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

Anonymous Author(s)*

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 are modeled and implemented as use cases, using an open-source blockchain database, 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.

KEYWORDS

Blockchain, Declarative blockchain transactions, Decentralized marketplaces

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, [9, 33], supply chain [19, 42, 52], decentralized finance (Defi) [40, 48], governance [32], web browsing, gaming, social media, and file sharing/storage [10].

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* [43]. 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 [34] that resulted in a loss of approximately 3.6M ETH (about $6.8B). (iii) Many transactional behaviors in smart contracts, embedded in programming structures, remain hidden on the blockchain, hindering their utility in complex data analysis. (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 [14], domain-specific languages [50], and smart contract templates [29] aim to ease creation and verification processes. However, they fall short in addressing performance, throughput, queryability, and other transactional model challenges in smart contracts.

1.1 Contributions:

This paper endeavors to extend the range of native transaction behaviors in blockchains, aiming to lessen the dependency on smart contracts. We introduce a *declarative transactions* methodology, redefining blockchain transaction management to address several limitations of smart contracts. Our specific contributions include:

(1) a formal model for *declarative blockchain transactions* that is based on a *type system* for blockchain transactions and the introduction of a novel concept of *nested blockchain transactions*.

(2) based on 1., a formalization of a selected sample of transaction behaviors drawn from a common category of blockchain applications - auction marketplaces.

(3) an implementation framework for declarative transactions that involve typed transaction schemas and transaction validation algorithms that build on the blockchain validation framework of an open source blockchain database BigchainDB.

(4) a comparative performance and usability evaluation of the declarative transaction model vs. the smart contract model using Ethereum smart contracts as the baseline. The evaluation results demonstrate that the declarative transaction method significantly outperforms smart contracts, with improvements by factors of 635 and 3, respectively

The rest of the paper is organized as follows: Section 2 provides 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

```
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) }}}}
```

Non-native assets

Bid validation

Figure 1: Smart Contract sample implemented in Solidity

of the concepts presented in Section 3. Section 5 reviews the literature on the topic, while Section 6 reports on the comparative experiments conducted to evaluate our system and smart contract. Finally, we conclude the paper with a summary in Section 7.

2 MOTIVATION AND BACKGROUND

2.1 Smart Contracts in Blockchain Marketplaces

Most blockchains have a very limited set of *native* transaction types: typically just the TRANSFER transaction. (We note that Ethereum has introduced some additional types of transactions but these focus more on operational semantics than transaction behavior e.g. multi-sig transactions that allow multiple parties to sign a transaction rather than the default single party signing. Therefore, nearly all blockchain applications need to supplement transaction support with user-programmed ones - smart contracts, inheriting all its limitations. These limitations are better illustrated by example. Assume that we want to set up a decentralized marketplace for procurement and supply chain management with the following requirements.

Example. *Buyers can post requests for services they need e.g., manufacturing services, and service providers e.g., 3-D printer manufacturers, can post supply bids for those service requests. A service request can include descriptive metadata such as quantity, product type, deadline, etc., instantiated by a call to the* `createrfq` *method. Similarly, a supply bid will identify which service request it is responding to and include a reference to the asset that forms the basis for the bid - some representation of its production capabilities such as capability certifications, work history, and so on, using* `createbid`*. (In a traditional product sale auction, 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.

Observations: Firstly, whereas for the native transaction TRANSFER transaction, transaction validation against the *double spend* error comes out-of-the-box, for smart contracts, the code for transaction validation must be included in the contract by the developer. Our example shows some of the methods responsible for that `checkValidBid()`. In an auction context, some of the checking might include that all non-winning bids be refunded (if escrow deposits were required), authorization checks like ensuring that the asset a bidder uses to support their bid is indeed theirs, if bid withdrawals are allowed, then a check that the right bid owner that is issuing a call to withdraw a bid, as well as, that only the auction creator can destroy it. Secondly, smart contracts have most transaction and asset metadata represented in program methods and variables that are not directly visible on the blockchain. In addition, they enable the introduction of new categories of non-native digital assets such as user content like documents, audio/videos and pictures [56], digital twins for physical assets like diamonds everledger[6], cars and houses [54], etc., ownership certificates for physical assets credential assets such as academic certificates [35], products like baby formula to be tracked in a supply chain [36, 37] and many others. Rather, they are stored in dedicated persistent structures requiring an understanding of low-level storage details to navigate them. In our example, metadata for requests, bids, and their underlying assets are represented as the struct variables, `request`, `bid`, and `asset` resp. 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 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 showed a 40% increase in GAS - a unit of execution in Ethereum). This overhead translates not just to transaction latencies but also financial costs - the fees for executing native transactions are typically flat while smart contract execution fees are based on the runtime behavior of programs which can be hard to anticipate.

2.2 Rationale for Approach

Introducing more native transaction types is one way to minimize this dependence on smart contracts. However, there are key questions that need to be answered to be able to achieve this:

(1) What new transaction primitives can be added to reduce the burden of always requiring smart contracts in the development of blockchain applications?
(2) How can these new primitives be effectively integrated into blockchains?

To answer 1., we can leverage the fact that most blockchain applications are forms of marketplaces where assets are traded or tracked. Consequently, common marketplace transactions e.g. buy, sell, bid, etc, are likely to be good candidates for frequently desired

blockchain transaction behavior. Indeed, an analytical study [53]of the smart contracts on Ethereum revealed such smart contract method calls to be dominant. With respect to 2., there are some choices for an integration implementation strategy. Our introductory discussion already highlighted the limitation of an imperative specification model such as smart contracts where the user is responsible for low-level, detailed implementation of transaction behavior. An alternative model is one that proved to be a major contributor to the success of relational database systems - *declarative* specifications. Here, users only specify what they want not how to implement it. This is typically specified in terms of constraints that define forbidden behavior and then build into the system, behavior validation capabilities. Another strength of declarative models is that it is possible to allow a single specification map to multiple possible runtime execution algorithms. With this approach, an optimizer is developed that can select the optimal execution strategy based on runtime conditions and cost models. Approaches that permit such automatic optimization usually have some advantages over imperative specifications that are completely prescriptive. In addition, it is possible to have a declarative specification model that is extensible such that multiple simple conditional expressions can be combined using operators to create more complex expressions.

The second part of 2. is coming up with an implementation strategy which in turn is determined by the target blockchain. Largely, blockchains have architectures that are either native or database-based. In the former, the engine of the blockchain is built from the ground up e.g. Ethereum. Database-based blockchains essentially build blockchain functionality on top of existing large-scale databases such as BigchainDB. BigchainDB [1] is a multi-asset blockchain database that possesses blockchain characteristics such as *decentralization*, *immutability*, and owner-controlled assets. In addition, it has database properties like *high transaction rate*, *low latency*, *indexing*, and *querying* of structured data. Each node in BigchainDB network runs three independent services, BigchainDB *Server*, *Tendermint* (consensus parameter), and *MongoDB* (local database). BigchainDB *Server*, the core of the application stack of the node that accepts, validates and passes the transactions to other services. *Tendermint* [5] is a Byzantine Fault Tolerant (BFT) state-machine replication engine used for networking and consensus. *Tendermint* consists of two major components: *Tendermint* Core (consensus engine) and the Application BlockChain Interface (ABCI). *Tendermint* core uses a Proof-of-Stake (PoS) consensus mechanism (there is no mining), and validators may or may not have equal voting power.

Naturally, the database-based blockchains are a more natural fit for a declarative model since the underlying database likely supports a declarative model. Consequently, we select an approach that builds our declarative transaction approach on top of database-based blockchains, in particular, BigchainDB.**Note that** is it neither necessary nor feasible for blockchains to offer a complete set of all possible transaction types and is not at all our goal here. Rather, we posit that a reasonable core can be developed initially and then extended over time (very similar to how query languages like SQL evolved).

3 APPROACH

Our approach is to introduce SmartchainDB, which is an extension of BigchainDB, with additional blockchain transaction primitives that can be used to specify complex transactional behavior and workflows rather than the use of imperative specifications, i.e., smart contracts. Specifically, we introduce a *"declarative" blockchain transaction* model on which different kinds of blockchain *transaction types* can be based. We then propose some concrete primitive *transaction types* based on the proposed model as well as their implementation strategies. We use decentralized marketplaces as our discussion context, using some marketplace transaction behavior as examples, because of their popularity as blockchain applications. However, the ideas here can be generalized to other contexts as well.

3.1 Formal Conceptual Model for Blockchain Transactions

Our formal transaction model encapsulates key components of a transaction which include: the asset being manipulated by the transaction, the accounts involved in the transaction (usually identified by public keys), the type of transaction behavior and the rules for automatic validation of transaction semantics e.g. similar to automatic validation of TRANSFER transaction for the *double spend* error. 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 $\mathtt{ms}_{\mathtt{i,j,k}}$ 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 $< (\mathtt{k_i}, \mathtt{v_i}), \mathtt{amt} >$ where $(\mathtt{k_i}, \mathtt{v_i})$ is a nested set of key-value pairs such that each $\mathtt{k_i} \in \{\mathcal{L} - \mathcal{RK}\}$ and $\mathtt{v_i} \in \mathcal{L} \cup \mathtt{A}$ 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 $\{o_1, o_2, ...o_m\}$. $T.o_k$ is used to denote the k^{th} output of transaction T. Since assets can be divisible, the different outputs can hold different numbers of shares of some asset A_i. Consequently, each of T's output object o_j is a tuple $< pb_i, A_i.amt, pb_i^{prev} >$, where $A_i.amt$ is the number of shares of A_i associated with the j^{th} output of T, i.e., $T.o_j$, pb_i is a set of public keys of the owners or controllers of those shares, and 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 $T.i_k$ to denote the k^{th} input of transaction T. Each input object i_i is a tuple $< T_a.o_b, ms_{u,v,w} >$, where $T_a.o_b$ is the output that is being "spent" by this input (in this case the o_b is an output of T_a) and $ms_{u,v,w}$ is the signature string formed from the private keys that should be the signatures of the assets' owners.
- Ch - a set of transaction outputs called children transactions dependent on I and O of parent transaction T.
- 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:

- $|CH| \geq 1$. at least one child transaction
- parent transaction is said to be committed if and only if all its children are committed
- $\forall T_{parent}, \exists T \in CH : T_{parent}.o \subset T.o$ Every output of a parent transaction is a subset of a child transaction's output

3.2 SmartchainDB 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 $\tau_\alpha = < T_\alpha, C_\alpha >$ where τ_α is the subset of transactions in $\mathcal{T}$ that have OP = α and a set of conditions C_α defined in terms of a transaction's inputs and outputs. In SmartchainDB 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 $\tau_\alpha = < T_\alpha, C_\alpha >$ if it meets all the conditions in C_α. 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 [8].

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 $\tau_{BID} = < T_{BID}, C_{BID} >$ where: $T_{BID} = <$ ID, BID, A, O, I, Ch, R $>$.

ID = 95879..., OP = BID, A = $\{asset\_id : 65be4..\}$
I = $\{< KmSd2..., 1 >, ms_{YM2sd4hn...}\}$,
O = $\{ < [7EAsH..], [1] >\}$, Ch = $\{\emptyset\}$, R = $[6ae47...]$

For example, the tuple above represents a BID transaction with the ID 95879.., involving an asset identified by $65be4...$. The input $< KmSd2..., 1 >$ refers to the output and the amount that is being spent by BID with signature string $ms_{YM2sd4hn...}$ which acts as the cryptographic proof required to satisfy the conditions tied to the output. The output condition and the amount for this transaction to be spent are defined as $< [7EAsH..], [1] >$. There are no child transactions, Ch = $\{\emptyset\}$, as BID is not *Nested* transaction, and it references a transaction with ID $64e47...$.

A BID transaction has the following set of boolean validation conditions C_{BID}:

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

$\tau_{ACCEPT\_BID} = < T_{ACCEPT\_BID}, C_{ACCEPT\_BID} >$ where $T_{BID} =$ < ID, ACCEPT_BID, A, O, I, Ch, R > with the following set of boolean validation conditions C_{ACCEPT_BID}

(1) $|I| == n$ i.e. where n is the number of BIDs for 1 REQUEST
(2) $|R| = 1$ i.e. reference vector must contain exactly 1 element
(3) $\exists!T_{ACCEPT\_BID} \in R : T.OP ==$ REQUEST, i.e., there exists exactly 1 REQUEST transaction in reference vector
(4) $|CH| == |I|$ number of elements in children set is equal to the number of input objects
(5) $\forall i \in I$, verify$(s_i, pb_i, m_i) == True$
(6) $\forall T \in CH : T_{ACCEPT\_BID}.o \supset T.o$, i.e., The output of parent ACCEPT_BID is a proper superset of every transaction's output in the children set
(7) $\forall k \in [1, |I|], T.i_k[1][1] == \mathcal{PBPK} - \mathcal{Res}$, i.e. each input has to spend an output of some TRANSFER transaction that has an account owner $\mathcal{PBPK}$-$\mathcal{Res}$
(8) $\forall j \in [1, |O|] \forall k \in [1, |I|] : T.o_j[1][1] == T.i_m[1][3]$ where $T.o_m[1][3] \wedge T.o_j.ID \neq T_{ACCEPT\_BID}.A.ID \wedge T.i_k.ID \neq T_{ACCEPT\_BID}.A.ID$ is pb_i^{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) $\exists!T.o : T.o[1] == T_{ACCEPT\_BID}.R.o[1]$, 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 T_{ACCEPT_BID}:

ID = $b64c6...$, OP = ACCEPT_BID, A = $\{win\_bid\_id : 95879...\}$
I = { <HmkC1...,1>, $ms_{HmkC1...}$, <MfcDL...,1>, $ms_{HmkC1...}$}
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 $T_1, T_2, ..., T_n$ where T_1 is *head* that initiates the workflow and T_n is *tail* of the sequence. The following condition must be true for a transaction in the sequence:

- $T_1.i = \emptyset$ Input of the transaction initiating workflow is null.
- $\forall\{T_j.i - \{T_1\}\}\exists T.o_k$ where $T.o_k$ 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 strategy enhances the BigchainDB platform, modifying its key architectural components, except for the consensus layer (*Tendermint*). Figure 2 depicts the transaction processing workflow and fundamental elements of SmartchainDB, our extended system. 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.

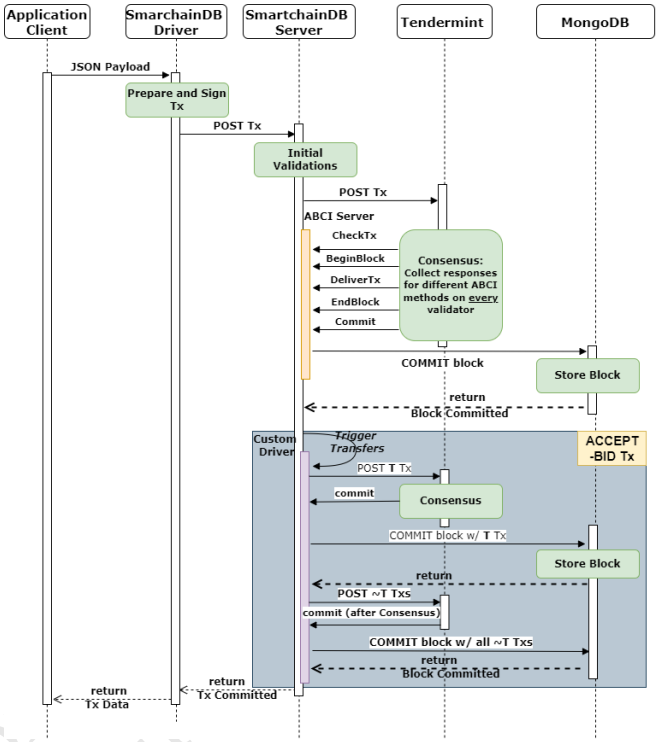


Figure 2: SmartchainDB Transaction Life cycle

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 $\texttt{validateT}_\alpha$ method used by the validator nodes at the *Server* layer, e.g., $\texttt{validateT}_{\texttt{BID}}$ for the BID transaction. At the network validator node, a transaction undergoes a secondary set of validation checks triggered by the CheckTx function. This step is implemented to verify that the validator node did not temper 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.

In the following, we elaborate on the implementation of the transaction validation algorithm

```
title: Transaction Schema
required:
  - id
  - inputs
  - outputs
  - operation
  - metadata
  - asset
  - version
properties:
  id:
    anyOf:
      - "$ref": "#/definitions/sha3_hexdigest"
      - type: "null"
  operation:
    "$ref": "#/definitions/operation"
  asset:
    "$ref": "#/definitions/asset"
  outputs:
    ...
  inputs:
```

```
definitions:
  operation:
    type: string
    enum:
      - CREATE
      - TRANSFER
      - REQUEST
      - BID
      - ACCEPT
      - RETURN
  asset:
    type: object
    additionalProperties: false
    properties:
      id:
        "$ref": "#/definitions/sha3_hexdigest"
      data:
        anyOf:
          - type: object
            additionalProperties: true
          - type: "null"
```

Figure 3: Transaction Schema

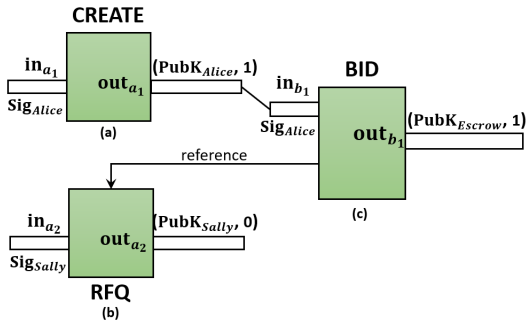


Figure 4: Example of BID

4.1 Implementation of *Non-nested* blockchain transactions

Fig. 3 presents a segment of a YAML snippet that plays a crucial role in defining the transaction schema in SmartchainDB. This schema acts as a blueprint for the formation, validation, and processing of transactions, ensuring conformity to a standardized format. It specifies a rigid structure for transactions, delineating mandatory fields such as id, inputs, outputs, operation, metadata, asset, and version, each with detailed constraints including types, patterns, and additional references within the schema.

Subsequently, each transaction is subjected to a schema validation method upon arrival at the *Server*. This method employs a schema validation algorithm 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

Algorithm 1: 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 >);
```

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. 4 illustrates the transaction dependencies (spending and reference) in the example. The permission dependencies for *Alice* are shown by the relationship PubK$_{Alice}$ and the input signature Sig$_{Alice}$) while *Sally*'s ownership of the REQUEST transaction is indicated by her signature Sig$_{Sally}$ 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 1 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. 2). 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 (response after validation confirmation from the SMARTCHAINDB server).

Imagine a scenario where only a subset 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 address this, we propose a *Non-blocking* transaction execution approach, allowing the parent transaction to be committed 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.

4.2.1 Implementation NBT. 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 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:

Algorithm 2: validateT$_{ACCEPT_BID}$

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

```
RFQTx = getTxFromDB(rfq_id);
WinTx = getTxFromDB(win_bid_id);
BidsForCurrentRFQ = getLockedBids(rfq_id);
if RFQTx AND WinTx txs are not committed then
    throw ValidationError;
if signer(Accept-bid) != signer(RFQ) then
    throw ValidationError;
DuplicateAcceptTx = getAcceptTxForRFQ(rfq_id);
if DuplicateAcceptTx is in the database then
    throw DuplicateTransactionError;
if WinTx is not found
  in EscrowHeldBidsForCurrentRFQ then
    throw ValidationError;
return validateTransferInputs (RFQTx, WinTx);
// Block commit is the final step in consensus
Commit(BlockTxs: List < TxObject >):
    for every tx in BlockTxs do
        if tx is of type ACCEPT_BID then
            ReturnTxs = List < TxObject >;
            r = deterRtrnTxs(WinTx, getPubKey(RFQTx))
            ReturnTxs.append(r);
            for every returnTx in ReturnTxs do
                ReturnQueue.put(returnTx)
            logAcceptBidTxUpdForRecovery(tx, status :
              commit)
```

   - 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. 2 shows the extra phases required to validate *Nested* transactions using the Algorithm 2, 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**. 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 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 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 [49] and ERC-20 [46] prescribe the minimum set of methods (signatures and behaviors) for specific classes of smart contracts, e.g., fungible tokens. *Smart contract templates* [15] 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 [30], SPECS [27], Findel [13], Contract Modeling Language (CML) [51], ADICO [21] 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, these techniques 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* [22, 24, 47], but most have focused on the problem of identifying bugs or attack vulnerabilities.

Addressing performance challenges: Several solutions have been proposed [38] to address the throughput and latency limitations of current blockchains, including sophisticated *consensus algorithms* [25, 26] in Hyperledger Fabric, *adjusting block size* which is prone to security vulnerabilities due to the increase in the propagation delay [7] and *reducing block data* which provides a limited increase in throughput [31]. *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 [28, 47, 55]. Some recent work [44] 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.

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. [18] 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, [12] propose 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 [11, 17, 39]. However, these techniques are still in their early phases and often use read-write sets to define conflicts. Furthermore, empirical results [39] 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 [41, 45].

6 EVALUATION

In our evaluation, we compare blockchain transactional behavior using our proposed declarative approach versus traditional smart contracts. Our comparison not only focuses on performance but also usability aspects. Usability is crucial since smart contracts necessitate software development skills for both creation and verification, posing challenges for contract owners and users. Therefore, methods that minimize or eliminate coding for transaction specification are notably advantageous. In our evaluation plan,

- *Usability* was assessed by the lines of code required to create a marketplace, reflecting the effort involved.
- *Performance* involved measuring transaction latency and throughput, along with the financial efficiency of smart contracts. Transaction fees on blockchains, often linked to the blockchain's fee structure, depend on smart contract length and efficiency, where longer or less efficient code means higher fees. Conversely, native transactions incur fixed costs. Financial performance was evaluated in terms of "GAS," the Ethereum unit for performance and fee measurement.

6.1 Experimental results

In our comprehensive comparative experiments within a reverse auction context, we evaluated SmartchainDB against Smart Contracts. Key findings include:

- SmartchainDB outperformed Smart Contracts across all transaction types. Its superiority was particularly pronounced when handling transactions exceeding 45 capabilities and in processing volumes ranging from 1000 to 10,000 transactions.
- Smart Contracts' throughput declined significantly with larger transaction sizes (above 30 capabilities), whereas SmartchainDB maintained stable throughput and was at least double that of Smart Contracts in the highest value tested.
- We also investigated the steadiness of SmartchainDB's throughput by conducting an experiment with with varying numbers of load-generating clients, showed a quadratic increase in throughput, affirming SmartchainDB's robust performance

- In a scalability test of a SmartchainDB cluster with four nodes, handling increasing transaction loads from 10 clients, the cluster notably surpassed the throughput of both Ethereum Smart Contracts and a single SCDB instance. With peak throughput reaching 3.2 TPS, the SCDB cluster demonstrated its capacity for efficiently managing growing workloads, benefitting from the integration of more workload-generating clients

6.2 Setup

6.2.1 Experiment environment.

The experiments were run on Digital Ocean Cloud infrastructure. We selected dedicated CPU-optimized virtual machines (VMs) under Ubuntu 20.04 (LTS) x64 operating system with 16 vCPUs, 32 GB of RAM, and 200 GB of SSD storage. The number of VMs utilized varied depending on the experiment.

6.2.2 Implementation of Approaches.

Smart Contract Implementation: For our reverse auction marketplace contract (ETH-SC), we employed Solidity, 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.

We utilized the Truffle framework [4] for automated testing and deployment of the smart contract in JavaScript. Additionally, Ganache [2], a local blockchain environment for Ethereum development, was employed in conjunction with an Ethereum client to generate workload for ETH-SC. Ganache's feature of instant block creation upon transaction submission facilitated development by **bypassing actual mining and consensus process on** Ethereum.

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. Unlike Ganache, which serves as a local testing client without consensus mechanisms, our setup involved a network configuration of a four-node *Server* cluster, incorporating a consensus protocol. This integration introduces various overheads, including computational, bandwidth, and latency considerations, among others.

6.2.3 Workloads.

In our study, unlike using established benchmarks like YCSB [16] or BigBench [23], we recognized the critical role of transaction validation semantics in blockchain performance, including aspects like access rights, asset conditions, and transaction dependencies. This complexity, especially in smart contracts, extends beyond simple *read* and *write* operations.

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, producing diverse workload sizes. We developed two distinct workloads, *A* and *B*, each comprising a mix of transaction types. The mix ratio was 1 REQUEST: 25 BIDs: 25 CREATEs: 1 ACCEPT_BID. Workload *A* included 1050 sequential transactions following a reverse sealed-bid auction workflow (REQUEST – CREATE – BID – ACCEPT_BID), while Workload *B* featured interleaved transactions from multiple rounds of this workflow. The total number of transactions varied significantly in our experiments, ranging from over 1,000 to more than 100,000 transactions."

6.3 Experiments and analyses

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

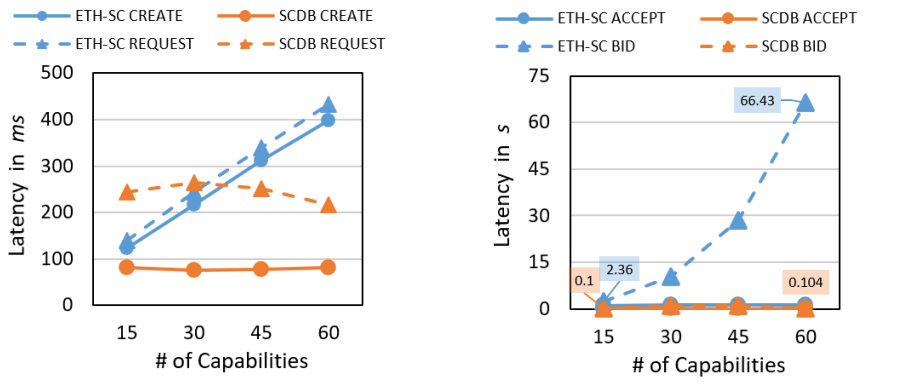
(1) Experiment 1 and 2: Aimed to evaluate latency and throughput by varying transaction sizes and counts, respectively, in both systems.
(2) Experiment 3: Assessed the impact of load-generating clients on SmartchainDB's throughput and latency.
(3) Experiment 4: Involved a network of four *Server* nodes, focusing on throughput and latency across the cluster. The choice of four nodes is based on Byzantine Fault Tolerance principles, requiring $3f + 1$ nodes to counteract f faulty ones, $f \in \mathbb{N}$. This ensures consensus even with f compromised nodes."

6.3.1 Experiment 1 - Latency and Throughput Analysis with Varied Metadata Sizes. To assess the average latency and throughput, we used workload *A* with varying string sizes in the *metadata* of REQUEST and CREATE transactions representing the manufacturing capabilities being requested and created in a digital representation resp., shown in the horizontal axis of Fig. 5a and 5b.

The data, illustrated in Figs. 5a and 5b, 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 a linear increase in latency for CREATE and REQUEST transactions as the capability count 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 quadratic growth with increasing capabilities; at 60 capabilities, 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.

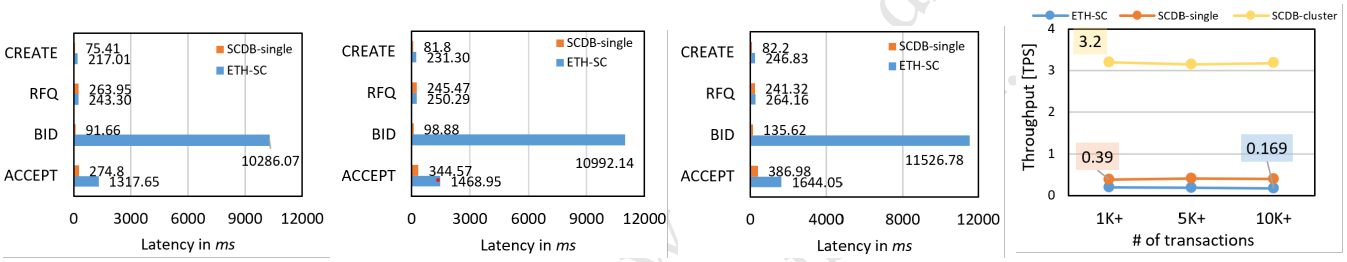
Furthermore, results in Fig. 5c indicate that SCDB's throughput stayed consistent despite the growing number of capabilities, primarily influenced by the workload feed rate. A notable observation was the inverse relationship between asset size and throughput in ETH-SC, where throughput decreased from an initial 0.72 transactions per second (tps) to 0.032 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 capabilities, suggesting scalability issues under heavier workloads. This escalation, particularly for CREATE and REQUEST transactions (Fig. 5a), 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



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

Figure 5: The Effect of the Number of Capabilities



(a) Latency of 1000+ transactions (b) Latency of 5000+ transactions (c) Latency of 10000+ transactions (d) Throughput (for cluster the # of transaction 10x more)

Figure 6: The Effect of the Number of Transactions

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. Table 1 details the GAS consumption and corresponding costs for each transaction type.

Note: GAS price is updated every 5 seconds and for the moment of writing cost of 1 unit of GAS was 14 gwei [20]

6.3.2 Experiment 2 - Analyzing Transaction Volume Impact on Latency and Throughput. This experiment assessed how the total number of transactions influences latency and throughput in both SCDB and ETH-SC, using Workload B. Average latency per transaction type, as shown in Figs. 6 (a - c), indicated that CREATE transactions consistently exhibited the lowest latency in both systems. This efficiency is attributed to the fewer validation and verification steps required for CREATE transactions, as detailed in Section 3.2.

Table 1: GAS and $ used by each transaction type in ETH-SC

	CREATE	REQUEST	BID	ACCEPT_BID
GAS	833599	857998	3828784	226180
$	19.12	19.68	87.83	5.18

Notably, ETH-SC's BID transactions experienced significantly longer processing times than other types, due to their more complex computational and verification requirements. Overall, ETH-SC showed considerably higher processing times across all transaction types compared to SCDB, particularly for BID. This difference is attributed to ETH-SC's virtual machine execution, which introduces extra computational load.

In scalability terms, SCDB demonstrated superior performance, maintaining relatively stable throughput for all transaction types. In contrast, throughput in ETH-SC increased under the same conditions, suggesting better scalability of SCDB in handling varying workloads.

6.3.3 Experiment 3 - Impact of Increasing Transaction-Sending Nodes. In this experiment, we investigated the influence of the number of transaction-sending nodes (clients) on SCDB's throughput and latency. Starting with a single client, we incrementally increased the count to 16 clients, at 15-minute intervals. Employing Workload B with over 5000 transactions, as in *Experiment 2*, we observed the relationship between throughput and client count (Fig. 7). Our findings indicate an exponential growth pattern, with each additional client contributing to a proportional rise in transactions per second. This pattern underscores SCDB's capacity to significantly boost throughput in response to more load-generating clients.

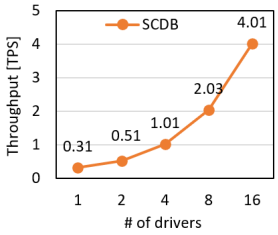


Figure 7: The Effect of Number of Client nodes

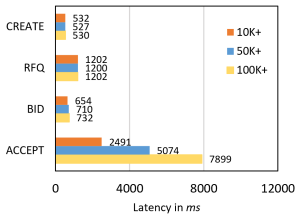


Figure 8: Cluster Latency

6.3.4 Experiment 4 - Scalability Analysis of Cluster Under Varied Transaction Loads. We conducted the scalability assessment of a SmartchainDB cluster comprising four nodes. The aim was to evaluate how SmartchainDB cluster handles increasing transaction loads. A set of 10 clients generated a workload, adhering to the structure of Workload *A* across three distinct scenarios: with each client initiating 1000+, 5000+, and 10,000+ transactions respectively. Fig. 6d illustrates the throughput performance of SCDB-cluster in comparison with ETH-SC and a single instance of SCDB. It is evident from the data that SCDB-cluster attains superior throughput levels, surpassing those of ETH-SC and SCDB-single. The stable throughput manifested by SCDB and SCDB-cluster across varying transaction volumes, especially the peak throughput of 3.2 TPS by SCDB-cluster, is indicative of their robust capacity to manage escalating workloads efficiently. This enhanced throughput can be attributed to the incorporation of a greater number of workload-generating clients

Fig. 8 illustrates the latency performance within the four-node SmartchainDB cluster, CREATE and REQUEST transactions maintained stable latency, reflecting a robust distribution system that efficiently handles task allocation and data retrieval without overloading individual nodes. The BID transactions, however, showed a modest latency increase, which is the result of more complex consensus being stressed under higher transaction volumes — the latency rose slightly from 654 ms to 732 ms as transactions scaled from 10K+ to 100K+. The ACCEPT_BID transactions, in contrast, revealed a pronounced spike in latency, which climbed steeply from 2491 ms at 10K+ transactions to 7899 ms at 100K+, indicating a significant computational overhead. This spike can be mainly attributed to the necessity of computing the appropriate RETURNs for BID, a process that becomes increasingly demanding as the number of transactions escalates

Usability. To measure the usability of these approaches, the number of lines of code required to implement a new marketplace was counted. SmartchainDB didn't require any user-implemented code, whereas the equivalent smart contract required 175 lines of code to establish one marketplace.

7 CONCLUSION AND ACKNOWLEDGEMENT

This paper introduces the concept of declarative blockchain transactions and outlines a methodology to implement it using an open-source blockchain database. By analyzing common transaction behaviors, we demonstrate this concept's advantages over smart contracts. Recognizing that some smart contract languages are Turing-complete and thus more expressive, our aim is not to replace them but to offer a suite of foundational primitives that reduce reliance on smart contracts for blockchain applications. A good parallel here is the SQL query that provides a set of primitives as a reasonable core for data processing within a Turing-complete environment. Our work serves as a foundation for blockchain transaction management, paving the way for future research to broaden these primitives' flexibility and introduce new transactional elements.

We want to acknowledge that our work was partially funded by the National Science Foundation CSR grant (number 1764025).

REFERENCES

[1] [n. d.]. BigchainDB. https://www.bigchaindb.com/.
[2] 2022. Ganache. https://trufflesuite.com/ganache/.
[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] 2022. TRUFFLE. Smart Contracts Made Sweeter. https://trufflesuite.com/truffle/.
[5] 2022. What is Tendermint. https://docs.tendermint.com/master/introduction/what-is-tendermint.html.
[6] 2023. Everledger. https://everledger.io/.
[7] 2023. The Peer-to-Peer Electronic Cash System for Planet Earth. https://www.bitcoinunlimited.info/.
[8] 2023. SmartchainDB: github repo. https://github.com/korchiev/smartchaindb.
[9] Cornelius C Agbo, Qusay H Mahmoud, and J Mikael Eklund. 2019. Blockchain technology in healthcare: a systematic review. In *Healthcare*, Vol. 7. MDPI, 56.
[10] Jameela Al-Jaroodi and Nader Mohamed. 2019. Blockchain in industries: A survey. *IEEE Access* 7 (2019), 36500–36515.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] 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).
[16] 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.
[17] 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.
[18] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2020. Adding concurrency to smart contracts. *Distributed Computing* 33, 3 (2020), 209–225.
[19] 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.
[20] Etherscan. [n. d.]. *Ethereum Gas Tracker*. Retrieved January, 2023 from https://etherscan.io/gastracker
[21] 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.
[22] 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.
[23] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 1197–1208.
[24] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep* (2018), 1–41.
[25] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. Fault-tolerant distributed transactions on blockchain. *Synthesis Lectures on Data Management* 16, 1 (2021), 1–268.
[26] 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.
[27] 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.
[28] Jelle Hellings and Mohammad Sadoghi. 2021. Byzantine Cluster-Sending in Expected Constant Communication. *arXiv preprint arXiv:2108.08541* (2021).
[29] Kai Hu, Jian Zhu, Yi Ding, Xiaomin Bai, and Jiehua Huang. 2020. Smart contract engineering. *Electronics* 9, 12 (2020), 2042.
[30] 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.
[31] Sergio Demian Lerner and RSK Chief Scientist. 2017. Lumino transaction compression protocol (LTCP).
[32] Fabrice Lumineau, Wenqian Wang, and Oliver Schilke. 2021. Blockchain governance—A new way of organizing collaborations? *Organization Science* 32, 2 (2021), 500–521.
[33] 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.
[34] 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.
[35] 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.
[36] Arim Park and Huan Li. 2021. The effect of blockchain technology on supply chain sustainability performances. *Sustainability* 13, 4 (2021), 1726.
[37] 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.
[38] 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.
[39] Vikram Saraph and Maurice Herlihy. 2019. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376* (2019).
[40] 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.
[41] 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.
[42] 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.
[43] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First monday* (1997).
[44] 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.
[45] 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.
[46] Fabian Vogelsteller and Vitalik Buterin. 2015. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20.
[47] 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.
[48] 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).
[49] Jacob Evans William Entriken, Dieter Shirley and Nastassia Sachs. 2018. EIP-721: Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721.
[50] 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.
[51] 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.
[52] 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.
[53] 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.
[54] 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).

[55] 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.

[56] 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.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009