Dependency-Aware Execution Mechanism in Hyperledger Fabric Architecture

Anonymous Authors

No Institute Given

Abstract. Hyperledger Fabric is a leading permissioned blockchain framework for enterprise use, known for its modular design and privacy features. While it strongly supports configurable consensus and access control, Fabric can face challenges in achieving high transaction throughput and low rejection rates under heavy workloads. These performance limitations are often attributed to endorsement, ordering, and validation bottlenecks. Further, optimistic concurrency control and deferred validation in Fabric may lead to resource inefficiencies and contention, as conflicting transactions are identified only during the commit phase.
To address these challenges, we propose a dependency-aware execution model for Hyperledger Fabric. Our approach includes: (a) a dependency flagging system during endorsement, marking transactions as independent or dependent using a hashmap; (b) an optimized block construction in the ordering service that prioritizes independent transactions; (c) the incorporation of a Directed Acyclic Graph (DAG) within each block to represent dependencies; and (d) parallel execution of independent transactions at the committer, with dependent transactions processed according to DAG order.
Incorporated in Hyperledger Fabric v2.5, our framework was tested on workloads with varying dependency levels and system loads. Results show up to 40% higher throughput and significantly reduced rejection rates in high-contention scenarios. This demonstrates that dependency-aware scheduling and DAG-based execution can substantially enhance Fabric's scalability while remaining compatible with its existing consensus and smart contract layers.

1 Introduction

Hyperledger Fabric is a permissioned blockchain platform widely used in enterprise applications [1]. Its modular design, with pluggable consensus and privacy features, enables industrial-grade flexibility. Fabric operates via a three-phase transaction pipeline: endorsement, ordering, and validation. Clients submit proposals to endorsers, who simulate transactions and return read-write sets. The ordering service collects these into blocks and broadcasts them to committing peers, who validate and apply them. Depending on its configuration, a peer node in the network can perform one or more of these roles: *endorser*, *orderer*, or *committer*.

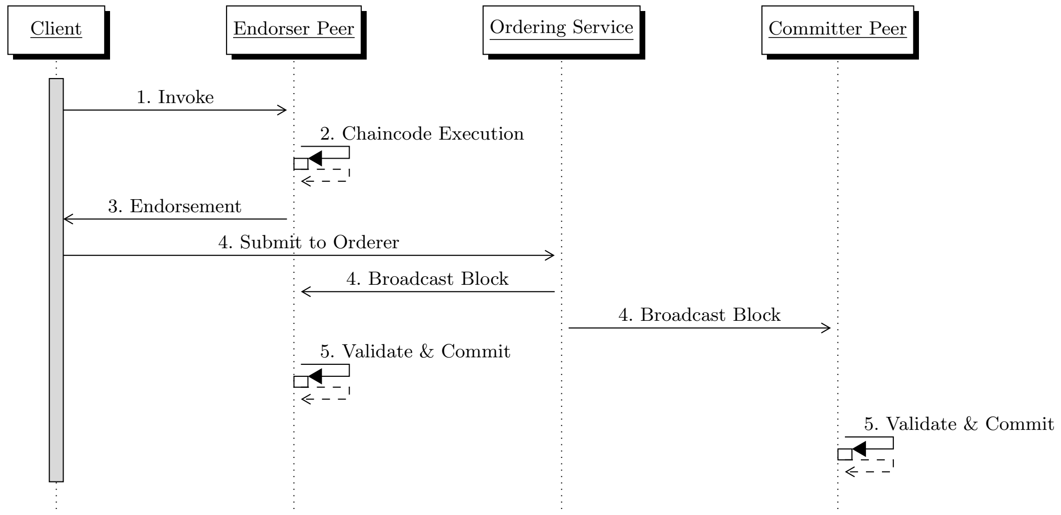


Fig. 1: Fabric transaction flow by Androulaki et.al [1]

Hyperledger Fabric Architecture: The transaction flow in Hyperledger Fabric, as illustrated in Fig. 1, begins when the client (**Step 1**) sends a transaction proposal to one or more *endorser* peers. Each endorser peer then simulates the transaction by executing the specified chaincode (smart contract code, **Step 2**), generating a read/write set and an endorsement signature. The endorser returns this endorsement to the client (**Step 3**), and the client collects enough endorsements as required by the policy. The client then submits the endorsed transaction to the *ordering service* (**Step 4**), which sequences transactions and packages them into blocks. The ordering service broadcasts the new block to all relevant peers, including both endorsers and *committers* (**Step 4**). Finally, each committer peer validates the transactions in the block, checking endorsement policies and for conflicts, and commits valid transactions to the ledger (**Step 5**).

While this architecture performs well in low-contention environments, it exhibits significant limitations under high transaction volume. A key bottleneck is the lack of early dependency detection and parallelism during the commit phase. Fabric employs an optimistic concurrency model with versioning, where all transactions are allowed to proceed until final validation. Conflicts are detected only during the commit stage, based on world-state version checks. As a result, conflicting transactions are often rejected later in the commit stage, leading to increased retries and wasted compute resources.

For example, consider 100 transactions that want to reduce the same asset by a value of 100. Suppose the initial value of the asset of also 100. In such a scenario, only the first transaction commits successfully and updates the state to zero, while the remaining 99 are rejected due to stale versioning. Despite being endorsed and ordered, these transactions fail at the commit stage (**Step**

5), resulting in unnecessary latency and reduced throughput. In this paper, we wish to address this shortcoming.

Another constraint is Fabric's strictly sequential commit logic, which executes all transactions in a block one after another—even if many are independent. This underutilizes multicore systems and limits concurrency, making Fabric inefficient in high-dependency workloads.

2 Problem Statement and Contributions

Hyperledger Fabric follows an *execute-order-validate* model [1] for processing transactions. Clients submit proposals that are simulated by endorsing peers. The results are passed to the ordering service, which sequences them into blocks. These blocks are then validated and committed by peers based on endorsement policies and version control. Although this pipeline is designed to decouple execution from ordering, it does not utilize parallelism effectively, especially at the commit stage.

The *order-execute-validate* model is the traditional transaction processing paradigm adopted by early blockchains such as Bitcoin and Ethereum. In this approach, transactions are first collected and ordered into blocks through a consensus protocol. Once ordered, every node in the network sequentially executes all transactions in the agreed order, updating their local copy of the ledger state. Finally, nodes validate the results to ensure consistency across the network. This model enforces strict determinism in smart contract execution to guarantee that all nodes reach the same state. While conceptually simple, this approach imposes limitations on scalability and performance, as transactions must be executed sequentially and by all nodes. Furthermore, the requirement for deterministic code restricts the flexibility of smart contract development.

The *execute-order-validate* model, pioneered by Hyperledger Fabric [1], reorders the transaction processing steps to address the limitations of the traditional approach. In this model, transaction proposals are first executed (or simulated) by a subset of peers, known as endorsers, which generate read/write sets and endorsements. These endorsed transactions are then ordered via a consensus service. After ordering, each peer validates the transactions against endorsement policies and checks for conflicts before committing them to the ledger. This separation of execution and ordering enables parallel transaction execution, improves throughput, and allows the use of general-purpose programming languages for smart contracts, as non-deterministic transactions can be detected and filtered out before ordering. The model is particularly suited for permissioned blockchains, where performance, flexibility, and strong consistency are essential.

A key issue lies in how Fabric handles transaction dependencies. Fabric employs an optimistic concurrency control mechanism with versioning, where all transactions are allowed to proceed to the commit phase regardless of conflicts. Final validation is deferred to the commit phase (stage 4), where transactions are checked for version consistency. If multiple transactions operate on the same key, only the one with the latest version is committed, while the rest are marked invalid. This leads to increased latency, unnecessary retries, and wasted compute resources for transactions that are eventually discarded.

For example (briefly discussed in Section 1), consider an application using the asset-transfer chaincode where 100 concurrent transactions each attempt to deduct 100 units from the same asset, which has an initial balance of 100. Each transaction is endorsed based on the initial state and appears valid in isolation. However, during commit, only the first transaction updates the ledger successfully. The remaining 99 are rejected due to state version mismatches, since their endorsements were based on stale data. This rejection is detected only in the commit phase and thus increasing the response time which could have been avoided had it been detected earlier in the endorsement phase. This illustrates the importance of conflict-aware application logic in Fabric and highlights how concurrent writes to the same key can degrade performance if not handled carefully.

Another major bottleneck is the strictly sequential commit logic. All transactions within a block are processed one after another, even if many are independent and could be executed concurrently. This prevents Fabric from leveraging multi-core architectures or exploiting transaction-level parallelism. The system lacks any mechanism to distinguish between independent and dependent transactions during execution.

These limitations result in two main issues:

1. **High rejection rates under contention**: As shared state access increases, the likelihood of invalid transactions grows, leading to frequent rejections at the commit stage.
2. **Under-utilization of system resources**: Independent transactions are serialized unnecessarily, limiting concurrency and throughput.

These problems are particularly pronounced in enterprise applications, where multiple clients frequently interact with overlapping state variables. While previous enhancements to Fabric have targeted endorsement and ordering optimizations, they do not resolve the core inefficiencies in the commit phase. To support high-throughput workloads with mixed transactional dependencies, a fundamentally new execution model is required—one that introduces minimal architectural changes and remains compatible with Fabric's existing endorsement policies and smart contract interfaces.

In this paper, we address these issues. Specifically, we present a dependency-aware transaction execution mechanism in the Fabric's architecture. Our approach detailed in Section 3 includes: (a) a dependency flagging system during endorsement, marking transactions as independent or dependent using a hashmap; (b) an optimized block construction in the ordering service that prioritizes independent transactions; (c) the incorporation of a Directed Acyclic Graph (DAG) within each block to represent dependencies; and (d) parallel execution of independent transactions at the committer, with dependent transactions processed according to DAG order.

3 Proposed Solution

We introduce a transaction dependency-aware execution mechanism for Hyperledger Fabric that enhances parallelism and reduces transaction rejections. The objective is to detect dependencies early in the transaction lifecycle and use that information during commitment phase to reduce transaction rejections while enabling parallel execution wherever possible. These enhancements integrates cleanly within the existing Fabric architecture, requiring minimal disruption to its consensus or chaincode interfaces.

The proposed transaction flow for Hyperledger Fabric, as depicted in Fig. 2, introduces a leader endorser and parallel execution mechanisms to enhance scalability and efficiency. The leader endorser coordinates the endorsement process across all the transaction received by various endorsing peers by maintaining the dependency relation. The process begins when the client sends a transaction proposal to an endorser peer, which then forwards the request to a designated leader endorser. The leader endorser simulates the transaction, flags any dependencies, and returns both the endorsement and dependency information to the original endorser. The endorser relays this endorsement back to the client, who submits the endorsed transaction to the ordering service. After ordering, the service broadcasts the block to the committer peer. The committer constructs a directed acyclic graph (DAG) based on the flagged dependencies and executes transactions in parallel where possible, committing valid results to the ledger. This architecture aims to maximize throughput by leveraging dependency-aware parallelism during the commit phase, while maintaining the integrity and consistency of the ledger state.

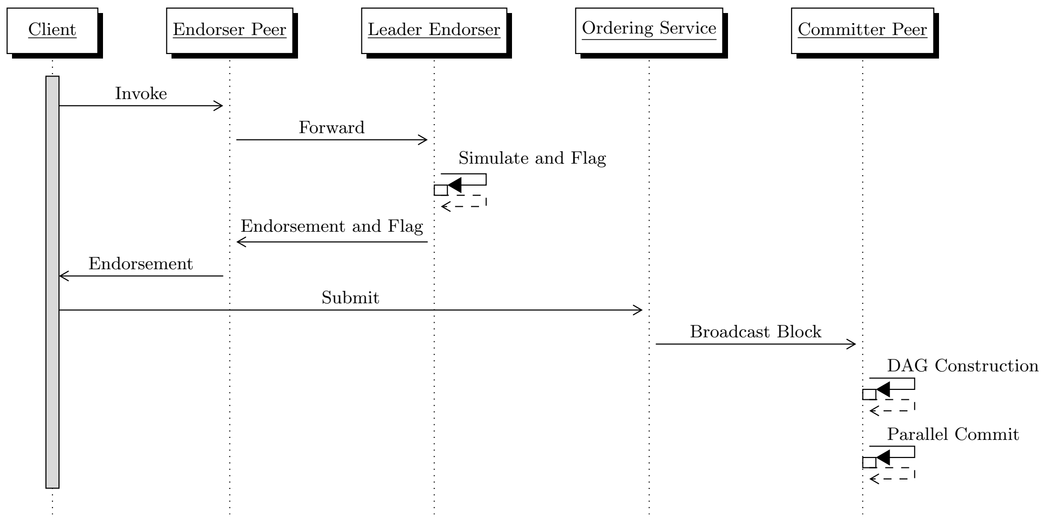


Fig. 2: Proposed Fabric transaction flow

3.1 Key Enhancements

The solution introduces modifications across the endorsement, ordering, and commit phases, as described below:

Dependency Flagging at Endorsement: We introduce a flagging mechanism at the leader endorser node. For each incoming transaction, we simulate its execution and inspect the key-value pairs it reads and writes. If a key has been recently modified or is part of an active endorsement set, the transaction is flagged as dependent (`flag = 1`); otherwise, it is marked as independent (`flag = 0`). The endorsement, along with the flag, is sent back to the client. Using this information, the client can decide whether to proceed with transaction submission or delay/abort it. By empowering the client with early conflict information, we reduce unnecessary network and validation overhead. This mechanism directly addresses the concurrency and commit-phase rejection issues detailed in Section 2.

A hashmap is maintained at the endorser to track recent key usage, logging transaction IDs and their accessed keys. Expiry timers are attached to purge stale metadata. This dependency flag is embedded into the transaction proposal response and is propagated through the pipeline.

For example, in a scenario where 100 transactions each attempt to reduce an asset's value by 100 (starting from a value of 100), the first transaction—having no conflicting predecessors—is flagged as independent (`flag = 0`), while the rest are flagged as dependent (`flag = 1`) due to access on a stale world state.

Propagation Through Ordering Phase: The ordering service remains mostly unchanged, maintaining compatibility with existing configurations (e.g., Raft, Kafka). However, it is modified to preserve and forward the dependency metadata. Custom fields in the block structure store transaction flags and associated key identifiers, ensuring the committer can reconstruct dependencies accurately without recomputation.

DAG Construction and Parallel Execution at Commit: To enable fine-grained concurrency while preserving Fabric's consistency guarantees, we enhance the committer to construct a lightweight Directed Acyclic Graph (DAG) from the transactions in each block. In this DAG, each transaction is represented as a node, and edges denote key-level dependencies. An edge from transaction A to B indicates that B depends on A's outcome and must be executed afterward. A modified topological sort organizes transactions into DAG levels, where independent transactions (`flag = 0`) form the base level and can be executed in parallel, while dependent transactions (`flag = 1`) are layered above based on their dependency chains.

We update the commit worker logic to process transactions in DAG-level batches. Each level is handled using a thread pool, allowing concurrent execution of all transactions in that level. Once a level is completed, the next level is scheduled, respecting dependency constraints. If a transaction fails validation (e.g., due to a version mismatch), it is rejected, and its downstream transactions are re-evaluated with updated context. This ensures consistency while avoiding cascade failures. All concurrency is managed within the peer node—no client-side changes are needed.

Endorsement Expiry and Conflict Cleanup: To prevent stale metadata from affecting performance, expiry timers are set on all endorsement flags and dependency entries. If a transaction remains uncommitted beyond a defined threshold, its dependency information is purged. This ensures the system remains responsive and avoids memory bloat. In such cases, the client must resubmit the transaction as a new request. Any endorser receiving a transaction with expired metadata will discard it.

3.2 Proposed Architecture

In this section we detail the modified architecture of Hyperledger Fabric as proposed in this work. The key idea is to integrate a dependency-aware execution mechanism while retaining the core stages of Fabric's pipeline: endorsement, ordering, and commit. The modified setup introduces changes to each phase, focused on tracking dependencies, constructing a DAG from transaction metadata, and enabling level-wise parallel execution at the commit stage.

Modified Endorsement Logic: In the standard Fabric flow, endorsers simulate the transaction and return the read-write sets to the client without explicitly tracking dependencies. We extend the logic of the **leader endorser** to inspect whether the keys involved in the transaction have already been accessed by other active transactions.

A key-value hashmap is maintained to store active keys and the latest transaction IDs associated with them. The endorsement logic is modified as follows:

PCode: Leader Endorser The pseudocode for the leader endorser's transaction endorsement with dependency flagging is outlined in Algorithm 1. Upon receiving a transaction T (line 1), the leader endorser first simulates its execution (line 2). If the simulation fails (line 3), the transaction is rejected and the client is notified (lines 4–5). If the simulation succeeds (line 6), the algorithm checks whether any variable accessed by T already exists in the HASHMAP (line 7). If such a variable is found, a dependency flag is set ($flag \leftarrow 1$), and a reference to this dependency is added to the HASHMAP (lines 8–9). Otherwise, the flag is set to zero and a new entry for the variable is created (lines 10–11). The endorser then sets an expiry timer for the endorsement (line 12), signs the transaction along with the flag and dependency metadata (line 13), and returns the result to the client (line 14). This process enables the system to efficiently track and signal transaction dependencies, facilitating subsequent parallel execution while preserving consistency.

This logic ensures that transactions accessing shared state are marked as dependent, and the information is embedded directly into the transaction metadata returned to the client.

Algorithm 1 Transaction Endorsement with Dependency Flagging

```
Require: Transaction $T$ received
 1: Simulate $T$
 2: if simulation fails then
 3:     Reject $T$
 4:     Notify client
 5: else
 6:     if variable in $T$ exists in HASHMAP then
 7:         $flag \leftarrow 1$
 8:         Add dependency reference to HASHMAP
 9:     else
10:         $flag \leftarrow 0$
11:         Add variable entry to HASHMAP
12:     end if
13:     Set endorsement expiry timer
14:     Sign $T$ with $flag$ and dependency metadata
15:     Return to client
16: end if
```

DAG-Based Block Construction and Commitment: After transactions are ordered into blocks, they are forwarded to committing peers. The commit logic is extended to construct a lightweight Directed Acyclic Graph (DAG) using the metadata in the block.

Algorithm 2 describes the DAG-based parallel transaction processing approach used by the committer peer. Upon receiving a block from the ordering service (line 1), the committer constructs a directed acyclic graph (DAG) to capture transaction dependencies (line 2). For each transaction t in the block (line 3), if the dependency flag is zero, t is added to Level 0 of the DAG, indicating it is independent (lines 4–5). Otherwise, an edge is added from t to its parent transaction(s) based on the recorded dependency information (lines 6–7). Once the DAG is constructed (line 9), transactions are processed level by level (line 10). Within each level, all transactions are validated and executed in parallel (lines 11–12). If a transaction is valid, the world state is updated and peers as well as the ordering service are notified (lines 13–15); invalid transactions are rejected (lines 16–17). This method enables safe parallelism by ensuring that dependent transactions are executed only after their prerequisites, thus improving throughput while maintaining correctness.

Transactions with `flag = 0` are processed in parallel immediately. Transactions with `flag = 1` are scheduled based on DAG dependencies. If two dependent transactions are at the same level and do not share keys, they can be executed in parallel.

Architecture Overview: Figure 3 provides a high-level view of the modified architecture.

The architecture retains Fabric's existing modular design. Clients continue to interact with the peer nodes through standard SDK interfaces. The only

Algorithm 2 DAG-based Parallel Transaction Processing

```
Require: Block received from ordering service
 1: Construct DAG:
 2: for all transaction $t$ in block do
 3:     if flag($t$) == 0 then
 4:         Add $t$ to Level 0 (independent)
 5:     else
 6:         Add edge from $t$ to parent transaction(s) based on dependency
 7:     end if
 8: end for
 9: Process DAG:
10: for all DAG level $L$ do
11:     for all transaction $t$ in $L$ in parallel do
12:         Validate $t$
13:         if valid then
14:             Update world state with $t$
15:             Notify peers and ordering service
16:         else
17:             Reject $t$
18:         end if
19:     end for
20: end for
```

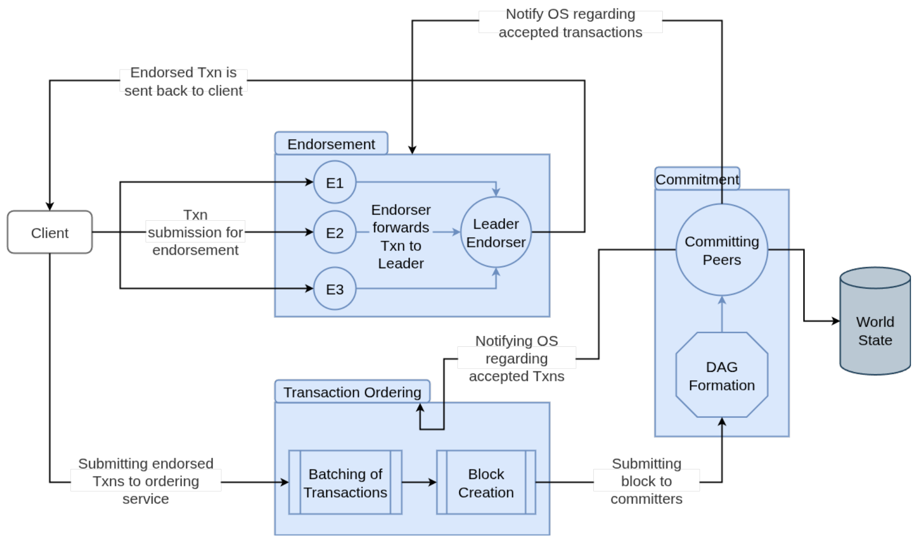

Notify OS regarding accepted transactions
Endorsed Txn is sent back to client
Endorsement
E1
E2
E3
Endorser forwards Txn to Leader
Leader Endorser
Client
Txn submission for endorsement
Commitment
Committing Peers
DAG Formation
World State
Notifying OS regarding accepted Txns
Transaction Ordering
Batching of Transactions
Block Creation
Submitting endorsed Txns to ordering service
Submitting block to committers

Fig. 3: Proposed DAG-aware Fabric Architecture

change is in how dependencies are internally detected, propagated, and processed. Endorsement flagging, DAG construction, and parallel validation are all encapsulated within the peer logic.

Commit Thread Scheduling: We integrate a thread pool at the committer that processes DAG levels concurrently. This thread pool dynamically scales with the number of level-0 and level-n transactions. The scheduler ensures that all parent transactions of a node are completed before the child transaction is validated.

The DAG is constructed using adjacency lists, and a topological sort is applied to determine the execution sequence. Transactions in the same level are passed to available threads in the pool.

Compatibility and Modularity: The proposed enhancements are backward-compatible with existing chaincode logic and network configurations. No changes are required in chaincode interface, world state structure, or ordering service APIs. This ensures that networks already using Fabric can adopt the new model with minimal transition effort.

Expected Impact and Benefits: The proposed enhancements deliver the following benefits over the default Fabric execution model:

- **Lower rejection rates:** Early dependency detection reduces commit-time rejections, especially under high contention.
- **Better resource utilization:** Independent transactions are executed in parallel, improving CPU and memory usage.
- **Higher throughput and lower latency:** DAG-level concurrency significantly improves transaction throughput and reduces average response time (ART), as demonstrated by up to 40% performance gain in our experiments.
- **Minimal architectural disruption:** The enhancements require only internal changes to peer logic; existing chaincode, clients, and ordering services remain untouched.
- **Scalability with workload dynamics:** The system adapts well to mixed workloads with varying dependency ratios.

To evaluate the effectiveness of our proposed enhancements, we have carefully designed a series of experiments that simulate realistic and diverse blockchain workloads with varying levels of contention. These experiments are tailored to assess whether the expected benefits, such as reduced transaction rejection rates, improved resource utilization, and increased throughput, are achieved in practice. Section 4 details the experiments design, the implementation platform, and the observations from results.

4 Performance Analysis

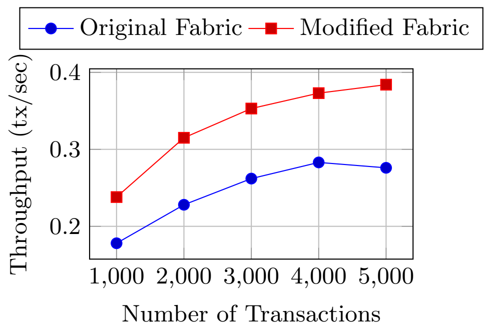
All experiments were conducted on a local development environment configured for reproducibility and performance isolation. The setup details are summarized in Table 1. Table 2 and Table 3 in the Appendix give a brief summary of the Fabric components modified by the authors, as well as the parameters set by the authors for workload simulation.

Component	**Specification**
Processor	AMD Ryzen 5 5500U, 2.1GHz, 6 cores / 12 threads
Memory	14 GB DDR4 RAM
Disk	512 GB NVMe SSD
Operating System	Ubuntu 24.04 LTS

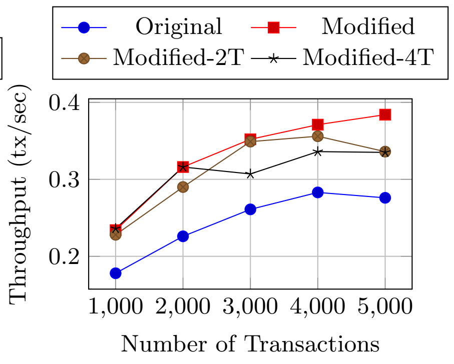
Table 1: Hardware and Software Configuration

Note: All results presented here are based on the Voting Contract. Additional preliminary results for the Asset-Transfer Contract and Wallet Contract, are provided in the Appendix for further comparison and analysis.

Experiment Settings: The experiments compare three threading strategies. (a) The first is the *Original Fabric*, which serves as the baseline implementation without any concurrency optimizations. (b) The second approach, *Modified Fabric (Dynamic Threads)*, employs a dynamic number of threads equal to the number of transactions at each DAG level, with the count capped by the number of physical cores to prevent oversubscription. (c) The third strategy, referred to as *2-threaded / 4-threaded Fabric*, uses a fixed number of threads per DAG level, irrespective of the transaction count.



(a) Original vs Modified Fabric

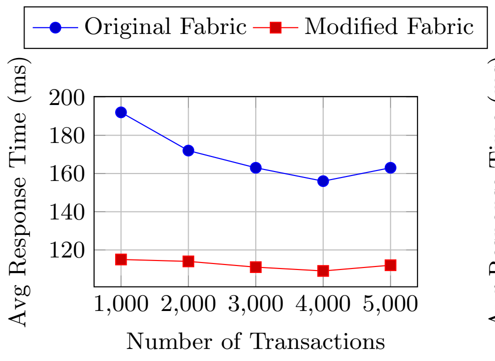


(b) Parallelism in Modified Fabric

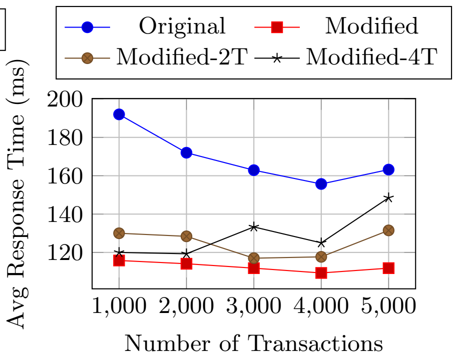
Fig. 4: Experiment 1: Impact of Number of Transaction on Throughput

Experiment 1: Impact of Number of Transaction on Throughput
This experiment illustrated in Figure 4 evaluated the throughput across increasing transaction loads. The Modified Fabric (Dynamic Threads) consistently achieved higher throughput than the Original Fabric. At 5000 transactions, throughput improved from 0.276 tx/sec to 0.384 tx/sec (approx. 39% gain).

Introduction of fixed-thread variants showed limited or mixed improvements. While 2-threaded execution showed some benefit, the 4-threaded variant often plateaued or underperformed due to thread contention or underutilization depending on DAG width.



(a) Original vs Modified Fabric



(b) Parallelism in Modified Fabric

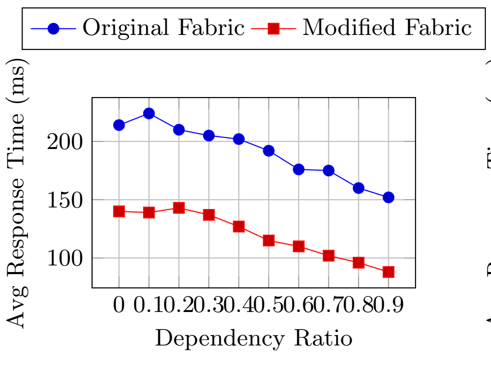
Fig. 5: Experiment 2: Latency Reduction Analysis

Experiment 2: Latency Reduction Analysis
This experiment (Figure 5) focused on average transaction latency. The Modified Fabric showed substantial latency reduction—for example, reducing latency from 192 ms to 115 ms at 1000 transactions. The 2-threaded and 4-threaded variants displayed varied performance. The 2-threaded variant showed modest improvements, while the 4-threaded version achieved lowest latency in some high-load cases, but exhibited inconsistent behavior due to rigid thread allocation at each DAG level.

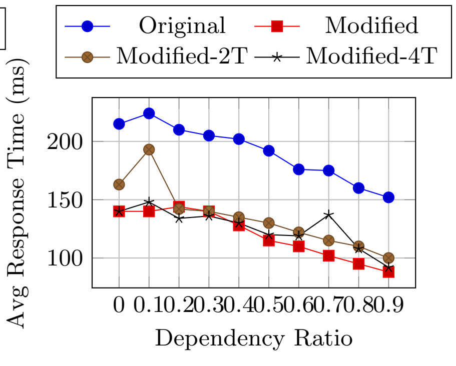
Experiment 3: Impact of Dependency Ratio on Latency
This experiment (Figure 6) analyzed performance under varying inter-transaction dependencies, measured as a dependency ratio from 0 to 0.9. As dependency increased, latency in the Original Fabric increased significantly, while the Modified Fabric with Dynamic Threads maintained much lower and more stable latencies. The 2-threaded and 4-threaded variants showed mid-range performance, with the 4-threaded approach performing best at high dependency levels (e.g., 92 ms at 0.9), though not always better than the dynamic strategy.

In addition to the core experiments, we evaluated the system performance by analyzing average response times for both committed and aborted transactions. These metrics provide further insight into the user-perceived latency and system efficiency.

Average Response Time for Committed and Aborted Transactions:
The Modified Fabric consistently reduced the average response time for both

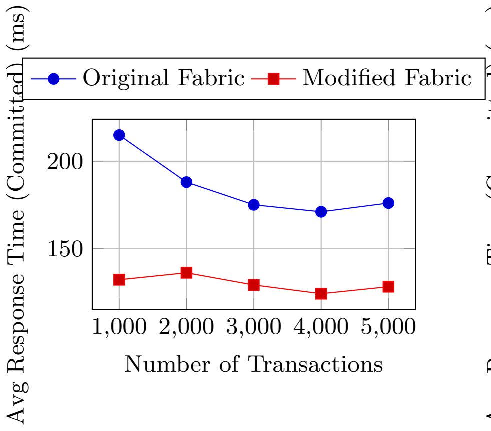


(a) Original vs Modified Fabric

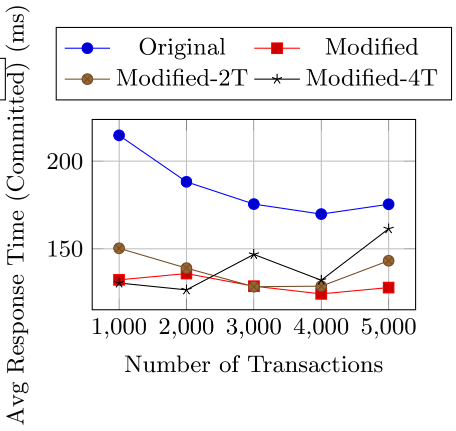


(b) Parallelism in Modified Fabric

Fig. 6: Experiment 3: Impact of Dependency Ratio on Latency

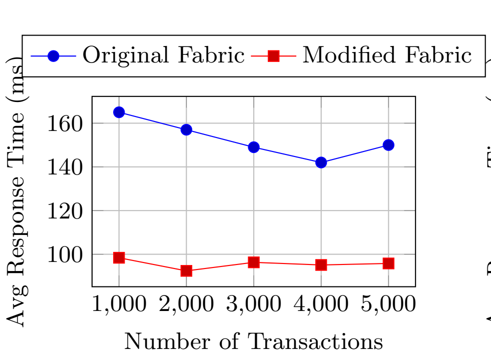


(a) Original vs Modified Fabric

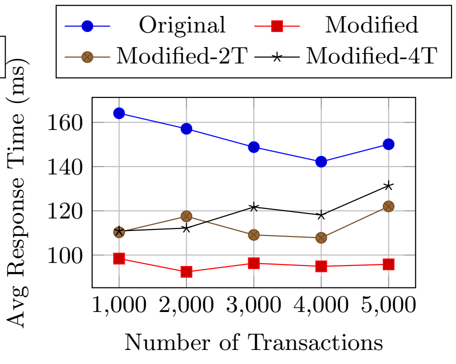


(b) Parallelism Comparison

Fig. 7: Committed Transactions Results



(a) Original vs Modified Fabric



(b) Parallelism Comparison

Fig. 8: Aborted Transactions Results

committed and aborted transactions compared to the Original Fabric across all transaction volumes (see Figure 7 and Figure 8). At 1000 transactions, the average response time for committed transactions dropped from 215 ms to 132 ms, while for aborted transactions, it fell from 165 ms to 98.4 ms. Notably, the 4-threaded variant sometimes matched or slightly outperformed the dynamic-threaded Modified Fabric in handling committed transactions, suggesting that fixed parallelism can occasionally align well with the underlying DAG structure. However, at higher volumes (e.g., 5000 transactions), the dynamic approach maintained lower and more stable response times, especially for committed transactions. In contrast, the fixed-threaded setups—particularly the 4-threaded configuration—showed elevated response times for aborted transactions (e.g., 131.4 ms at 5000 transactions), likely due to increased thread contention and overhead. These results emphasize the adaptability and responsiveness of dynamic threading under varying workloads and transaction outcomes.
Key Insights from Experiments: The following observations are derived from performance evaluations conducted on the **Voting smart contract**. Each experiment examines a specific aspect of system performance under various execution strategies, comparing the baseline Original Fabric with optimized versions. All performance metrics reflect either throughput (transactions per second) or latency (in milliseconds).

The Modified Fabric with Dynamic Threading consistently outperforms the baseline across all experiments. It adapts to DAG width dynamically, providing better utilization of parallelism without overwhelming system resources. Fixed-thread approaches (2T/4T per level) can offer benefits but are sensitive to DAG structure and thread scheduling efficiency. These results highlight the value of adaptive concurrency in smart contract execution models.

5 Related Work

Recent research has proposed various improvements for permissioned blockchains. Androulaki et al. [1] introduced Fabric's execute-order-validate architecture. DAG-based approaches have emerged as effective alternatives to improve concurrency [8]. For instance, BlockPilot [11] and RT-DAG [10] enable concurrent execution using dependency-aware scheduling. Constructing order restricted DAG for execution and sharing the DAG for validation stage are explored for generic blockchains framework [7, 2]. Coloring the contructed DAG to rearrange the DAG levels to get optimized performance is introduced throug Batch-schedule-Execute [4] recently.

Particularly for Hyperledger Fabric, Fabric++ [6] and FastFabric [3] enhance throughput via endorsement and validation optimizations, though they retain sequential commitment. Public blockchain systems like Omniledger [5], RapidChain [9], and NutBolt [12] introduce parallel models with distinct trust and consensus assumptions.
Unlike prior approaches that require significant architectural changes or new consensus layers, our method preserves full compatibility with Fabric's chaincode,

client APIs, and consensus modules. We implement our modifications on Fabric v2.5 and validate them through experiments on a local Docker-based testbed with varying transaction volumes and dependency levels. Results show up to 40% improvement in throughput and significant reduction in average response time for both committed and aborted transactions.

6 Conclusion and Future Work

Hyperledger Fabric has emerged as a prevalent permissioned blockchain framework; however, its transaction processing mechanism lacks concurrency-awareness, limiting performance under load. This paper introduced a dependency-aware transaction execution scheme for Fabric that identifies inter-transaction dependencies and uses DAG-based block structures to enable parallelism.
Our proposed enhancements include: dependency flagging at the endorsement phase, dependency-preserving block construction at the ordering service, and DAG-guided execution at the committer. The implementation extends the existing Fabric codebase with minimal intrusion, ensuring compatibility and maintainability. Experimental evaluations demonstrate that the modified Fabric significantly outperforms the standard implementation across key metrics. Notably, throughput increases by up to 25%, response times are consistently lower, and commit rates remain higher under varying transaction loads and dependency ratios. These improvements are especially evident in scenarios with high contention, where the original Fabric struggles to maintain performance.

In future work, we plan to integrate dynamic dependency detection using static code analysis or machine learning to remove the need for client-declared read/write sets. Additionally, incorporating fault-tolerant and Byzantine-resilient endorser leader election and DAG commit logic would make the solution more robust in distributed deployments. Further exploration of integration with Fabric private data collections and channels could enhance support for confidential multi-party workflows.

References

1. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S.W., Yellick, J.: Hyperledger fabric: A distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference. pp. 30:1–30:15. EuroSys '18, ACM (2018). https://doi.org/10.1145/3190508.3190538
2. Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts. In: PDP. pp. 83–92 (2019)
3. Gorenflo, C., Lee, S., Golab, L., Keshav, S.: Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In: Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation. pp. 125–140. USENIX (2020)

4. Hay, Y., Friedman, R.: Batch-schedule-execute: On optimizing concurrent deterministic scheduling for blockchains. In: 2024 43rd International Symposium on Reliable Distributed Systems (SRDS). pp. 163–174 (2024). https://doi.org/10.1109/SRDS64841.2024.00025
5. Kokoris-Kogias, E., Jovanovic, P., Gailly, L., Gasser, L., Ford, B.: Omniledger: A secure, scale-out, decentralized ledger. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 406–421. ACM (2020). https://doi.org/10.1145/3341301.3359631
6. Nasir, Z., Zhang, X., Yu, L., Ren, K.: Fabric++: Optimizing smart contract execution in hyperledger fabric. In: 19th USENIX Symposium on Networked Systems Design and Implementation. pp. 251–266. USENIX (2022)
7. Piduguralla, M., Chakraborty, S., Anjana, P.S., Peri, S.: Dag-based efficient parallel scheduler for blockchains: Hyperledger sawtooth as a case study. In: Cano, J., Dikaiakos, M.D., Papadopoulos, G.A., Pericàs, M., Sakellariou, R. (eds.) Euro-Par 2023: Parallel Processing. pp. 184–198. Springer Nature Switzerland, Cham (2023)
8. Wang, Q., Yu, J., Peng, Z., Bai, V.C., Guo, Z., Liang, W., Yang, G.: Sok: Dag-based blockchain systems. ACM Computing Surveys **56**(5), 1–35 (2024). https://doi.org/10.1145/3576899
9. Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: Scaling blockchain via full sharding. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 931–948. ACM (2018). https://doi.org/10.1145/3243734.3243853
10. Zhang, Z., Hong, C., Zhou, J., Ahmad, I., Zheng, Z., Chen, S.: Rt-dag: Dag-based blockchain supporting real-time transactions. IEEE Transactions on Parallel and Distributed Systems **35**(8), 1349–1363 (2024). https://doi.org/10.1109/TPDS.2024.3416751
11. Zhao, H., Li, Q., Wang, Z., Liu, F., Yang, H., Xu, C., Li, M.: Blockpilot: A proposer-validator parallel execution framework for blockchain. In: Proceedings of the 52nd International Conference on Parallel Processing. pp. 641–651. ICPP '23, ACM (2023). https://doi.org/10.1145/3605573.3605621
12. Zheng, S., Wang, J., Ren, K., Yuan, Y., Wang, B.: Nutbolt: Efficient and scalable execution of smart contracts. In: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation. pp. 755–768. USENIX (2018)

A Appendix

Appendix Table of Contents

A.1 Set-up Summary

This environment was used consistently across all experiments to ensure a fair comparison of the original and modified Fabric behavior. Several key components of Hyperledger Fabric were modified to support the evaluation of dependency-aware transaction processing. The modifications are detailed in Table 2. We have introduced the concept of leader endorser to coordinate the dependency for all the transactions received by the endorsing peers. To simulate realistic and diverse workloads, the experimental setup varied several parameters, detailed in Table 3. Each block contained between 1000 and 5000 transactions, with the proportion of transactions having dependencies ranging from 0% (no inter-transaction dependencies) to 90% (highly dependent workloads).

Component	**Modification**
Leader Endorser	Added logic to detect key conflicts and tag transactions with dependency flags
Committer	Implemented DAG constructor and thread pool-based executor for level-wise validation
Protobuf Files	Extended block metadata to include transaction dependency markers

Table 2: Key Modifications in Hyperledger Fabric

Parameter	**Value / Range**
Transactions per block	1000 – 5000
Dependency ratio	0% – 90%
Retry logic	Disabled
Threads per committer	2, 4 and dynamic threads in DAG levels
Transaction logic	Mix of read/write operations on shared key-value space

Table 3: Workload Simulation Parameters

A.2 Security and Correctness

Any architectural modification to a permissioned blockchain must maintain the integrity, determinism, and fault tolerance of the original system. Our proposed enhancements to Hyperledger Fabric were designed with these properties in mind, ensuring that the changes do not compromise the security guarantees already provided by the platform.

Data Consistency and Validation: Fabric maintains correctness through version checks and endorsement policies. Our model preserves this by not altering the semantics of transaction simulation, endorsement, or validation. Each transaction still undergoes the same chaincode-based simulation at endorsers. The DAG formation and utilisation remove the need for version check without altering the logic of the version check (in case that would have been in place). For example, let's suppose we have 2 endorsed transactions which update the same asset in the world state. In the case of original logic, the first transaction will succeed, but the second transaction will be rejected. This is because the version of the asset will be updated after the completion of the first transaction. The endorsed version in the second transaction would not match the new version of the asset now. In our updated code base, the second transaction would be checked as per the chain code. Passing the check would mean that the transaction would not cause any conflicts and could be executed without any issues. Hence, the proposed architecture reduces transaction rejections without accepting any conflicting transactions. The dependency flagging added during endorsement is the metadata and is used only for scheduling. It does not change the transaction's logic or state transitions.

Hence, incorrect or conflicting transactions are filtered out before updating the world state. This ensures that the integrity of the ledger is preserved, even when multiple transactions are executed in parallel.

Determinism via DAG Execution: The DAG structure enforces a partial order among dependent transactions. The use of topological sorting ensures that parent transactions are always processed before their children. This guarantees that the final ledger state is deterministic and consistent across all committing peers, regardless of thread scheduling or parallelism.

The DAG is generated from explicit flags and known dependencies, and therefore remains the same across all peers. No randomness or non-deterministic ordering is introduced in the execution pipeline.

Concurrency Control: Only transactions that are marked as independent and conflict-free are allowed to execute in parallel. Any transaction with dependencies is delayed until its parent transaction(s) complete. This avoids race conditions and ensures that no transaction is executed in an invalid state.

All shared structures used during DAG scheduling and commit (e.g., state buffers, world state locks) are synchronized using thread-safe mechanisms, ensuring that data races or interleaved commits do not corrupt state.

Endorsement Validity and Expiry: To prevent stale or reused endorsements from affecting correctness, each endorsed transaction includes a validity timeout. If a transaction does not get committed within this time window, its endorsement is invalidated and the transaction is dropped. This mechanism prevents accumulation of expired or invalid dependency links and keeps the endorsement context clean.

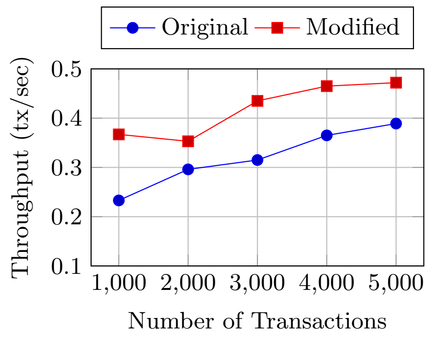
Security Summary: In summary, our changes do not bypass any of the validation or consensus stages in Fabric. They preserve:

- **Ledger correctness:** All committed transactions are validated and applied in a version-consistent order.
- **Execution determinism:** DAG execution guarantees identical ordering and results across peers.
- **Concurrency safety:** All parallel execution is strictly controlled using dependency-aware scheduling.
- **Peer consistency:** All peers construct the same DAG and validate transactions independently.

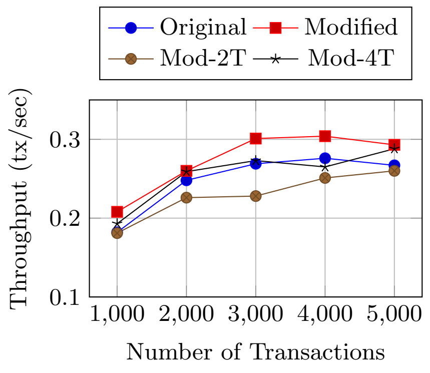
Thus, the proposed system maintains the security, correctness, and trust guarantees of Hyperledger Fabric while improving its performance and scalability.

A.3 Asset-Transfer Contract Results

This section summarizes the experimental findings for the Asset-Transfer smart contract, evaluating performance across different execution strategies: the baseline Original Fabric, the Modified Fabric with Dynamic Threads, and fixed-threaded variants (2 threads and 4 threads per DAG level). The results are grouped by transaction volume and dependency ratio to assess performance under varying conditions.



(a) Throughput vs Transactions

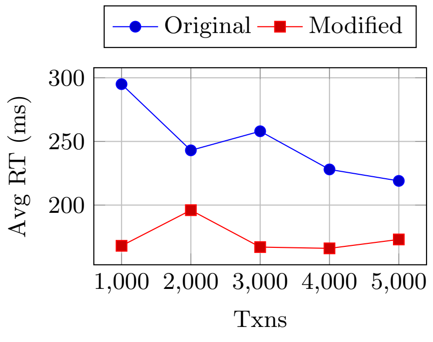


(b) Throughput vs Threads

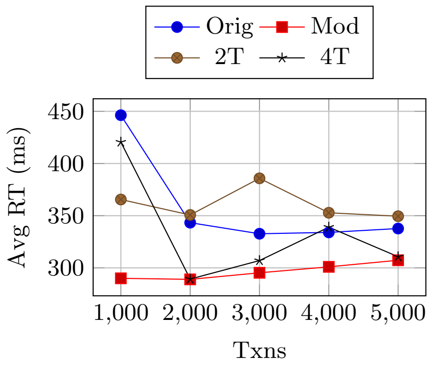
Fig. 9: Throughput Analysis of Asset Transfer Contract on Fabric Variants

Experiment 1: Throughput vs Number of Transactions
The Modified Fabric demonstrated consistent throughput improvement over the Original Fabric. For example, at 5000 transactions, throughput increased from 0.389 tx/sec to 0.472 tx/sec. The fixed-threaded versions (2T and 4T) showed moderate performance improvements but were generally less adaptive. The 4-threaded variant showed better throughput than the 2-threaded one, but the

Modified Fabric with dynamic thread allocation outperformed both by adjusting concurrency per DAG level based on available cores and transaction density.



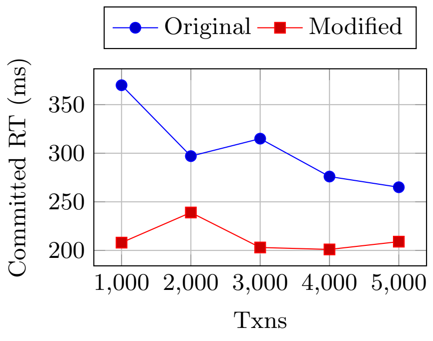
(a) All Txns



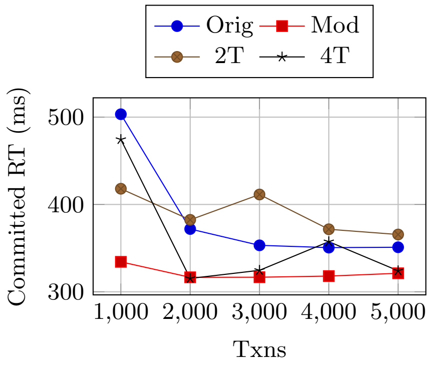
(b) All Txns - Threads

Fig. 10: Average Response Time - All Transactions

Experiment 2: Average Response Time vs Number of Transactions In terms of average response time across all transactions, the Modified Fabric offered lower latency in most scenarios. For instance, at 1000 transactions, latency dropped from 295 ms to 168 ms. However, both 2-threaded and 4-threaded versions showed inconsistent behavior: sometimes approaching the Modified Fabric's performance and other times suffering from overhead, particularly in the 4T configuration due to excessive context switching and synchronization.



(a) Committed Txns



(b) Committed Txns - Threads

Fig. 11: Average Response Time - Committed Transactions

Experiment 2 (Committed Transactions):
The response time for committed transactions also improved significantly with the Modified Fabric. For instance, at 1000 transactions, latency dropped from 370 ms (Original) to 208 ms (Modified). While the 2T and 4T versions yielded occasional benefits (e.g., 315.3 ms for 4T at 2000 transactions), they frequently underperformed compared to the adaptive Modified Fabric, which maintained a consistent edge.

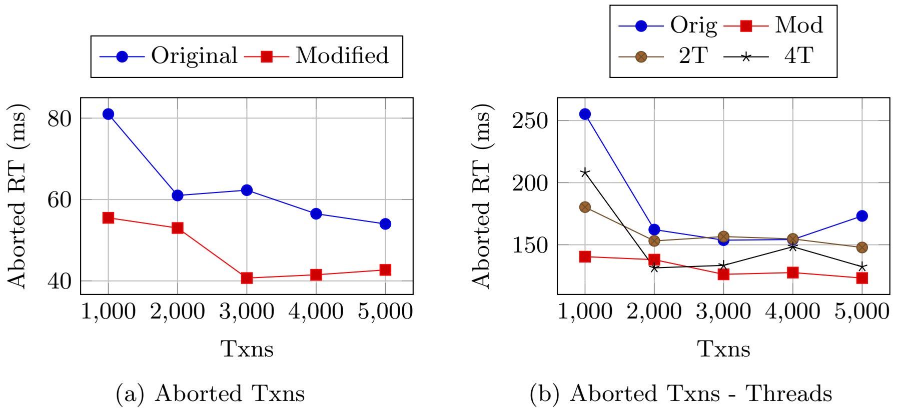


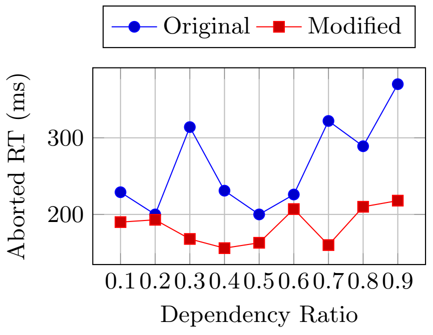
Fig. 12: Average Response Time - Aborted Transactions

Experiment 2 (Aborted Transactions):
The Modified Fabric also reduced the response time for aborted transactions. At 1000 transactions, the average latency dropped from 81 ms to 55.5 ms. The fixed-threaded approaches lagged behind in this category, with the 4-threaded variant recording 208.1 ms at the same transaction level, indicating inefficiencies in abort handling with rigid thread allocation.

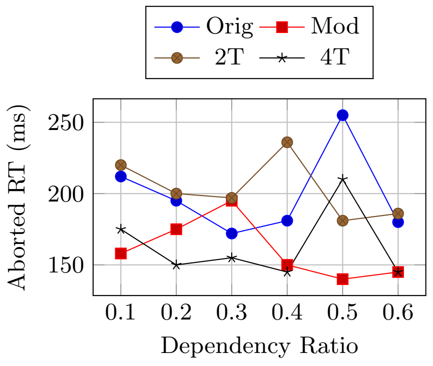
Experiment 3: Response Time vs Dependency Ratio
As inter-transaction dependencies increased, the Original Fabric showed a dramatic increase in latency (e.g., 642 ms at dependency ratio 0.3). In contrast, the Modified Fabric was significantly more stable, maintaining response times in the 290–375 ms range across all ratios. The 2T and 4T variants showed mixed outcomes—performing well at lower dependency ratios but struggling at higher ones, likely due to rigid thread scheduling unable to adapt to dependency-heavy DAG structures.

Experiment 3 (Committed Transactions):
The Modified Fabric consistently outperformed the baseline in response time for committed transactions under all dependency ratios. At 0.3, the Original took 707 ms versus 353 ms for Modified. Again, fixed-threaded approaches sometimes

Fig. 13: Experiment 3: Dependency Ratio vs Average Response Time



(a) Aborted Txns



(b) Aborted Txns - Threads

Fig. 14: Avg Response Time vs Dependency Ratio for Aborted Transactions

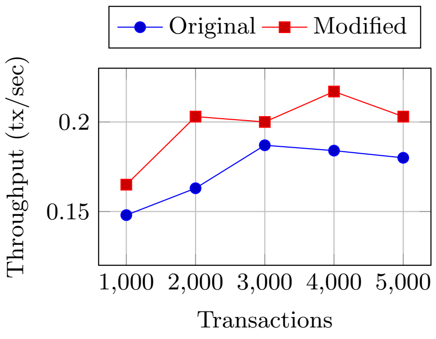
matched the Modified Fabric (e.g., 344 ms for 4T at 0.2) but failed to maintain consistent gains due to their static nature.

Experiment 3 (Aborted Transactions):
Aborted transaction latency also favored the Modified Fabric. For example, at dependency ratio 0.3, the Modified Fabric recorded 168 ms, while the Original showed 314 ms. The 2T and 4T variants occasionally yielded comparable results (e.g., 145 ms for 4T at 0.6), but performance varied based on workload characteristics, again showing the superior adaptability of the dynamic threading model.

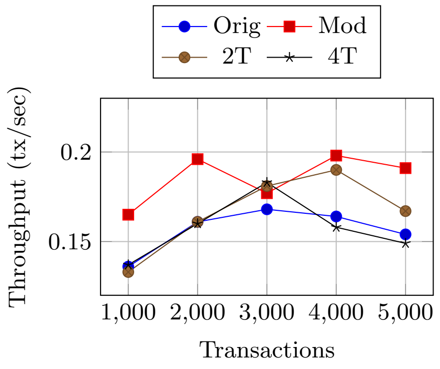
A.4 Wallet Contract Results

This section of appendix presents a detailed performance evaluation of the **Wallet Contract** deployed on Hyperledger Fabric. The results compare the **Original Fabric** against an optimized **Modified Fabric** version, including multi-threaded variants using **2-thread** and **4-thread** configurations.
Experiment 1: Throughput vs Number of Transactions analyzes throughput (in transactions per second) as the number of submitted transactions scales from 1,000 to 5,000.

– **Original Fabric** achieved a maximum throughput of **0.187 tx/sec**.
– **Modified Fabric** improved performance across all cases, peaking at **0.217 tx/sec**.
– The multi-threaded configurations did not consistently improve throughput; in some cases, throughput degraded due to thread contention or synchronization overhead.



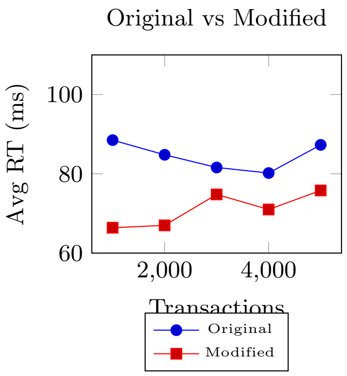
(a) Original vs Modified



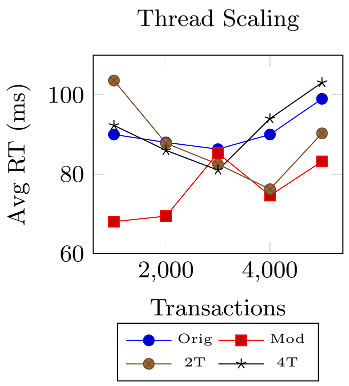
(b) Modified Fabric with Threads

Fig. 15: Wallet Contract - Experiment 1: Throughput vs Number of Transactions

Fig. 16: Wallet Contract - Experiment 2: Avg Response Time for All Transactions



(a) All Txns



(b) All Txns - Threads

Experiment 2: Average Response Time vs Number of Transactions measures the average response time (in milliseconds) under increasing transaction volume, separated by:

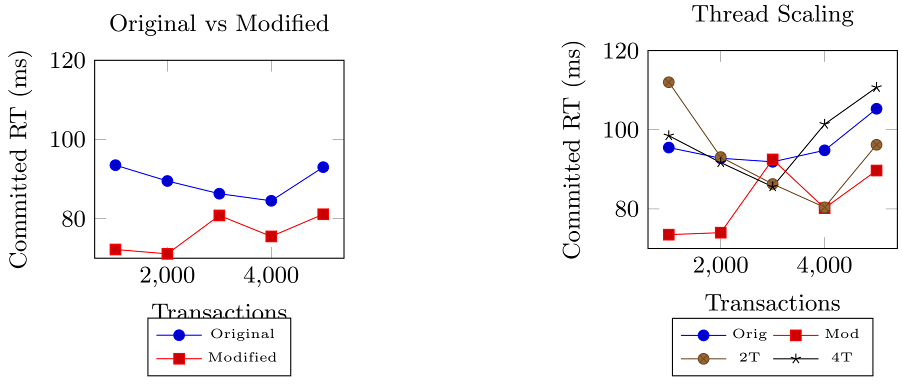
All Transactions:

- The Modified Fabric reduced average response time significantly (e.g., **66.4 ms** vs **88.5 ms** at 1,000 transactions).
- The 2-thread configuration often provided a slight benefit; the 4-thread version showed inconsistent gains.

Committed Transactions

- The Modified Fabric consistently outperformed the Original, with improvements up to **20%**.
- The 2-thread configuration provided better latency under higher load compared to 4-thread, which occasionally suffered from synchronization delays.

Fig. 17: Wallet Contract - Experiment 2: Avg Response Time for Committed Transactions



(a) Committed Txns (b) Committed Txns - Threads

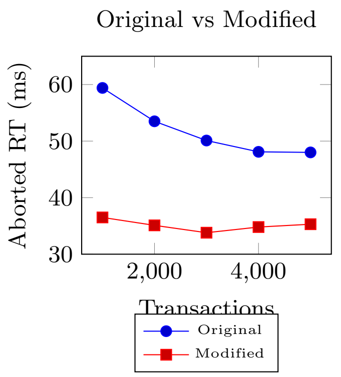
Aborted Transactions

- A marked improvement was seen in the Modified Fabric, reducing response time from **59.4 ms** to as low as **33.8 ms**.
- Multi-threaded versions did not consistently improve performance for aborted transactions and occasionally increased variability.

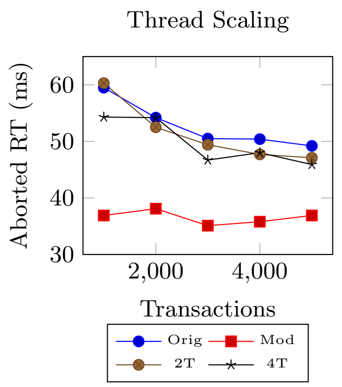
Experiment 3: Average Response Time vs Dependency Ratio evaluates system latency across increasing dependency ratios (0.0 to 0.9), simulating transaction contention and logical dependencies.

All Transactions:

Fig. 18: Wallet Contract - Experiment 2: Avg Response Time for Aborted Transactions

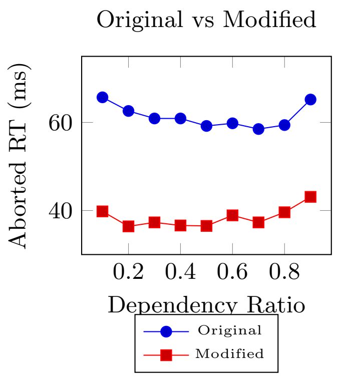


(a) Aborted Txns

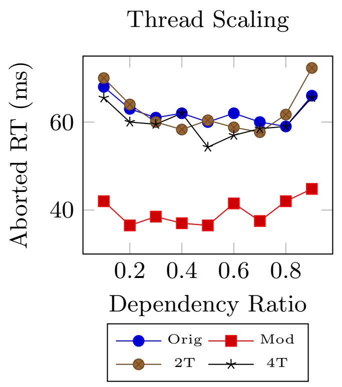


(b) Aborted Txns - Threads

Fig. 19: Wallet Contract - Experiment 3: Average Response Time vs Dependency Ratio



(a) Aborted Txns



(b) Aborted Txns - Threads

(c) Avg Response Time for Aborted Transactions

- The Modified Fabric consistently outperformed the Original, with response times ranging from **62.6 ms** to **73.3 ms**.
- Multi-threaded variants yielded mixed results—sometimes improving response time (e.g., at ratio 0.3), but occasionally increasing latency due to scheduling inefficiencies.

Committed Transactions:

- The Modified Fabric offered smoother response time curves across varying ratios.
- The 2-thread and 4-thread variants provided benefits at some ratios (e.g., 0.3 to 0.6), though performance often peaked unpredictably at higher contention levels.

Aborted Transactions:

- The Modified Fabric demonstrated significant improvements, with response times reducing from over **65 ms** to as low as **36 ms**.
- Multi-threaded variants exhibited moderate benefits in some cases, but also introduced occasional spikes in response time.