Fabric-X: Redesigning Hyperledger Fabric Architecture for High-throughput Regulated Asset Exchange Applications

E. Androulaki, M. Brandenburger, M. Buzaglo, A. De Caro, K. Elkhiyaoui, A. Filios, L. Funaro, Y. Manevich, H. Meir, S. Natarajan, M. Sethi, Y. Tock
IBM Research

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, while conceptually suitable, are limited by architectural bottlenecks in their monolithic peer design and consensus mechanisms, preventing them from achieving the required scale.

This paper presents a fundamental re-architecture of Hyperledger Fabric that addresses these challenges end-to-end. 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 200,000 transactions per second (TPS)—a two-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.

PVLDB Reference Format:
E. Androulaki, M. Brandenburger, M. Buzaglo, A. De Caro, K. Elkhiyaoui, A. Filios, L. Funaro, Y. Manevich, H. Meir, S. Natarajan, M. Sethi, Y. Tock
IBM Research. Fabric-X: Redesigning Hyperledger Fabric Architecture for High-throughput Regulated Asset Exchange Applications. 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 URL_TO_YOUR_ARTIFACTS.

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 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 [citations]. Early results indicate the potential for 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. It involves distributing operational components across multiple independent nodes, reducing reliance on a single point of failure. This does not imply decentralized governance but rather a design choice to mitigate risks like cyberattacks, natural disasters, and insider threats. For example, a CBDC system with a centralized architecture in a single data center is highly vulnerable to disruptions. In contrast, a decentralized deployment spreads components across dispersed nodes, incorporating redundancy and mechanisms like Byzantine fault tolerance to maintain security and functionality. While the central bank governs the system, a decentralized deployment enhances resilience, protects against threats, and fosters stakeholder trust. Centralized governance and decentralized deployment together ensure regulatory oversight and operational stability.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Consequently, financial institutions [1] and the research community (e.g., [2]) 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 open to anyone and typically rely on energy-intensive consensus mechanisms like Proof-of-Work. The public transparency of data and transaction logic in these systems raises concerns about data leakage. This openness, coupled with the limitations of serial transaction execution, restricts their capacity and makes them less suitable for regulated financial applications.

Permissioned blockchains, such as Hyperledger Fabric [citation], 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. In this architecture, a transaction's lifecycle consists of three pipelined phases: (1) Execution: A client sends a transaction proposal to endorsing peers specified by an endorsement policy. These peers execute the smart contract triggered by the proposal, recording its effects in terms of read and write sets. Transactions from different clients may be parallelized during this phase. (2) Ordering: A consensus protocol establishes a total order for the endorsed transactions, grouping them into blocks. This order is then broadcast to all peers. (3) Validation: Each peer validates the state changes from the endorsed transactions with respect to the endorsement policy and serializability.

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:

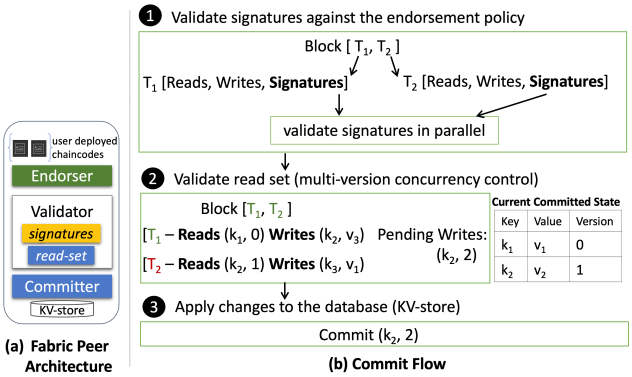


Figure 1: Architecture of Fabric Peer and Commit Flow within a Peer

(1) A High-Performance Fabric Redesign: We decouple the monolithic Fabric peer into a set of independently scalable microservices for endorsement, validation, and committing. By introducing parallel validation of transactions across blocks using a dependency graph and employing threshold signatures to reduce computational overhead, our design breaks existing bottlenecks and achieves a throughput of over 100,000 TPS.
(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 with strong identity management, non-repudiation, and accountability mechanisms. It utilizes an innovative *execute-order-validate* [3] transaction model, which replaces the traditional *order-execute* approach. In this model, transactions are speculatively executed before ordering. Any transaction using stale data is rejected during the validation. This design supports parallel execution, improving throughput, and allows developers to write execution logic in their preferred programming language.

Figure 1(a) illustrates the architecture of a Fabric peer, which includes components such as an endorser, signature validator, read-set validator, committer, local key-value database, and user-deployed chaincodes. Figure 2 depicts Hyperledger Fabric's (HLF) transaction flow, which begins with chaincodes—smart contracts written in conventional programming languages. They execute during the transaction's execution phase. Each chaincode is tied to specific endorsers according to an endorsement policy, which is a subset of network participants responsible for its execution. Chaincodes operate within defined namespaces that specify the ledger partitions they can modify and are subject to validation through the endorsement policy.

The transaction process begins when a client submits a transaction proposal to the endorsers, initiating chaincode execution.

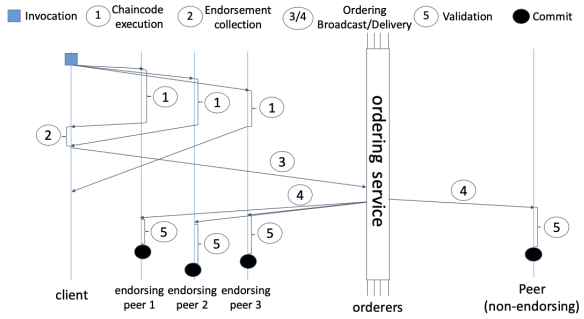


Figure 2: Transaction Lifecycle in Hyperledger Fabric.

The endorsers speculatively execute the chaincode, producing a read-set (dependencies) and a write-set (proposed changes), collectively forming a read-write set. This set is signed by the endorsers and returned to the client. The client then constructs a transaction that includes the signed read-write set along with the endorsers' signature.

In the ordering phase, orderers—authorized network participants—determine the sequence of transactions. These transactions are batched into blocks and distributed to peers.

Upon receiving a block, each peer independently executes the commit process shown in Figure 1(b). This involves three steps:

(1) *Signature Validation*: The signature validator verifies that the endorsers' signatures comply with the chaincode's endorsement policy.
(2) *Read-Set Validation*: Transactions that pass signature validation proceed to this phase, where the read versions of keys are compared with their committed versions. If the versions match, the transaction's write-set is added to the pending writes list. For example (refer to Figure1(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 pending writes are applied to the local key-value database, finalizing the transaction.

This structured process ensures the integrity and consistency of the ledger, even under concurrent operations.

2.2 Assessing Hyperledger Fabric for CBDC Requirements

The architectural design of Hyperledger Fabric (HLF) provides several features that make it a strong conceptual candidate for building CBDC and other regulated financial systems. Its permissioned model ensures that only known, authorized participants can access the network, which is a prerequisite for financial compliance. The platform's core features offer a suitable starting point:

Flexible Smart Contracts: Chaincodes in HLF can be written in general-purpose programming languages to implement essential CBDC operations like issuance, transfer, and compliance checks. Modular Consensus: HLF's modular design supports swappable consensus protocols, allowing a system to transition between Crash Fault Tolerant (CFT) and Byzantine Fault Tolerant (BFT) mechanisms to adapt to evolving security and performance requirements. Granular Control: Compared to other enterprise platforms, HLF's endorsement policies allow for granular control over transaction validation, providing a mechanism to enforce complex business rules and security checks.

While these conceptual strengths make HLF an attractive foundation, its standard implementation struggles to meet the stringent, real-world demands of a large-scale CBDC. When benchmarked against the required throughput, latency, and scalability for a national payment system, the current architecture reveals several fundamental bottlenecks. Therefore, addressing these implementation-level deficiencies is essential before the platform can be considered viable for such a critical role.

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.

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

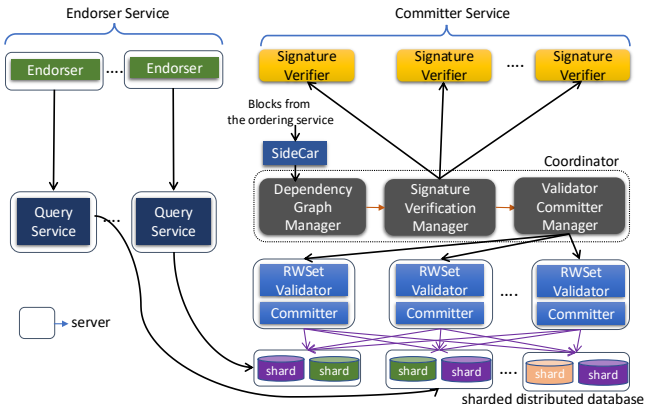


Figure 3: Proposed Fabric Peer Architecture

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 3.5.

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 significant 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. In summary, the proposed architecture directly addresses the first three limitations detailed in Section 2.3. It enhances scalability through the independent deployment of services, reduces resource contention via service isolation, and improves data management performance by incorporating a distributed database with sophisticated concurrency control. Consequently, this re-architected Fabric Peer provides a foundation for a more robust, scalable, and efficient blockchain platform.

3.2 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 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.

3.3 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

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)	✓

be validated and committed/aborted before T_j' can be considered for validation. We define the following three dependencies which sufficient to represent any potential conflict that could arise from concurrent access to shared state

(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 previous version of state k. If T_i is valid, T_j must be invalid because it read an outdated version.

(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.

It's important to note that 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 acyclic nature 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.

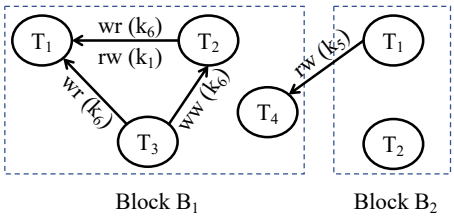


Figure 4: Dependency graph of transactions in Table 1.

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.

3.4 Consistency Proof

Our goal is to prove that parallel validation and commitment of transactions using our dependency graph produces the same final state as serial execution. In serial execution, transactions are processed one by one, each validated against the state resulting from the preceding ones. In contrast, our parallel approach validates transactions concurrently, respecting dependency constraints, and commits them in a topological order derived from the graph. We assume that validation and commitment operations are deterministic and that our dependency graph accurately captures all read-write, write-read, and write-write dependencies between transactions.

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, it can be validated immediately in parallel against the initial state. While this differs from serial execution, where it's validated against the state after committing k transactions, the lack of dependencies signifies that there is no shared state between T_{k+1} and the prior transactions. Consequently, the validation result and the final state remain consistent regardless of when T_{k+1} is committed after T_k.

Therefore, in both scenarios, the outcome for T_{k+1} is consistent between serial and parallel execution. By the principle of mathematical induction, we conclude that for any set of transactions, parallel validation and commitment using the dependency graph produce the same final state as serial execution. The dependency graph ensures that the order of validation and commitment respects all inter-transaction dependencies, effectively mimicking serial execution while enabling the performance benefits of parallelism. This guarantees the consistency and integrity of the blockchain's state under our proposed parallel execution model.

3.5 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 3.3 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 sends verified transactions to the Validator Committer Manager (in step ❺), which routes them to available Validator-Committer services (in step ❻). The underlying data is organized across several tables. There are two primary system tables: one for storing the status of each transaction (`txID, status`), with `txID` being unique, and another for storing namespace details, including its endorsement policy (e.g., verification key) and version. Additionally, each namespace has its own data table with columns for `key`, `value`, and `version`, where the `key` column has a unique constraint.

These Validator-Committer services execute a three-phase pipelined process to validate and commit transactions using this data structure (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, which are identified by a `nil` version signifying the state did not exist when the contract was executed; and existing state updates, which are identified by a `non-nil` version.

(2) *Validator Phase*: The reads are then validated against the committed version of the data in the ledger. This separation of inserts and updates enables an optimization: for new state inserts (nil version), the service does not need to check if the state already exists to detect staleness. Transactions that fail validation due to stale reads are marked with an invalid status, and their associated writes are discarded.

(3) *Commit Phase*: Valid transactions proceed to the final commit phase. The service commits the results by first storing the transaction's final status and its identifier in the (txID, status) table. It then applies the writes to the relevant namespace table. For new state inserts, it adds a new row with the version set to 0; if this key already exists, the database's unique constraint raises a conflict that invalidates the transaction. For existing state updates, it modifies the row for the given key with the new value while incrementing its current version by 1.

Because the dependency graph guarantees that concurrently processed transactions are conflict-free, a straightforward concurrency control mechanism, such as Read Committed Isolation, can be employed in the distributed database to enhance performance.

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 steps 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 failure/recovery, and forwards the completed block to registered clients, informing them whether their transactions were committed or aborted.

3.6 Failure and Recovery

During the processing lifecycle, any of the microservices, including the sidecar, coordinator, signature verifier, and validator-committer, can potentially fail. The system is designed with specific mechanisms to handle these failures gracefully and ensure correctness.

Failure of a Signature Verifier: The Signature Verifier Manager maintains a list of transactions assigned to each verifier for processing. If a verifier fails and its connection drops, the manager detects this and automatically resubmits the pending transactions to another available verifier. This design also robustly handles transient network issues. If the original verifier was not actually down and eventually returns a response, the manager will no longer find a matching pending transaction in its list for that verifier and will simply ignore the late response, preventing any duplicate processing.

Failure of a Validator-Committer Service: In a similar manner, the Validator Committer Manager maintains a list of transactions sent to each Validator-Committer (VC) service. If a VC service fails, its pending transactions are resubmitted to another

one. This can create a scenario where, due to a transient network issue, the same transaction is processed by more than one VC service. This is addressed at commit time to prevent incorrect validation or duplicate writes. When a VC service attempts to store the final transaction status, it checks if a status for that specific txID already exists in the database. If the txID, block number, and transaction index match an existing record, the service identifies it as a resubmission from a failure/recovery event and reuses the already committed status, ensuring the commit operation is idempotent.

Failure of Coordinator: The sidecar periodically stores the last fully committed block number in sequence into the state database. When the coordinator restarts after a failure, it will read the last fully committed block number and set it as the next expected block number from th e sidecar. As a result, the sidecar would start to pull blocks from the ordering service, starting with the next expected block by the coordinator. Given that the architecture allows validation and commit of transactions across blocks, it is possible that some transactions in the block being fetched are already committed. Note that our validator-committer services allow resubmission of a transaction and detect the existing status by comparing the txID, block number, and transaction index within the block and return the existing status without actually recommitting the transaction.

Failure of Sidecar: When the sidecar fails and restarts, it would ask the coordinator for the next expected block number and compare it against the last block stored in its block store. If there is a gap, it would fetch those blocks from the ordering service and statuses of those transactions from the state database through the coordinator to bring the block store up to date. Then, it would start to fetch from the next expected block from the ordering service.

Failure of Nodes in Distributed Database: To handle database node failures in a distributed system, the replication of shards is employed. When a shard fails, other replicas would exist. When the whole state database fails or gets corrupted, the whole state has to be rebuilt by fetching blocks from the beginning from the ordering service.

4 A SCALABLE OREDERING SERVICE

In this section we introduce "Arma", a scalable, high performance Byzantine fault tolerant (BFT) ordering service for HLF. Our goal was to design an ordering service with an order of magnitude better throughput than what is currently in HLF, while maintaining reasonable latency for the CBDC use case. We also strive to maintain API compatibility with the current HLF ordering service, although some minor changes were introduced.

Starting from v3, HLF provides the option for a BFT ordering service (see [4, 5]). HLF's ordering service uses the SmartBFT library [6, 7], which is based on PBFT [8] and BFT-SMaRt [9]. In HLF v3, the ordering service is a cluster of ordering service nodes (OSNs), each of which is a process (server) which may be governed by a different organization. In order to tolerate f Byzantine failures, the cluster must have at least $n = 3f + 1$ OSNs (see top of Figure 5). In Arma, each HLF v3 OSN is replaced by a "party", which is a distributed implementation of an OSN. A party has multiple sub-components, is controlled by a single organization, and is a "unit of failure"; that is, if any sub-component of the party is faulty, we consider the party as faulty. As in SmartBFT, Arma requires $n = 3f + 1$ parties

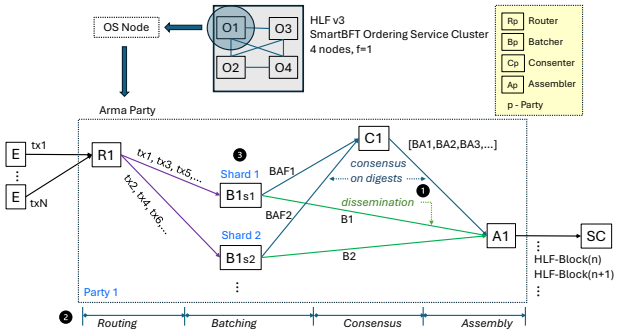


Figure 5: Arma party composition. An Arma party is equivalent to an HLF v3 OS Node. An Arma party is composed of 4 sub-components: a router (R), one or more batchers (B), a consenter (C), and an assembler (A). Endorsing clients (E) submit transactions to the router. The router spreads transactions across 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 signed digest (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.

in order to tolerate f Byzantine faults (for more details on BFT in blockchain systems see [10] and references therein).

In Arma, three design principles help us achieve the performance goals. First, we separate transaction ordering from transaction dissemination (Figure 5 ❶), ordering digests rather than full transactions, thus alleviating network bottlenecks. Second, we decompose transaction ordering into multiple sequential stages that can operate in parallel (Figure 5 ❷), pipelining the ordering service. And third, we divide key elements of processing which can be done independently of each other into shards (Figure 5 ❸), which can execute in parallel. Pipelining and sharding are established techniques that increase parallelism, and allow us to scale the performance of Arma by adding hardware resources. The design of Arma draws inspiration from "Narwal and Tusk" [11], yet is adapted to the special needs of the HLF architecture.

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 [12], 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.

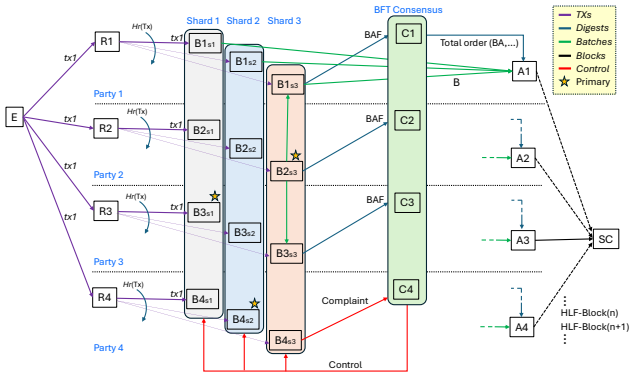


Figure 6: Arma architecture. A correct endorsing client (E) tries to submit a TX to all parties. Routers validate and dispatch the TX to a shard according to hash function H_r. The primary batcher in a shard (e.g. $B2_{s3}$ in shard 3) bundles TXs in a batch, persists it, and broadcasts it to the secondary 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 receive a stream of BAs from consensus and collate them with matching batches they pull from the shards. Assemblers 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.

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.

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 Aram, 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 details 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. Transaction validation consists of checking that the received transaction is well formed and submitted by an authorized Arma client. Client authentication and authorization (A&A) can be done on the session level or per transaction. The early identification and exclusion of invalid transactions protects Arma's resources against misuse and denial of service attacks. The approach in HLF is to verify a client signature on every transaction, and to carry the client certificate in every transaction. This approach is costly; although we support it, for performance we prefer session level A&A. Nevertheless, given that transaction validation is stateless, it is amenable to parallelization and horizontal scaling; an Arma party may add router instances to scale the CPU intensive task of verifying signatures, and therefore, speed up the processing of incoming transactions.

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 $H_r(TX) \rightarrow shard\text{–}ID$ should be consistent across all parties, and deterministic. In other words, on input of a transaction, a correct execution of this mapping process will always yield the same target $shard\text{–}ID$. In order to effectively load balance the shards, the output of this mapping should also be uniformly distributed. The hash function H_r does not need to be secure; an efficient candidate could be based on CRC checksum.

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 4.3.4). 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*.

4.3.1 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 poll, 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 [11]). 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.

4.3.2 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

$$\langle \texttt{shard}, \texttt{primary}, \texttt{seqN}, \texttt{digest} \rangle$$

(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.

4.3.3 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.

4.3.4 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 (BAFs and complaints) and output signed *decisions*.

4.4.1 Batch Attestation Fragments 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 $\langle \texttt{shard}, \texttt{primary}, \texttt{seqN}, \texttt{digest} \rangle$) have been totally ordered, the consensus nodes collaborate to 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.

4.4.2 Ordering and Collecting Complaint Votes. A secondary batcher that detects primary misbehavior generates a complaint vote and submits the 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 batchers 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

Here we informally explain the BFT features of Arma. The core of the Arma protocol is the consensus cluster, which implements, using the state machine replication paradigm, two basic functions. First, it collects BAFs, counts them, and when they cross the $f + 1$ threshold, totally orders BAs with corresponding block headers. Second, it controls the function of every shard by collecting complaints and changing primaries if needed. The consensus cluster is built using the SmartBFT library [12] which like PBFT [8], provides safety and liveness based on the eventual-synchrony model. To carry the safety and liveness features all the way to the assemblers we must show that given a BA and corresponding block header provided by consensus, an assembler would be able to find the corresponding batch in a shard. This is implemented jointly by the behavior of the batchers and the state machine in the consensus cluster: correct batchers submit a BAF to consensus after they had validated and persisted a batch (either as primary or secondary); since consensus emits a BA after collecting $f + 1$ signed BAFs, we are guaranteed that at least one correct batcher has said batch. An assembler can always contact all batchers in a shard and request each one for a certain batch identifier $\langle$`shard`, `primary`, `seqN`, `digest`$\rangle$, and get at least one copy of said batch.

Moreover, a correct Arma client is expected to try and submit to all parties. Censorship resistance in Arma relies on that, since this guaranties that at least $f + 1$ correct batchers in a shard would get the transaction, and hence would be able to complain if the primary censors said transaction. This is also the key to liveness in the shards, since given incoming transactions, correct batchers expect batches from the primary. If a primary stops progress, it would be eventually replaced. For a more details see [13].

5 CBDC: A TOKENIZED UTXO APPROACH

This section introduces a token-based architecture for a Central Bank Digital Currency (CBDC). Here, a token, defined as (owner, value), represents the CBDC asset. A distributed ledger maintains the state of these tokens, updated through "Issue" (creation) and "Transfer" (ownership change) operations, submitted as transactions. Our architecture employs the Unspent Transaction Output (UTXO) model, offering enhanced privacy and concurrency control compared to traditional account models. Although Hyperledger Fabric primarily uses an account model, its key-value data model and chaincode functionality allow us to implement a UTXO model at the application layer. This involves user wallets and a specialized "token chaincode" on endorser nodes.

This section details the transaction flow, covering both Issue and Transfer transactions. The transaction flow begins with a user wallet creating a client transaction. This is submitted to endorser nodes for processing by the token chaincode, which validates it and creates a settlement transaction. This is then sent to Hyperledger Fabric's ordering service, which orders it into a block. Finally, committer nodes perform a final validation before committing the transaction to the ledger.

5.1 Transaction Flow

5.1.1 User Wallet and Client Transaction Creation. The user wallet, held by each participant (issuers, payers, or payees), is responsible for translating user instructions into transactions and securely storing sensitive information. The wallet stores the user's unique identifier (eid), their secret key (sk) for signing transactions, and for each owned token τ, the token itself (containing owner and value) along with its unique key in the ledger (key—hash of the token τ).

Users interact with their wallets to initiate either Issue or Transfer transactions. For an Issue transaction, initiated by an authorized issuer, the wallet creates a transaction containing the new tokens to be created, each defined as (owner, value), along with the issuer's digital signature, cryptographically proving their authorization. For a Transfer transaction, initiated by a payer to transfer value to a payee, the wallet constructs a transaction containing the tokens to be spent along with their corresponding unique keys in the ledger, the outputs which are the tokens designated for the payee and the "change" returned to the payer, and signatures generated by the payer, cryptographically proving their ownership of the input tokens.

5.1.2 Token Chaincode: Validation and Settlement Transaction Creation. The client transaction is submitted to the token chaincode, which is responsible for validating the transaction and creating a corresponding settlement transaction. For Issue transactions, the chaincode verifies the issuer's signature, ensuring that only authorized entities can create new tokens. For Transfer transactions, the chaincode performs several checks. It verifies the correctly computed transaction ID (TxID). It also checks that the inputs are correctly formatted as (token, key) pairs, with the key being the hash of the token. Finally, it verifies the signatures using the public keys of the owners of the input tokens.

If all validation checks are successful, the chaincode creates a settlement transaction. For Issue transactions, the read-set is empty, and the write-set includes entries to add the new output tokens to the ledger: $< key, ser\_out >$, where key is a unique key for the output token, and ser_out is the serialized representation of the output token. For Transfer transactions, the read-set includes entries of the form $< key, 0 >$, indicating that when the transaction is committed, the ledger state for each key should be at version 0 (unspent). The write-set includes entries to add the new output tokens to the ledger and entries to mark the input tokens as deleted.

5.1.3 Hyperledger Fabric: Settlement Transaction Validation. The signed settlement transaction is processed by Hyperledger Fabric. The ordering service orders transactions into blocks, and committers perform the final validation. For Issue transactions, committers simply apply the write-set as there are no double-spending checks applicable. For Transfer transactions, this includes a critical double-spending check. Committers verify that the input tokens referenced in the read-set ($keys$) are indeed at version 0 in the ledger. If not, the transaction is rejected. If the double-spending check (and other endorsement checks) are successful, the committers apply

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 (Section ??).

the write-set to the ledger: adding new output tokens and marking input tokens as deleted.

6 EXPERIMENTAL EVALUATION

In this section we evaluate the performance characteristics of our system's architecture as described in Section 3 and 4. In Section 6.1 we evaluate the transaction *execute* phase capturing stateless checks of transaction validity. We proceed in Section 6.2 with the performance evaluation of the *order* phase. Finally, in Section 6.3 we evaluate the *validate* phase with our enhanced HLF committers

Experimental Setup We implemented the user wallets, the endorsers, and the certifiers using the Fabric Smart Client [15] and the Token SDK [16], and extended the HLF ordering service [17] to support the Arma consensus protocol [18]. We implemented the committers as a distributed service in Go [19] using GRPC [20] for network communication, and yugabyteDB [21] as shard database to persist ledger states. To evaluate various aspects of the settlement engine, we implemented a workload generator that can produce synthetic workloads to simulate millions of users and stress individual components of the settlement engine.

Performance metrics (i.e., throughput and 99^{th} percentile latency) are collected via Prometheus [22] with a sample rate of one second. Each reported data point is the average of at least fifteen minutes running time after a warm-up phase.

Finally, we deployed the components of the settlement engine on IBM Cloud [23] 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 [24]. 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.

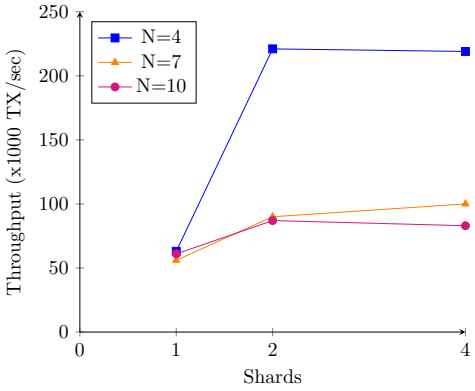


Figure 7: Ordering service throughput with the Arma consensus protocol.

6.2 Transaction Ordering

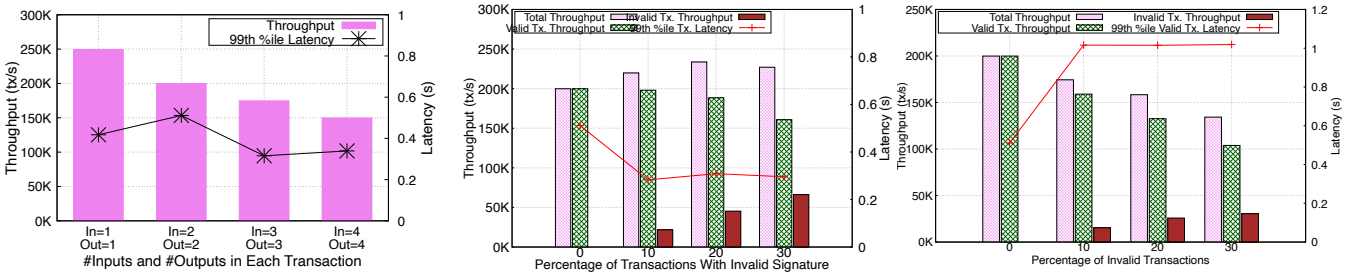
Next, we investigate the efficiency of the settlement engine's ordering using the newly proposed consensus algorithm, Arma [18]. Figure 7 illustrates the throughput of the ordering service instantiated with a varying number of parties and a varying number of shards, with a latency of approximately 1 second.

We observe that the best throughput of up to 220, 000 tx/s (with about 1 second latency) is achieved with 4 parties and 2 shards (1 router, 2 batchers, 1 consenter, and 1 assembler per party). The throughput increases when using 2 shards instead of only one shard, however with 4 shards there is no noticeable improvement. The throughput decreases when deploying more parties. With 7 parties and 4 shards the throughput is 100, 000 tx/s. It is unclear why with 7 parties we get such a decreases in performance relative to 4 parties, we expected to see a smaller difference. This behavior can be due to the spread of parties across different data centers. This warrants further investigation that we plan to report in future versions of this paper.

The reported throughput of SmartBFT [12], the consensus algorithm currently deployed by HLF, is about 2, 500 tx/s; Arma achieves a 20-fold speedup. This effect can be attributed to the separation of transaction ordering from transaction dissemination, sharding, and pipelining of transaction processing, as explained in Section 4. Note that in contrast to classical HLF transactions, which require a size of about 3.5 KB, the compact form of the settlement transaction only requires less than 300 B to perform the same operation (spending two inputs and creating two outputs), thereby reducing the network load. This explains some (but not all) of the difference in the performance achieved by Arma compared to the current BFT HLF ordering service.

6.3 Transaction Validation

We evaluate the validation phase using the implementation of our enhanced committer, as outlined in Section 3. We measure the impact on transaction throughput and latency, of the following parameters: transaction size, invalid signatures, and double spending. We also evaluate the impact of number of database nodes on the performance of validator-committer services. We deployed the



(a) (b) (c)

Figure 8: 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.

components of the committer and the workload generator on multiple servers within the same region. Unless otherwise specified, we deployed the committer with three signature verifiers, nine validator-committer services colocated with nine nodes in a distributed database, one coordinator, and one sidecar. Note that when we overload the committer beyond its capacity, a queue begins to form, leading to an increase in throughput but with an increased latency. Throughout all experiments, we shaped the workload to keep the latency below one second.

Impact of transaction size To understand the effect of transaction complexity on system performance, we varied the number of inputs (read-set) and outputs (write-set) per transaction. As shown in Figure 8a, increasing the transaction size from one input and one output (1/1) to four of each (4/4) resulted in a 40% decrease in throughput, from 250,000 to 150,000 transactions per second (tps).

This performance decline is primarily due to the increased overhead in managing the dependency graph at the coordinator. Larger transactions introduce more complex dependencies, which in turn require more time for graph construction and updates. For instance, when increasing the input/output count from 1/1 to 4/4 for a batch of 500 transactions, the graph construction time increased from 1.6 ms to 3 ms, and the post-commit graph update time rose from 2.8 ms to 4.5 ms. Furthermore, the dependency graph is protected by a synchronization primitive to ensure data integrity, which serializes access and can become a point of contention, further limiting throughput.

Bottleneck Analysis. To confirm that the coordinator is the primary bottleneck, we independently benchmarked the other key components of the system: the signature validators and the validator-committers. Our findings indicate that these components are not the limiting factor. The signature verification process scales linearly with the number of verifiers; a single verifier can handle 150,000 tps, and two can achieve 300,000 tps. Similarly, scaling the validator-committer services and distributed database nodes from 6 to 15 increased shard throughput from 200,000 to 500,000 tps. Given that both the validator-committer and signature validation components demonstrate scalable performance well above the observed system throughput, we conclude that the coordinator is the current bottleneck in our system.

Impact of faulty transactions To analyze system performance when handling faulty transactions, we conducted an experiment by submitting a mix of valid and invalid transactions.

Invalid signatures. First, we focused on transactions with invalid threshold signatures, which can result from insufficient endorsements or malformed content. In this scenario, all transactions were configured with two inputs and two outputs, while the proportion of invalid transactions was varied from 0% to 30%. As illustrated in Figure 8b, we observed a counter-intuitive increase in overall throughput from 200,000 tps to 240,000 tps as the percentage of faulty transactions grew. This performance gain occurs because transactions with invalid signatures are rejected early in the pipeline by the signature validators. This pre-processing significantly reduces the workload on the validator-committer, as it does not need to process the state changes for these invalid transactions. This efficiency is also reflected in the transaction latency. As the share of invalid transactions increased from 0% to 30%, the average commit latency for a batch of 1000 transactions decreased by over 50%, from 375 ms to 175 ms. The primary reason for this improvement is the reduction in write operations; for an invalid transaction, the committer only records its identifier and status instead of applying the full set of state changes, resulting in a notable performance enhancement.

Double spendings. This scenario evaluates the system's performance when processing transactions with double-spend attempts. A double-spend occurs when a transaction attempts to use an input that has already been confirmed in the ledger. As illustrated in Figure 8c, the presence of these conflicting transactions initiates a significant performance degradation, which stems from two distinct bottlenecks. First, as the number of conflicting inputs rises, the complexity of dependencies between transactions within the batch increases. This directly slows down the dependency graph management, causing the graph construction time for a 500-transaction batch to increase from 1.6 ms to 4 ms and the post-commit update time to rise from 2.8 ms to 6 ms. Second, and concurrently, the validator-committer service must perform significant extra work when a transaction conflicts with the already committed state in the ledger——that doubles the internal latency of the validator-committer service from 300 ms to 600 ms. It is the combination of these two factors—slower intra-batch dependency processing and the intensive work of resolving conflicts against the ledger.

This internal processing bottleneck translates directly into the severe reduction in system-wide performance shown in the figure. As the proportion of double-spend transactions rises to 30%, the total throughput falls by 33%, from 200,000 tps to 134,000 tps. More

critically, the throughput of valid transactions, or "goodput," experiences a far more dramatic collapse of nearly 48%, plummeting to just 103,000 tx/s. This shows that the system's capacity for useful work is disproportionately impacted. This inefficiency is also reflected in the end-user latency, which doubles from 511 ms to over 1000 ms and stabilizes at this higher plateau, confirming that the entire commit process becomes fundamentally more expensive when conflicts are present.

Metric	coordinator	signature verifier	vcservice+ database-node
cpu	17%	38%	71%
memory	7 GB	3 GB	32 GB
disk	-	-	29%
network send	1.7 Gbps	0.05 Gbps	0.8 Gbps
network recv	1.2 Gbps	0.3 Gbps	0.86 Gbps

Table 3: Resource utilization of the coordinator, signature verifier, and Validator-Committer service (vcservice) with its co-located database node.

Resource Utilization To understand the resource footprint of our system, we measured the CPU, memory, disk, and network utilization across its core components, as detailed in Table 3. The Validator-Committer service (vcservice) is the most resource-intensive component. Each vcservice instance is co-located with a distributed database node, and this combination drives the highest CPU usage to 71% and memory consumption to 32 GB due to the intense I/O, processing, and memory demands of the database. In contrast, the coordinator is primarily network-bound. It consumes the most bandwidth because it must receive blocks from the sidecar, distribute transactions to all signature verifiers and validator-committers, aggregate their responses, and forward the final statuses back to the sidecar. This heavy communication workload results in high data transmission rates (1.7 Gbps send and 1.2 Gbps receive). Finally, the signature verifier is the most lightweight of the three components, operating with modest resource requirements across all metrics.

7 RELATED WORK

Performance Optimization of Hyperledger Fabric Hyperledger 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 [25], while other analyses framed Fabric's concurrency challenges as classic database problems [26]. Building on these insights, targeted optimizations have improved performance within the existing model. Gorenflo et al.'s influential "FastFabric" [27] 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 [29] or by applying transactional dependency analysis [28]—a concept that informs our own work. More recent research has proposed fundamental architectural changes. Notably, Thakkar et al. introduced "Sparse Peers" [30], 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 are ultimately constrained by a focus on specific parts of the system. Our research advances this trajectory by presenting a holistic, end-to-end redesign. We propose a more granular decoupling of the peer into microservices and introduce a fully redesigned, scalable ordering service. This comprehensive approach allows our framework to achieve a level of performance an order of magnitude beyond what has been previously reported.

CBDC Architectures and Privacy-Preserving Payments Prior work on Central Bank Digital Currencies (CBDCs) includes early centralized models [31] and high-throughput processors like the Hamilton Project [32], which, while performant, lack the full decentralization and BFT-resilience of our approach. Addressing the critical requirement of privacy, other systems like Platypus [33] offer compliance features but lack parallel processing, while other privacy-focused frameworks [34] have not been experimentally benchmarked. Meanwhile, powerful privacy-enhancing technologies like Zerocoin [35] and Zcash [37] use computationally expensive cryptography to ensure confidentiality. The significant overhead of these methods makes them impractical for large-scale systems without an exceptionally performant underlying platform. Our work provides this foundation; by delivering high throughput, our architecture serves as a critical enabler for integrating such intensive privacy features, complementing systems focused on auditable privacy [14] by providing the infrastructure they require.

8 CONCLUSION

To address the significant performance gap between existing DLTs and the demands of a Central Bank Digital Currency, this paper presented a fundamental re-architecture of Hyperledger Fabric. We introduced a decoupled microservices peer with a parallelized commit flow and Arma, a novel, scalable Byzantine Fault Tolerant ordering service. Our experimental evaluation confirms this design achieves a peak throughput exceeding 200,000 transactions per second—a two-orders-of-magnitude improvement over standard Fabric. This work demonstrates that permissioned DLTs can be engineered to meet the rigorous demands of modern financial systems, providing a scalable and resilient foundation for practical CBDC deployments and the future integration of advanced privacy-enhancing technologies.

REFERENCES

[1] 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] Alin Tomescu, Adithya Bhat, Benny Applebaum, Ittai Abraham, Guy Gueta, Benny Pinkas, and Avishay Yanai. Utt: Decentralized ecash with accountable privacy. Cryptology ePrint Archive, Paper 2022/452, 2022. https://eprint.iacr.org/2022/452.

[3] 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. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18. ACM, 2018.

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

[5] Hyperledger fabric. https://github.com/hyperledger/fabric/releases/tag/v3.0.0, 2023.

[6] Artem Barger, Yacov Manevich, Hagar Meir, and Yoav Tock. A byzantine fault-tolerant consensus library for hyperledger fabric. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2021.

[7] The SmartBFT library, open-source repository. https://github.com/hyperledger-labs/SmartBFT, 2024.

[8] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[9] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. 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*, pages 355–362. IEEE Computer Society, 2014.

[10] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. Bft in blockchains: From protocols to use cases. *ACM Comput. Surv.*, 54(10s), September 2022.

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

[12] Artem Barger, Yacov Manevich, Hagar Meir, and Yoav Tock. 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*, 2021.

[13] Yacov Manevich, Hagar Meir, Kaoutar Elkhiyaoui, Yoav Tock, and May Buzaglo. Arma: Byzantine fault tolerant consensus with horizontal scalability, 2024.

[14] Elli Androulaki, Jan Camenisch, Angelo De Caro, Maria Dubovitskaya, Kaoutar Elkhiyaoui, and Björn Tackmann. 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, 2020.

[15] https://github.com/hyperledger-labs/fabric-smart-client.

[16] https://github.com/hyperledger-labs/fabric-token-sdk.

[17] https://github.com/hyperledger/fabric.

[18] Yacov Manevich. Arma: Byzantine fault tolerant consensus with linear scalability. *ArXiv*, abs/2312.13777, 2023.

[19] https://go.dev/doc/devel/release#go1.20.

[20] https://grpc.io.

[21] https://github.com/yugabyte/yugabyte-db.

[22] https://prometheus.io.

[23] https://www.ibm.com/cloud.

[24] Threshold signature scheme library. https://github.com/IBM/TSS/.

[25] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 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)*, pages 264–276, 2018.

[26] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 105–122, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 455–463, 2019.

[28] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 543–557, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. Xox fabric: A hybrid approach to blockchain transaction execution. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2020.

[30] Parth Thakkar and Senthilnathan Natarajan. Scaling blockchains using pipelined execution and sparse peers. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 489–502, New York, NY, USA, 2021. Association for Computing Machinery.

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

[32] James Lovejoy, Madars Virza, Cory Fields, Kevin Karwaski, Anders Brownworth, and Neha Narula. 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, 2023.

[33] Karl Wüst, Kari Kostiainen, Noah Delius, and Srdjan Capkun. 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, 2022.

[34] Aggelos Kiayias, Markulf Kohlweiss, and Amirreza Sarencheh. 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, 2022.

[35] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[36] Andrew Poelstra, Adam Back, Mark Friedenbach, Gregory Maxwell, and Pieter Wuille. Confidential assets. In *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2019.

[37] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 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, 2014.