Taming the Beast of User-Programmed Transactions on Blockchains: A Declarative Transaction Approach

Nodirbek Korchiev
North Carolina State University
Raleigh, North Carolina, USA
nkorchi@ncsu.edu

Akash Pateria
Oracle
Seattle, Washington, USA
pateria.akash77@gmail.com

Vodelina Samatova
North Carolina State University
Raleigh, North Carolina, USA
vsamato@ncsu.edu

Sogolsadat Mansouri
North Carolina State University
Raleigh, North Carolina, USA
smansou2@ncsu.edu

Kemafor Anyanwu
North Carolina State University
Raleigh, North Carolina, USA
kogan@ncsu.edu

ABSTRACT

Blockchains are being positioned as the "technology of trust" that can be used to mediate *transactions* between non-trusting parties without the need for a central authority. They support transaction types that are *native* to the blockchain platform or *user-defined* via user programs called *smart contracts*. Despite the significant flexibility in transaction programmability that smart contracts offer, they pose several usability, robustness, and performance challenges.

This paper proposes an alternative transaction framework that incorporates more primitives into the native set of transaction types (reducing the likelihood of requiring user-defined transaction programs often). The framework is based on the concept of *declarative blockchain transactions* whose strength lies in the fact that it addresses several of the limitations of smart contracts, simultaneously. A formal and implementation framework is presented and a subset of commonly occurring transaction behaviors is modeled and implemented as use cases, using an open-source blockchain development platform, BigchainDB as the implementation context. A performance study comparing the declarative transaction approach to equivalent smart contract transaction models reveals several advantages of the proposed approach.

1 INTRODUCTION

Blockchains, as a technology for mediating and managing transactions between non-trusting parties, is becoming an increasingly popular concept. They are decentralized, fully replicated, append-only databases of *transactions* that are *validated* through a large, distributed consensus. These characteristics ensure that blockchain contents are tamper-proof and that no single authority controls a blockchain's operation and contents, conferring a good degree of trust in them.

Initially aimed at cryptocurrency, blockchain technology now extends to areas seeking data control and ownership decentralization, primarily for privacy and efficiency. This includes healthcare, [7, 30], supply chain [17, 39, 49], decentralized finance (Defi) [37, 45], governance [29], web browsing, gaming, social media, and file sharing/storage [8].

Blockchain transactions typically involve digital asset management aligned with business activities. The fundamental transaction type is asset TRANSFER between accounts, a *native* function in most blockchains. To address the diverse needs of modern applications, blockchains have evolved to include user-designed transactions known as *smart contracts* [40]. These contracts execute business operations and adhere to specific conditions. Examples include auction bidding and regulated patient record management. Recent survey [3] indicates the existence of over 44 million smart contracts on the Ethereum blockchain alone.

Problem: Smart contracts, despite their flexibility, face adoption barriers due to several issues: (i) They require significant effort in creation and verification, offer limited reusability across platforms, and constrain automatic optimization possibilities. (ii) Vulnerable to user errors and security breaches, they pose financial risks, exemplified by the DAO attack [31] that resulted in a loss of approximately 3.6M ETH (about $12.02B[1]). (iii) Many transactional behaviors in smart contracts, embedded in programming structures, remain hidden on the blockchain, hindering their utility in complex data analysis. Blockchain ledgers record method calls, but the semantics of user-programmed behavior remain opaque because call signatures alone don't reveal full logic. Understanding often requires source code, which is optional to publish and frequently omitted since only executables are required for deployment. (iv) Their execution involves higher latency and costs compared to native transactions. The lack of validation semantics for these user-programmed transactions complicates concurrency conflict management, leading most platforms, including Ethereum, to adopt sequential execution, which lowers throughput.

Declarative smart contracts [12], domain-specific languages [47], and smart contract templates [26] aim to ease creation and verification processes. However, they fall short in addressing performance, throughput, queryability, and other transactional model challenges in smart contracts.

To address these limitations, we propose the concept of declarative transactions, inspired by database systems, as a way to abstract transactional behavior on blockchains. Similar to how SQL revolutionized relational databases by allowing users to specify what data to retrieve or manipulate without detailing how to execute the query, declarative transactions enable developers to specify what transactional behavior is desired, while the system determines how to execute it. This abstraction offers several advantages over imperative approaches, including automatic **optimization of transaction execution**, improved code **reusability**, better **scalability**, and **enhanced queryability** for complex data analysis.

[1] As of December 20, 2024, 1 ETH = $3,338.66 USD. Source: Investing.com.

1.1 Contributions:

This paper investigates the feasibility and impact of lifting transactional behaviors typically found in smart contracts into the core blockchain layer as *native* transactions. Specifically, we propose:

(1) a *declarative and typed* blockchain transaction model that includes the novel concept of *nested blockchain transactions*, as a foundation for modeling transactional behavior on blockchains.

(2) concrete declarative blockchain transaction modeling of a sample transactional behavior represented in many smart contracts of the most popular blockchain application category - marketplaces.

(3) an implementation framework for declarative blockchain transactions that builds on BigchainDB blockchain development platform [1], extending its transaction modeling and validation infrastructure.

(4) a comparative performance and usability evaluation of the declarative transaction model vs. imperative transaction model that uses Ethereum smart contracts as the basis. The evaluation results demonstrate that the declarative transaction method significantly outperforms smart contracts, achieving improvements by a factor of 635 in latency and a minimum of 60 in throughput.

The rest of the paper is organized as follows: Section 2 provides motivation and background information on blockchain native transactions, smart contracts, and BigchainDB. Section 3 introduces the formal blockchain transaction model and novel concepts of *Non-nested* and *Nested* transactions. Section 4 provides implementation details of the concepts presented in Section 3. Section 5 reports on the comparative experiments conducted to evaluate declarative and imperative approaches. Section 6 reviews the literature on the topic, while Section 7 discusses the limitations and scenarios where declarative transactions may face challenges. Finally, we conclude the paper with a summary and future work in Section 8 and acknowledgement in Section 9.

2 MOTIVATION AND BACKGROUND

2.1 Smart Contracts in Blockchain Marketplaces

Most blockchain platforms typically support only basic transactions like TRANSFER, with Ethereum adding more complex types, such as multi-signature transactions that focus more on operational semantics rather than behavior. Consequently, most applications rely on smart contracts to extend functionality, which comes with inherent limitations. For example, in setting up a decentralized marketplace for procurement and supply chain management, smart contracts are needed for actions like posting service requests by buyers or supply bids by providers, involving complex metadata management through user-programmed methods.

Example. *Buyers can post requests (e.g., for manufacturing services), and providers (e.g., 3-D printer manufacturers) can respond with bids. These transactions involve detailed metadata such as quantity, product type, and deadlines, managed through the* `createrfq` *method for requests and* `createbid` *for bids, which also includes the asset's production capabilities like certifications and work history. This setup mimics traditional auctions where the asset that forms the basis of a bid is some form of payment. Fig. 1 shows the skeleton of an* Ethereum *smart contract modeling such a procurement reverse auction marketplace.*

```
contract smartAuction {
struct request { ... }
struct asset { ... }
struct bid { ... }
mapping (uint => request) public requests;
mapping (uint => asset) public assets;
... ... ... ...
constructor() public {
    ...
function createrfq (unit rfqID, string[] memory rfqCapabilities) public{
    // creating new rfq by the requestor }
function createasset (unit assetID, string assetCapabilities) public {
    // creating new asset for the bid by a bidder }
function createbid(uint bidasset_id, uint request_id) public payable {
    ...
    // internal function call
    function checkValidBid(uint bidasset_id, uint request_id)
    emit bidCreated(bidasset_id, bid_count, msg.sender);  }
function checkValidBid(uint bidasset_id, uint request_id)
        public view returns (bool isValid)
    if (!requests[request_id].isSet) {
        revert("Request does not exist."); }
        ...
            if (compareStrings(requests[request_id].rfq_capabilities[i],
            ...
    if (flag >= requests[request_id].rfq_capabilities.length) {
        ...
    } }
function transferasset (address newOwner, unit rfqID, unit assetID) {
    // transferring of the bid's asset ownership
        (to the escrow account holder when the bid is submitted or
        to the requestor when the bid is accepted) }}}}
```


Figure 1: Smart Contract sample implemented in Solidity

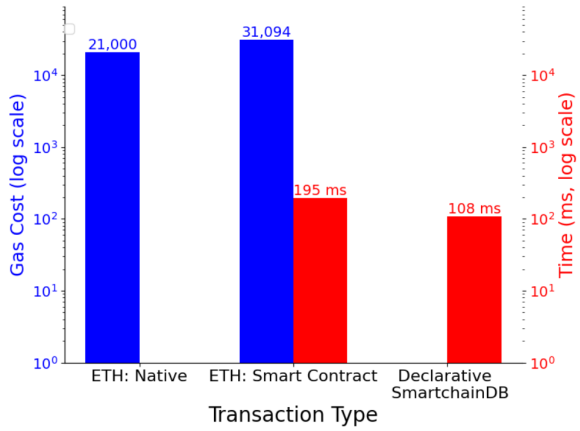


Figure 2: Comparison of runtime (in milliseconds) and gas cost for executing a TRANSFER **transaction using Ethereum native transactions, Ethereum Smart Contracts, and Declarative transactions. The left y-axis represents gas costs, while the right y-axis represents execution time.** ***Note: For ETH: Native and Declarative SmartchainDB, certain metrics are not measured due to the lack of programmable execution (time) or fee structure (gas).***

Observations:

Native transactions such as TRANSFER automatically handle validation against errors like double-spending. However, with smart contracts, developers must manually code such validations, as seen with methods like `checkValidBid()`. In an auction context, this includes ensuring all non-winning bids are refunded (if escrow deposits were required), verifying ownership of bidding assets, and managing bid withdrawals and deletions by authorized parties only.

Smart contracts also manage a broad range of transaction and asset metadata, which are not directly visible on the blockchain. This includes everything from user content (e.g., documents, audio/videos) [53] to digital twins of physical assets like diamonds

[4], cars and houses [51] etc., even ownership certificates for various physical assets [32–34]. These assets are stored in complex structures that require deep technical knowledge to navigate. In our example, metadata for requests, bids, and their underlying assets are represented as the struct variables, `request`, `bid`, and `asset` respectively. The mappings of accounts to bids and requests are implemented using program map data structures (e.g. *requests*) and not wallet accounts. Consequently, a query like finding open service requests for 3-D printing manufacturing capabilities may be of interest to 3-D printing manufacturing providers. However, this query involves specifying conditions on the metadata of the service request that are not queryable on the blockchain. Even more complex queries are critical for supporting tasks like fraud analysis or other business decision-making tasks, but unfortunately, they cannot be supported easily. Thirdly, the smart contract execution model has more overhead than that of native transactions. An experiment comparing the native TRANSFER transaction to its smart contract equivalent in Figure 2 showed that using smart contracts instead of native transaction primitives increased GAS costs by 40% in Ethereum, reflecting higher transaction latencies and variable execution fees that depends on the contract's runtime behavior. Ethereum native transactions show zero time because they rely on predefined primitives that do not involve programmable execution. In contrast, the declarative SmartchainDB lacks a measurable "gas cost" because it operates without a fee structure. Unlike Ethereum's native transactions, smart contract performance can be unpredictable because it's tied to network conditions rather than fixed processing rules.

2.2 Rationale for Approach

Introducing more native transaction types is one way to minimize this dependence on smart contracts. However, the exact set of native transaction types necessary to mitigate the burden of smart contracts is unclear. Further, the best strategy for extending mainstream blockchain transaction models requires some investigation. In this paper, we exploit the fact that a large percentage of the transaction calls on the blockchain are associated with marketplace applications [50]. Consequently, we focus our attention on transaction types such as BID, ACCEPT_BID, REQUEST etc. With respect to strategy, we admit a bias towards declarative specifications with the hope of ultimately enabling some of the database-style optimization strategies for improving performance and scalability of blockchains. To support the exploration of these ideas, a platform with a stratified or layered and flexible architecture would be ideal. These requirements led to the choice of BigchainDB as our implementation context. BigchainDB is not an actual blockchain (e.g. does not have any native assets) but a blockchain development platform that can be used to implement different kinds of blockchains from permissioned to public blockchains. Using such a platform allows us to insulate our efforts from unrelated issues.
Importantly, this choice does not limit the applicability of our approach. Our techniques can be implemented on other blockchain systems as well.

Comment about objective: This paper does not aim to suggest some minimal set of blockchain transaction primitives. Rather, it aims to demonstrate how declarative blockchain transaction modeling can be achieved and its potential benefits. To this end, it introduces a set of primitives relevant to marketplace applications. The hope is that this set can be extended over time resulting in a corresponding decrease in the dependence on smart contracts, at least for some categories of blockchain applications.

3 APPROACH

Our approach is based on extending the BigchainDB platform which provides some foundational blockchain components but is not a concrete blockchain. It provides support for usual transaction types: CREATE and TRANSFER. Its architecture supports an extensible and customizable transaction model through the transaction schema layer. For transaction execution, BigchainDB's architecture runs on a network where each node operates three services: BigchainDB server, *Tendermint*, and *MongoDB*. The BigchainDB server processes transactions, while *Tendermint*, a Byzantine Fault Tolerant engine, handles consensus without mining, using a Proof-of-Stake mechanism.

Our approach introduces a conceptual transaction model that encapsulates the existing two transaction types in BigchainDB, *in addition to, introducing additional ones.* The extended transaction model also introduces the concept of *nested blockchain transactions* to allow modeling of more complex transaction types. It also formalizes the validation schemes for each transaction type. The second dimension of our contributions is the *extensions to transaction execution infrastructure - specifically the blockchain server and the blockchain storage* to provide support for the newly proposed transaction types. A novel concept that is introduced in this context is a *recovery model for nested transactions.*

3.1 Formal Conceptual Model for Blockchain Transactions

Our formal transaction model defines key components necessary for a transaction: the asset involved, the participating accounts identified by public keys, the type of transaction, and protocols for automated validation of semantics, such as preventing *double spend* errors in TRANSFER transactions. The model is based on the number of sets:

- a set $\mathcal{PBPK} = \{\mathsf{pbpk_i} = < \mathsf{pb_i}, \mathsf{pk_i} >\}$ of public-private key pairs. The pair $< \mathsf{pb_i}, \mathsf{pk_i} >$ represents account/owner *i*. We denote a subset $\mathcal{PBPK}\text{-}\mathcal{R}es \subseteq \mathcal{PBPK}$ as reserved accounts i.e. system or admin accounts.
- sets $\mathcal{L}$ – a set of literals. $\mathcal{RK}, \mathcal{RV} \subseteq \mathcal{L}$ is a set of string literals that are reserved keywords and values, respectively. We assume a specific subset of reserved values $\mathcal{OP} \subseteq \mathcal{RV}$ that are the names of transaction operations, e.g., CREATE, TRANSFER, and so on.
- a set $\mathcal{S} \subseteq \mathcal{L}$ is a set of strings that are called digital signatures, which are associated with two functions such that given a message string $\mathtt{m}$: $\mathtt{sign}(\mathsf{pk}, \mathtt{m})$ returns a signature string $\mathtt{s} \in \mathcal{S}$ and $\mathtt{verify}(\mathtt{s}, \mathsf{pb}, \mathtt{m})$ is a boolean function that returns `True` if the corresponding public key can be used to decrypt the signature and recreate the signed message $\mathtt{m}$. We can also have a more complex string made up as a function of multiple signatures. This is used in the case where an asset is controlled by a group of entities who must sign transactions on the asset. We use $\mathsf{ms_{i,j,k}}$ to denote such a multi-signature string from using signatures generated with private keys $\mathsf{pk_i}$, $\mathsf{pk_j}$, $\mathsf{pk_k}$.
- $\mathcal{AS}$ – the set of all blockchain assets where each blockchain asset A is a tuple $< (\mathsf{k_i}, \mathsf{v_i}), \mathtt{amt} >$ where $(\mathsf{k_i}, \mathsf{v_i})$ is a nested set of key-value pairs such that each $\mathsf{k_i} \in \{\mathcal{L}$ - $\mathcal{RK}\}$ and $\mathsf{v_i} \in \mathcal{L} \cup \mathtt{A}$ and `amt` is a non-negative number of shares that an asset holds.

- $\mathcal{T}$ - set of all blockchain transactions

Definition 1. (Transactions). A transaction $\mathtt{T} \in \mathcal{T}$ is a complex object $< \mathtt{ID}, \mathtt{OP}, \mathtt{A}, \mathtt{O}, \mathtt{I}, \mathtt{Ch}, \mathtt{R} >$ s.t.:

- ID - a globally unique string identifier
- $\mathtt{OP} \in \mathcal{OP}$ i.e. the name of transaction operation
- $\{\mathtt{A}\} \subseteq \mathcal{AS}$ - set of assets
- {O} - a set of transaction output objects $\{o_1, o_2, ...o_m\}$. $\mathtt{T}.o_k$ is used to denote the k^{th} output of transaction some T . Since assets can be divisible, the different outputs can hold different numbers of shares of some asset $\mathtt{A}_i$. Consequently, each of T's output object o_j is a tuple $< pb_i, \mathtt{A}_i.amt, pb_i^{prev} >$, where $\mathtt{A}_i.amt$ is the number of shares of A_i associated with the j^{th} output of T, i.e., $\mathtt{T}.o_j[1]$ that denotes pb_i is a set of public keys of the owners or controllers of those shares, and $\mathtt{T}.o_j[3]$ that denotes pb_i^{prev} is a set of public keys of previous owners.
- I - a set of transaction input objects $\{i_1, i_2, ...i_n\}$. We use $\mathtt{T}.i_k$ to denote the k^{th} input of some transaction T. Each input object i_k is a tuple $< \mathtt{T}'.o_b, ms_{u,v,w} >$, where $\mathtt{T}'.o_b$ is the output that is being "spent" by this input (in this case, the o_b is an output of some T′) can be referenced by the notation $\mathtt{T}.i_k[1]$ meaning it is the first element of the input $\mathtt{T}.i_k$. $ms_{u,v,w}$ is the signature string formed from the private keys that should be the signatures of the assets' owners.
- Ch - A set of children transactions. A child transaction is a transaction that depends on the outcome of a preceding parent transaction. It is triggered by the results or changes initiated by the parent transaction, ensuring that subsequent steps align with established rules and maintain workflow integrity.
- R - a reference vector of referenced transactions by their ID. Referencing a transaction differs from spending it, as referencing does not result in the consumption of its output.

Definition 2. (Nested transactions). Blockchain transaction T is *Nested* transaction if the following conditions are satisfied:

- It contains at least one child transaction, denoted as $|Ch| \geq 1$.
- The parent transaction is considered committed if and only if all its child transactions have been committed.
- For any parent transaction $\mathtt{T}_{parent}$, there exists at least one child transaction T within its children set Ch such that every output of $\mathtt{T}_{parent}$ is included within the outputs of T, expressed as $\forall \mathtt{T}_{parent}, \exists \mathtt{T} \in \mathtt{Ch} : \mathtt{T}_{parent}.o \subseteq \mathtt{T}.o$.

Nested blockchain transactions, as defined, incorporate the principle of *eventual commit* semantics, a commitment that is realized through the strategic use of *escrow* mechanisms. This guarantees that a parent transaction is committed only after the successful commitment of all its child transactions.

3.2 Declarative Transaction Types and Transaction Workflow

We introduce a novel typing scheme over the set of all blockchain transactions $\mathcal{T}$ that defines a blockchain transaction type $\tau_\alpha = < \mathtt{T}_\alpha, \mathtt{C}_\alpha >$ where τ_α is the subset of transactions in $\mathcal{T}$ that have OP = α and a set of conditions $\mathtt{C}_\alpha$ defined in terms of a transaction's inputs and outputs. There are *Non-nested*: CREATE, TRANSFER, REQUEST, BID RETURN, and *Nested*: ACCEPT_BID transaction types. We say a transaction T is *valid* with respect to a transaction type $\tau_\alpha = < \mathtt{T}_\alpha, \mathtt{C}_\alpha >$ if it meets all the conditions in $\mathtt{C}_\alpha$. For brevity, we present one representative transaction type from the *Non-nested* and *Nested* transaction categories, BID and ACCEPT_BID, respectively. The formal models for the remaining transaction types are available in the extended version of our paper [6].

Definition 3. (BID **Transaction type**). BID transaction is usually an offer transaction for something being sold or in the context of our procurement example, a REQUEST being made. We make the assumption that typically some asset is used to guarantee a bid and is typically held in some form of escrow account. Given this perspective, a BID can be represented as $\tau_{\mathtt{BID}} = < \mathtt{T}_{\mathtt{BID}}, \mathtt{C}_{\mathtt{BID}} >$ where: $\mathtt{T}_{\mathtt{BID}} = <$ ID, BID, A, O, I, Ch, R $>$.
ID = 95879..., OP = BID, A = {*asset_id* : 65*be*4..}
I ={< *KmSd2*..., 1 >, $ms_{\mathtt{YM2sd4hn...}}$ },
O = { < [7*EAsH*..], [1] >}, Ch = {∅}, R = [6*ae*47...]


```
{"asset": {
    "data": {
        "asset_id":"65be4152228c6eb63dkl...",
    }
 },
 "R" :["6ae4736e02d396fc8aa74....]"
 "id": "9587986b9ffa81261ff19528eaf1f1...",
 "inputs": [{
   "fulfillment": "KmSd2PnKjebRlzr3...",
   "fulfills": {
       "output_index": 0,
       "transaction_id": "7241s152228c6eb..."},
   "owners_before": [ "YM2sd4hngh2Drd..." ]
 }],
 "metadata": { "about": "..."  },
 "operation": "BID",
 "children" : {}
 "outputs": [{
   "amount": "1",
   "condition": {
     "details": {
       "public_key": "7EAsHUGQ15LdS2NoX...",
        "type": "ed25519-sha-256" },
     "public_keys": [ "7EAsHUGQ15LdS2NoX..." ]
 }]}
```


Figure 3: BID transaction type

For example, the tuple above represents a BID transaction, with its details illustrated in Fig 3. In this BID transaction, the key behavior involves transferring an asset to an escrow account. The escrow account is defined by the output field, where the public_key of the escrow account is specified. Here's a detailed explanation of the process:

- The input asset, identified by the asset_id 65*be*4..., is transferred to an escrow account. This is achieved by transferring ownership of the asset through the output section, where the public_key of the escrow account is set as the new owner.
- The cryptographic fulfillment, indicated by *KmSd2P*..., ensures that the previous owner's conditions are satisfied before the asset can be transferred. This signature proves that the previous owner authorizes the transfer of the asset to the escrow account.
- The output condition and amount for this transaction, [7*EAsH*...] for the public key and 1 for the amount, define how much of the asset is being transferred and to whom (the escrow in this case).

This BID transaction is tied to a prior REQUEST transaction, identified by R = [6*ae*47...], which references the request being made. This is essential for validating the context of the BID, ensuring that the assets and references are properly linked to the request.

A BID transaction has the following set of boolean validation conditions $\mathtt{C}_{\mathtt{BID}}$:

(1) $|\mathtt{I}| \geq 1$ i.e. must be at least 1 input object
(2) $|\mathtt{R}| \geq 1$ i.e. reference vector must contain at least 1 element
(3) $\exists!\mathtt{T} \in \mathtt{R} : \mathtt{T.OP} == \text{REQUEST}$, i.e., there exists exactly 1 REQUEST transaction in reference vector
(4) $\exists \mathtt{i} : \mathtt{T_{BID}.i[1].A.amt} > 0$ i.e. there exists at least one input object with none-null asset
(5) $\forall \mathtt{i} \in \mathtt{I}$, verify$(\mathtt{s_i}, \mathtt{pb_i}, \mathtt{m_i}) == True$
(6) $\forall \mathtt{j} \in \mathtt{T.o} : \mathtt{T.o_j[1]} = \mathcal{PBPK}\text{-}\mathscr{Res}$ i.e. The output of every BID transaction has to be sent to $\mathcal{PBPK}\text{-}\mathscr{Res}$ account
(7) $\mathtt{T.R.A} \subseteq \bigcup_{j=1}^{|\mathtt{I}|} \mathtt{T.i_j[1].A}$, where T.R.OP == REQUEST. The amount of the requested asset(s) must be a subset of the union of input bid assets.
(8) $\forall \mathtt{i} \in [1, |\mathtt{I}|]$, $\mathtt{T.i} == \mathtt{T.o_j}$, i.e., every transaction input i has to spend some transaction's $\mathtt{j}^{\mathtt{th}}$ output

Definition 4. (ACCEPT_BID **Transaction type**). ACCEPT_BID transaction is a *Nested* transaction that takes one or more BID as the parameters. Its semantics is to transfer the winning bid to the requester while unaccepted bids are transferred back the original bidders.

Formally, $\tau_{\text{ACCEPT\_BID}} = < \mathtt{T}_{\text{ACCEPT\_BID}}, \mathtt{C}_{\text{ACCEPT\_BID}} >$ where
$\mathtt{T_{BID}} = <$ ID, ACCEPT_BID, A, O, I, Ch, R $>$ with the following set of boolean validation conditions $\mathtt{C}_{\text{ACCEPT\_BID}}$
(1) $|\mathtt{I}| == n$ i.e. where n is the number of BIDs for 1 REQUEST
(2) $|\mathtt{R}| = 1$ i.e. reference vector must contain exactly 1 element
(3) $\exists!\mathtt{T}_{\text{ACCEPT\_BID}} \in \mathtt{R} : \mathtt{T.OP} == \text{REQUEST}$, i.e., there exists exactly 1 REQUEST transaction in reference vector
(4) $|\mathtt{CH}| == |\mathtt{I}|$ number of elements in children set is equal to the number of input objects
(5) $\forall \mathtt{i} \in \mathtt{I}$, verify$(\mathtt{s_i}, \mathtt{pb_i}, \mathtt{m_i}) == True$
(6) $\forall \mathtt{T} \in \mathtt{Ch} : \mathtt{T}_{\text{ACCEPT\_BID}}\mathtt{.o} \supset \mathtt{T.o}$, i.e., The output of parent ACCEPT_BID is a proper superset of every transaction's output in the children set
(7) $\forall \mathtt{k} \in [1, |\mathtt{I}|]$, $\mathtt{T.i_k[1][1]} == \mathcal{PBPK} - \mathscr{Res}$, i.e. each input has to spend an output of some TRANSFER transaction that has an account owner $\mathcal{PBPK}\text{-}\mathscr{Res}$
(8) $\forall \mathtt{j} \in [1, |\mathtt{O}|] \forall \mathtt{k} \in [1, |\mathtt{I}|] : \mathtt{T.o_j[1][1]} == \mathtt{T.i_m[1][3]}$ where $\mathtt{T.o_m[1][3]} \wedge T.o_j.ID \neq T_{\text{ACCEPT\_BID}}.A.ID \wedge T.i_k.ID \neq T_{\text{ACCEPT\_BID}}.A.ID$ is $\mathtt{pb}_\mathtt{i}^{\mathtt{prev}}$ previous owner of $T.i_m$, i.e., every unaccepted output of ACCEPT_BID transaction must be transferred back to the original bidder
(9) $\exists!\mathtt{T.o} : \mathtt{T.o[1]} == \mathtt{T}_{\text{ACCEPT\_BID}}\mathtt{.R.o[1]}$, i.e., there exists exactly one output transaction that transfers asset to the requester.

In a similar way to BID the ACCEPT_BID can be represented in the following tuple form $\mathtt{T}_{\text{ACCEPT\_BID}}$:
ID = $b64c6$..., OP = ACCEPT_BID, A = $\{win\_bid\_id : 95879...\}$
I = { <HmkC1...,1>, $\mathtt{ms}_{\mathtt{HmkC1...}}$, <MfcDL...,1>, $\mathtt{ms}_{\mathtt{HmkC1...}}$}
O = {<[HmkC1..], [1]>}
Ch = { <[HmkC1..], [1]>, <[fPjsA..], [1]> } , R = [$6ae47$..]

The tuple above can be described in the following way. For brevity, we will omit previously described general fields like ID, OP (operation), and O (output) and focus on transaction-specific fields. The asset A field anchors the transaction to the specific bid with id 95879... that has won acceptance, forming a bridge to the original offer. This transaction includes two Inputs, Ch, <[HmkC1..], [1]>, <[fPjsA..], [1]>, each representing the outputs from two different BID transactions for the same REQUEST, as indicated in the reference vector R.

Sometimes, complex transaction behavior may require composing multiple transactional primitives into a workflow which we define as follows:

Definition 5 (Blockchain transaction workflow). Transaction workflow is a sequence of transactions $\mathtt{T}_1, \mathtt{T}_2, ..., \mathtt{T}_\mathtt{n}$ where $\mathtt{T}_1$ is *head* that initiates the workflow and $\mathtt{T}_\mathtt{n}$ is *tail* of the sequence. The following condition must be true for a transaction in the sequence:

- $\mathtt{T}_1\mathtt{.i} = \emptyset$ Input of the transaction initiating workflow is null.
- $\forall\{\mathtt{T_j.i} - \{T_1\}\}\exists \mathtt{T.o_k}$ where $\mathtt{T.o_k}$ is committed. The input of any transaction in the sequence, except the *head* transaction, must come from a committed transaction.

Transaction workflow refers to a series of executions of different types of transactions in a specific order. The exact number of transaction types involved may vary depending on the workflow. An example can be the utilization of a reverse auction workflow within the context of supply chain procurement. Where the only valid workflows can be, CREATE, CREATE – TRANSFER, CREATE – REQUEST – BID – ACCEPT_BID – TRANSFER. This is a multistage process, where one side can REQUEST an execution of a particular item/task and the suppliers can show their interest through BID for this REQUEST, and, eventually, if the other side accepts the bid the workflow ends. Other scenarios may include a different number of steps, and/or their structure will be different, but the main point is that they all involve the same primitives.

4 TRANSACTION MODEL IMPLEMENTATION

Our implementation builds upon the BigchainDB platform by extending and modifying its core architectural components, while retaining its original consensus layer (*Tendermint*). Figure 4 illustrates the transaction life cycle and the key elements of SmartchainDB, the enhanced system developed as part of this work. At the core of our approach, transactions are defined using YAML schemas. Each transaction is validated according to its specific schema type by the *Driver* before submission to the *Server*. We have enriched the *Server* with specialized transaction validation algorithms for each type, enabling automatic transaction validation.

In terms of schemas, we have expanded the existing BigchainDB transaction types CREATE and TRANSFER, incorporating new components detailed in the transaction model (Section 3.1). This extension also includes schemas for new transaction types like REQUEST. On the storage front, the *MongoDB* collections within BigchainDB have been adjusted and expanded to support the novel transaction structures introduced in our model. Additionally, the *Server* component has been fortified with unique *transaction validation algorithms* for each transaction type, facilitating an automated and efficient validation process.

The transaction life cycle begins with the *client* providing a serialized transaction payload in JSON format. Subsequently, *Driver* utilizes the received payload to generate a transaction by employing pre-existing templates customized to each transaction type and signs it before submitting it to the *Server* (*"Prepare and Sign"*). At this stage, one of the validator nodes is chosen at random to act as the *receiver node*, which is responsible for the semantic validation of the transaction according to the rules for its type. Each transaction has associated $\mathtt{validateT}_\alpha$ method used by the validator nodes at the *Server* layer, e.g., $\mathtt{validateT_{BID}}$ for the BID transaction. At the network validator node, a transaction undergoes a secondary set of validation checks triggered by the

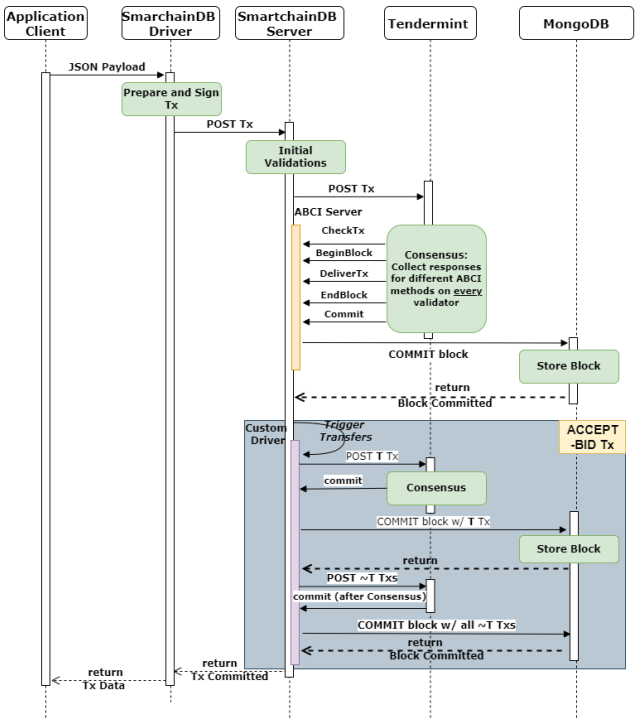


Figure 4: SmartchainDB **Transaction Life cycle**

`CheckTx` function. This step is implemented to verify that the validator node did not tamper the transaction and to add valid transactions to the local *mempool*. Once a transaction is successfully *committed* on more than 2/3 of validators, the final, third set of validation checks take place at `DeliverTx` stage before mutating the state. After accumulating validated transactions, the *Server* issues a *commit* call to store the newly accepted block to the local *MongoDB* storage. The *Server* awaits the response from *MongoDB* about the commit state. Depending on the transaction type, it may end the cycle and inform the *client* about the transaction's status or proceed to the internal *process* from where the response is returned with a successful transaction commit message. The *Driver* usually attaches a callback to the request, thus, the respective callback method is invoked when the transaction is committed or if any validation error is raised.

Observation: Blockchain transactions differ from distributed transactions in that they have a transaction "validation" phase done by each peer independently and this phase is delineated from the distributed consensus and commit phases. In the following, we elaborate on the implementation of the transaction validation algorithms for both Nested and Non-Nested transactions.

4.1 Implementation of *Non-nested* blockchain transactions

Fig. 5 illustrates a portion of the YAML schema that defines the transaction structure in SmartchainDB. This schema serves as the foundational framework for creating, validating, and processing transactions, ensuring that each transaction adheres to a uniform format. The schema enforces a strict structure by specifying required fields, including **id**, **inputs**, **outputs**, **operation**, **metadata**, **asset**, **children**, and **version**. Each of these fields comes with clearly defined constraints, such as data types, patterns, and references to other components within the schema.

```yaml
title: Transaction Schema
required:
  - id
  - inputs
  - outputs
  - operation
  - metadata
  - asset
  - children
  - version
properties:
  id:
    anyOf:
      - "$ref": "#/definitions/sha3_hexdigest"
      - type: "null"
  operation:
    "$ref": "#/definitions/operation"
  asset:
    "$ref": "#/definitions/asset"
  inputs:
    type: array
    items:
      "$ref": "#/definitions/input"
  outputs:
    type: array
    items:
      "$ref": "#/definitions/output"
  children:
    type: array
    items:
      "$ref": "#/definitions/children"
  metadata:
    "$ref": "#/definitions/metadata"
  version:
    type: string
```

```yaml
definitions:
  sha3_hexdigest:
    pattern: "[0-9a-f]{64}"
    type: string
  operation:
    type: string
    enum:
      - CREATE
      - TRANSFER
      - REQUEST_FOR_QUOTE
      - BID
      - ACCEPT
      - RETURN
  asset:
    type: object
    properties:
      id:
        $ref: "#/definitions/sha3_hexdigest"
      data:
        anyOf:
          - type: object
          - type: "null"
  output:
    type: object
    required: [amount, condition, public_keys]
  input:
    type: object
    required: [owners_before, fulfillment]
  children:
    type: array
  metadata:
    anyOf:
      - type: object
        additionalProperties: true
      - type: "null"
```

Figure 5: Transaction Schema in YAML

This ensures that every transaction meets predefined standards, making it easier to validate and interpret transactions across the system. Additionally, the schema supports flexibility through object references, allowing for modular and scalable transaction definitions while maintaining consistency.

Subsequently, each transaction is subjected to a schema validation method upon arrival at the *Server*. This method employs a schema validation algorithm, described in Algorithm 1, that receives a transaction object as input and yields a boolean variable, which signifies the transaction's validity as per the defined schema. The algorithm ensures structural adherence of the JSON transaction payload to the established blueprint.

For example, the `id` within the asset definition field imposes constraints that it must adhere to a specific format, as indicated by the reference to a *'sha3_hexdigest'*, ensuring that each transaction can be uniquely identified and verified. The operation field is restricted to only predefined operations like CREATE, TRANSFER, REQUEST, BID, etc. This constraint ensures that only allowable transaction types are processed within the SmartchainDB ecosystem. If an operation does not match this predetermined set, it is rejected during schema validation and is prevented from proceeding to the semantic validation phase.

During *semantic validation*, rules about permissions, required dependencies between transactions and conditions about assets are checked. For example, assume Alice responds to a REQUEST for bids by Sally with a BID transaction. Some of the required conditions to check about the bid include (i.) ensuring that Alice owns the asset used to support the bid i.e. she has the permission to spend the output of the CREATE transaction that created the asset; (ii.) and that the bid is in response to some request and meet some conditions (we ignore additional details). Fig. 6 illustrates the transaction dependencies (spending and reference) in the example. The permission dependencies for *Alice* are shown by the relationship $\texttt{PubK}_{\texttt{Alice}}$ and the input signature $\texttt{Sig}_{\texttt{Alice}}$) while *Sally*'s ownership of the REQUEST transaction is indicated by her signature $\texttt{Sig}_{\texttt{Sally}}$ on its input. The output of BID is owned by escrow (one of the system accounts) which holds bids until a winning bid is selected.

Algorithm 2 provides a high-level implementation for BID, incorporating semantic validation based on the validation conditions (VC) outlined in subsection 3.2. The primary function,

validateBidTx(), is executed during the initial validation on the receiver node and twice in the consensus phase on validator nodes (as depicted in Fig. 4). Initially, a MongoDB query (line 1) retrieves the REQUEST transaction for the specified rfq_id. The algorithm's first major check (line 4) confirms all transaction inputs, addressing semantics in VC 1-3. Ensuring input transaction correctness, related to VC 4-6, is covered in lines 6-8. A crucial aspect of BID validation, checking if a BID asset meets the required "capabilities", is based on VC 7 and implemented in lines 14-16. Finally, as BID entails aspects of a TRANSFER transaction, it undergoes additional semantic validation (VC 8) in the algorithm's concluding step (line 13).

4.2 *Nested* blockchain transactions (NBT)

The traditional "nested transaction" semantics is that a parent transaction is not committed unless child transactions have been committed so that parent transaction blocks on child transactions. A typical concern is the semantics of nested transactions in the presence of failures. For blockchain contexts, we not only have to worry about being able to recover from failure but also to ensure that security vulnerabilities that allow violation of transaction semantics do not occur due to a failure.

Example. Consider a sealed-bid auction with suppliers $Sup_1, Sup_2, ..., Sup_n$ submitting bids $T_{B_1}, T_{B_2}, ..., T_{B_n}$ in response to a REQUEST transaction $T_{REQUEST}$. The requester initiates an ACCEPT_BID transaction $T_{ACC}(T_{B_1})$, choosing T_{B_1} as the winning bid. Correctly, T_{ACC} should initiate one TRANSFER of the winning bid to the requester and $n-1$ RETURNs $T_{R_1}, ..., T_{R_n}$ back to the original bidders, all from the $\mathcal{PBPK}$-$\mathcal{Res}$ account. These TRANSFER transactions must be written to the blockchain before committing the parent transaction. Transactions can be executed in *sync* (immediate response before validation) or *async* mode

Algorithm 1: validate$T_{BID-schema}$

Input: TxnObject
Output: Boolean variable

```
1  validateSchema(loadSchema(bid.yaml), TxnObject)
2  validateTxObj(asset, TxnObject[asset], data, validateKey)
3  validateTxObj(metaData, TxnObject[metaData],
     data, validateKey)
4  validateLanguageKey(TxnObject, data)
5  validateLanguageKey(TxnObject, metaData)
6  return True
```

Algorithm 2: validateT_{BID}

Input: rfq_id, asset_id, TxnObject, CurrentTxs : List < TxObject >
Output: Boolean variable

```
1  RFQTx = getTxFromDB(rfq_id);
2  AssetTx = getTxFromDB(asset_id);
3  if RFQTx AND AssetTx txs are not committed then
4  |   throw InputDoesNotExistError;
5  for every output in TxnObject.outputs do
6  |   if output.pubKey is not EscrowPubKey then
7  |   |   throw ValidationError;
8  RequestedCaps = getCapsFromRFQ(RFQTx);
9  AssetCaps = getCapsFromAsset(AssetTx);
10 if RequestedCaps is not subset of AssetCaps then
11 |   throw InsufficientCapabilitiesError;
12 return validateTransferInputs
     (TxnObject, CurrentTxs : List < TxObject >);
```

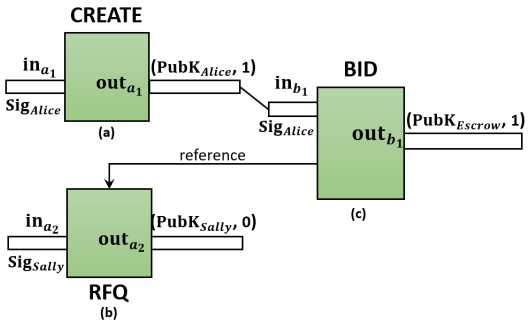


Figure 6: Example of BID

(response after validation confirmation from the SmartchainDB server).

Imagine a scenario where only a subset s of child transactions, say $T_{R_1}, T_{R_2}, T_{R_3}$, completes before a failure occurs. Due to blockchain immutability, transactions in s cannot be undone. A potential issue arises if the $T_{ACC}(T_{B_4})$ transaction is reinitiated with a different winning bid; it's not a duplicate since it wasn't committed, creating a security risk where the requester might receive both winning bids.

To tackle this, we propose a *Non-blocking* transaction execution approach, allowing the parent transaction to be committed (no lock) to the blockchain even if child transactions are pending. This method enforces 'eventually commit' semantics for the child transactions, ensuring transaction integrity and preventing such vulnerabilities.

Building on this, we address the possibility of failures during the execution of Nested blockchain transactions, we propose a *Redo Recovery strategy* that is mediated by an escrow account - special system account. The escrow account uses a logging protocol to keep track of pending nested transactions. Access authorization for assets of pending nested transactions always remains with the escrow account to ensure that it has the necessary permissions to retry transactions until they commit. Fig. 7 illustrates the workings of our recovery scheme. A dedicated recovery log (accept_tx_recovery) tracks the state of all nested transactions, capturing which child transactions have been successfully committed and which remain incomplete. In the event of a failure, the escrow-mediated logging protocol ensures recovery without undoing previously committed actions by identifying incomplete child transactions using logged states and automatically re-initiating them with a binary exponential backoff timeout.

Furthermore, the Byzantine Fault Tolerant (BFT) consensus mechanism protects the system by pausing consensus during faults, preventing invalid transactions until a sufficient quorum is restored. While the escrow mechanism effectively resolves failures within the nested transaction framework, the BFT mechanism ensures robustness at the consensus layer. The following subsection delves into how our system handles failures under BFT constraints, particularly focusing on scenarios where more than 1/3 of validator nodes go offline.

4.2.1 Implementation NLT. Non-blocking approach was examined under two scenarios regarding system failures: (1) a positive case without any failures, (2) a case with a possible crash while processing the transaction when more than 1/3 (BFT) of voting power goes offline simultaneously. Under case (1) with no failures, after receiving the transaction payload and performing schema validation, the receiver node logs and sends

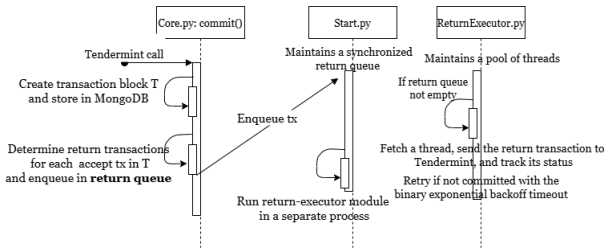


Figure 7: Recovery Flow for NBT

ACCEPT_BID for the consensus without waiting for the children's transactions to be determined and validated, contrary to *blocking* approach. After consensus has been reached, each child transaction, i.e, TRANSFER is enqueued into a task queue during the commit phase by the receiver node. Multiple parallel workers execute the queued jobs asynchronously. Such an approach enables quick commit of the ACCEPT_BID transaction to the blockchain because it gets committed first, allowing committing all the incoming returns after it in an asynchronous way. Under case (2), when 1/3 of the validator nodes go offline, there are two possible sub-cases: (a) receiver node excluded from the set of the crashed nodes, then the process will resume as soon as sufficient voting power is attained, and (b) receiver node included to the set of the crashed nodes. The possible node crash times and crash handling techniques under sub-case (2.b) are provided below:

(1) while processing a parent transaction:
- if a crash happens during the initial validation phase, the driver will re-trigger ACCEPT_BID after the timeout interval.
- if a crash happens on *Tendermint* in mempool, the election process will be resumed as soon as the quorum of nodes is back online

(2) while enqueueing RETURN transactions:
- enqueue all the RETURNs using the recovery log when the receiver node comes up online

(3) while processing RETURN transactions:
- All the RETURN transactions already persist in the queue for the execution. RETURNs are sent to a randomly selected validator node to track its commit status and to retry them if needed. Once the chain resumes, they will end up in the mempool and get committed

Algorithm discussion. The gray shaded area in Fig. 4 shows the extra phases required to validate *Nested* transactions using the Algorithm 3, that can be divided into two parts. In the first part, parent transaction ACCEPT_BID gets validated according to the conditions from subsection 3.2 **Definition** – 4. The conditions and errors that can be thrown by this function are readily comprehensible through the pseudo-code provided. For example, if REQUEST and winning BID transactions are not committed or the signer of the ACCEPT_BID transaction is different from the signer of REQUEST transaction, a validation Error is thrown. In the second part, all the appropriate children transactions are determined and written to the blockchain via the invocation of the `commit()` method. The `commit()` method is called on the receiver node as the last step of the consensus process to trigger children transactions. The function `deterRtrnTxs()` determines unaccepted BIDs for particular REQUEST given the winning BID. Once the list of the *n-1* RETURN transactions has been identified, they all are enqueued to the `ReturnQueue` allowing the system to asynchronously send them without blocking the actual flow. To

Algorithm 3: validateT$_{\text{ACCEPT_BID}}$

Input: rfq_id, win_bid_id, TxnObject, CurrentTxs : List < TxObject >
Output: Boolean variable

```
1  RFQTx = getTxFromDB(rfq_id);
2  WinTx = getTxFromDB(win_bid_id);
3  BidsForCurrentRFQ = getLockedBids(rfq_id);
4  if RFQTx AND WinTx txs are not committed then
5  |   throw ValidationError;
6  if signer(Accept-bid) != signer(RFQ) then
7  |   throw ValidationError;
8  DuplicateAcceptTx = getAcceptTxForRFQ(rfq_id);
9  if DuplicateAcceptTx is in the database then
10 |   throw DuplicateTransactionError;
11 if WinTx is not found
     in EscrowHeldBidsForCurrentRFQ then
12 |   throw ValidationError;
13 return validateTransferInputs (RFQTx, WinTx);
   // Block commit is the final step in consensus
14 Commit(BlockTxs: List < TxObject >):
15 |   for every tx in BlockTxs do
16 |   |   if tx is of type ACCEPT_BID then
17 |   |   |   ReturnTxs = List < TxObject >;
18 |   |   |   r = deterRtrnTxs(WinTx, getPubKey(RFQTx))
19 |   |   |   ReturnTxs.append(r);
20 |   |   |   for every returnTx in ReturnTxs do
21 |   |   |   |   ReturnQueue.put(returnTx)
22 |   |   |   logAcceptBidTxUpdForRecovery(tx, status :
   |   |   |     commit)
```

monitor the status of unaccepted BIDs and to conduct the recovery process, a new collection named `accept_tx_recovery` was introduced in the *MongoDB* database model. Furthermore, the employed storage model enables reliable queryability facilitating the ability to answer various inquiries.

5 EVALUATION

In our evaluation, we analyze the performance and usability of blockchain transaction mechanisms by comparing our proposed declarative transactions with *equivalent transactional behavior implemented as smart contracts i.e. imperative specifications.* Our marketplace case study focused on transaction behaviors in reverse-auction contexts because they have slightly more complex transactional requirements than the traditional forward auction. Our evaluation focuses on assessing the performance of declarative and imperative blockchain transaction models by measuring their *latency* and *throughput* under varying workloads and cluster configurations. While performance metrics are explicitly measured, usability is discussed qualitatively based on the effort required to instantiate and customize transaction workflows.

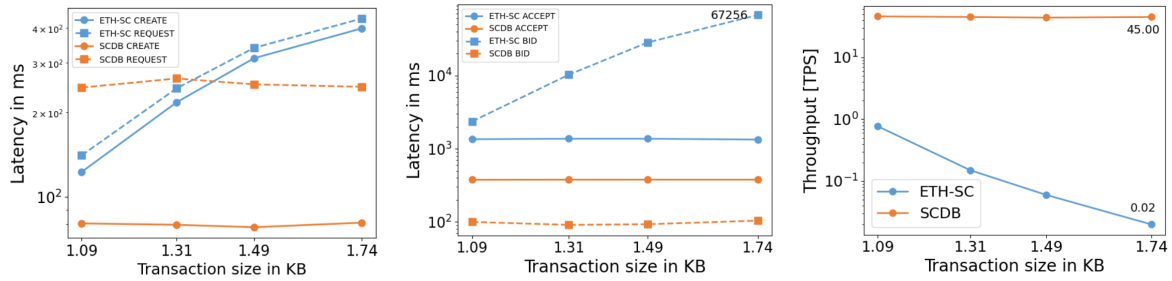
5.1 Setup

5.1.1 Experiment environment.

The experiments were run on Digital Ocean Cloud using virtual machines (VMs) under Ubuntu 20.04 (LTS) x64 operating system with 8 vCPUs, 16 GB of RAM, and 200 GB of SSD storage. The number of VMs utilized varied depending on the experiment.

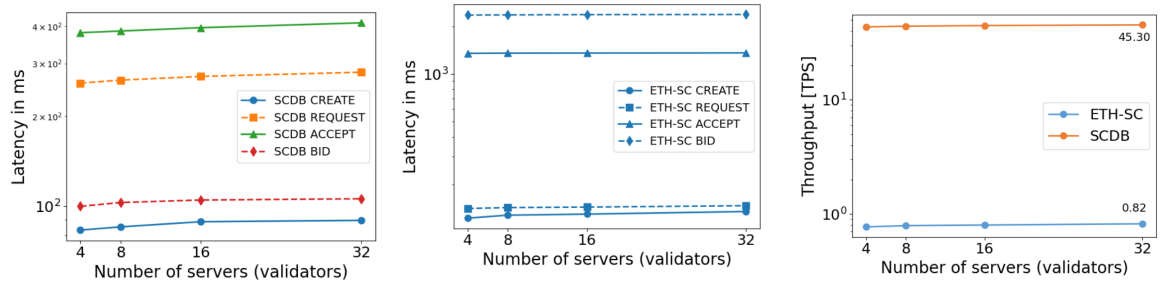
5.1.2 Implementation of Approaches.

Smart Contract Implementation: We implemented reverse auction marketplace contract (ETH-SC), we employed Solidity,



(a) Latency of REQUEST **and** CREATE **(b) Latency of** BID **and** ACCEPT_BID **(c) Throughput**

Figure 8: The Effect of Transaction Size



(a) Latency of SCDB transaction types (b) Latency of ETH-SC transaction types (c) Throughput

Figure 9: The Effect of Cluster Size

Ethereum's Turing-complete, statically-typed, and compiled language designed for smart contract development. Our implementation incorporated standard data structures like *struct* to manage user-defined assets, including bids, with transactional functions defined as methods within the contract.

To thoroughly test the consensus mechanism in a multi-node environment, we integrated GoQuorum [2], a permissioned blockchain platform. Both GoQuorum and BigchainDB prioritize scalability and performance over the financial incentivization model of traditional blockchains such as Ethereum. Moreover, both systems employ Byzantine Fault Tolerance (BFT), ensuring transaction finality and low latency in permissioned environments. This makes them comparable in terms of usability and throughput, as neither imposes the gas cost overhead, allowing for a fair evaluation of declarative versus imperative transaction models in similar environments.

We deployed the reverse auction smart contract on this network and conducted multiple transaction rounds. GoQuorum fully enforced the protocol, offering a realistic assessment of transaction throughput and latency. This setup allowed us to measure the consensus overhead and compare the performance of our declarative approach with traditional smart contract execution, highlighting the trade-offs between usability and the operational costs of maintaining consensus.

For our declarative transactions approach, we leveraged SmartchainDB-*Server*, implemented in Python, alongside SmartchainDB-*Driver*, developed in Java. Notably, Python, being dynamically typed and interpreted, contrasts with Solidity's compiled nature. Our setup involved a network configuration of a different number of *Server* clusters, incorporating a consensus protocol. This integration introduces various overheads, including computational, bandwidth, and latency considerations, among others.

5.1.3 *Workload.*

Blockchains don't have a standard transaction size, so comparing Blockchain X's throughput with Blockchain Y's throughput isn't straightforward because transaction sizes can differ significantly. Consequently, transactions of larger size may require a longer duration for disk writing operations.In our study, unlike *Ge et al., 2022* [20] that used established benchmark YCSB [14] to evaluate the performance of hybrid blockchains, we recognized the critical role of transaction validation semantics in blockchain performance, including aspects like access rights, asset conditions, and transaction dependencies. This complexity extends beyond simple *read* and *write* operations, especially in smart contracts, and requires a more complex workload.

To accurately evaluate smart contracts, we devised a synthetic workload generator tailored for the declarative transaction approach. This generator creates synthetic payloads varying in data size across different transaction fields. We have sent 110,000 transactions to each system comprising of CREATE: 50,000, BID: 50,000, REQUEST: 5000, ACCEPT_BID: 5000.

5.1.4 *Metric calculation.*

Transaction latency was computed by measuring the time elapsed from the moment the transaction was received to its final commitment.

Throughput was calculated by counting the number of transactions that were successfully committed within a time frame, defined as the interval between the reception of the first and the commitment of the last transaction.

5.2 Experiments and analyses

The experiments simulate a reverse auction workflow within the manufacturing domain. We conducted four sets of experiments:

- Experiment 1: Aimed to evaluate latency and throughput by varying transaction sizes in both systems. The cluster of four nodes was used for both systems.
- Experiment 2: Involved a various network size of *Server* validator nodes to simulate real-world scenarios, evaluating how well the system scales, focusing on throughput and latency across the cluster.

5.2.1 Experiment 1 - Latency and Throughput Analysis with Varied Transaction Sizes. To assess the average latency and throughput, we put a list of strings of various sizes in the *metadata* of REQUEST and CREATE transactions representing digital manufacturing capabilities being requested and created respectively.

The data, illustrated in Figs. 8a and 8b, reveal that transaction size had minimal impact on the latency in SmartchainDB (SCDB), remaining nearly constant across all transaction types. Conversely, Ethereum-based Smart Contracts (ETH-SC) exhibited an increase in latency for CREATE and REQUEST transactions as the transaction weight increased, with latency for CREATE transactions becoming nearly five times, and for REQUEST transactions, twice that of SCDB. Additionally, the latency for BID transactions in ETH-SC showed substantial growth with increasing transaction size; at 1.74 KB, ETH-SC's latency was 635 times higher (66.43 seconds) compared to SCDB's 0.104 seconds. For ACCEPT_BID transactions, latency remained stable in both systems, although ETH-SC was over four times slower than SCDB.

Furthermore, results in Fig. 8c indicate that SCDB's throughput stayed consistent despite the growing size of transactions. A notable observation was the inverse relationship between asset size and throughput in ETH-SC, where throughput decreased from an initial 0.72 transactions per second (tps) to 0.02 tps by the end of the experiment.

Analysis SCDB vs. ETH-SC: SCDB leverages BigchainDB's execution architecture, which enhances transaction processing through efficient indexing for database queries, built-in caching for quick data access, and pipelined execution. These features mitigate the transaction payload size's impact on latency. Conversely, in ETH-SC, we observed a consistent rise in latency across all transaction types with an increasing number of transaction size, suggesting scalability issues under heavier workloads. This escalation, particularly for CREATE and REQUEST transactions (Fig. 8a), ties back to the smart contract's storage structure, comprising a vast array of 2^{256} slots. For dynamic data structures like *mappings*, Solidity's hash function computes storage locations, but each map item's retrieval takes $O(n)$ time. Additionally, the complexity of smart contract logic exacerbates latency and throughput issues. The quadratic time complexity ($O(n^2)$) for BID transactions results from a nested loop comparing each CREATE asset capability with every REQUEST capability to validate BIDs. The validation also employs a costly `compareStrings()` function in terms of GAS usage.

5.2.2 Experiment 2 - Analyzing Impact of Cluster Size on Latency and Throughput. This experiment assessed how the number of validator nodes in the cluster affects latency and throughput in both SCDB and ETH-SC. Throughout the experiment, the transaction size was kept constant at 1.09KB to ensure consistent conditions for evaluation. As shown in Figs. 9a and 9b , despite the increased complexity and number of validators, the latency for various transaction types remained relatively stable for both SCDB and ETH-SC across increasing numbers of validator nodes (from 4 to 32). While adding more validator nodes typically introduces more communication overhead in decentralized networks, the results indicate that ETH-SC's latency does not significantly increase as more nodes are added. This could be due to the efficient finality properties of the IBFT consensus mechanism, which ensures low-latency agreement among nodes. However, despite the stable latency across varying node counts, the baseline latency for ETH-SC is still significantly higher compared to SCDB, particularly for BID and REQUEST transactions.

As depicted in Figure 9c, throughput shows a slight steady increase from 43.5 TPS with 4 nodes to 45.3 TPS with 32 nodes. This incremental improvement in throughput can be attributed to the system's ability to leverage blockhain pipelining technique,which enhances scalability during the voting process for new blocks. With more nodes available, SCDB can distribute the workload more effectively, allowing multiple transactions to be processed simultaneously across different validators. While adding more nodes generally improves throughput, it also introduces potential challenges. Typically, increasing the number of nodes leads to more communication and data exchange among validators, which can slow down the consensus process. These factors account for the steady, incremental increase in throughput, illustrating SCDB's ability to balance performance enhancements with the given complexities.

In comparison, ETH-SC exhibits significantly lower throughput, beginning at 0.77 TPS with 4 nodes and showing no substantial improvement as the cluster size increases. This limitation stems from the added computational overhead of GoQuorum's IBFT consensus mechanism, which handles Ethereum smart contracts and privacy layers, making it more resource-intensive. The difference is further amplified by SCDB's streamlined transaction model, which avoids the computational complexity of smart contract execution.

Usability. Instantiating and customizing a transaction in our model is simply a matter of providing metadata for the appropriate transaction type - no programming expertise is required. Smart contracts (even with the help of templates) often require programming expertise to identify and, potentially customize, suitable smart contract codes. They must also deal with other programming-related tasks such as the deployment of contracts which may produce errors.

6 RELATED WORK

Different efforts have been made to address some of the limitations of smart contracts as a mechanism for specifying transaction behavior.

Addressing usability and interpretability challenges: Standardized function interfaces or *tokens* such as ERC-721 [46] and ERC-20 [43] prescribe the minimum set of methods (signatures and behaviors) for specific classes of smart contracts, e.g., fungible tokens. *Smart contract templates* [13] are similar in spirit to token interfaces but incorporate methods for linking legal contracts written in prose to methods in a contract so that execution parameters are extracted from the legal prose and passed to the smart contract code to drive execution. *Domain-Specific Languages* (DSLs) such as Marlowe [27], SPECS [24], Findel [11], Contract Modeling Language (CML) [48], ADICO [18] are programming languages with limited expressiveness that provide high-level abstractions and features optimized for a specific class of problems (typically in a specific domain such as finance or law). DSLs allow the possibility of domain experts rather than

programmers to implement smart contracts using graphical user interfaces that can be translated to smart contract code via the DSL. However, while efforts like (DSLs) and formal verification tools improve usability and correctness within specific domains, they lack generality and composability. Further, they still require a non-trivial amount of manual code implementation which is vulnerable to the risk of errors and inefficiencies. Further, being imperative specifications, they are less amenable to querying and analysis. Some contributions have been made in the area of *smart contract code analysis* [19, 21, 44], but most have focused on the problem of identifying bugs or attack vulnerabilities.

Addressing performance challenges: Several solutions have been proposed [35] to address the throughput and latency limitations of current blockchains, including sophisticated *consensus algorithms* [22, 23] in Hyperledger Fabric, *adjusting block size* which is prone to security vulnerabilities due to the increase in the propagation delay [5] and *reducing block data* which provides a limited increase in throughput [28]. *Sharding* divides the network into different subsets (i.e., shards) and distributes workloads among shards to be executed in parallel. This provides processing and storage scalability, although cross-shared communication overhead is often a major challenge. Further, poor shard design may lead to a 1% attack and other security issues [25, 44, 52]. Some recent work [41] on a distributed and dynamic sharding scheme that reduces communication cost and improves reliability has been proposed. However, these efforts do not directly address the sequential execution of smart contracts adopted by most platforms which limits their throughput.

Declarative smart contracts, such as those (DeCon) proposed by Chen et al., [12], represent a significant shift from traditional imperative programming models. By enabling developers to specify high-level rules and conditions rather than procedural logic, DeCon aims to simplify the creation and validation of smart contracts. This approach reduces the need for detailed coding and allows for a more transparent specification of transaction behaviors.
However, while DeCon simplifies smart contract creation, achieving functional parity with manually written contracts still requires significant effort. For instance, even after using DeCon to model a contract, we found it necessary to modify parts of the generated contract to align it with the functional requirements of a custom-created contract. Changes included adjusting data structures, refining method references, and ensuring the generated logic adhered to the desired transactional workflow. This indicates that declarative frameworks like DeCon still require manual intervention to create contracts that meet specific use cases, limiting their usability in complex or highly tailored scenarios. Furthermore, declarative frameworks like DeCon fail to address several critical challenges inherent in the transactional model of smart contracts.

With respect to *parallel execution* of smart contracts, the main challenge is dealing with the conflicts and dependencies between smart contracts, given that they have a shared state. [16] propose the use of pessimistic transactional memory systems for concurrent execution of *non-conflicting* smart contracts. They suggest achieving parallelism with lower latency by two steps: first, involving a serializable schedule for miners and, second, executing this sequence of transactions deterministically for parallel validating to avoid the synchronization excessive costs. However, this approach implies that validating should be performed significantly more times than mining. On the other hand, [10] proposed optimistic transactional memory systems which guarantee correctness through opacity rather than serializability. *Speculative concurrent execution* of smart contracts proposed in which transactions are executed in parallel, and if a conflict occurs, by tracing write and read sets, one of the transactions is committed, and the other is discarded to rerun later. Speculative strategies usually perform reasonably well when the rate of conflicts is low [9, 15, 36]. However, these techniques are still in their early phases and often use read-write sets to define conflicts. Furthermore, empirical results [36] suggest that this notion might be too aggressive, resulting in many unnecessary conflicts detected, suggesting the need for reasoning about conflicts at a slightly higher level of abstraction.

Alternative strategies such as aggressive caching and parallel validation using validation system chaincode have also been used in Hyperledger Fabric [38, 42].

7 LIMITATIONS

Declarative transactions excel in scenarios with well-defined and standardized operations. However, they might lack the flexibility required for handling complex, dynamic, or unique transactions that do not fit neatly into predefined patterns. Also, for certain applications requiring fine-grained control over individual steps or transactions, the declarative model might not offer the level of granularity needed.

8 CONCLUSION AND FUTURE WORK

This paper introduces the concept of *declarative blockchain transactions*, emphasizing an alternative approach to modeling transaction behavior on blockchains. By leveraging declarative principles, we aim to address the usability and performance challenges associated with traditional imperative approaches, such as smart contracts. The proposed methodology, implemented by extending an open-source blockchain database, demonstrates how pre-defined primitives can simplify transaction workflows and improve efficiency.

Our experimental results validate the benefits of this approach, highlighting its potential to improve usability and performance in specific application contexts. However, the variability among blockchain systems underscores the need for cautious interpretation of the results. Future work will focus on generalizing the modeling framework to support more complex transaction workflows and exploring its application across diverse blockchain domains.

This work lays a foundation for developing advanced transaction optimization strategies, particularly in the context of typed transaction models. Such models have the potential to enable reasoning about semantic conflicts during concurrent execution, paving the way for more efficient transaction processing in distributed systems. Another promising direction is to explore implementation strategies for extending the declarative transaction paradigm to other blockchain architectures. This would allow us to assess the applicability and benefits of our approach in a broader range of blockchain contexts, addressing diverse requirements and use cases.

9 ACKNOWLEDGEMENTS

Our work was partially funded by a National Science Foundation CSR grant # 1764025.

REFERENCES

[1] [n.d.]. BigchainDB. https://www.bigchaindb.com/.

[2] [n.d.]. ConsenSys GoQuorum. https://docs.goquorum.consensys.io/.

[3] 2022. Over 44 Million Contracts Deployed to Ethereum Since Genesis: Research. https://cryptopotato.com/over-44-million-contracts-deployed-to-ethereum-since-genesis-research/.

[4] 2023. Everledger. https://everledger.io/.

[5] 2023. The Peer-to-Peer Electronic Cash System for Planet Earth. https://www.bitcoinunlimited.info/.

[6] 2023. SmartchainDB: github repo. https://github.com/korchiev/smartchaindb.git.

[7] Cornelius C Agbo, Qusay H Mahmoud, and J Mikael Eklund. 2019. Blockchain technology in healthcare: a systematic review. In *Healthcare*, Vol. 7. MDPI, 56.

[8] Jameela Al-Jaroodi and Nader Mohamed. 2019. Blockchain in industries: A survey. *IEEE Access* 7 (2019), 36500–36515.

[9] Parwat Singh Anjana, Hagit Attiya, Sweta Kumari, Sathya Peri, and Archit Somani. 2021. Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory. In *International Conference on Networked Systems*. Springer, 77–93.

[10] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2019. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 83–92.

[11] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure derivative contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*. Springer, 453–467.

[12] Haoxian Chen, Gerald Whitters, Mohammad Javad Amiri, Yuepeng Wang, and Boon Thau Loo. 2022. Declarative smart contracts. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 281–293.

[13] Christopher D Clack, Vikram A Bakshi, and Lee Braine. 2016. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771* (2016).

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

[15] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 303–312.

[16] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2020. Adding concurrency to smart contracts. *Distributed Computing* 33, 3 (2020), 209–225.

[17] Christian F Durach, Till Blesik, Maximilian von Düring, and Markus Bick. 2021. Blockchain applications in supply chain transactions. *Journal of Business Logistics* 42, 1 (2021), 7–24.

[18] Christopher K Frantz and Mariusz Nowostawski. 2016. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 210–215.

[19] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2019. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 394–397.

[20] Zerui Ge, Dumitrel Loghin, Beng Chin Ooi, Pingcheng Ruan, and Tianwen Wang. 2022. Hybrid blockchain database systems: design and performance. *Proceedings of the VLDB Endowment* 15, 5 (2022), 1092–1104.

[21] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep* (2018), 1–41.

[22] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. Fault-tolerant distributed transactions on blockchain. *Synthesis Lectures on Data Management* 16, 1 (2021), 1–268.

[23] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. Rcc: Resilient concurrent consensus for high-throughput secure transaction processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1392–1403.

[24] Xiao He, Bohan Qin, Yan Zhu, Xing Chen, and Yi Liu. 2018. Spesc: A specification language for smart contracts. In *2018 IEEE 42nd Annual computer software and applications conference (COMPSAC)*, Vol. 1. IEEE, 132–137.

[25] Jelle Hellings and Mohammad Sadoghi. 2021. Byzantine Cluster-Sending in Expected Constant Communication. *arXiv preprint arXiv:2108.08541* (2021).

[26] Kai Hu, Jian Zhu, Yi Ding, Xiaomin Bai, and Jiehua Huang. 2020. Smart contract engineering. *Electronics* 9, 12 (2020), 2042.

[27] Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon Thompson. 2020. Marlowe: implementing and analysing financial contracts on blockchain. In *International Conference on Financial Cryptography and Data Security*. Springer, 496–511.

[28] Sergio Demian Lerner and RSK Chief Scientist. 2017. Lumino transaction compression protocol (LTCP).

[29] Fabrice Lumineau, Wenqian Wang, and Oliver Schilke. 2021. Blockchain governance—A new way of organizing collaborations? *Organization Science* 32, 2 (2021), 500–521.

[30] Thomas McGhin, Kim-Kwang Raymond Choo, Charles Zhechao Liu, and Debiao He. 2019. Blockchain in healthcare applications: Research challenges and opportunities. *Journal of network and computer applications* 135 (2019), 62–75.

[31] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. 2019. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. *Journal of Cases on Information Technology (JCIT)* 21, 1 (2019), 19–32.

[32] Xinping Min, Lanju Kong, Qingzhong Li, Yuan Liu, Baochen Zhang, Yongguang Zhao, Zongshui Xiao, and Bin Guo. 2022. Blockchain-native mechanism supporting the circulation of complex physical assets. *Computer Networks* 202 (2022), 108588.

[33] Arim Park and Huan Li. 2021. The effect of blockchain technology on supply chain sustainability performances. *Sustainability* 13, 4 (2021), 1726.

[34] Michael Rogerson and Glenn C Parry. 2020. Blockchain: case studies in food supply chain visibility. *Supply Chain Management: An International Journal* 25, 5 (2020), 601–614.

[35] Abdurrashid Ibrahim Sanka and Ray CC Cheung. 2021. A systematic review of blockchain scalability: Issues, solutions, analysis and future research. *Journal of Network and Computer Applications* 195 (2021), 103232.

[36] Vikram Saraph and Maurice Herlihy. 2019. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376* (2019).

[37] Asad Ali Siyal, Aisha Zahid Junejo, Muhammad Zawish, Kainat Ahmed, Aiman Khalil, and Georgia Soursou. 2019. Applications of blockchain technology in medicine and healthcare: Challenges and future perspectives. *Cryptography* 3, 1 (2019), 3.

[38] Harish Sukhwani, Nan Wang, Kishor S Trivedi, and Andy Rindos. 2018. Performance modeling of hyperledger fabric (permissioned blockchain network). In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 1–8.

[39] Tobias Sund, Claes Lööf, Simin Nadjm-Tehrani, and Mikael Asplund. 2020. Blockchain-based event processing in supply chains—A case study at IKEA. 65 (2020), 101971.

[40] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First monday* (1997).

[41] Yuechen Tao, Bo Li, Jingjie Jiang, Hok Chu Ng, Cong Wang, and Baochun Li. 2020. On sharding open blockchains with smart contracts. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1357–1368.

[42] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 2018. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th international symposium on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS)*. IEEE, 264–276.

[43] Fabian Vogelsteller and Vitalik Buterin. 2015. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20.

[44] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[45] Sam M Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J Knottenbelt. 2021. Sok: Decentralized finance (defi). *arXiv preprint arXiv:2101.08778* (2021).

[46] Jacob Evans William Entriken, Dieter Shirley and Nastassia Sachs. 2018. EIP-721: Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721.

[47] Maximilian Wöhrer and Uwe Zdun. 2020. Domain specific language for smart contract development. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.

[48] Maximilian Wöhrer and Uwe Zdun. 2020. Domain Specific Language for Smart Contract Development. *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (2020), 1–9.

[49] Hanqing Wu, Jiannong Cao, Yanni Yang, Cheung Leong Tung, Shan Jiang, Bin Tang, Yang Liu, Xiaoqing Wang, and Yuming Deng. 2019. Data management in supply chain using blockchain: Challenges and a case study. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–8.

[50] Zhiying Wu, Jieli Liu, Jiajing Wu, Zibin Zheng, Xiapu Luo, and Ting Chen. 2023. Know Your Transactions: Real-time and Generic Transaction Semantic Representation on Blockchain & Web3 Ecosystem. In *Proceedings of the ACM Web Conference 2023*. 1918–1927.

[51] Victor Zakhary, Mohammad Javad Amiri, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. 2019. Towards global asset management in blockchain systems. *arXiv preprint arXiv:1905.09359* (2019).

[52] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 931–948.

[53] Yan Zhu, Yao Qin, Zhiyuan Zhou, Xiaoxu Song, Guowei Liu, and William Cheng-Chung Chu. 2018. Digital asset management with distributed permission over blockchain and attribute-based access control. In *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 193–200.