dominate performance under realistic cross-shard access patterns [108], since state migration requires subcommittee to subcommittee BFT coordination. These challenges make sharding operationally complex, and recent production roadmaps decide to shift away from sharding [27].

We argue there is an alternative scale-out approach: *scale out execution within a single validator.* Rather than repartitioning state across multiple validator committees, we can focus on the execution bottleneck of a single validator and distribute the workload across a cluster of machines, while preserving the consensus protocol, validator set, and trust assumptions. This design allows a validator, which is a single administrative and trust domain, to scale its execution capacity elastically with available compute resources, narrowing the widening gap between consensus throughput and single-node execution.

> **Takeaway 1**: *The computational need of modern smart contracts goes beyond the capacity of single-node validators, making the scale-out design necessary. Scale-out intra-validator execution can retain the trust model and operational requirements of modern blockchains, unlike inter-validator sharding approaches.*

2.2 Design Requirements and Prior Work Limitations

Focusing on a scale-out architecture of a *single validator*, we enumerate the key requirements that guide such a design. To do so, we revisit related work across blockchains and distributed databases/systems to highlight where prior approaches fall short and to surface the practical considerations for real deployments.

R1: Strict Determinism. Preserving the consensus-established order, i.e., strict determinism [20], is the fundamental correctness requirement for blockchain execution. Given a globally agreed-upon transaction order, a strictly deterministic execution layer guarantees that the resulting effects are identical to those of sequential execution. This corresponds to *strictly-deterministic serializability* [20]: for an order of transactions $O = \langle T_1, \ldots, T_n \rangle$, execution effect is equivalent to sequentially executing and committing O in that order. Other prior work also formalizes the specific total order into the correctness target as known as *Byzantine Ordered Consensus* [115, 116]. This requirement sharply differentiates blockchains from conventional deterministic systems within a trusted administrative domain, where controlled reordering can be used to improve performance by reducing dependencies or object transfers [67, 71, 89].

In a blockchain setting, strict adherence to the consensus order is critical for two reasons. First, replicas are operated by mutually untrusted parties, and the established order of transactions can have direct financial consequences in order-sensitive applications (e.g., auctions, flash minting) [68, 87]. Second, blockchains are designed as permanent, decades-spanning ledgers requiring continuous public auditability. If execution engines were allowed to apply different reorderings over time, validators would need to record additional per-block metadata to reproduce the chosen order, complicating verification and weakening the replayability. Strict adherence to consensus order is thus the current industry standard.

R2: Efficiency and Cost-Effectiveness. A scale-out execution layer should convert added machines into execution throughput, not duplicated control-plane work. Calvin [95] pioneered distributed deterministic execution by enforcing a global transaction order to eliminate two-phase commit. In Calvin, every node receives the fully replicated, ordered batch of transactions, even though each node manages only a partition of the object space. To maintain determinism, nodes process only the transactions relevant to their partition, strictly following the global order. Transactions execute in-place on partitioned state, with remote reads and coordination handled during batched phases.

Calvin-style designs have been widely followed by later systems [19, 67, 76, 78, 83, 89], which keep a *symmetric* architecture: every node sequences, schedules, and executes. This preserves determinism but duplicates work, i.e., each node processes the full batch of transactions, runs the same dependency analysis, and schedules, even though it executes only a subset. The result is wasted CPU, higher coordination overhead, and higher per-node provisioning cost (full metadata/control stack on every machine). As the cluster or throughput grows, these replicated costs dominate, so scale-out shows diminishing returns (quantified in §6.2).

R3: Workload Adaptiveness. A scale-out execution layer should adapt online to workload skew and shifting hotspots, minimizing distributed transactions by favoring locality, while still balancing loads. Prior work shows that performance in distributed databases is highly sensitive to data partitioning [67, 89], as it largely determines the fraction of distributed versus single-node transactions. Although offline profiling or periodic repartitioning can improve locality [86, 89], real workloads are dynamic and hard to predict [79, 99, 113]. Even workload-driven repartitioning and live object migration [2, 23] can fall short in complex deployments, as prior work [67] highlights. Hence, recent systems combine repartitioning with on-demand migration so that objects flow across nodes according to the workload needs as part of transaction processing [48, 66, 67].

However, freely allowing object migrations across nodes can lead to further inefficiencies. On one hand, load-driven policies, which try to equalize the load among nodes, trigger frequent state transfers and “ping-pong” effects [43, 67], causing stalls and inefficiency. On the other hand, locality-driven policies, which seek to minimize object transfers, collapse active records onto a single node in the presence of skewed workloads, as shown in §6.3. Skewed workloads are the norm in modern blockchains [10, 65]. Some designs consider both locality and load, but are either non-deterministic [117] or heavily rely on reordering [67, 89], thus preventing their use in blockchain systems that require strict determinism.

R4: Fault Tolerance. Scale-out execution introduces additional failure modes: execution nodes may crash, reboot, or be temporarily unavailable. Our goal is to scale *intra-validator* execution, without interfering with the consensus layer. BFT remains provided by the blockchain's consensus across validators, and our design should not change its fault threshold or safety guarantees. Given that one validator is a single administrative and trust domain, we target *crash fault tolerance* for execution nodes. Specifically, we aim to preserve availability and avoid reducing a validator's mean-time-to-failure (MTTF) due to the increase in the number of components that can fail independently, rather than to defend against intra-validator compromise.

R5: Elastic Autoscaling. Production workloads are bursty and time-varying. Thus, a practical execution layer should be elastic to scale resources up/down and adapt placement online without disruptive global reconfiguration. Past permission systems [99–101] emphasize the need for elasticity, yet they focus on the *whole chain* and make substantial and potentially non-compatible changes to the execution, architecture, and all validators. This work instead targets *single-validator* elasticity arguing that each validator should scale its execution capacity independently without changing the consensus layer or affecting other validators.

> **Takeaway 2**: *Scale-out blockchain execution requires a careful co-design for strict determinism, architecture efficiency, workload adaptiveness, fault tolerance, and elasticity as independent design choices might lead to requirement violations.*

3 Remora Insights

Before diving into the Remora design, we highlight two crucial observations that are unique to blockchain systems and substantially influence the proposed architecture.

3.1 Stateless-Stateful Separation

We observe that smart contract execution naturally decomposes into two components: *stateless* operations, dominated by cryptographic verification of transaction authenticity, and *stateful* operations, which execute the contract logic and modify the blockchain state. This separation creates an opportunity to scale stateless work independently of stateful execution. Stateless work is completely independent of the smart contract logic or the blockchain state, imposing no ordering or placement constraints. Hence, it can be executed on any available compute resource in the validator cluster.

Figure 2 breaks down the execution time of three representative smart contracts: a simple transfer, an NFT mint, and a Uniswap [70] trade. Two smart contract VMs are used: SuiVM [57], a production-grade MoveVM powering the Sui [56] blockchain, and PEVM [84], a recent parallel EVM implementation written in Rust. We pair each stateful part with one of three primary authentication methods [55] in modern blockchains: (1) simple signatures (e.g., Ed25519 [13] or BLS [12]), (2) key-less authentication via external identity providers (e.g., zkLogin [9]), and (3) multi-signatures combining several instances of these methods. Differences in stateful execution time across VMs reflect their varying runtime overheads and optimizations—lighter VMs reduce execution costs by design, making the stateful component more efficient.

We observe that stateless compute often dominates total execution time of the smart contract: 1 ms for BLS [15, 64], 1.6 ms for zkLogin [15, 32], and up to 16 ms for multi-signatures involving ten zkLogin proofs. While simple signatures remain most common, keyless authentication is rapidly gaining adoption. We observe roughly 10k zkLogin-based transactions per day on a production blockchain validator [56]. Multi-signatures are also used, albeit less frequently, with about three transactions per minute requiring verification of multi-signatures comprising two to ten signatures or zkLogin proofs. As on-chain workloads expand to include post-quantum cryptography, the stateless portion of execution is expected to become increasingly compute-intensive [3, 36, 94].

> **Insight 1**: Smart contract execution consists of a stateless and a stateful part, with the stateless portion being increasingly compute-intensive yet runnable without ordering or placement constraints on any available compute resource.

3.2 Consensus Window

Prior work [18] has identified there is a window between the time a transaction is proposed till it is finally committed that can be leveraged for speculative execution that can accelerate the performance of the blockchain infrastructure. In state-of-the-art production-level consensus systems [8], this window can last over 300ms, which is a substantial amount of time compared to the duration of the smart contract execution.

To evaluate the opportunity, we study existing blockchain consensus algorithms and deployed systems and observe the following. Existing consensus protocols exhibit a high degree of predictability in practice: proposals are committed in the expected order in the overwhelming majority of cases. Deviations occur only when (i) a block proposal is invalid (e.g., due to faulty validators), or (ii) network asynchrony delays or reorders commitments. Empirical evidence from Sui's L1 blockchain infrastructure shows that 98.66% of blocks are committed directly in the proposed order [56]. Moreover, after a block proposal collects a quorum of votes [14] in the first phase of the protocol [8, 14, 88, 107], only extreme network asynchrony can prevent it from being committed as expected.

Leveraging the consensus window becomes considerably more challenging in the context of scale-out execution. Prior work [18, 112] exploits this window for speculative smart contract execution. This is feasible in their setting because they assume a single-node validator, which holds the entire blockchain state locally. In a scale-out setting, however, state is partitioned across nodes. Speculative execution would then require speculative object transfers across partitions to resolve dependencies. These transfers not only impose significant communication cost, but also create the risk of cascading aborts: a single misprediction in the final commit order can invalidate multiple speculative transfers and their dependent computation across nodes. The combined cost of speculative state movement and large-scale rollbacks can easily outweigh the performance gains, rendering naive scale-out execution impractical.

To still harvest the benefits of the consensus window, we identify two parts of validator execution that can be performed speculatively without requiring access to the underlying blockchain state. These are: (i) the stateless part of a smart contract and (ii) the scheduling

logic that determines which transactions each node will execute. The first requires only the transaction input, while the second relies on placement metadata and prior scheduling decisions. Crucially, neither depends on the distributed blockchain state. By only optimistically performing these state-independent components, we capture the benefits of the consensus window without costly transfers or cascading aborts.

> **Insight 2**: In a scale-out design, validator tasks that can be efficiently performed in the consensus window are stateless execution and transaction scheduling.

4 The Remora Design

We design Remora, a scale-out execution engine for a single validator, to cater to the needs of modern blockchain infrastructure. Remora should: (i) follow a modular architecture that decouples consensus from smart contract execution and preserve strict determinism; (ii) leverage stateful and stateless execution separation; (iii) allow to implement various pre- and post-consensus scheduling algorithms; (iv) enable seamless elasticity by dynamically adapting to load without upfront costs; (v) tolerate node failures within a trusted validator. We provide comprehensive formalized proofs in an extended version of this paper (https://zenodo.org/records/18436963).

4.1 High-Level Overview

Architecture. Remora adopts an *asymmetric* architecture that splits validator functionality across specialized nodes, in contrast to prior symmetric designs (§2.2) where every node redundantly performs ordering, scheduling, and execution. A single Coordinator node handles sequencing and scheduling, while a pool of Worker nodes split the object state among them and execute smart contracts. The Coordinator is the only participant in the consensus protocol, from which it receives the ordered sequence of transactions. Then, based on the scheduling logic, it dispatches transactions to the Workers for deterministic scale-out execution. The Coordinator also manages elasticity and resource allocation, persisting durably execution results, and coordinating failure recovery.

Workflows. Figure 3 gives an overview of the main components and the transaction execution workflow in Remora. The Coordinator receives transactions from clients (❶), and the consensus module runs a sequencing protocol to determine their global order according to the specific blockchain deployment. Remora separates the stateless and stateful components early: it forwards the stateful part to the scheduler for speculative planning (2a), and dispatches the stateless part directly to Workers according to the scheduling policy (2b). Both steps leverage the consensus window to overlap with consensus and reduce latency. After consensus finalizes the transaction sequence, Remora assigns object versions to each transaction (❸) to enforce a deterministic execution schedule preserving the established order. The Coordinator then routes the stateful components to Workers using the scheduler's precomputed plan. Workers execute only the transactions that the Coordinator assigns to them (❹) and communicate with their peers when remote state is needed (❺). They also periodically report load and execution state snapshots (❻), enabling autoscaling to adjust cluster size based on utilization. The persistence module in the Coordinator safely stores these execution state snapshots to ensure crash recovery for Workers, alongside the transaction logs determined by the consensus.

This asymmetric design effectively fulfills our design goals. First, the consensus layer is agnostic to Remora, making the architecture reusable across different blockchain deployments. Second, concentrating scheduling in the Coordinator avoids duplicating scheduling logic at every Worker and enables new policies without Worker changes. Third, with its global view of cluster utilization, the Coordinator can make accurate autoscaling decisions. Finally, persisting results asynchronously combined with deterministic execution allows for efficient Worker fault tolerance.

Threat Model. Remora scales out execution within a single trust domain, the validator. The cross-domain BFT guarantees are not affected by this design as they remain intact and provided by the cross-validator consensus algorithm in conjunction with the strict determinism of Remora. The Coordinator is the validator's sole externally visible interface to consensus. Thus, a Byzantine or permanent crash failure of the Coordinator is simply a Byzantine/crash failure of the validator itself (as in conventional single-node validators). Workers, in contrast, are internal execution resources under the same administrative trust domain as the Coordinator (analogous to threads/cores in a scale-up validator), so we assume they are trusted but crash-stop, i.e., they may fail by crashing, but do not behave maliciously.

4.2 Enforcing Strict Determinism

Remora assumes a widely used smart contract model [69, 75] in which contracts execute atop a flat key-value store and provides ACID transactional guarantees. Each contract declares its read and write sets upfront, an assumption shown to hold for most transactional workloads (further discussed in §7). The Coordinator acts as the global owner of all objects but delegates execution by leasing objects to Workers. Each Worker holds the leased state in memory and periodically reports updates back to the Coordinator. The Coordinator maintains metadata tracking the current leaseholder for each object. Based on the scheduling policy, the Coordinator forwards a transaction to a single Worker for execution. If all objects reside in the delegated Worker, execution runs to completion. Otherwise, the Worker initiates lease transfers and fetches remote objects from their current owners before execution.

Object Versioning. Remora's scale-out design introduces challenges in guaranteeing deterministic execution in a distributed setting, beyond the scope of single-node deterministic parallel execution [31, 34, 69]. To enforce strict determinism in a scale-out manner, Remora leverages the Coordinator as a serialization point. For each transaction in the consensus-ordered batch, the Coordinator assigns a unique version to every accessed object (read or write) by incrementing its per-object counter. This ensures that all accesses to the same object are consistent with the consensus-established order, guaranteeing strict determinism. Version assignment requires only a single linear scan over per-transaction objects within the consensus-ordered batch, incurring negligible overhead. Note that this version is validator-local and does not have to be the same across different validators. This version-based design is inspired by single-node deterministic parallel systems, such as Bohm [30], where a single thread assigns object versions before execution. Remora generalizes this principle to distributed execution.

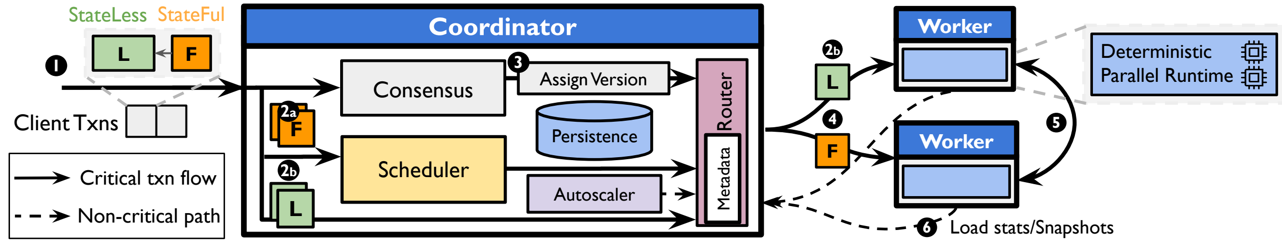

Figure 3: Remora Architecture.

Centralized version assignment simplifies the design of each Worker's local runtime, which must execute transactions in parallel while preserving determinism. When the Coordinator dispatches a transaction to a Worker, it explicitly defines the specific object versions the transaction needs to access along with their owner node. A transaction does not compute dependencies by identifying the last writer of an object; instead, its dependencies are fully determined by the assigned versions. A transaction becomes runnable once all required versions are available, either locally produced or fetched from remote workers. It observes the shared state defined by input versions, and produces the next versions for its accessed objects. Per-object version stream assigned by the Coordinator ensures that execution respects the consensus order without requiring additional coordination or locking. Furthermore, this naturally enables *inter-block* parallelism [79], as Worker execution proceeds independently of block boundaries.

Object versions have explicit lifetimes, making garbage collection trivial. The Coordinator assigns a unique version per transaction access, ensuring each version is consumed *exactly once*. Once the consuming transaction executes, the Worker safely discards the old version. A new version is created even for reads, which simplifies memory management but limits traditional read-sharing optimizations. The performance cost of this trade-off is minimal: hot objects in blockchain workloads, such as balances, order books, and NFTs, are update-heavy, and prior measurements show that write-write conflicts are far more prevalent than read-write conflicts [10].

Object Ownership and Leasing. Remora adopts a strict object ownership model: at any point in time, each object resides on a single Worker, which holds an exclusive lease to access and modify it, or the Coordinator. Static partitioning or fixed sharding performs poorly for dynamic workloads with distributed transactions [23], as discussed in §2.2. Thus, instead of periodic repartitioning as in dynamic sharding schemes [2, 86, 109], Remora transfers ownership on demand through transaction dispatch. This design avoids unnecessary object movement and adapts naturally to access patterns, a strategy adopted by several recent systems [48, 66, 67]. The Coordinator maintains per-object metadata tracking the current version and leaseholder. When scheduling a transaction to a Worker that does not own all required objects, the Coordinator annotates the request with the current owners for each object version. The receiving Worker fetches the necessary state directly from the owning nodes without Coordinator's intervention and becomes the new owner. Upon transaction dispatch, the Coordinator updates its metadata accordingly indicating the new Worker as the leaseholder.

4.3 Stateless-Stateful Separation

Remora leverages a structural property of modern smart contracts: their execution can be naturally decomposed into a *stateless* and a *stateful* part (**Insight 1**). The stateless part performs compute-heavy cryptographic verification and produces a validation token, which serves as an input to the stateful execution. Before executing the stateful part, the corresponding Worker fetches this token either locally or remotely from the Worker that ran the stateless part. If the stateless validation fails, the transaction is aborted. To maintain version order in such cases, the Worker executing the stateful part, performs a no-op that touches the relevant objects, thereby advancing their versions without applying any state changes.

This separation enables different scheduling policies for stateless and stateful parts. The router module dispatches stateless parts with a focus on *load balancing*, since they have no data dependencies. This approach can reactively fill utilization gaps when some Workers are temporarily occupied with stateful execution due to locality-aware scheduling. We describe how Remora schedules stateful parts in §4.5.

4.4 Leveraging the Consensus Window

Consensus finalization in modern blockchains often takes several hundred milliseconds (e.g., 300 ms). Rather than leaving this interval idle, Remora exploits it to hide latency and shift expensive work off the post-consensus critical path (**Insight 2**). Specifically, during this window, Remora performs stateless execution and stateful scheduling, as illustrated in Figure 3 (steps 2a and 2b).

Pre-consensus Stateless Execution. The lack of dependencies in the stateless part allows it to be executed in parallel to consensus. This overlap leverages the consensus window to reduce end-to-end transaction execution latency. While this can lead to wasted computation if a block is eventually rejected, such occurrences are rare, thus will have minimal impact on performance or utilization.

Pre-consensus Stateful Scheduling. In addition to stateless execution, Remora also runs its stateful scheduling logic during the consensus window. The Coordinator forwards each proposed block simultaneously to the consensus and scheduling modules. While consensus runs, the scheduler analyzes transaction dependencies, and devises a scheduling plan. Overlapping this with consensus allows Remora to implement even more computationally expensive scheduling algorithms without affecting end-to-end latency and overall throughput. To guarantee correctness, the router dispatches the transactions via pre-computed plan only after consensus finalizes. In the rare occasion that consensus rejects the block, the pre-consensus stateful routing plan is invalidated and dropped. To handle this case, Remora maintains two versions of metadata: one

Algorithm 1: SFS (Subgraph-First Scheduling)

Input: Batch B (consensus order), per-worker ownership sets $\{O_i\}$, loads $\{P_i\}$

Output: Assignment of subgraphs to workers

1 Build dependency graph $D(B)$ and extract subgraphs $\mathcal{G}$

2 **foreach** *subgraph* $G \in \mathcal{G}$ **do**

3 $\quad K \leftarrow \bigcup_{t \in G} \mathrm{RW}(t)$

4 $\quad$ **foreach** *worker* i **do**

5 $\quad\quad R_i \leftarrow |K \cap O_i| / |K|$

6 $\quad\quad L_i \leftarrow 1 - P_i / \max_j P_j$

7 $\quad\quad S_i \leftarrow 0.5 * R_i + 0.5 * L_i$

8 $\quad i^\star \leftarrow \arg\max_i S_i$ (break ties randomly)

9 $\quad$ Assign G to $i^\star$; update $O_{i^\star}$

for the scheduler and one for the router. The router updates its version when dispatching transactions. If a block is dropped, the scheduler pauses, synchronizes the two metadata versions, discards intermediate schedules, and then resumes.

We intentionally avoid pre-consensus stateful execution. Unlike the stateless part, stateful execution is less compute-intensive and tightly coupled with shared state dependencies. Speculating on it would largely complicate system design triggering speculative object transfers and cascading aborts in the distributed setting while yielding marginal latency benefits.

4.5 Subgraph-First Scheduling (SFS)

Now we describe how Remora efficiently schedules stateful transactions among workers. As discussed in §2, an effective scheduling policy must simultaneously satisfy three requirements: strict determinism, locality awareness, and load balance. Existing approaches fall short of meeting all three at once. For uniform workloads, a naive load balancing policy (e.g., random) would suffice. The challenge lies in handling skewed workloads with hot objects. Leveraging the consensus window, we design a new policy, subgraph-first scheduling (SFS), that achieves all three objectives.

Our key insight is to leverage **subgraphs** as the unit of scheduling. For each batch of transactions from the consensus output, Remora constructs a dependency graph (vertices are transactions and edges capture immediate dependencies induced by shared objects), identifies disjoint subgraphs, and dispatches those to the Workers. The intuition is that transactions within a subgraph already share objects and ordering constraints. Thus, collocating them on the same Worker avoids cross-node coordination. In practice, subgraphs typically arise from contention on hot objects, making them natural candidates for collocation.

Remora avoids further partitioning such subgraphs [79], because it would not gain performance while introducing overhead. For extremely contended workloads, distributed stateful execution yields no benefits since the maximum concurrency is limited by the workload itself. Therefore, the contribution of this scheduling scheme is on low to medium level of contention. This idea aligns with prior work in distributed transactions: skewed, contended workloads benefit from single-node execution, while uniform workloads distribute effectively across nodes [60].

Algorithm 1 summarizes SFS. For each consensus batch, the Coordinator builds the dependency graph and schedules each disconnected subgraph G as a unit. It first computes the subgraph's object footprint K as the union of the read/write sets $\mathrm{RW}(t)$ of all transactions $t \in G$. Then, for each Worker i, SFS computes a *locality* score R_i as the fraction of K already owned by i (using the ownership map O_i), and a *load* score L_i from i's current load P_i. The final scheduling score S_i is the equal-weighted combination of locality and load (line 7). This jointly optimizes both, i.e., locality score reduces state movement and cross-Worker coordination, while load score prevents persistent skew and state accumulation on a single Worker [66, 67]. SFS assigns the entire subgraph to the Worker with the highest score and updates the ownership metadata accordingly. Stateless work is dispatched purely by load, which helps utilize idle Workers even when locality concentrates some stateful subgraphs.

4.6 Handling Failures

Remora scales out single-validator execution and adopts the same failure model as existing blockchains that allows for validator crashes. In Remora, we treat the failure of Coordinator, the sole externally visible interface to consensus, equivalent to a single-validator failure. However, to avoid reducing the validator's MTTF due to the distributed architecture, we carefully design Remora to tolerate Worker failures.

Periodic snapshotting. Remora implements *periodic snapshotting* to keep the Coordinator loosely but consistently synchronized with the execution state in the Workers. Snapshotting operates in *epochs*. The Coordinator has full flexibility in determining the epoch size, choosing boundaries that either align with consensus rounds or adapt to workload characteristics, and explicitly notifies the Workers when an epoch ends. This exposes a tunable trade-off: longer epochs reduce snapshot overhead but increase recovery cost, while shorter epochs inverts this trade-off. Workers keep track of the objects they modify within an epoch and once the epoch finishes, they send the Coordinator an *incremental* update containing only the latest versions of those modified objects. This incremental checkpointing on Workers amortizes frequent updates to hot objects without communicating every versioned update. With its global view of the system, the Coordinator knows exactly which object versions it needs at the end of each epoch. After collecting updates from all Workers, it atomically updates its state and advances `persist_index`, which marks the log position up to which all preceding transaction effects have been durably persisted.

Failure recovery. When a Worker fails, the Coordinator initiates recovery as follows. It first spawns a new Worker and determines the transaction *replay set*, defined as the range between the current dispatch index and the `persist_index`. From the log view, state up to `persist_index` is durable at the Coordinator. So, replaying after that point is required to reconstruct the parts of the missing suffix. The Coordinator reclaims ownership of all objects previously assigned to the failed Worker that are already durably persisted, and then dispatches the entire replay set to the new Worker. During replay, the Coordinator attaches required state that it has durably persisted, avoiding extra round-trip fetches. This procedure brings the new Worker online lazily, driven by transaction demand, to reduce stalls of excessive state transfers. After dispatching the replay

set, the Coordinator can continue processing incoming transactions. We conservatively replay the entire replay set rather than attempting selective replay, as transactions may have been partially executed by either healthy or failed Workers and failures can occur at arbitrary points during execution. Notably, our design allows for skipping re-running all stateless parts as long as the Coordinator receives acknowledgement upon successful validation, which we leave for future work.

4.7 Elasticity

Remora supports seamless elasticity by automatically scaling the Worker pool in response to workload fluctuations. The Coordinator orchestrates resource allocation by periodically monitoring per-Worker load and deciding when to spawn or retire Workers. Crucially, autoscaling should not stall foreground execution or introduce any explicit rebalancing phase: Remora leverages its lease-based ownership model to migrate objects across Workers and Coordinator dynamically and lazily.

Scale-out. The Coordinator spawns a new Worker and begins dispatching transactions to it. Initially, the new Worker has no local state. As it executes transactions, it fetches any objects it does not own from other Workers or from the Coordinator. Over time, state naturally migrates to the new Worker as leases are transferred during normal execution. Note that a new Worker can immediately start helping with stateless execution.

Scale-in. The Coordinator retires a Worker *gracefully* in two steps. First, it stops assigning new transactions to the retiring Worker, but keeps it online for a grace period to serve in-flight requests and inter-Worker lease transfers. Second, to complete retirement and reclaim ownership atomically, the Coordinator terminates the current snapshotting epoch for that Worker, forcing it to report all objects modified in that epoch back to the Coordinator. This yields an epoch-aligned handoff: hot objects migrate away naturally as their leases are transferred to other Workers, while cold objects reclaimed by the Coordinator via the epoch snapshot where future transactions fetch them directly from the Coordinator.

5 Implementation

We implemented Remora in 13k LoC of Rust. This includes a modular implementation of the Coordinator and Workers. We use tokio [93] for asynchronous network IO across nodes which communicate over TCP sockets.

Coordinator. Each module runs as a long-lived task pinned to a dedicated core. Modules on the critical transaction path (Figure 3) are connected by bounded channels, forming a pipeline. The router module spawns lightweight dispatching tasks to forward transactions according to the scheduling policy for stateful and stateless parts. These tasks run to completion in parallel on a thread pool, occupying all remaining cores, with one thread pinned per core. The autoscaler module periodically monitors the incoming load rate. Coordinator detects Worker failures by observing lost network connections and triggers the failure handling scheme (§4.6).

The Coordinator maintains metadata for each object and each node in the cluster. The object metadata take the form of a map from object names to a tuple of `(current_version, current_owner)`. The node metadata maintain per-Worker load information, which the stateless and stateful routing leverage to make load-aware decisions. The metadata footprint is modest: for 10M objects, the size of ownership metadata totals around 100 MB. This is in line with production chains; for example, although hundreds of millions of addresses have appeared on Ethereum, daily active addresses peak around 1.4M [28].

Worker. Each Worker node in Remora runs a deterministic parallel runtime that handles both stateless and stateful compute. The runtime implements a custom asynchronous thread pool to drive the task execution. The design is agnostic to the underlying smart contract VM and focuses on parallel execution with strict ordering guarantees. Each execution unit, stateless or stateful, is an asynchronous task. Stateless tasks are immediately runnable and can run in parallel. Stateful tasks depend on (i) their corresponding stateless tasks (e.g., authentication) and (ii) any previous tasks accessing the same objects. The runtime uses a dynamic DAG to keep track of the dependencies among tasks based on the specific object versions they access as assigned by the Coordinator. A certain task can only be executed when all its prior dependencies are satisfied.

The intra-Worker runtime implements the task DAG via `tokio::sync::Notify` [21], a lightweight primitive for synchronization. `Notify` carries no payload and is used purely for readiness signaling. Concretely, the runtime maintains one `Notify` instance per object version. When a transaction arrives with its required versions explicitly annotated, the runtime spawns an asynchronous task. This task (1) `await`s on the `Notify` instances for all required input versions, ensuring all dependencies are satisfied; (2) once the dependencies become available executes the transaction to completion on the thread pool; (3) signals the `Notify` instances corresponding to the versions it produces, thereby unblocking dependent tasks. Remote dependencies work similarly, as when the Worker fetches the object, it signals the equivalent `Notify` instance. Finally, since each version is consumed exactly once, the runtime garbage-collects the corresponding `Notify` entry and the object version immediately after it has been consumed. `Notify` objects have a unique name within the Worker and the runtime creates them atomically the first time it processes the producer or the consumer task. This design naturally integrates dependency enforcement into the asynchronous runtime, enabling deterministic parallel execution without any invasive changes to the existing scheduler, i.e., Tokio's in the current implementation.

6 Evaluation

Our evaluation aims to answer the following questions:

- Does Remora's *asymmetric* architecture improve efficiency? (§6.2)
- How does SFS compare against other distributed (non-)deterministic scheduling schemes? (§6.3)
- How much does the stateless-stateful separation improve performance? (§6.4)
- What is the benefit of leveraging the consensus window? (§6.5)
- How efficiently does Remora recover from Worker failures? (§6.6)
- How does Remora autoscale under varying loads? (§6.7)
- Can Remora handle realistic dynamic workload hotspots? (§6.8)
- How scalable is Remora's Coordinator? (§6.9)

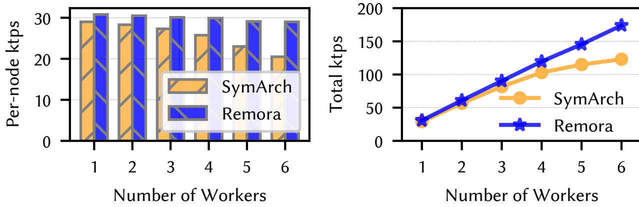

Figure 4: Comparison with other work using symmetric architecture.

6.1 Experimental Setup

Testbed. We run our experiments on an 8-node cluster on AWS, using `m5d.8xlarge` instances in the same region. One node serves as the client node which generates transactions based on an open-loop Poisson process, one node serves as the Coordinator, and the others as Workers. Each machine is equipped with a 10Gbps NIC, 32 vCPUs (16 physical cores), and 128GB memory, which aligns with the recommended hardware configuration for typical validators [90–92]. In our experiments, given that we focus on a single validator, we implement a mock consensus module that adds a constant delay of 300*ms* equivalent to the consensus execution [8].

Workloads. Following the evaluation methodology adopted in related work [20, 67, 89], we evaluate our system using the YCSB [22] and TPC-C benchmarks [1], and traces from Ethereum mainnet [84]. We implement a YCSB-style smart contract with a global mapping that represents a key value store over a 10M keyspace. To evaluate performance under contention and skew, we vary the number of keys accessed per transaction and consider both uniform and Zipfian access distributions. We also implement a TPC-C-like smart contract that supports New-Order and Payment transactions, which account for 88% of the workload and contain remote data access [66], and configure 40 warehouses per Worker. To factor out VM-specific effects (as Remora is a generic scale-out system agnostic to the choice of smart contract VM), we model execution as a stateless stage plus a stateful stage. Unless otherwise specified, YCSB microbenchmarks use 0.5 ms synthetic time for both stateless and stateful stage, consistent with §2 and prior measurements on common smart contracts [97]. For TPC-C, the stateful stage runs the real business logic, while the stateless stage uses the same synthetic 0.5 ms cost. We mainly use TPC-C benchmark in evaluating scheduling schemes (§6.3) and emulating dynamic hotspots (§6.8). We further demonstrate Remora's practicality on SuiVM [57], a production-grade MoveVM, on real-world traces.

6.2 Architectural Efficiency

Methodology. We start by comparing Remora's *asymmetric* architecture with a single node in charge of scheduling to prior Calvin-inspired [95] *symmetric* schemes [67, 76, 78, 83, 89] where every node accepts and deterministically schedules all transactions. To evaluate the efficiency of these two choices, we implement a baseline, `SymArch`, where each Worker performs both deterministic scheduling and execution to represent this line of work. Since sequencing in our setting is established via consensus, we exclude sequencing from both systems to ensure a fair comparison. We use the YCSB benchmark with uniform distribution and a round-robin scheduling policy instead of SFS, while disabling the stateful-stateless separation. In this configuration, the ideal per-Worker throughput is about 32k TPS, given 32 vCPUs and a 1 ms per-transaction service time. We vary the number of Workers and measure the maximum throughput.

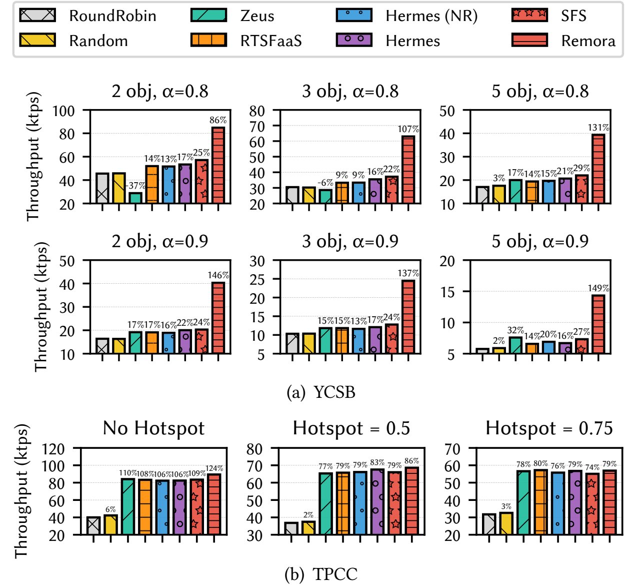

Figure 5: Performance comparison with other policies. Percentage denotes the improvement over round-robin policy.

Results. Figure 4 shows that Remora delivers higher throughput per deployed node than `SymArch` as the number of Workers increases. In `SymArch`, every Worker repeatedly pays the cost of deterministic scheduling, consuming CPU cycles that could otherwise execute transactions. This redundant overhead directly reduces the useful work obtained from each provisioned machine. In contrast, Remora offloads all scheduling to the Coordinator, allowing Workers to dedicate their full capacity to transaction execution, improving resource efficiency and translating the same hardware budget into more throughput. Although Remora requires an additional coordination node, the net deployment cost is still lower for a target throughput: for example, a 4-worker Remora configuration already outperforms a 5-worker `SymArch`. We expect the efficiency gap to widen further with more advanced scheduling policies that are more computationally expensive compared to round robin.

6.3 Scheduling Schemes Performance

Methodology. We compare SFS against several state-of-the-art scheduling schemes, including both deterministic and non-deterministic designs by measuring the max achieved throughput. **Hermes**[67] represents the state-of-the-art for deterministic scheduling. It considers both load and locality but violates strict determinism, since it relies on transaction reordering to improve performance. We also implement a strictly-deterministic variant, **Hermes (NR)**, that disables reordering in Hermes. **RTSFaaS** [117] is another scheme that considers both locality and load balance initially designed for FaaS workloads, yet is not deterministic. **Zeus** [48] is a locality-only baseline that aggressively migrates objects to the Worker executing each distributed transaction. We also include simple load-balancing policies: RoundRobin and Random. Notably,

under uniform or low-skew workloads, all policies achieve similar performance as transactions are evenly distributed and require minimal object migration. This experiment thus focuses on contended workloads. For an apples-to-apples comparison of scheduling policies, we disable Remora's use of the consensus window and the stateful-stateless separation. We use a relatively smaller pool of three Workers given that the workloads have limited parallelism. We also keep this 3-Worker configuration for the remaining experiments to focus on mechanism-level comparisons (stateless-stateful separation, consensus-window, recovery, etc.) that are not qualitatively affected by adding more Workers.

YCSB. As shown in Figure 5, naive policies perform the worst due to frequent distributed transactions and object transfers. Zeus ignores load imbalance and gradually migrates all objects to a single Worker. After warm up, only one Worker is busy while others remain idle, which is consistent with prior observations [67]. However, under high contention (3 or 5 objects with $\alpha = 0.9$) in which the concurrency level in the workload is extremely limited , the benefits of scale-out diminish and single-node execution outperforms others. RTSFaaS and Hermes (NR) achieve similar performance (9% to 15% improvement over round-robin) by considering both load and locality. The full Hermes policy benefits further from reordering up to 21%, yielding the best baseline performance.

SFS consistently outperforms all baselines (up to 29%), including non-deterministic ones, across all contended configurations. Whereas Hermes and RTSFaaS balance load by migrating individual transactions, SFS schedules at the subgraph level, colocating dependent and locality-sharing transactions. This granularity is dependency-aware and minimizes remote state transfer, confirming that subgraphs are the right unit for scale-out scheduling. We also include Remora, which is the SFS policy with the optimizations of stateful-stateless separation and consensus window execution.

TPCC. For TPC-C benchmarks, we pre-partition warehouses among Workers, and construct the hotspot by setting 50% and 75% of transactions targeting at the warehouses pre-assigned on the same Worker. SFS does not yield noticeable throughput gains over the baselines on TPC-C. This is expected because TPC-C uses warehouse-based partitioning by design, and most transaction state accesses are by the home warehouse/district while cross-warehouse accesses are relatively limited (default 15% in NewOrder and Payment transactions). Consequently, under TPC-C the bottleneck is dominated by warehouse-level load skew rather than distributed execution, so different locality-aware policies converge in performance. Our observation is consistent with prior work [67, 89].

6.4 Impact of Stateless-Stateful Separation

Methodology. We study the benefits of separating the stateless from the stateful part of a transaction in the achieved throughput. We consider two separation modes: (i) *Worker-based Sep*, where only Workers apply stateless–stateful separation during execution. This configuration could even be applied in single-node runtimes or distributed runtimes without centralized scheduling. (ii) *Coordinator-based Sep*, in which the Coordinator can schedule stateless and stateful parts on different Workers. We use the SFS scheduling policy and the YCSB benchmark to study such impact.

Results. Figure 6 shows that *Worker-based Sep* improves throughput by up to 2×, as separating execution at Workers unlocks greater

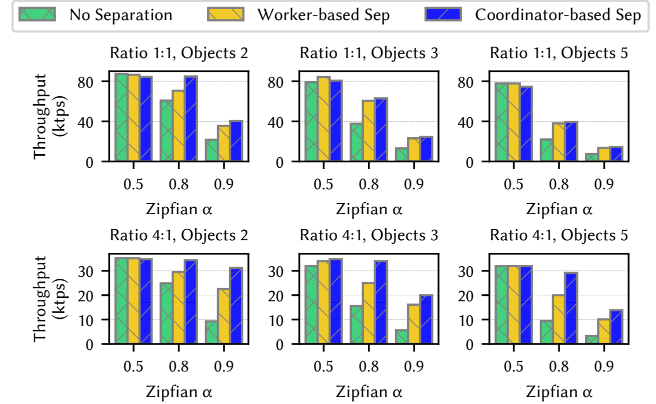

Figure 6: Benefits of stateless-stateful separation. Ratio denotes the duration ratio of stateless and stateful part in each transaction, in which stateful part always takes 0.5ms.

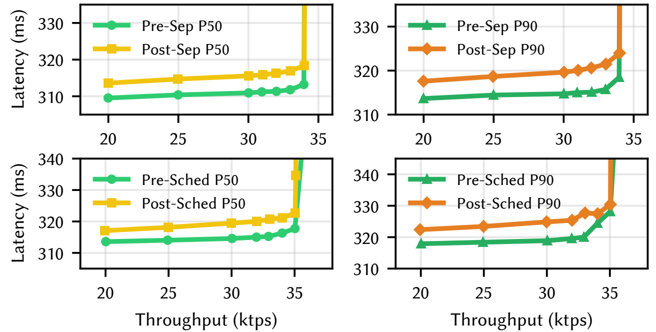

Figure 7: Benefits of consensus window. *Pre-Sep* denotes pre-consensus stateless execution, while *Post-Sep* denotes post-consensus stateless execution; *Pre-Sched* denotes pre-consensus scheduling, while *Post-Sched* denotes post-consensus scheduling.

parallelism across available cores. *Coordinator-based Sep* achieves up to 3× improvement, with the most pronounced advantage when the stateless dominates computation. These results highlight that under skewed workloads, separation at Coordinator effectively harnesses otherwise idle compute capacity across Workers. In contrast, under low contention the benefits diminish, as load is balanced.

6.5 Benefits of Consensus Window

Methodology. Remora leverages the consensus window to perform stateless execution and stateful scheduling. We study the effect of this design choice on latency. Given that the benefits are independent of scheduling schemes, we use the uniform YCSB benchmark, 2ms of stateless, and 0.5ms of stateful processing, and SFS scheduling. As a baseline, we disable pre-consensus scheduling and decoupled stateless execution and measure the end-to-end latency with achieved throughput.

Pre-consensus stateless execution. In Figure 7, *Pre-Sep* demonstrates clear latency improvement by overlapping stateless execution with the consensus window. *Post-Sep* incurs an additional 2~5ms at both P50 and P90 latency. Given that the stateless portion must always be executed and is fully parallelizable, the performance gain from pre-consensus execution is guaranteed, and becomes even more pronounced as the stateless workload grows.

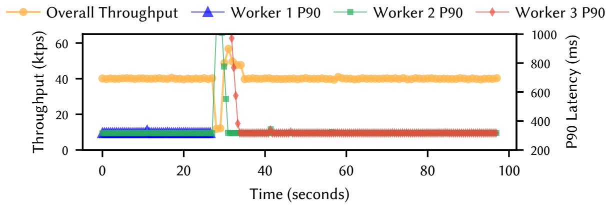

Figure 8: Failure recovery. Worker 1 failed and Worker 3 is spawned.

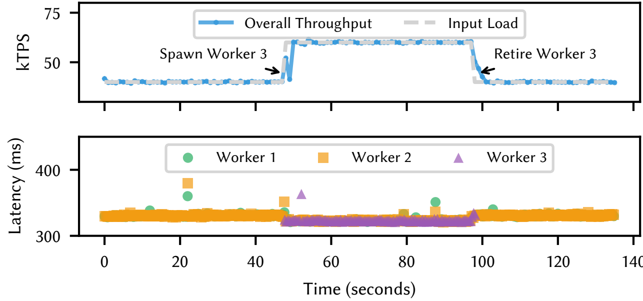

Figure 9: Elastic scaling for load fluctuations.

Pre-consensus stateful scheduling. A similar effect arises when scheduling is shifted into the consensus window. To isolate this effect, we keep stateless execution in post-consensus for this experiment. Although our scheduling policy is lightweight (~4ms per batch), latency reduction of *Pre-Sched* indicates that the consensus window provides sufficient slack to accommodate more sophisticated scheduling strategies off-path, e.g., ML-based ones.

6.6 Failure Recovery

We evaluate Remora's failure recovery by subjecting it to a constant YCSB uniform workload while injecting a Worker failure at a random time. Figure 8 illustrates the recovery process. Initially, Worker 1 and Worker 2 jointly handle the incoming requests. Upon detecting the failure of Worker 1 at $t = 28s$, Remora immediately switches on Worker 3 as a replacement. We omit the cost of provisioning an AWS EC2 VM and instead pre-launch an idle Worker before the experiment. During recovery, the system experiences a brief throughput degradation of approximately 2 seconds while it determines the replay set, transfers object ownership, and dispatches the replay set to Worker 3. Subsequently, Worker 2 and Worker 3 enter a catch-up phase, processing requests at their maximum throughput capacity, which temporarily elevates latency. Once the backlog is cleared, the performance stabilize at the pre-failure levels, demonstrating Remora's resilience to Worker failures.

6.7 Elasticity

We study Remora's elasticity capabilities by applying load that changes over time via uniform YCSB workloads. Figure 9 shows the input load pattern: throughput increases sharply from 40k TPS to 60k TPS and then decreases to 40k TPS. Once the autoscaler detects the load spike which exceeds the processing capacity of existing Workers, it starts a new Worker instance (pre-launched as same as the failure experiment). The Coordinator immediately begins forwarding load to the new Worker, which incrementally fetches required state from existing nodes on demand. When the load drops to 40k TPS, the autoscaler initiates scale-in and gracefully retires

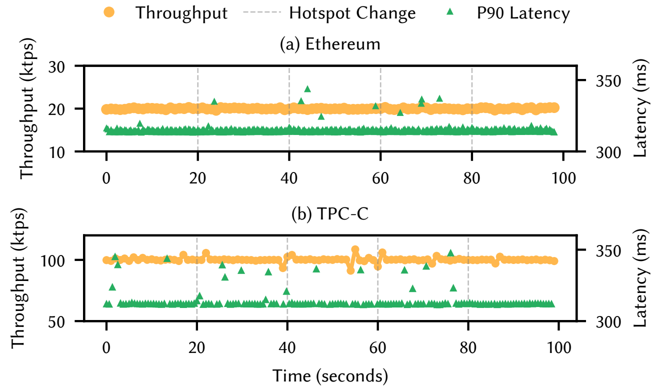

Figure 10: Dynamic hotspot workloads.

Table 1: Scalability of the Coordinator.

Policy	Round robin	Zeus	RTS FaaS	SFS	Remora
Max TPS	380k	360k	354k	348k	251k

Worker 3. Across both transitions, latency remains low and stable, showing that Remora can autoscale without stalling foreground execution or triggering an explicit rebalancing phase.

6.8 Handling Dynamic Hotspots

We evaluate Remora's ability to handle dynamic hotspots, i.e., object popularity changes, via both real-world traces from the Ethereum mainnet [84] and TPC-C benchmarks. For the Ethereum traces, we derive object access distributions from different blocks spanning different periods of time to capture realistic contention. We run this workload on a production-grade smart contract VM, SuiVM [57]. For the TPC-C experiment, we target 50% of the transactions to warehouses on a single Worker node to emulate hotspots. The workload changes every 20s, causing hotspots to shift dynamically across Workers and mimicking temporal locality observed in practice. We set the input rate to 80% of the measured system capacity for each workload (accounting for available parallelism and VM overhead) based on offline profiling. Remora shows only occasional high-latency outliers (Figure 10).

6.9 Scalability

While Remora's asymmetric architecture enables near-linear scale-out with the number of Workers (§6.2), centralizing transaction dispatch at the Coordinator raises the concern that it could become a throughput bottleneck. We therefore stress-test the Coordinator under a range of scheduling policies. To isolate dispatching overhead, we use uniform distribution with zero service time, ensuring that Workers do not limit throughput. As shown in Table 1, with a simple round-robin policy the system sustains nearly 380k TPS. Adding computation for locality calculation and dependency analysis reduces throughput only slightly, to around 350k TPS. Separating stateless and stateful execution reduces throughput to 250k TPS due to the extra message serialization, I/O, and syscall overhead. This overhead is largely an engineering artifact: syscall batching and kernel-bypass or specialized network stacks could mitigate it, which we leave for future work. We also observe all Coordinator cores are saturated, indicating that larger EC2 instances would yield

proportionally higher throughput, as the network communication tasks are parallelizable. Given that state-of-the-art consensus modules achieve 200~300k TPS [7, 8], we conclude that the Coordinator in our design is sufficient for practical deployment.

7 Discussion

Limitation: read/write set assumption. Remora assumes that each smart contract pre-declares its read/write set. This assumption is widely adopted in deterministic database systems [30, 31, 40, 69, 76, 78, 80, 83, 95], blockchain systems [4, 19, 79], and has been shown to hold for a majority of transactional workloads in a recent study [75]. While this restricts expressiveness, it aligns with most common smart contracts, including token transfers, decentralized exchanges, NFT mints, and lending protocols. Several prior systems [33, 59, 71] overcome this limitation while introducing violation to strict determinism. Recent work [6, 29] further shows that access specifications can benefit optimistic execution by reducing aborts. This constraint simplifies our enforcement of determinism and enables efficient scheduling with locality awareness.

Scaling L1 on-chain compute. Emerging on-chain workloads (AI inference, ZK proofs, and post-quantum crypto [3, 25, 55, 98]), are compute-heavy and currently pushed to L2s or off-chain due to incapability of current L1 infrastructure. Remora removes this reliance via elastic scale-out, improving L1 performance while avoiding fragmentation and enabling new L1-native use cases.

DoS resilience. Stateless validation often involves expensive cryptographic checks, making it a common vector for DoS attacks. Remora mitigates this risk by elastically scaling stateless execution across nodes, absorbing heavy or adversarial workloads without introducing coordination bottlenecks. This elasticity ensures system responsiveness and robustness even under targeted load. As blockchains adopt quantum-safe cryptography and other compute-intensive primitives, such resilience becomes increasingly essential.

8 Related Work

Deterministic Distributed Transactions. Most scale-out deterministic transactional systems adopt Calvin [95]'s *symmetric* architecture, unlike Remora's asymmetric design. Some focus on improving the sequencing layer, while others optimize the partitioning and scheduling scheme. SLOG [83] and Detock [76] decentralize the sequencer to support geo-distributed deployments. Caerus [40] minimizes the round-trip traverse by deterministically merging per-region partial sequences using a new ordering protocol. Q-Store [78] bypasses the single-threaded sequencing and scheduling parts via constructing fine-grained execution queues.

Deterministic Single-Node Transactions. Deterministic parallelism has been widely studied in databases and systems. A broad line of work follows a pessimistic *schedule-then-execute* pattern, with mechanisms spanning dependency-graph analysis [31, 41, 51, 69, 71], multi-versioning [30, 80], and per-object queues [16]. Remora similarly decouples scheduling from execution. Other work adopts optimistic *execute-then-validate* schemes [47, 71], which avoid pre-declared read/write sets but can incur aborts under contention and relax strict adherence to the agreed-upon order.

Non-Deterministic Distributed Transactions. This line of work relies on variants of two-phase commits (2PC) to coordinate multi-partition transactions. Some accelerate 2PC using new hardware technologies such as RDMA [26, 111] and CXL [44, 45] to reduce communication costs. Eris [62] exploits programmable switches for transaction sequencing. Centralized transaction routing with ownership-based data migration is also adopted in several shared-storage databases [43, 61]. GaussDB [61] employs a centralized routing model performing inference while deploying training elsewhere to avoid the on-path overhead. Our proposed consensus window can allow such computationally expensive and ML-based routing schemes. Pegasus [63] uses an in-network directory to track objects location, similar to how Coordinator tracks ownership. Lotus [118] optimizes multi-partition transactions by introducing granule locks to mitigate distributed commit overhead. Besides, epoch-based commit/recovery and asynchronous persistence in prior work [72, 114] are similar to our design.

Smart Contract Execution. Existing efforts primarily extract multi-core parallelism for deterministic execution. Recent work targets finer-grained operation-level optimizations to mitigate rollback overhead under optimistic concurrency control [20, 33, 65, 96, 102]. OptME [85] builds key-based dependency graphs to derive parallel execution schedules while extensively using re-ordering. Some other work leverages new programming semantics to enhance performance, including commutativity [82, 97] and deferred objects [74]. Spectrum [20] carefully designs rollback and re-execution mechanisms to support unknown read/write sets while preserving strict determinism. Pre-execution is also used to synthesize accelerated EVM code by caching speculative results for all possible branches [18, 112]. DeCl [106] adopts software-based sandboxing to perform deterministic gas metering on untrusted machine code. In Vegeta [105], each validator speculatively executes its proposed blocks to generate hints, and after consensus all validators replay all blocks. These mechanisms are orthogonal to the design of Remora.

Blockchain Architectures. AdaptChain [104] scales throughput by adjusting concurrent block generation based on demand. Some systems move state off-chain to increase throughput [17, 103]. Pilotfish [49] adopts Calvin architecture and uses fixed data partitioning, thus inelastic and inadaptive to workloads. In permissioned systems, SChain [79] adopts a similar architecture in its single trusted organization and decouples execution into compute and storage roles. FlexChain [99] further leverages hardware disaggregation to improve resource utilization. Remora differs in leveraging domain insights and focusing on the performance and operational practicality of a modular execution layer, instead of the whole chain.

9 Conclusion

In this work, we make the case for a scale-out blockchain execution layer that preserves strict determinism. We show why prior deterministic transactional systems fall short and identify opportunities unique to blockchains. Based on these insights, we design Remora, a scale-out smart contract execution engine. Remora adopts an asymmetric architecture for efficiency and cost-effectiveness, and employs versioning with ownership to guarantee determinism. Remora separates stateless part from smart contract transactions, and leverages consensus window to perform stateless execution and stateful scheduling. Remora also demonstrates its elasticity and workload adaptiveness. We believe Remora paves the way for the next generation of scalable blockchain systems.

References

[1] 2023. TPC-C Benchmark. https://www.tpc.org/tpcc/. Accessed: December 2, 2023.

[2] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: automatic physical design metamorphosis for distributed database systems. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3573–3587.

[3] Algorand Foundation. 2025. Leading on post-quantum technology. https://algorand.co/technology/post-quantum.

[4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.

[5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 international conference on management of data*. 76–88.

[6] Parwat Singh Anjana, Matin Amini, Rohit Kapoor, Rahul Parmar, Raghavendra Ramesh, Srivatsan Ravi, and Joshua Tobkin. 2025. Efficient Parallel Execution of Blockchain Transactions Leveraging Conflict Specifications. In *7th Conference on Advances in Financial Technologies (AFT 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 29–1.

[7] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2025. Shoal++: High Throughput {DAG}{BFT} Can Be Fast and Robust!. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 813–826.

[8] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2023. Mysticeti: Reaching the limits of latency with uncertified dags. *arXiv preprint arXiv:2310.14821* (2023).

[9] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. 2024. zklogin: Privacy-preserving blockchain authentication with existing credentials. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3182–3196.

[10] Dvir David Biton, Roy Friedman, and Yaron Hay. 2025. Ethereum Conflicts Graphed. arXiv:2507.20196 [cs.DC] https://arxiv.org/abs/2507.20196

[11] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. 2024. Sui Lutris: A Blockchain Combining Broadcast and Consensus. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Salt Lake City, UT, USA, 2606–2620. https://doi.org/10.1145/3658644.3670286

[12] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short signatures from the Weil pairing. *Journal of cryptology* 17, 4 (2004), 297–319.

[13] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. 2021. The provable security of ed25519: theory and practice. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1659–1676.

[14] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. USENIX Association, 173–186.

[15] Kostas Kryptos Chalkias, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Alberto Sonnino, and Joy Wang. 2024. Fastcrypto: Pioneering cryptography via continuous benchmarking. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*. 227–234.

[16] Chen Chen, Xingbo Wu, Wenshao Zhong, and Jakob Eriksson. 2024. Fast Abort-Freedom for Deterministic Transactions. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 692–704.

[17] Wuhui Chen, Ding Xia, Zhongteng Cai, Hong-Ning Dai, Jianting Zhang, Zicong Hong, Junyuan Liang, and Zibin Zheng. 2024. Porygon: Scaling blockchain via 3d parallelism. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1944–1957.

[18] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. 2021. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 570–587.

[19] Zhihao Chen, Xiaodong Qi, Xiaofan Du, Zhao Zhang, and Cheqing Jin. 2021. Peep: A parallel execution engine for permissioned blockchain systems. In *International Conference on Database Systems for Advanced Applications*. Springer, 341–357.

[20] Zhihao Chen, Tianji Yang, Yixiao Zheng, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2024. Spectrum: Speedy and strictly-deterministic smart contract transactions for blockchain ledgers. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2541–2554.

[21] Tokio Contributors. [n.d.]. Notify in tokio::sync. Rust crate documentation on docs.rs. https://docs.rs/tokio/latest/tokio/sync/struct.Notify.html Accessed: 2025-12-31.

[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[23] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).

[24] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.

[25] DFINITY Foundation. 2024. The Next Step for DeAI: On-Chain Inference Enabling Face Recognition. https://medium.com/dfinity/the-next-step-for-deai-on-chain-inference-enabling-face-recognition-589183203fc2. DFINITY Blog.

[26] Tamer Eldeeb, Xincheng Xie, Philip A Bernstein, Asaf Cidon, and Junfeng Yang. 2023. Chardonnay: Fast and general datacenter transactions for {On-Disk} databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 343–360.

[27] Ethereum.org. 2025. *Ethereum roadmap*. https://ethereum.org/roadmap/ Accessed: 2026-01-14.

[28] Etherscan. [n.d.]. Ethereum Active Addresses Chart. Etherscan. https://etherscan.io/chart/active-address Accessed: 2025-12-31.

[29] François Ezard, Can Umut Ileri, and Jérémie Decouchant. 2025. NEMO: Faster Parallel Execution for Highly Contended Blockchain Workloads. In *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 1–5.

[30] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1190–1201.

[31] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[32] FastCrypto. 2025. FastCrypto Benchmark. https://mystenlabs.github.io/fastcrypto/benchmarks/criterion/reports/BN254%20Verification/Groth16%20verify%20with%20processed%20vk/8/index.html. Accessed: July 22, 2025.

[33] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. 2022. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In *Proceedings of the 44th international conference on software engineering*. 2315–2326.

[34] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. In *PPoPP*. ACM, 232–244.

[35] Sara Gherghelas. 2024. DappRadar Games Report – 2023 Overview. https://dappradar.com/blog/dappradar-games-report-2023-overview. DappRadar × BGA Blog.

[36] Paul Grau. [n.d.]. Lessons learned from making a Chess game for Ethereum. https://medium.com/@graycoding/lessons-learned-from-making-a-chess-game-for-ethereum-6917c01178b6. Accessed: 2025-09-08.

[37] Yaron Hay and Roy Friedman. 2024. Batch-Schedule-Execute: On Optimizing Concurrent Deterministic Scheduling for Blockchains. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 163–174.

[38] Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, and Roger Wattenhofer. 2023. Defi and nfts hinder blockchain scalability. In *International conference on financial cryptography and data security*. Springer, 291–309.

[39] Jelle Hellings and Mohammad Sadoghi. 2021. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2230–2243.

[40] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2023. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proceedings of the VLDB Endowment* 17, 3 (2023), 469–482.

[41] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. 2013. KuaFu: Closing the parallelism gap in database replication. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1186–1195.

[42] Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. 2024. GriDB: Scaling blockchain database via sharding and off-chain cross-shard mechanism. *arXiv preprint arXiv:2407.03750* (2024).

[43] Chunyue Huang, Shuang Liu, Xinyi Zhang, Wenhao Li, Wei Lu, and Xiaoyong Du. 2025. Chimera: Mitigating Ownership Transfers in Multi-Primary Shared-Storage Cloud-Native Databases. *Proceedings of the VLDB Endowment (PVLDB)* 18, 10 (2025), 3368–3381. https://doi.org/10.14778/3748191.3748201

[44] Yibo Huang, Haowei Chen, Newton Ni, Vijay Chidambaram, Dixin Tang, Emmett Witchel, Zhiting Zhu, and Zhipeng Jia. 2025. Tigon: A distributed database for a CXL pod. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25), Boston, MA*.

[45] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. 2025. Pasha: An efficient, scalable database architecture for cxl pods. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[46] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[47] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about eve:{Execute-Verify} replication for {Multi-Core} servers. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 237–250.

[48] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 145–161.

[49] Quentin Kniep, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. 2024. Pilotfish: Distributed transaction execution for lazy blockchains. *arXiv preprint arXiv:2401.16292* (2024).

[50] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 583–598.

[51] Ramakrishna Kotla and Michael Dahlin. 2004. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*. IEEE, 575–584.

[52] Aptos Labs. 2025. Aptos. https://aptoslabs.com/. Accessed: July 22, 2025.

[53] Aptos Labs. 2025. Aptos Code Repository. https://github.com/AptosLabs/aptos-core. Accessed: July 22, 2025.

[54] Cronos Labs. [n.d.]. Cronos. https://cronos.org/. Accessed: 2025-09-05.

[55] Mysten Labs. 2025. FastCrypto GitHub Repository. https://github.com/MystenLabs/fastcrypto. Accessed: July 22, 2025.

[56] Mysten Labs. 2025. Sui. https://sui.io/. Accessed: July 22, 2025.

[57] Mysten Labs. 2025. Sui Code Repository. https://github.com/MystenLabs/sui. Accessed: July 22, 2025.

[58] Solana Labs. 2025. Solana Code Repository. https://github.com/solana-labs/solana. Accessed: July 22, 2025.

[59] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When private blockchain meets deterministic database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–28.

[60] Jennifer Lam, Jeffrey Helt, Wyatt Lloyd, and Haonan Lu. 2024. Accelerating Skewed Workloads With Performance Multipliers in the {TurboDB} Distributed Database. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1213–1228.

[61] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3786–3798.

[62] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 104–120.

[63] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with {In-Network} coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 387–406.

[64] Zhuolun Li, Alberto Sonnino, and Philipp Jovanovic. 2023. Performance of eddsa and bls signatures in committee-based consensus. In *Proceedings of the 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*. 1–5.

[65] Haoran Lin, Hang Feng, Yajin Zhou, and Lei Wu. 2025. ParallelEVM: Operation-Level Concurrent Transaction Execution for EVM-Compatible Blockchains. In *Proceedings of the Twentieth European Conference on Computer Systems*. 211–225.

[66] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1659–1674.

[67] Yu-Shan Lin, Ching Tsai, Tz-Yin Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don't look back, look into the future: Prescient data partitioning and migration for deterministic database systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1156–1168.

[68] Yushi Liu, Liwei Yuan, Zhihao Chen, Yekai Yu, Zhao Zhang, Cheqing Jin, and Ying Yan. 2023. ChainDash: An Ad-Hoc Blockchain Data Analytics System. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4022–4025.

[69] Zhengqing Liu, Musa Unal, Matthew J Parkinson, and Marios Kogias. 2025. DORADD: Deterministic Parallel Execution in the Era of Microsecond-Scale Computing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 282–296.

[70] Yuen C Lo and Francesca Medda. 2021. Uniswap and the Emergence of the Decentralized Exchange. *Journal of financial market infrastructures* 10, 2 (2021), 1–25.

[71] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. (2020).

[72] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.* 14, 5 (2021), 743–756. https://doi.org/10.14778/3446095.3446098

[73] Bhavana Mehta, Nupur Baghel, Mohammad Javad Amiri, Boon Thau Loo, and Ryan Marcus. 2025. Adaptive Sharding in Untrusted Environments. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–28.

[74] George Mitenkov, Igor Kabiljo, Zekun Li, Alexander Spiegelman, Satyanarayana Vusirikala, Zhuolun Xiang, Aleksandar Zlateski, Nuno P Lopes, and Rati Gelashvili. 2024. Deferred objects to enhance smart contract programming with optimistic parallel execution. *arXiv preprint arXiv:2405.06117* (2024).

[75] Cuong DT Nguyen, Kevin Chen, Christopher DeCarolis, and Daniel J Abadi. 2025. Are Database System Researchers Making Correct Assumptions about Transaction Workloads? *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[76] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[77] Okta, Inc. 2024. What is Decentralized Identity? https://www.okta.com/blog/identity-security/what-is-decentralized-identity/. Okta Blog.

[78] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication.. In *EDBT*. 73–84.

[79] Xiaodong Qi, Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. 2023. Schain: Scalable concurrency over flexible permissioned blockchain. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1901–1913.

[80] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.

[81] Kaihua Qin, Stefanos Chaliasos, Liyi Zhou, Benjamin Livshits, Dawn Song, and Arthur Gervais. 2023. The blockchain imitation game. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3961–3978.

[82] Geoffrey Ramseyer and David Mazières. 2024. Groundhog: Linearly-scalable smart contracting via commutative transaction semantics. *arXiv preprint arXiv:2404.03201* (2024).

[83] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[84] risechain. 2025. risechain/pevm: Blazingly fast Parallel EVM. https://github.com/risechain/pevm. Accessed: 2025-09-05.

[85] Donghyeon Ryu and Chanik Park. 2024. Toward High-Performance Blockchain System by Blurring the Line between Ordering and Execution. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[86] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[87] Zeshun Shi, Cees De Laat, Paola Grosso, and Zhiming Zhao. 2022. Integration of blockchain and auction models: A survey, some applications, and challenges. *IEEE Communications Surveys & Tutorials* 25, 1 (2022), 497–537.

[88] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.

[89] Yuan Sui, Xiaochun Yang, Bin Wang, Yujie Zhang, and Baihua Zheng. 2025. Wait and See: A Delayed Transactions Partitioning Approach in Deterministic Database Systems for Better Performance. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–27.

[90] The Aptos Team. 2025. Validator Deployment amd Configuration. https://aptos.dev/network/nodes/validator-node/node-requirements.

[91] The Sui Team. 2025. Validator Deployment amd Configuration. https://docs.sui.io/guides/operator/validator/validator-config.

[92] The Solana Team. 2025. Validator Deployment amd Configuration. https://docs.solanalabs.com/operations/requirements.

[93] The Tokio Team. [n.d.]. Tokio: an asynchronous Rust runtime. https://tokio.rs/. Accessed: 2025-09-08.

[94] Justin Thaler. 2025. Quantum computing and blockchains: Matching urgency to actual threats. a16z crypto. https://a16zcrypto.com/posts/article/quantum-computing-misconceptions-realities-blockchains-planning-migrations/ Accessed 2026-01-13.

[95] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[96] Xing Tong, Zheming Ye, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2024. TELL: Efficient Transaction Execution Protocol Towards Leaderless Consensus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1902–1915.

[97] Hao Wang, Minghao Pan, and Jiaping Wang. 2025. Crystality: A Programming Model for Smart Contracts on Parallel EVMs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*.

412–425.

[98] Zhipeng Wang, Rui Sun, Elizabeth Lui, Tuo Zhou, Yizhe Wen, and Jiahao Sun. 2025. AIArena: A blockchain-based decentralized AI training platform. In *Companion Proceedings of the ACM on Web Conference 2025*. 1375–1379.

[99] Chenyuan Wu, Mohammad Javad Amiri, Jared Asch, Heena Nagda, Qizhen Zhang, and Boon Thau Loo. 2022. FlexChain: an elastic disaggregated blockchain. *Proceedings of the VLDB Endowment* 16, 1 (2022).

[100] Chenyuan Wu, Mohammad Javad Amiri, Haoyun Qin, Bhavana Mehta, Ryan Marcus, and Boon Thau Loo. 2024. Towards Full Stack Adaptivity in Permissioned Blockchains. *Proceedings of the VLDB Endowment* 17, 5 (2024).

[101] Chenyuan Wu, Bhavana Mehta, Mohammad Javad Amiri, Ryan Marcus, and Boon Thau Loo. 2023. AdaChain: A Learned Adaptive Blockchain. *Proceedings of the VLDB Endowment* 16, 8 (2023), 2033–2046.

[102] Jiang Xiao, Shijie Zhang, Zhiwei Zhang, Bo Li, Xiaohai Dai, and Hai Jin. 2022. Nezha: Exploiting concurrency for transaction processing in dag-based blockchains. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 269–279.

[103] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. 2021. SlimChain: Scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2314–2326.

[104] Jie Xu, Qingyuan Xie, Sen Peng, Cong Wang, and Xiaohua Jia. 2023. Adaptchain: Adaptive scaling blockchain with transaction deduplication. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2023), 1909–1922.

[105] Tianjing Xu, Yongqi Zhong, Yiming Zhang, Ruofan Xiong, Jingjing Zhang, Guangtao Xue, and Shengyun Liu. 2025. Vegeta: Enabling Parallel Smart Contract Execution in Leaderless Blockchains. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 795–811. https://www.usenix.org/conference/nsdi25/presentation/xu-tianjing

[106] Zachary Yedidia, Geoffrey Ramseyer, and David Mazières. 2025. Deterministic Client: Enforcing Determinism on Untrusted Machine Code. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 633–649.

[107] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM symposium on principles of distributed computing*. 347–356.

[108] Dachao Yu, Hao Xu, Lei Zhang, Bin Cao, and Muhammad Ali Imran. 2021. Security analysis of sharding in the blockchain system. In *2021 IEEE 32nd Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 1030–1035.

[109] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 511–526.

[110] Jianting Zhang, Zhongtang Luo, Raghavendra Ramesh, and Aniket Kate. 2024. Optimal Sharding for Scalable Blockchains with Deconstructed SMR. *arXiv preprint arXiv:2406.08252* (2024).

[111] Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. 2023. Efficient distributed transaction processing in heterogeneous networks. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1372–1385.

[112] Shijie Zhang, Ru Cheng, Xinpeng Liu, Jiang Xiao, Hai Jin, and Bo Li. 2024. Seer: Accelerating Blockchain Transaction Execution by Fine-Grained Branch Prediction. *Proceedings of the VLDB Endowment* 18, 3 (2024), 822–835.

[113] Shijie Zhang, Jiang Xiao, Enping Wu, Feng Cheng, Bo Li, Wei Wang, and Hai Jin. 2024. Morphdag: A workload-aware elastic dag-based blockchain. *IEEE Transactions on Knowledge and Data Engineering* 36, 10 (2024), 5249–5264.

[114] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent state machines for recoverable in-memory storage systems with {NVRam}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1029–1046.

[115] Yunhao Zhang, Haobin Ni, Soumya Basu, Shir Cohen, Maofan Yin, Lorenzo Alvisi, Robbert van Renesse, Qi Chen, and Lidong Zhou. 2025. Ordered Consensus with Equal Opportunity. arXiv:2509.09868 [cs.DC] https://arxiv.org/abs/2509.09868

[116] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 633–649.

[117] Jianjun Zhao, Haikun Liu, Shuhao Zhang, Haodi Lu, Yancan Mao, Zhuohui Duan, Xiaofei Liao, and Hai Jin. 2025. Towards {High-Performance} Transactional Stateful Serverless Workflows with {Affinity-Aware} Leasing. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1535–1551.

[118] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2022. Lotus: scalable multi-partition transactions on single-threaded partitioned databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2939–2952.