Fabric-X: Scaling Hyperledger Fabric for Asset Exchange

Elli Androulaki
lli@zurich.ibm.com
IBM Research
Zurich, Switzerland

Marcus Brandenburger
bur@zurich.ibm.com
IBM Research
Zurich, Switzerland

May Buzaglo
may.buzaglo@ibm.com
IBM Research
Haifa, Israel

Angelo De Caro
adc@zurich.ibm.com
IBM Research
Zurich, Switzerland

Kaoutar El Khiyaoui
kao@zurich.ibm.com
IBM Research
Zurich, Switzerland

Alexandros Filios
alexandrosfilios@gmail.com
IBM Research
Zurich, Switzerland

Liran Funaro
liran.funaro@ibm.com
IBM Research
Haifa, Israel

Yacov Manevich
yacov.manevich@gmail.com
IBM Research
Haifa, Israel

Hagar Meir
hagar.meir@ibm.com
IBM Research
Haifa, Israel

Senthilnathan Natarajan
snatara7@in.ibm.com
IBM Research
Bangalore, India

Manish Sethi
manish.sethi1@ibm.com
IBM
Raleigh, United States

Yoav Tock
tock@il.ibm.com
IBM Research
Haifa, Israel

Abstract

The adoption of Distributed Ledger Technology (DLT) for critical financial infrastructures like Central Bank Digital Currencies (CBDCs) is hindered by a significant performance gap. Permissioned blockchains such as Hyperledger Fabric, which are conceptually suitable and have become popular platform used in many production deployments today, are nevertheless limited by architectural bottlenecks. Their monolithic peer design and consensus mechanisms prevent them from achieving the required scale for such demanding applications.

This paper presents Fabric-X, a fundamental re-architecture of Hyperledger Fabric that addresses these challenges end-to-end. We have open-sourced Fabric-X as part of the Linux Foundation Decentralized Trust at github.com/hyperledger/fabric-x. We decompose the monolithic peer into independently scalable microservices for endorsement, validation, and committing. To maximize parallelism, we introduce a transaction dependency graph that enables the safe, concurrent validation of transactions across multiple blocks. Complementing the peer redesign, we introduce Arma, a novel sharded Byzantine Fault Tolerant (BFT) ordering service that dramatically increases throughput by ordering compact transaction digests rather than full transaction payloads. We implemented and benchmarked this framework with a UTXO-based CBDC application. Our evaluation demonstrates a peak throughput exceeding 100,000 transactions per second (TPS)—a multiple-orders-of-magnitude improvement over the standard implementation. This work proves that permissioned DLTs can be engineered for national-scale payment systems, providing a resilient and highly performant foundation for practical CBDC deployments and the integration of advanced, computationally intensive features.

CCS Concepts

• **Information systems** → **Database transaction processing**; • **Security and privacy** → *Digital signatures*; • **Networks** → *Peer-to-peer protocols.*

Keywords

Blockchain, Hyperledger Fabric, CBDC, Scalability, BFT Consensus, UTXO

ACM Reference Format:
Elli Androulaki, Marcus Brandenburger, May Buzaglo, Angelo De Caro, Kaoutar El Khiyaoui, Alexandros Filios, Liran Funaro, Yacov Manevich, Hagar Meir, Senthilnathan Natarajan, Manish Sethi, and Yoav Tock. 2026. Fabric-X: Scaling Hyperledger Fabric for Asset Exchange. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31–June 5, 2026, Bengaluru, India.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3788853.3803092

1 Introduction

Distributed ledger technology (DLT) has generated significant interest in tokenization and programmable ledgers, particularly within regulated financial market infrastructures. Tokenization enables the digital representation of assets and their governing rules on a ledger, while programmable ledgers use smart contracts to automate actions like ownership transfer and delivery versus payment. This potential for automation promises increased speed and reduced errors, leading financial institutions—including central banks, commercial banks, and clearing houses—to explore new financial market

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

SIGMOD Companion '26, Bengaluru, India.

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3803092

Authors are listed in alphabetical order by last name.

infrastructures based on these technologies. These institutions aim to address existing inefficiencies in regulated finance, including payment systems, by leveraging tokenization and smart contracts.

A prime example of this exploration is central bank digital currency (CBDC), a tokenized form of central bank liability on par with cash and reserves. Over 130 central banks are investigating CBDC through experimentation, reports, and pilot launches [3]. Early results indicate faster and less risky wholesale payments, reduced fees, and greater financial inclusion for retail payments. Another example is regulated liability networks, platforms where CBDC, tokenized commercial bank money, and tokenized securities coexist on a programmable ledger to automate operations such as delivery versus payment and payment versus payment. Platforms facilitating the exchange of these tokenized assets operate within diverse legal and regulatory frameworks. While jurisdictional variations exist, a common thread is a centralized or federated governance model, often with the central bank as a key stakeholder. This is understandable given the potential impact on monetary policy and financial stability. However, it's crucial to distinguish between governance and deployment. While a central authority may oversee the system, the operational architecture can be designed for enhanced resilience. Resilience, in this context, refers to the system's ability to maintain functionality amidst disruptions, including technical failures, cyberattacks, and insider threats.

A decentralized deployment model is critical for ensuring such resilience: distributing operational components across independent nodes with Byzantine fault tolerance reduces reliance on any single point of failure, while centralized governance is preserved to ensure regulatory oversight.

Consequently, financial institutions [1] and the research community (e.g., [37]) have been exploring decentralized consensus-based transaction processing systems, especially in the context of CBDC. DLT systems can be broadly categorized as permissionless (public) or permissioned (private). Permissionless blockchains like Bitcoin and Ethereum are unsuitable due to their openness, energy-intensive consensus, and limited throughput from serial execution.

Permissioned blockchains, such as Hyperledger Fabric [12], are designed for specific groups with access controls and employ more efficient consensus mechanisms. This allows for greater scalability and performance due to factors like faster block generation, higher transaction throughput, and lower latency. Furthermore, permissioned DLTs offer flexibility in their transaction processing models. They can follow the traditional order-execute model or adopt the more efficient execute-order-validate (EOV) model.

Hyperledger Fabric supports the EOV architecture, which enhances confidentiality and exploits concurrency through three pipelined phases: *Execution* (parallel smart contract execution producing read-write sets), *Ordering* (consensus-based total ordering into blocks), and *Validation* (endorsement policy and serializability checks).

However, despite its promising architecture, the standard implementation of Hyperledger Fabric falls critically short of the requirements for a national-scale CBDC or other demanding financial use cases. Its limitations are significant:

(1) Performance and Scalability: The monolithic design of a Fabric peer, which combines endorsing, validating, and committing services on a single node, creates severe resource contention and I/O bottlenecks. This fundamentally constrains throughput and prevents horizontal scaling.
(2) Privacy: Standard endorsement policies can inadvertently leak the identities of transacting organizations, failing to provide the anonymity and unlinkability required in many financial scenarios.
(3) Efficiency: Transactions are bloated with multiple signatures and certificates, and the sequential validation process introduces significant latency, making it impossible to achieve the high throughput needed for real-time systems. Existing optimizations have failed to push performance beyond 20,000 transactions per second (TPS), a fraction of what a CBDC would demand.

This research directly addresses these deficiencies by presenting a fundamental re-architecture of Hyperledger Fabric designed to deliver the performance, scalability, and privacy necessary for modern financial infrastructures. Our key contributions are:

(1) *A High-Performance Fabric Redesign:* We decouple the monolithic Fabric peer into independently scalable microservices for endorsement, validation, and committing. Crucially, we redesign the Fabric orderer to separate transaction ordering from dissemination by ordering digests instead of full transactions, pipeline the service into sequential stages, and divide processing into parallel shards. Combined with parallel validation of transactions across blocks using a dependency graph and threshold signatures, our design breaks existing bottlenecks and achieves a throughput of over 100,000 TPS. Furthermore, we introduce an extended programming model for developers to directly integrate endorsement logic, reducing multi-transaction overhead in complex, multi-party workflows.
(2) *An Application-Layer UTXO Model for CBDCs:* We demonstrate the redesigned platform's capabilities by implementing a CBDC use case that utilizes an Unspent Transaction Output (UTXO) model. This showcases how a token-based system can be effectively built at the application layer on Hyperledger Fabric, validating that our high-performance architecture can support complex financial transaction logic.

2 Hyperledger Fabric

2.1 Background

Hyperledger Fabric (HLF) is a permissioned DLT system known for strong identity management and accountability. It features an innovative *execute-order-validate* [12] transaction model that replaces the traditional *order-execute* approach. In this model, transactions are speculatively executed before being ordered, which supports parallel execution for improved throughput and allows developers to use conventional programming languages. Any transaction using stale data is rejected during a final validation phase.

A Fabric peer's architecture, illustrated in Figure 1(a), includes components like an endorser, validators, and a committer. The transaction flow, depicted in Figure 2, begins with chaincodes—smart contracts tied to specific endorsers via an endorsement policy. These

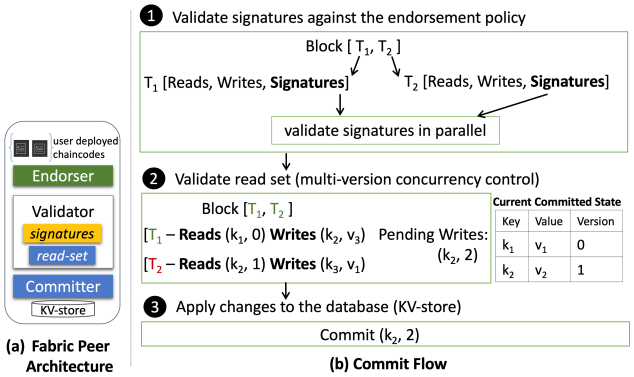


Figure 1: Architecture of Fabric Peer and Commit Flow within a Peer

chaincodes operate within defined namespaces, specifying which ledger partitions they can modify. The process starts when a client sends a transaction proposal to its designated endorsers. The endorsers speculatively execute the chaincode, generating a signed read-write set (dependencies and proposed changes). The client then assembles these signed sets into a transaction and submits it to the orderers. In the ordering phase, orderers sequence the transactions, batch them into blocks, and distribute them to all peers. Upon receiving a block, each peer independently performs the commit process as shown in Figure 1(b), which involves three steps:

(1) *Signature Validation*: The signature validator checks if the endorsers' signatures comply with the chaincode's endorsement policy.
(2) *Read-Set Validation*: The read-set versions of keys are compared with their committed versions in the ledger. For example (refer to Figure 1(b)), if transaction T_1 reads key k_1 at version 0, matching the committed version, its writes are added to the pending writes after incrementing the version of k_2. However, any transaction referencing stale data is marked invalid. For instance, T_2 reading k_2 at version 1 will be invalid as the pending writes indicate version 2 for k_2.
(3) *Commitment*: The valid, pending writes are applied to the local key-value database, finalizing the state changes.

This structured process ensures ledger integrity and consistency under concurrent operations.

2.2 Assessing Hyperledger Fabric for CBDC Requirements

The architectural design of HLF provides several features that make it a strong conceptual candidate for CBDC systems. Its permissioned model ensures only authorized participants access the network, a prerequisite for financial compliance:

(1) Flexible Smart Contracts: Chaincodes can be written in general-purpose languages to implement CBDC operations like issuance, transfer, and compliance checks. (2) Modular Consensus: HLF supports swappable consensus protocols, allowing transition between CFT and BFT mechanisms. (3) Granular Control: Endorsement policies allow granular control over transaction validation to enforce complex business rules.

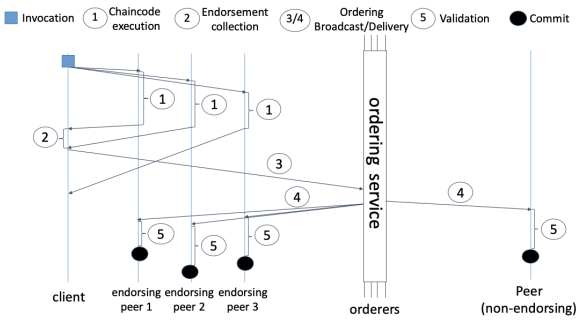


Figure 2: Transaction Lifecycle in Hyperledger Fabric.

However, HLF's standard implementation struggles to meet the demands of a large-scale CBDC, revealing fundamental bottlenecks in throughput, latency, and scalability.

2.3 Limitations for CBDC Use-Case

The current Hyperledger Fabric implementation delivers a throughput of only around 2,000 transactions per second (TPS), which significantly falls short of CBDC requirements due to the following limitations:

(1) Monolithic Peer Resource Contention: Each peer acts as both an endorser and a validator on a single node, leading to contention for CPU resources as these distinct workloads compete for a limited number of cores.
(2) State Database Contention: Transaction execution (by the endorser) and ledger updates (by the committer) are mutually exclusive operations that compete for access to the state database, limiting concurrency.
(3) Committer and Storage Bottlenecks: The committer's performance is constrained by the disk write bandwidth of a single node, and the storage capacity of a single peer is insufficient for a national-scale ledger.
(4) Large Transaction Size: Transactions are bloated with multiple signatures and x509 certificates, creating significant network and storage overhead that degrades the performance of the ordering service.
(5) Endorsement Policy Verification Overhead: Verifying multiple signatures for each transaction against its endorsement policy is computationally expensive and adds significant commit latency.
(6) Sequential Validation Execution: The validator processes the read-set of each transaction in a block sequentially to ensure serializability, creating a major bottleneck in the commit phase.
(7) Non-Pipelined Commit Flow: The validation and commit phases for consecutive blocks are not pipelined; block i can only be validated after block $i-1$ is fully committed, increasing overall latency.
(8) Limited Consensus Throughput: Traditional BFT consensus protocols used by HLF, while secure, do not scale well and become a bottleneck as the number of ordering nodes increases.

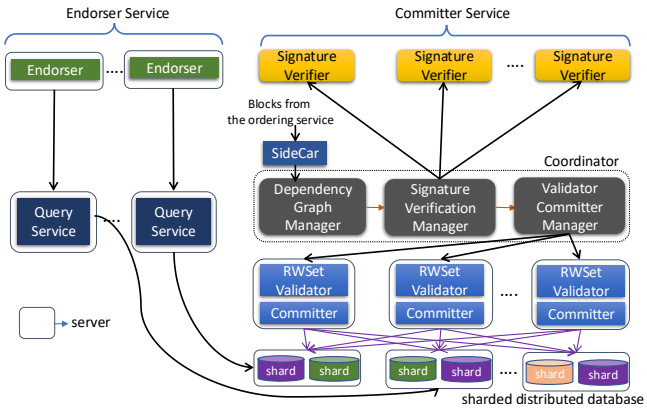


Figure 3: Proposed Fabric Peer Architecture

Existing literature suggests various optimizations for HLF, yet none have succeeded in achieving throughput beyond 20,000 TPS. A fundamental redesign of the Hyperledger Fabric architecture is therefore necessary to overcome these limitations and meet the demands of a CBDC system.

3 Redesigning Hyperledger Fabric Peer Architecture

To address the limitations found in Hyperledger Fabric (HLF), we propose a fundamental re-architecture of the Fabric Peer.

3.1 Micro-Service Architecture

The conventional Hyperledger Fabric peer is implemented as a monolithic service, as illustrated in Figure 1(a). We propose the disaggregation of this monolithic architecture into a set of decoupled microservices, depicted in Figure 3. This decomposition yields the following distinct components: an *Endorser* Service for transaction endorsement, a *Query* Service for read-only ledger queries, a *Coordinator* to orchestrate the transaction lifecycle, a *Signature Verifier* to verify endorsement signatures, a *Validator-Committer* Service for transaction validation and ledger commitment, and a *Sidecar*. The Sidecar receives blocks from the ordering service, forwards them to the Coordinator for processing, aggregates transaction statuses to determine block commitment, and notifies clients of final transaction outcomes (commit or abort). The functionalities of each service is detailed in Section 3.6.

To complement the microservice-based design, we replace the peer's local key-value store with a sharded, distributed database that provides native support for advanced concurrency control protocols. This disaggregated architecture yields several advantages. First, the decoupling of the Endorser and Signature Verifier services enables their independent deployment and scaling across multiple servers, thereby mitigating CPU resource contention. Second, the adoption of a sharded database supporting snapshot isolation obviates the requirement for exclusive ledger access by the Endorser and Committer services. This distribution of storage and I/O workload across multiple database shards enhances both write throughput and overall system scalability. This architecture directly addresses limitations 1–3 of Section 2.3.

3.2 Extended Programming Model

In traditional Hyperledger Fabric deployments, application developers express their business logic as chaincode that implements the shim interface, which is then installed and executed on designated endorsing peers. This model has proven effective for a broad range of use cases, offering deterministic execution, access control, and endorsement flexibility within a permissioned framework. However, this model introduces limitations in scenarios involving multi-party interactions. For example, selling a digital asset typically requires multiple back-and-forth exchanges between a buyer and a seller, with each step triggering a separate transaction and endorsement process. These intermediate transactions increase transactional overhead and fragment the business logic across several loosely connected ledger operations, making reasoning and validation more complex.

The new programming model introduced in this work extends the traditional chaincode-centric approach by allowing application developers to integrate the endorsement logic directly into their applications via an Endorser SDK. In this model, the application itself functions as the endorsing service, as described in Section 3.1. A key advantage of this architecture is its support for collaborative transaction assembly. Rather than relying on sequential client–chaincode interactions, business parties (e.g., buyer and seller) can interact directly to jointly construct a transaction and generate the required endorsements in a single, cohesive operation. This design streamlines complex workflows, eliminates redundant ledger writes, and preserves the atomicity of business actions.

3.3 Threshold-Based Signatures

Hyperledger Fabric's current namespace endorsement policy system is powerful and flexible, allowing for the definition of sophisticated rules that specify which set of organizations must endorse a transaction through logical constructs like `AND`, `OR`, and `n-OutOf`. However, this policy framework leads to two critical limitations. First, it contributes to large transaction size (Limitation 4), as each transaction must carry multiple signatures and their corresponding X.509 certificates. This bloats transaction size, creating significant network and storage overhead. An alternative, such as storing certificates as known identities, could reduce transaction size but would leak participant identities and would not reduce the number of signatures per transaction. Second, it results in significant endorsement policy verification overhead (Limitation 5). The multiple cryptographic operations required to verify each signature against the policy add computational overhead and increase commit latency. To address these challenges, we leverage the extended programming model (Section 3.2) and propose introducing a threshold-based signature scheme to operate in addition to the conventional mechanism, which fundamentally alters the verification process. In a (t, n) threshold scheme, any t participants from a group of n can collaboratively generate a single, valid signature for the group, while no individual can do so alone. This model yields a single, aggregated signature that consolidates endorsements into one compact proof, and its verification key can be inherently linked to the policy itself. By replacing multiple verifications with a single one, this approach reduces computational load, lowers latency, and improves transaction throughput. Furthermore, threshold signatures enhance privacy by concealing the identities of the specific

endorsing organizations. While a purely threshold-based scheme may lack the expressiveness of Fabric's native policies, we envision a hybrid model supporting both. This dual approach allows users to balance the flexibility of traditional policies with the efficiency and privacy of threshold signatures, creating a more versatile platform. Thus, the proposed design directly addresses limitations (4) and (5) as discussed in Section 2.3.

3.4 Parallel Validation and Pipelined Execution

Vanilla Fabric validates each transaction's read-set sequentially. This is because the validity of an earlier transaction can influence the validity of later transactions as shown in Figure 1(b). While a fully parallel validation approach, assuming no transaction conflicts, would be ideal, it's unrealistic to expect such behavior in practice. However, we can anticipate that a majority of transactions will not have conflicts.

To exploit this characteristic and enable parallel validation, we introduce a transaction dependency graph in the Coordinator. To ensure deterministic results across all nodes despite the inherent non-determinism of concurrency, this graph carefully tracks dependencies between transactions. Each node in the graph represents a transaction. Let's consider two transactions, T_i and T_j, where T_i appears earlier in the block ordering than T_j (either in the same block or a preceding block). An edge from T_j to T_i signifies that T_i must be validated and committed/aborted before T_j can be considered for validation. We define the following three dependencies which are sufficient to represent any potential conflict that could arise from concurrent access to shared state. Recall that in the EOV model, each transaction's read-set records the specific version of each key read during execution. During validation, these recorded versions are checked against the committed versions in the ledger—a form of multi-version concurrency control (MVCC). The three dependency types below capture conflicts that arise from these versioned reads and writes:

(1) ***read-write dependency*** ($T_i \xleftarrow{rw(k)} T_j$): T_i writes a new value to state k, updating its version. T_j reads the version of state k that was current before T_i's write. If T_i is valid and committed, its write updates k's committed version, making T_j's read stale—hence T_j must be invalid. Conversely, if T_i is invalid, its write is discarded and k's committed version remains unchanged; in this case T_j's read may still be fresh, and T_j can potentially be valid (subject to its other dependencies).

(2) ***write-read dependency*** ($T_i \xleftarrow{wr(k)} T_j$): T_j writes a new value to state k, updating its version. T_i reads the previous version of state k. Regardless of T_i's validity, T_j can be validated. However, this dependency ensures that T_j is not committed before T_i. If T_j were committed first, T_i would become invalid as it read an outdated version. This should not be allowed because T_i appeared in the block before T_j, and committing T_j first would lead to an incorrect result. We use this dependency to enforce the correct order of commit.

(3) ***write-write dependency*** ($T_i \xleftarrow{ww(k)} T_j$): Both T_i and T_j write to the same state k. Regardless of the validity of T_i, T_j can be validated. This dependency ensures that T_j is not committed before T_i to prevent T_j's write from being overwritten and lost.

Table 1: The read-write set of transactions in block B_1 and B_2.

Block	Tx.	Read Set (key, version)	Write Set (key, value)	valid
B_1	T_1	$(k_6, 1)$	(k_1, v_1)	✓
	T_2	$(k_1, 2)$	(k_6, v_6)	✗
	T_3	$(k_3, 1)$	(k_6, v_7)	✓
	T_4	-	(k_5, v_5)	✓
B_2	T_1	(k_5, nil)	(k_7, v_1)	✗
	T_2	$(k_4, 3)$	(k_4, v_4)	✓

An edge always points from a later transaction (T_j) to an earlier transaction (T_i) i.e., if $T_i \leftarrow T_j$ then $i < j$ because the commit order is determined during the ordering phase. Consequently, the dependency graph is always directed and acyclic (i.e., it contains no cycles). This structure is analogous to a serialization graph (or conflict graph) from classical concurrency control theory [16], adapted here for the EOV validation pipeline. Prior work has noted the connection between Fabric's concurrency challenges and database concurrency control [33]. The acyclic nature of the graph is crucial for enabling efficient parallel validation. The dependency graph of transactions in Table 1 is shown in Figure 4. The transactions T_1, T_4 in block B_1 and T_2 in block B_2 are dependency-free and can be processed in parallel. Other transactions have to wait for their dependencies to be validated and committed or aborted.

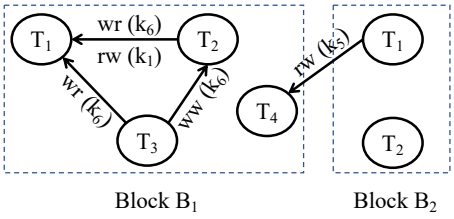


Figure 4: Dependency graph of transactions in Table 1.

When the *Dependency Graph Manager* receives new blocks, it initiates the construction of local dependency graphs, one for each block, where each graph captures dependencies between transactions within its respective block's scope. Importantly, these local dependency graphs can be constructed in parallel for multiple individual blocks simultaneously, enabling parallel graph construction and significantly accelerating the process compared to building a single global graph serially. Subsequently, these local graphs are merged into the global dependency graph, strictly respecting block order to accurately represent inter-block dependencies. A key optimization during this merge is that dependencies only need to be established between transactions in the local graph and those already residing in the global graph, since a new block's transactions can only depend on transactions within the same block or prior ones that are already part of the global graph. This focused approach minimizes the number of transactions the *Dependency Graph Manager* processes sequentially to construct the global dependency graph.

Transactions with no outstanding dependencies (i.e., those having no outgoing edges in the dependency graph, also known as

having an out-degree of zero) are selected for validation and commitment. To maximize parallelism, the validation of these independent transactions is distributed across multiple *Signature Verifier services* and *Validator-Committer* services. After valid transactions are committed to the ledger, the dependency graph is updated. This update may resolve dependencies for other transactions, making them eligible for validation in the next cycle. Since the dependency graph guarantees that concurrently validated transactions are conflict-free, we can safely pipeline the validation and commit phases, further enhancing performance. Thus, the proposed design directly addresses the limitation 6 and 7 discussed in Section 2.3.

3.5 Consistency Proof

We aim to prove that parallel validation using our dependency graph produces the same final state as serial execution. Serial execution processes transactions sequentially, while our parallel approach validates concurrently, respecting dependency constraints and committing in a topological order. This proof assumes deterministic operations and that the graph accurately captures all read-write, write-read, and write-write dependencies.

We employ induction for this proof. The base case with a single transaction is trivial: both serial and parallel execution yield identical results. Now, let's assume the inductive hypothesis: for k transactions, parallel execution with the dependency graph results in the same final state as serial execution. For the inductive step, we introduce T_{k+1} to the set. In serial execution, T_{k+1} is validated against the state after committing T_1 through T_k and then committed if valid. In parallel execution, the dependency graph is built for all $k + 1$ transactions. By our inductive hypothesis, the first 'k' transactions are validated and committed in parallel, but due to the dependency graph's constraints, the resulting state is identical to that achieved through serial execution. There are two possibilities for T_{k+1}: it either has dependencies on preceding transactions or it doesn't. If T_{k+1} depends on one or more of the preceding k transactions, it will only be validated after its dependencies are satisfied. This ensures validation against the same state as in serial execution, and due to deterministic operations, the validation and commitment outcomes will be identical. If T_{k+1} has no dependencies, the lack of shared state with prior transactions means it can be validated against any state, yielding the same result. Since both scenarios are consistent with serial execution, we conclude by induction that our parallel model produces the same final state for any set of transactions. The dependency graph ensures all inter-transaction dependencies are respected, effectively mimicking serial execution's logic while enabling the performance benefits of parallelism, thus guaranteeing state consistency and integrity.

3.6 Transaction Commit Flow

The commit process, with steps depicted in Figure 3, begins with the *Sidecar*, which fetches blocks from the ordering service (in step ❶). The Sidecar maintains a list of active transactions, allowing it to identify and mark any with duplicate identifiers before forwarding the block to the *Coordinator* for processing (in step ❷). The Coordinator sends the transactions to the *Dependency Graph Manager*, which follows the steps described in Section 3.4 to continuously identify dependency-free transactions and forward them to the *Signature Verifier Manager* (in step ❸). This manager distributes the transactions to available *Signature Validators* for preliminary validation and signature verification. Transactions with structural errors are immediately marked as invalid, and all results are returned to the Coordinator (in step ❹). The *Coordinator*, in turn, sends the successfully verified transactions to the *Validator Committer Manager* (in step ❺), which routes them to available *Validator-Committer* services (in step ❻). These services execute a three-phase pipelined process to validate and commit transactions (in step ❼):

(1) Preparer Phase: The service first batches incoming transactions. In this phase, the read-write set of each transaction is divided into three categories: reads that need validation against the committed state, new state inserts, and updates to existing states.
(2) Validator Phase: The reads are then validated against the committed version of the data in the ledger. Transactions that fail this check due to stale reads are marked with the appropriate invalid status, and their associated writes (inserts and updates) are discarded.
(3) Commit Phase: Valid transactions proceed to the final commit phase. Here, the service commits the results by storing the transaction's final status along with its identifier, inserting any new states, and applying updates to existing states as specified in the write-sets.

A key property of this design is that multiple Validator-Committer (VC) instances operate concurrently, yet consistency is maintained without explicit cross-instance coordination. This is possible because the dependency graph (Section 3.4) guarantees that transactions dispatched simultaneously to different VC instances are conflict-free—they have no overlapping read or write keys. Consequently, each VC instance can independently validate and commit its assigned transactions against the distributed database without risk of conflicting writes or inconsistent reads from another instance. The distributed database employs Repeatable Read Isolation as a safety net for each individual VC transaction. In the event of a VC instance failure, the idempotent commit mechanism described in Section 3.7 ensures that resubmitted transactions are not duplicated or incorrectly revalidated.

Following validation, transaction identifiers (txID) and their statuses are sent from each *Validator-Committer* service back to the *Validator Committer Manager* and then to the *Coordinator* (in step ❽ and ❾). This outcome is also notified back to the dependency graph, which removes the completed transaction to free up any dependent transactions. The *Coordinator* forwards these statuses back to the *Sidecar* (in step ❿), which aggregates the status for each block, as transactions may be committed across block boundaries. Once the next block in sequence is fully committed, the Sidecar appends the block with its final transaction statuses to an append-only block store, stores the last fully committed block number in the state database for use during failure/recovery of services, and forwards the completed block to registered clients, informing them whether their transactions were committed or aborted.

3.7 Failure and Recovery

The system is designed to gracefully handle failures in any microservice—including the sidecar, coordinator, signature verifier, and validator-committer—and ensure correctness.

Failure of a Signature Verifier. The Signature Verifier Manager tracks transactions per verifier. If a verifier fails, the manager detects this and resubmits its pending transactions to an available replica. To prevent duplicate processing from transient issues, the manager ignores late responses for transactions that have already been reassigned.

Failure of a Validator-Committer Service. Similarly, the Validator Committer Manager resubmits pending transactions from a failed Validator-Committer (VC) service to another replica. To prevent incorrect validation or duplicate writes from this resubmission, the commit operation is idempotent. When a VC service attempts to store a transaction's status, it first checks if a record matching the txID, block number, and transaction index already exists. If a match is found, the service reuses the existing committed status instead of reprocessing the transaction.

Failure of Coordinator. The sidecar periodically stores the last fully committed block number. After a restart, the coordinator reads this number to determine the next expected block from the sidecar, which then begins fetching blocks from that point. Because transactions can be committed across blocks, it is possible some transactions in a fetched block are already committed. The idempotent design of the validator-committer services handles this: they detect existing transaction statuses (via txID, block number, and index) and return the stored status without recommitting.

Failure of Sidecar. When the sidecar restarts, it gets the next expected block number from the coordinator and compares it to the last block in its local store. If there is a gap, the sidecar fetches the missing blocks from the ordering service and their transaction statuses from the state database to update its store. It then resumes fetching new blocks from the ordering service.

Failure of Nodes in Distributed Database. Database node failures are handled by shard replication; if a shard fails, its replicas remain available. A catastrophic failure or corruption of the entire state database requires rebuilding the state by reprocessing all blocks from the ordering service.

4 A Scalable Ordering Service

This section introduces Arma, a scalable, high-performance Byzantine Fault-Tolerant (BFT) ordering service for HLF, designed to achieve significantly higher throughput with reasonable latency for the CBDC use case while maintaining near-API compatibility with HLF's existing service. HLF v3 offers a BFT ordering service [23, 29] using the SmartBFT library [14, 25], which builds on PBFT [18] and BFT-SMaRt [17]. In this model, a cluster of $n = 3f + 1$ Ordering Service Nodes (OSNs) is required to tolerate f Byzantine failures (see top of Figure 5). Arma replaces each OSN with a party, a distributed set of components controlled by a single organization that acts as a single unit of failure. Arma also adheres to the $n = 3f + 1$ BFT requirement [38]. Arma's performance gains are based on three design principles inspired by "Narwhal and Tusk" [19] and adapted for HLF. First, it separates transaction ordering from dissemination by ordering digests instead of full transactions, alleviating network bottlenecks (Figure 5 ❶). Second, it pipelines the ordering service by decomposing it into sequential stages that operate in parallel (Figure 5 ❷). Third, it divides independent processing into parallel shards (Figure 5 ❸). Together, pipelining and sharding increase parallelism and allow Arma to scale with added hardware resources.

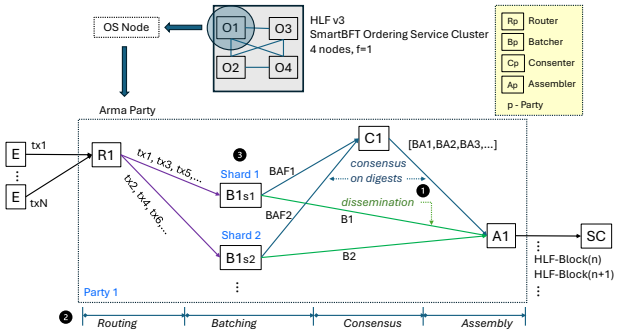


Figure 5: An Arma party is equivalent to an HLF v3 OS Node. An Arma party is composed of 4 sub-components: a router (R), one or more batchers (B), a consenter (C), and an assembler (A). Endorsing clients (E) submit transactions to the router. The router spreads transactions across the shards, sending only to the batcher (B) of its own party (p) in a given shard. Batchers aggregate transactions in to batches (B1,B2), and compute a batch attestation fragment on the batches (BAF1,BAF2, resp.). The consensus cluster collects BAFs and emits a total order of batch attestations (BAs). The assembler consumes both totally ordered BAs from consensus, and batches from the shards. It then assembles HLF blocks according to the order induces by consensus and the content received from the shards. The HLF blocks are then ready for the scalable committer (SC) to pull.

4.1 Arma Overview

Arma runs in four stages **routing**, **batching**, **consensus**, and **assembly** (Figure 5). When *clients* submit transactions to Arma, the transactions enter the routing stage, during which they are validated against some pre-defined static system rules (e.g., admissible transaction format and access control rules). Valid transactions then move onto batching, whereby transactions are bundled into batches and persisted to disk. During consensus, the batches are *totally ordered* using a BFT consensus protocol. Arma utilizes the SmartBFT library [14], which is a BFT consensus protocol adapted for blockchains. However, for efficiency reasons, consensus does not order the batches in their entirety; instead it orders their much shorter *signed digests*, which are computed during batching. Thanks to this optimization, we amplify the throughput of the SmartBFT consensus protocol. Consensus then delivers *signed block headers*, which are compressed representation of the ordered transactions for block assembly, during which the batches corresponding to a received block header are fetched from disk and the full block is constructed.

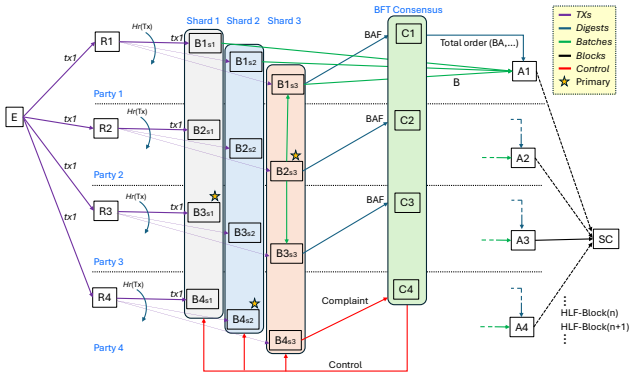


Figure 6: A correct endorsing client (E) tries to submit a TX to all parties. Routers validate and dispatch the TX to a shard according to hash function H_r. The primary batcher in a shard (e.g. $B2_{s3}$ in shard 3) bundles TXs in a batch, persists it, and broadcasts it to the secondary batchers. Batchers that persist a batch send a BAF to the consensus cluster. Upon receiving enough BAFs, the consensus cluster emits a total order of BAs. Assembler nodes collate the stream of BAs with matching batches, and then append HLF blocks to their ledger and make it available to the scalable committer (SC). Finally, a batcher that suspects misbehavior of the primary may complain to the consensus cluster (e.g. $B4_{s3}$ in shard 3), which given enough distinct complaints will exert control and change the primary of a shard.

To increase the degree of parallelism and speed up transaction ordering, an Arma participant, called thereafter *party*, will host multiple nodes such that each node is dedicated to one of Arma's stages. Notably, a party runs four nodes: (1) a *router* which performs validation tasks; (2) a *batcher* which is in charge of batching; (3) a *consensus node* which participates in consensus; and finally, (4) an *assembler* which is responsible for block assembly. To further scale Arma, we divide the transaction space into *shards* of equal size, and have each party run one batcher per shard. That is, if Arma has 3 shards, then a party will run a router, 3 batchers (instead of one), a consensus node, and an assembler. Now when a client submits a transaction to a party, the transaction is first validated by that party's router, which if the transaction is admissible, deterministically assigns it to a shard and dispatches it to the relevant batcher. In the following, we describe the four stages of Arma in more detail and discuss its BFT properties (see Figure 6).

4.2 Routing

Upon receiving a client's transaction, a router proceeds with *transaction validation* followed by *transaction routing*.

Transaction Validation. The router validates that transactions are well-formed and from authorized clients. For performance, we prefer session-level authentication over HLF's default per-transaction signature verification. Since validation is stateless, router instances can be scaled horizontally.

Transaction Routing. A router maps each valid transaction to an Arma shard and forwards it to the relevant batcher, which is the batcher that the party managing the router has devoted to that shard. The hash function $H_r(TX) \rightarrow shard\text{–}ID$ must be deterministic, consistent across all parties, and uniformly distributed for load balancing; a CRC checksum suffices.

4.3 Batching

Each shard in Arma comprises as many batchers as parties in the system, and designates one batcher as *primary* while the rest of the batchers are *secondaries*. The primary batcher is in charge of bundling transactions it receives into batches. A primary batcher is ousted and a new primary is elected if it (1) fails to include transactions of correct clients into batches or (2) pollutes the batches with bogus transactions (see Section 4.3). We call the time a batcher serves as a primary a *term*. The term of a primary starts when it is elected and it ends when it is removed. In the following, we identify four batcher operations: *transaction dissemination*, *batch attestation*, *batch verification*, and *primary batcher rotation*.

Transaction Dissemination. A correct client considers a transaction to be successfully submitted if it has been delivered to at least $N-f \geq 2f+1$ parties (i.e., the routers of $N-f$ parties acknowledged receipt). This guarantees that the routers of at least $f+1$ correct parties have received the transaction. Each such router will subsequently map the transaction to a shard and forward it to the relevant batcher – i.e., the batcher that is dedicated to the shard and managed by the same party as the router. If the batcher is a *primary*, it adds the transaction to its memory pool, and then extracts a batch of transactions from the memory pool, and persists the batch to disk. A batch is uniquely identified by the tuple $\langle$`shard`, `primary`, `seqN`, `digest`$\rangle$, where `shard` and `primary` identify the batcher generating the batch, `seqN` is a monotonically-increasing sequence number generated by said primary, and `digest` is a secure cryptographic hash of the batch (SHA256 on a serialized form of ordered $\langle$`length`, `value`$\rangle$ tuples, to prevent a second pre-image attack). If the batcher is a *secondary*, it adds the transaction to its memory pool. Secondary batchers replicate every persisted batch from the primary through a reliable broadcast protocol (but not Byzantine atomic broadcast as in [19]). When a secondary batcher receives a batch from the primary, it validates it, persists it to disk, and removes the corresponding transactions from its memory pools. If a correct secondary has in its memory pool a transaction that has not been included in a batch by the primary for a period of time, it will directly forward said transaction to the primary batcher. This ensures that the primary will eventually receive the transaction. If the primary does not include the transaction in a future batch, then a correct secondary can safely conclude that the primary is misbehaving (censoring the transaction), and send a complaint vote against it to the consensus nodes. Figure 6 shows the transaction flow from clients to batchers through the router nodes, and also how transaction batches are disseminated among the batchers in each shard.

Batch Attestation. Once a batch B is persisted to disk, a correct batcher X, be it primary or secondary, creates a *batch attestation fragment* (BAF) by signing the message

$\langle$`shard`, `primary`, `seqN`, `digest`$\rangle$

(the unique batch identifier). Next, the correct batcher sends the BAF to all consensus nodes. This attests to the consensus nodes that said batch is persisted at X.

Batch & Transaction Verification. Note that transactions (TXs) arrive to the batchers in three different ways. First, from the router of the same party; those transactions are verified at the router. Second, a secondary receives batches from the primary, and third, a primary may receive forwarded TXs from a secondary. When a TX is received from a foreign party, it needs to be verified, to prevent that party from injecting junk TXs. This can be done by having the secondary batchers verify all of the transactions in their incoming batches, and primaries verifying forwarded TXs from secondaries. However, a shortcut is available. A secondary may inspect each transaction in an incoming batch, and first check if it is in the memory pool. If it is, said TX has been verified by its own router and need not be verified again. The same can be done by the primary when handling incoming forwarded TXs from secondaries. In both case, when verification is needed, it is the same set of checks done by a router when it processes incoming TXs. When a correct secondary detects a bogus transaction from a primary, it submits a complaint vote against the primary to the consensus nodes. As we will see next, if enough secondaries complain, the primary will be deposed and a new primary selected. This is why the primary needs to verify every TX forwarded to it; otherwise, a faulty secondary may corrupt a primary with a bogus TX. If that TX ends up in a batch, it would cause correct secondaries to depose said primary, which is a safety problem.

Primary Rotation. Batchers utilize the BFT consensus protocol executed by the consensus nodes as a bulletin board that tracks, for each shard, complaint votes against a given primary. We recall that whenever a correct secondary batcher detects misbehavior from the primary, the secondary sends a complaint vote to the consensus nodes for ordering and broadcast. We also recall that a primary misbehaves in one of two ways, either it includes bogus transactions in the batches it disseminates or it censors transactions from correct clients. A complaint vote from a batcher in Arma consists of its signature on the identifiers of the shard and the current primary (or more accurately the term). A primary change is initiated if $f + 1$ distinct batchers of a shard successfully submit complaint votes against the current primary (i.e. the current term) to the consensus nodes. This results in increasing the term by 1 and selecting a new batcher to be a primary. The selection of the new primary uses round robin. By defining the threshold to be $f + 1$, Arma ensures that (1) no coalition of f faulty parties (i.e., batchers) can overthrow a correct primary, and (2) a faulty primary will be overthrown only if at least 1 correct party detects its misbehavior.

4.4 Consensus

A consensus node in Arma participates in a *BFT protocol* to order the messages sent by the batcher nodes and output signed *decisions*.

Batch Attestation Fragments (BAF) Ordering and Arma Block Creation. The consensus nodes totally order BAFs using the BFT protocol. When $f + 1$ distinct BAFs of the same batch (or more accurately the same tuple $\langle$`shard`, `primary`, `seqN`, `digest`$\rangle$) are ordered, the nodes assemble a quorum of signatures over a block header that matches the batch associated with the `digest`. The signed block header contains the digest of the batch, and a hash pointer to the previous block header, thus extending the chain. Each decision may contain multiple block headers.

Ordering and Collecting Complaint Votes. A secondary batcher that detects primary misbehavior and submits a complaint vote to the consensus nodes. The latter nodes order the received vote that will eventually appear in a decision. If the consensus nodes order $f + 1$ distinct complaints on the same primary (from the same shard), the nodes inform the batchers of a term change (primary rotation in that shard). The batchers consume the state from the consensus nodes and track the current term. If a correct batcher sees a term change for its shard, it designates a new primary batcher. As mentioned earlier, Arma uses round-robin to designate the new primary in a *deterministic* fashion. Notice that ordering and outputting the complaint votes using BFT consensus combined with round-robin enable the batchers of a shard to agree on the term change with zero interaction. If the new primary is faulty, then the process of submitting and collecting complaint votes is repeated until a correct primary is selected.

4.5 Assembly

After each round of consensus, one or more Arma block headers are persistently stored by the consensus nodes. Each block header includes a digest referencing a batch, a monotonically increasing sequence number, and the previous block hash. The assembler nodes retrieve the Arma block headers from the consensus nodes, while the batches are retrieved from the batcher nodes. Each Arma block header is attached with a quorum of signatures from the consensus nodes, which the assembler nodes verify (unless it receives it from a consenter of its own party, which is trusted). This and the block sequence number guarantee that all assembler nodes commit the same headers in the same order. For each Arma block header, an assembler node is responsible for retrieving the corresponding batch, attaching them together to form a complete block and subsequently storing it. For performance, assemblers speculatively pre-fetch batches, but they can always request a specific batch by accessing all batchers in a shard.

4.6 BFT Properties of Arma

We informally explain Arma's BFT features. Arma's core is the consensus cluster, which uses state machine replication for two functions. First, it collects BAFs, and once $f + 1$ are received, totally orders the corresponding Batch Attestations (BAs) and block headers. Second, it controls shards by collecting complaints and managing primary rotation. Built on the SmartBFT library [14], the cluster provides safety and liveness (like PBFT [18]) based on the eventual-synchrony model.

To extend safety and liveness to assemblers, they must be able to retrieve the batch for any BA provided by consensus. This is guaranteed because correct batchers (primary or secondary) submit a BAF only after validating and persisting the batch. Since consensus requires $f + 1$ BAFs to emit a BA, at least one correct batcher is guaranteed to have persisted the batch. An assembler can thus always retrieve the batch by querying all batchers in the shard for the batch identifier $\langle$`shard`, `primary`, `seqN`, `digest`$\rangle$. Furthermore, correct Arma clients must submit transactions to all parties. This

ensures censorship resistance, as at least $f + 1$ correct batchers will receive the transaction and can complain if the primary censors it. This mechanism is also key to shard liveness; correct batchers expect batches for incoming transactions, and a primary that stops progress will eventually be replaced. For more details see [28].

5 CBDC: A Tokenized UTXO Approach

This section details a token-based CBDC architecture using the UTXO model, which offers enhanced privacy and concurrency over account-based systems. We implement this at the application layer of the account-based Hyperledger Fabric-X, leveraging its key-value data model via custom user wallets and a specialized "token validator" on endorsing nodes. The flow begins in the user wallet, which securely stores keys and tokens and constructs a transaction request with inputs to be spent, new outputs (for the payee and change), and the payer's digital signature. This request is sent to the token validator, which verifies the signature and formatting before creating a settlement transaction. This new transaction's read-set lists the input keys (to be checked as unspent), and its write-set specifies the new output tokens to be created and marks the input tokens for deletion. This settlement transaction is then sequenced into a block by Fabric-X's ordering service. Finally, committer nodes perform a critical double-spend check by verifying that the read-set's input keys are still unspent. If the check passes, the committers apply the write-set, adding new tokens and removing the spent ones; otherwise, the transaction is rejected as invalid .

6 Experimental Evaluation

This section evaluates our system architecture (Sections 3 and 4). We assess the transaction execute phase in Section 6.1, the order phase in Section 6.2, and the validate phase in Section 6.3. Our implementation uses the Fabric Smart Client [6] and Token SDK [7] for wallets and endorsers. We extended the HLF ordering service [5] to support the Arma consensus protocol [27]. The committers are a distributed service written in Go [10], using GRPC [4] for communication and yugabyteDB [11] for persistence. A custom workload generator was created to stress the system. Performance metrics, specifically throughput and 99th percentile latency, were collected via Prometheus [9]. Each reported data point is an average over at least five minutes, sampled at one-second intervals. Finally, we deployed the components of the settlement engine on IBM Cloud [8] in three different regions, namely, London, Paris, and Milan. The bare-metal servers are equipped with dual 48 core CPUs (Intel(R) Xeon(R) 8260 CPU @ 2.40 GHz), 64 GB RAM, 1 TB SSD (Raid 0), and 10 Gbps network with Ubuntu Linux 20.04 LTS Server.

6.1 Transaction Execution

In this section we analyze the transaction latency observed at the token endorser. The latency for token generation and validity check are 100 milliseconds and 274 milliseconds, respectively. Table 2 shows the latency for threshold signature generation with the combination of ($t = 3, 5, 10$) endorser shares using non-interactive threshold BLS signing implemented using the TSS library [2]. The execute phase scales horizontally as the performed operations are stateless. Our results show that, even when ZKPs are utilized for confidentiality and unlinkability in payment transactions, the *execute* phase does not go above 20 milliseconds, while latency is brought down to a few milliseconds when only anonymity of transactions is assured. Notice that the impact of threshold signing mechanisms is negligible on the overall latency of the execute phase, even when the threshold is 10 nodes.

Number of Endorsers	Signing	Aggregation
$t = 3$	0.147 ±4%	0.124 ±0%
$t = 5$	0.147 ±4%	0.241 ±1%
$t = 10$	0.147 ±4%	0.624 ±1%

Table 2: Latency for threshold signing and signature share aggregation for different thresholds.

6.2 Transaction Ordering

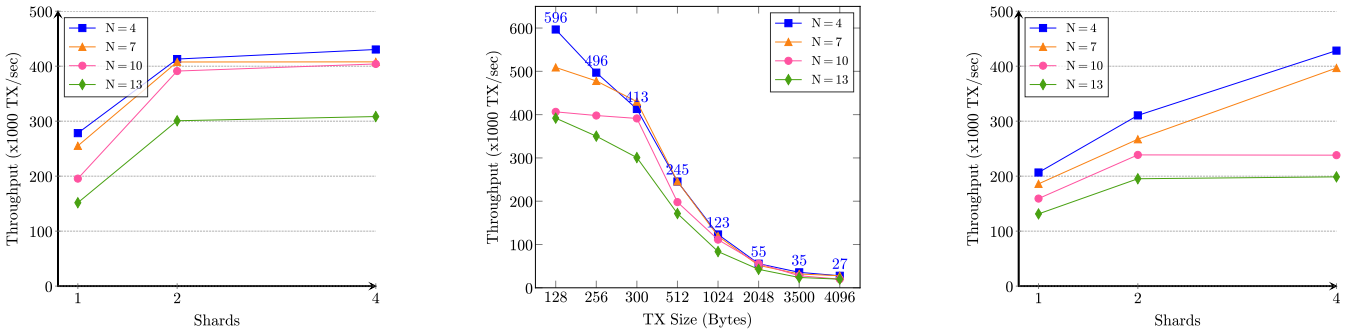
We evaluated the ordering service efficiency of the new consensus algorithm, Arma [27]. Figure 7a shows the service's throughput with varying numbers of parties and shards, maintaining a latency of approximately 0.6 seconds, using 300 B transactions. A peak throughput of 430,000 tps was achieved with 4 parties and 4 shards. Increasing the number of parties decreased performance as expected; for instance, throughput dropped to 308,000 tps with 13 parties and 4 shards. Figure 7b further reports on throughput with varying parties (with 2 shards) and transaction sizes. With increase in the size, the throughput dropped due to higher network bandwidth utilization. Figure 7c additionally outlines the throughput of ordering service running on multiple data centers, maintaining a latency of approximately 0.8 seconds, using 300 B transactions. These experiments ran on 4 AWS sites in Ohio, N. Virginia, N. California, and Oregon. Each VM had 32 vCPU, 64 GiB Memory, running RHEL 10, with EBS General Purpose (SSD), supporting 3000 IOPS and 125 GiB/S bandwidth.

Resource Utilization. Table 3 presents the system's resource utilization footprint. Routers are compute-intensive due to per-transaction processing. Primary batchers fan out to $|P| - 1$ secondaries and one assembler, consuming proportionally more send bandwidth. Assemblers persist all transactions and serve blocks to committers, consuming the most storage. The consenter is the most lightweight component.

Impact of Failures. As shown in Figure 8, the system is resilient to failures. Stopping the consensus leader (Party 1) halts throughput until a new leader is elected, while stopping a shard primary (Party 2 or 3) temporarily drops throughput for that shard

Metric	**R**	**Bp**	**Bs**	**C**	**A**
cpu	34%	22%	26%	2%	4%
memory (GB)	2.5	2.8	2.7	2.5	2.9
disk write (MBps)	-	17	17	0.2	34
network recv (MBps)	39	20	38	0.04	35
network send (MBps)	42	70	18	0.1	34

Table 3: Resource utilization of the ordering service. Load: 100K Tx/Sec, 300B TX size; with: 4 parties, 2 shards.



(a) varying number of shards – 300*B* TX size, *single* data center.

(b) varying transaction size – 2 shards, *single* data center.

(c) running on *multiple* data centers – 300*B* TX size.

Figure 7: Ordering service throughput.

until a new primary is elected via secondary complaints. Stopping a non-primary follower (Party 4) has no performance impact.

6.3 Transaction Validation

We evaluated the enhanced committer's validation phase (Section 3) by measuring the impact of transaction size, invalid signatures, and double spending on transaction throughput and latency. The system components were deployed on multiple servers within the same region. Our default configuration used three signature verifiers, nine validator-committer services co-located with nine distributed database nodes, one coordinator, and one sidecar. For all experiments, the workload was managed to keep latency below one second to prevent queuing from committer overload.

Impact of transaction size. We varied the number of inputs (read-set) and outputs (write-set) per transaction. As shown in Figure 9a, increasing the transaction size from one input/output (1/1) to four (4/4) decreased throughput by 41%, from 474,000 to 280,000 TPS, while the 99th percentile latency remained low, rising only from 83 ms to 101 ms. This performance decline is primarily due to increased overhead at the coordinator's dependency graph, as larger transactions create complex dependencies that require more time for graph construction and updates. Furthermore, a synchronization primitive protecting the graph serializes access, creating a contention point that limits throughput. A bottleneck analysis confirmed the coordinator as the primary limiting factor, as signature verification scales linearly and validator-committer throughput exceeds the observed system throughput.

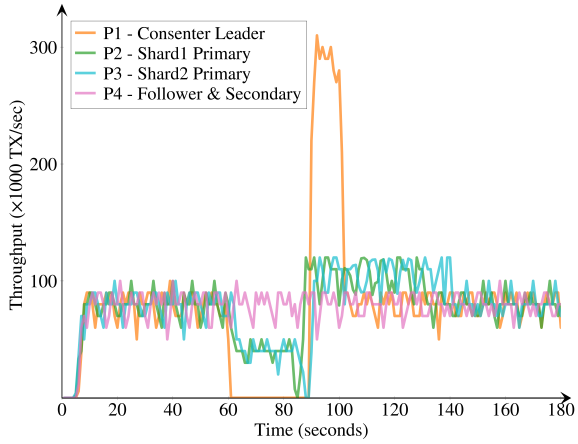


Figure 8: Ordering service throughput with party failures. Load: 80K Tx/Sec, 300B TX size; with: 4 parties, 2 shards.

Impact of faulty transactions. *Invalid signatures.* We analyzed transactions with invalid threshold signatures. Counter-intuitively, as invalid transactions rose from 0% to 30%, throughput increased from 419,000 to 459,000 tps (Figure 9b), a ~10% gain, because invalid signatures are rejected early, reducing validator-committer workload. The 99th percentile latency stayed between 75–83 ms, as only identifiers and statuses are recorded for invalid transactions.

Double spendings. The second scenario analyzed double spend incidents. Unlike invalid signatures, this significantly decreased throughput (Figure 9c): even at 10% double spends, throughput dropped 33% (from 419,000 to 280,000 tps) with 99th percentile latency rising from 85 ms to over 1,100 ms. At 30%, throughput fell 38% to 260,000 tps. Conflicts create serialization chains in the dependency graph that block dependent transactions (convoy effect), and each conflict doubles commit work due to rollback and retry.

Metric	coordinator	signature verifier	vcservice+ database-node
cpu	17%	55%	49%
memory	3 GB	1 GB	32 GB
disk	-	-	29%
network send	2.4 Gbps	0.06 Gbps	0.8 Gbps
network recv	1.55 Gbps	0.3 Gbps	0.86 Gbps

Table 4: Resource utilization of committer.

Resource Utilization. Table 4 shows distinct profiles: the signature verifier is most CPU-intensive (55%), the validator-committer+database unit dominates memory (32 GB) and disk (29%), and the coordinator is primarily network-bound (2.4 Gbps send) with low CPU (17%).

Impact of Failures. As shown in Figure 10, a single VC failure had no impact, two failures dropped throughput to 160,000 TPS

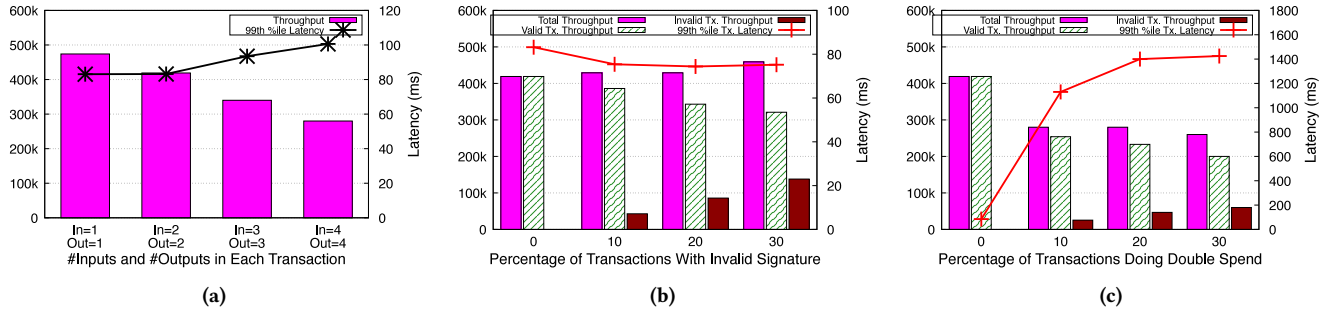


Figure 9: Impact on transaction throughput and latency of (a) varying number of inputs and outputs per transactions; (b) varying mix of valid and invalid transaction signatures; and (c) varying number of double spend transactions.

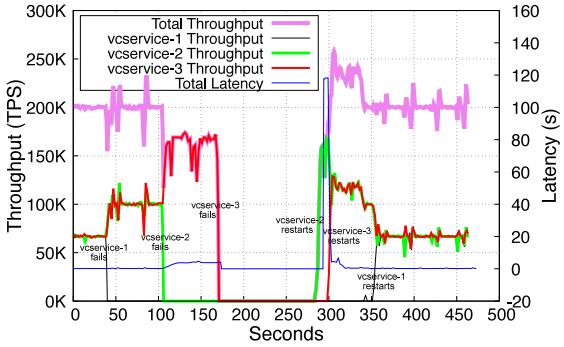


Figure 10: Impact of validator-committers failure on performance.

with 3s latency, and three failures halted processing entirely. Upon restart, a temporary latency spike occurred before returning to normal. Signature verifier failures showed comparable behavior. Coordinator or sidecar failure immediately halted all processing, with recovery after restart.

6.4 Fabric-X vs. Fabric: A Performance/Cost Analysis

Fabric-X achieves over 400,000 TPS—a 200-fold increase over Fabric's 2,000 TPS baseline—at the cost of 21 servers per organization compared to Fabric's 2, reflecting a clear trade-off between throughput and infrastructure.

7 Related Work

Performance Optimization of Hyperledger Fabric. Fabric's performance bottlenecks are well-studied, with research ranging from foundational analysis to architectural redesigns. Initial benchmarking work identified the commit phase as a primary bottleneck [36], while other analyses framed Fabric's concurrency challenges as classic database problems [33]. Building on these insights, targeted optimizations have improved performance within the existing model. Gorenflo et al.'s influential "FastFabric" [22] achieved 20,000 TPS through fixes like caching and removing redundant checks. Other efforts have focused on increasing parallelism by proposing hybrid execution models like XOX Fabric [21] or by applying transactional dependency analysis [32]—a concept that informs our own work. More recent research has proposed fundamental architectural changes. Notably, Thakkar et al. introduced "Sparse Peers" [35], where peers process only a subset of transactions, demonstrating that rethinking the peer's role is a viable path to scalability. While these efforts offer valuable gains, they focus on specific subsystems. Our work advances this trajectory with a holistic, end-to-end redesign—decoupling the peer into microservices and introducing a scalable ordering service—achieving performance an order of magnitude beyond prior work.

CBDC Architectures & Privacy-Preserving Payments. Prior work on CBDCs includes early centralized models [20] and high-throughput processors like the Hamilton Project [26], which lack full decentralization and BFT-resilience. Privacy-focused systems like Platypus [39] and PEReDi [24] offer compliance features but lack parallel processing or experimental validation. Technologies like Zerocoin [30] and Zcash [15] use expensive cryptography impractical at scale without a performant platform. Our work provides this foundation, enabling the integration of such privacy features and complementing auditable privacy systems [13].

Ordering Service. Performance bottlenecks of high throughput leader-based BFT protocols stem mainly from need of the leader to disseminate a batch of transactions to all participants. Some systems, like Honeybadger [31] and Mir-BFT [34], mitigate this problem by having multiple leaders. Recently Danezis et al. proposed "Narwhal and Tusk" [19] which separates consensus from dissemination and uses HotStuff [40] as the core consensus protocol. Arma adopts the design principles of "Narwhal and Tusk", improves upon them, and uses the SmartBFT library [14] as the core consensus protocol, as it is more suitable to the permissioned-blockchain setting.

8 Conclusion

This paper presented Fabric-X, a fundamental re-architecture of Hyperledger Fabric featuring a decoupled microservices peer with parallelized commit flow and Arma, a scalable BFT ordering service. This work has been open-sourced as part of Hyperledger [1]. Our evaluation confirms a peak throughput exceeding 200,000 TPS—a multiple-orders-of-magnitude improvement over standard Fabric—demonstrating that permissioned DLTs can meet the demands of modern financial systems and CBDC deployments.

[1]Project repositories are available at: github.com/hyperledger/fabric-x and related repositories therein.

References

[1] 2021. WHOLESALE CENTRAL BANK DIGITAL CURRENCY EXPERIMENTS WITH THE BANQUE DE FRANCE. https://www.banque-france.fr/sites/default/files/media/2021/11/09/821338_rapport_mnbc-04.pdf.

[2] 2024. Threshold Signature Scheme library. https://github.com/IBM/TSS/.

[3] 2025. CBDC Tracker. https://cbdctracker.org/. Accessed: 2025-01-15.

[4] 2025. gRPC: A High Performance, Open Source Universal RPC Framework. https://grpc.io. Accessed: 2025-01-15.

[5] 2025. Hyperledger Fabric. https://github.com/hyperledger/fabric. Accessed: 2025-01-15.

[6] 2025. Hyperledger Labs Fabric Smart Client. https://github.com/hyperledger-labs/fabric-smart-client. Accessed: 2025-01-15.

[7] 2025. Hyperledger Labs Fabric Token SDK. https://github.com/hyperledger-labs/fabric-token-sdk. Accessed: 2025-01-15.

[8] 2025. IBM Cloud. https://www.ibm.com/cloud. Accessed: 2025-01-15.

[9] 2025. Prometheus: Monitoring System and Time Series Database. https://prometheus.io. Accessed: 2025-01-15.

[10] 2025. The Go Programming Language. https://go.dev. Accessed: 2025-01-15.

[11] 2025. YugabyteDB: Distributed SQL Database. https://github.com/yugabyte/yugabyte-db. Accessed: 2025-01-15.

[12] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, Article 30. doi:10.1145/3190508.3190538

[13] Elli Androulaki, Jan Camenisch, Angelo De Caro, Maria Dubovitskaya, Kaoutar Elkhiyaoui, and Björn Tackmann. 2020. Privacy-Preserving Auditable Token Payments in a Permissioned Blockchain System. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies (AFT '20)*. ACM. doi:10.1145/3419614.3423259

[14] Artem Barger, Yacov Manevich, Hagar Meir, and Yoav Tock. 2021. A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2021, Sydney, Australia, May 3-6, 2021*. doi:10.1109/ICBC51069.2021.9461099

[15] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society. doi:10.1109/SP.2014.36

[16] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* (1981). doi:10.1145/356842.356846

[17] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 355–362. doi:10.1109/DSN.2014.43

[18] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[19] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. ACM. doi:10.1145/3492321.3519594

[20] George Danezis and Sara Meklejohn. 2016. Centrally Banked Cryptocurrencies. In *Network and Distributed System Security Conference*.

[21] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. 2020. XOX Fabric: A hybrid approach to blockchain transaction execution. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 1–9. doi:10.1109/ICBC48266.2020.9169478

[22] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2019. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. doi:10.1109/BLOC.2019.8751452

[23] Hyperledger Fabric v3 release notes 2023. Hyperledger Fabric. https://github.com/hyperledger/fabric/releases/tag/v3.0.0.

[24] Aggelos Kiayias, Markulf Kohlweiss, and Amirreza Sarencheh. 2022. PEReDi: Privacy-Enhanced, Regulated and Distributed Central Bank Digital Currencies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM. doi:10.1145/3548606.3560707

[25] Library Repository 2024. The SmartBFT Library, Open-Source Repository. https://github.com/hyperledger-labs/SmartBFT.

[26] James Lovejoy, Madars Virza, Cory Fields, Kevin Karwaski, Anders Brownworth, and Neha Narula. 2023. Hamilton: A High-Performance Transaction Processor for Central Bank Digital Currencies. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association. https://www.usenix.org/conference/nsdi23/presentation/lovejoy

[27] Yacov Manevich. 2023. Arma: Byzantine Fault Tolerant consensus with Linear Scalability. *ArXiv* abs/2312.13777 (2023). https://arxiv.org/abs/2312.13777

[28] Yacov Manevich, Hagar Meir, Kaoutar Elkhiyaoui, Yoav Tock, and May Buzaglo. 2024. Arma: Byzantine Fault Tolerant Consensus with Horizontal Scalability. arXiv:2405.16575 [cs.DC] https://arxiv.org/abs/2405.16575

[29] Yacov Manevich, Yoav Tock, and Hagar Meir. 2024. Hyperledger Fabric v3: Delivering Smart Byzantine Fault Tolerant consensus. https://www.lfdecentralizedtrust.org/blog/hyperledger-fabric-v3-delivering-smart-byzantine-fault-tolerant-consensus.

[30] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*. doi:10.1109/SP.2013.34

[31] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. doi:10.1145/2976749.2978399

[32] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A Transactional Perspective on Execute-order-validate Blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 543–557. doi:10.1145/3318464.3389693

[33] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 105–122. doi:10.1145/3299869.3319883

[34] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolic. 2022. [Solution] Mir-BFT: Scalable and Robust BFT for Decentralized Networks, Vol. 2. Journal of Systems Research. http://dx.doi.org/10.5070/SR32159278

[35] Parth Thakkar and Senthilnathan Natarajan. 2021. Scaling Blockchains Using Pipelined Execution and Sparse Peers. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. ACM. doi:10.1145/3472883.3486975

[36] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 2018. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 264–276. doi:10.1109/MASCOTS.2018.00034

[37] Alin Tomescu, Adithya Bhat, Benny Applebaum, Ittai Abraham, Guy Gueta, Benny Pinkas, and Avishay Yanai. 2022. UTT: Decentralized Ecash with Accountable Privacy. Cryptology ePrint Archive, Paper 2022/452. https://eprint.iacr.org/2022/452 https://eprint.iacr.org/2022/452.

[38] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. 2022. BFT in Blockchains: From Protocols to Use Cases. *ACM Comput. Surv.* 54, 10s, Article 209 (Sept. 2022), 37 pages. doi:10.1145/3503042

[39] Karl Wüst, Kari Kostiainen, Noah Delius, and Srdjan Capkun. 2022. Platypus: A Central Bank Digital Currency with Unlinkable Transactions and Privacy-Preserving Regulation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM. doi:10.1145/3548606.3560617

[40] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness *(PODC '19)*.