An Algorand-based Approach for Flexible Execution of BPMN Choreographies

Nawaz Abdullah Malla[1], Alessandro Marcelletti[1], Andrea Morichetta[1,*] and Francesco Tiezzi[2]

[1]University of Camerino, Camerino, Italy
[2]Università degli Studi di Firenze, Firenze, Italy

Abstract

Blockchain technology has enabled secure and decentralized execution of inter-organizational business processes. Such processes define distributed interactions which can be designed using Business Process Model and Notation (BPMN) choreographies, encoded as smart contracts and executed in the blockchain. However, such processes may need to change at runtime to deal with unforeseen situations. This leads to the need for flexibility mechanisms, permitting the adaptation of the choreography specification and consequently the blockchain implementation. However, the immutability of blockchain poses challenges in this direction, as smart contract code typically cannot be updated once deployed. While some solutions as redeployment and patterns, are available, these introduce high costs and increased complexity. With the recent advent of the Algorand blockchain, intrinsic update capabilities are available, opening for novel solutions addressing flexibility needs. For this reason, we use Algorand to enable flexible execution of BPMN choreographies. Our approach modularly encodes choreographies as Algorand smart contracts, giving the means to allow and control runtime adaptation of the implementation. By changing the initial choreography, corresponding updates are reflected in the underlying smart contract, providing a novel solution that eliminates the need for complex gas-intensive patterns.

Keywords

BPMN choreography, Flexibility, Smart contract update, Algorand, business processes, blockchain

1. Introduction

The advent of blockchain has enabled the creation of novel decentralized applications where participants have trust and accountability needs. Among the various domains, inter-organizational business processes have found in blockchain a proper solution to create technical infrastructures supporting process execution in decentralized and untrusted environments [1]. By encoding business logic into smart contracts, the latter enforce the proper execution of the process behavior, automatically executing constraints and advancing the state of the process. Blockchain immutability and transparency also play a crucial role, as they allow for keeping a secure and accessible track of past business interactions [2, 3].

Over the years, different approaches have emerged to leverage blockchain for process automation and coordination, typically relying on Ethereum-based platforms [4, 5]. Such solutions specify inter-organizational interactions employing the Business Process Model and Notation (BPMN) standard [6], which has been established over the years as a dominant solution both in industry and academia. In particular, *BPMN choreographies* allow for modeling inter-organizational interactions, specifying only the message exchange between participants without exposing their internal behavior. In this context, blockchain serves as a target technology, encoding the choreography and its interactions, supporting their distributed execution at runtime [7, 8].

While the adoption of blockchain has enabled a concrete and trustworthy enactment of choreographies, it has also led to several challenges, among which *flexibility* is a significant one. When modeling business interactions in terms of a choreography, not all scenarios can be foreseen. Thus, it is crucial to deal with possible factors that can affect a process at runtime, which may lead to the introduction

DLT2025: 7th Distributed Ledger Technology Workshop, June, 12-14 2025 - Pizzo, Italy
*Corresponding author.
nawaz.malla@unicam.it (N. A. Malla); alessand.marcelletti@unicam.it (A. Marcelletti); andrea.morichetta@unicam.it (A. Morichetta); francesco.tiezzi@unifi.it (F. Tiezzi)

of changes in the initially designed choreography model and, hence, the consequent update of the underlying implementation [9]. This ability to react to unforeseen scenarios is called *adaptation* [10].

However, in the context of blockchain-based execution of choreographies, flexibility remains a persistent and open challenge due to the native structure of blockchain [4]. Indeed, the inherent immutability of blockchain typically does not allow for direct updates to smart contract code once deployed, making it difficult to deal with runtime changes. This requires dedicated solutions, opening to new research directions. While updates can be implemented through new smart contract deployments, these come with significant costs and operational overhead, especially if adopting complex patterns [11]. Off-chain solutions have been explored to address this limitation [12], but they introduce their own set of challenges, including security risks and increased complexity in integration.

On the other hand, these limitations can be overcome nowadays by exploiting blockchain architectures that introduce mechanisms for more flexible and adaptive execution [13]. In particular, Algorand [14] is a blockchain platform that aims to address the 'blockchain trilemma', concerning the simultaneous satisfaction of the scalability, security, and decentralization challenges. Designed with a pure proof-of-stake consensus mechanism, Algorand provides instant transaction finality, high throughput, and robust fork resistance, which are key properties for enabling reliable, time-critical business workflows. Traditional blockchain platforms, such as Ethereum, often suffer from issues like unpredictable transaction fees and intricate upgrade procedures, which can limit scalability and reduce the flexibility needed for dynamic business process execution [15]. Instead, Algorand adopts a fixed, low-fee model that provides predictability and cost efficiency for developers and users alike. Furthermore, Algorand offers native features that facilitate updates of smart contracts and storage elements, without the use of proxies, enabling seamless and secure business process evolution and minimizing operational disruption [16]. These capabilities make Algorand particularly well-suited for implementing flexible choreographies, where adaptability, reliability, and low overhead are critical.

In this work, we explore how Algorand's design principles can be leveraged to realize **flexible and trusted execution of inter-organizational business processes specified in terms of BPMN choreographies**. To this purpose, we encode BPMN choreographies as smart contracts, deployed and executed on the Algorand blockchain. We show how the adaptation of a choreography at runtime is reflected and enacted on the corresponding smart contract. In fact, a choreography can be adapted to multiple perspectives of the process, and each change leads to a specific update in the underlying smart contract. By relying on the native Algorand update capabilities, we do not need complex and gas-consuming patterns or operations to update the underlying smart contract. In addition, using a native update mechanism allowed us to define a lean approach to control the conditions enabling smart contract updates, thus fostering the definition of update policies for regulating the choreography adaptation. Furthermore, the performance efficiency of Algorand permits dealing with adaptation without incurring high costs.

The rest of the paper is structured as follows. Section 2 provides an overview of the Algorand blockchain and BPMN choreographies. Section 3 introduces our approach and focuses on possible changes that can be applied to a choreography. Section 4 details the implementation of the approach, showing how smart contracts encode the business logic of choreographies and how they can be updated at runtime; the section also provides a cost analysis for the flexible execution approach we propose. Section 5 compares our approach with relevant related works, while Section 6 concludes the work and touches on future directions.

2. Background

In this section, we introduce basic notions of the Algorand blockchain, focusing on its peculiarities and technical features. We then describe BPMN choreography diagrams and their elements via a simple example.

2.1. Algorand

Algorand is a decentralized, permissionless blockchain platform designed for scalability, security, and true decentralization [17, 14]. Different to other blockchain technologies such as Ethereum-based ones, it introduces several novelties both at the protocol and implementation layers. As a consensus mechanism, Algorand relies on a Pure Proof-of-Stake (PPoS) protocol, which randomly selects a committee of users to propose and validate blocks depending on their stake amount in the network. This approach ensures fast finality, low latency, and resistance to forks while maintaining decentralization.

Algorand supports smart contracts (also referred to as *applications*) through the Algorand Virtual Machine (AVM). Developers can write smart contracts using Transaction Execution Approval Language (TEAL), a language optimized for safety and performance. To simplify development, Algorand also offers Python-based languages, i.e., PyTEAL and Algorand Python. In this work, we use the latter one.

Another key difference with respect to other platforms is the Algorand storage model, which includes three different types of storage: Local storage, Global storage, and Box storage. Specifically, the local storage is suited for storing user-specific data, the global storage for shared data accessible by all users, and the box storage for application-specific data. In this work, we mainly rely on the global storage, as it provides a persistent on-chain storage that serves as shared memory among business participants to keep exchanged information. More specifically, the global storage in Algorand is a key-value store directly associated with a smart contract application. Each application can store up to 64 key-value pairs, with each pair having a combined maximum size of 128 bytes (key + value $\leq$ 128 bytes), and a total storage limit of 8 KB shared across all pairs. Technically, this storage is maintained as part of the application's on-chain state, where keys are stored as byte slices and values can be either byte slices or uint64 integers. To modify global storage in a smart contract, developers must use the appropriate methods within the contract code. Global state can be managed using two approaches:

- *Implicit declaration* (`self.var = UInt64(1)`): This method directly assigns a typed value to the variable, offering concise syntax. It provides limited control over state behavior and obscures the underlying storage mechanism.
- *Explicit declaration* (`self.var = GlobalState(UInt64(1))`): This approach clearly defines the variable as part of the global state. It uses `.value` for access and modification, and provides a richer API and support for explicit deletions.

Both approaches use the same TEAL opcodes and incur identical minimum balance requirements. However, the explicit method is preferred for clarity, better error handling, and maintainability.

2.2. BPMN Choreographies

The BPMN standard provides the choreography diagram [6, Ch. 11] as an intuitive and graphical means to describe the interactions that should occur among the multiple participants of a distributed scenario. We illustrate some key elements of this kind of diagram by resorting to the choreography model in Fig. 1. We use this simple model throughout the paper as a running example.

The starting point of the choreography is represented by the *start* event (start1), which is drawn as a circle with an outgoing edge. As in any workflow diagram, *edges* are used to specify the execution flow by connecting choreography elements. Although edge names typically are not visualized in the graphical representation of the diagram, these names are always included in the underlying XML file produced by BPMN modelers; to improve the paper's readability, we reported edge names (using the gray color) in the model in Fig. 1.

The start event is connected by edge e1 to a *choreography task* element, which describes a message exchange between two participants involved in the choreography. The task is represented as rectangles divided into three bands: the central one contains the task's name (Task 1), the white band refers to the initiator participant (A), and the grey one refers to the recipient participant (B). The exchanged message, of type msg1, is graphically represented by a white envelope. Its payload is formed by a single field (x), whose actual content will be used subsequently in the choreography execution to make a choice.

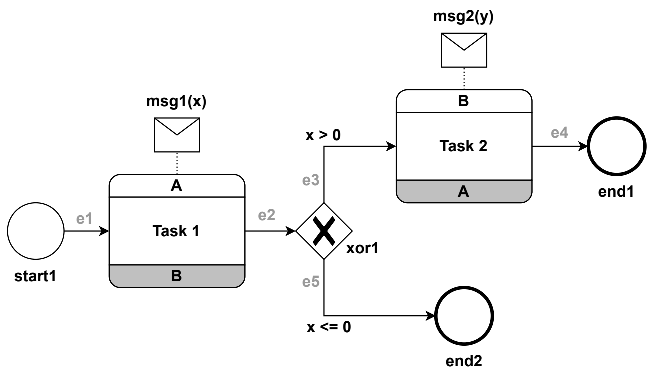

Figure 1: Running example: original choreography.

```python
@baremethod(allow_actions=["UpdateApplication"])
def update(self) -> None:
    assert Txn.sender == self.admin.value
    assert self.update_policy.value == UInt64(1)
    self.locked.value = UInt64(1) #Lock the contract

@arc4.abimethod
def update_global_store(self) -> None:
    assert Txn.sender == self.admin.value,

    #Added variables
    self.e6.value = UInt64(0)
    self.msg4_payload_k.value = UInt64(0)
    self.msg3_payload_z.value = UInt64(0)
    self.party_c.value = Account("OWSIGW6NHI...WIH775ZZLE")

    #Removed variables
    del self.msg2_payload_y.value

    self.locked.value = UInt64(0) # Unlock the contract
```

Listing 1: Update policy example.

The execution flow of the choreography is controlled by using *gateways*. Gateways act as either split nodes (forking into outgoing edges) or join nodes (merging incoming edges). In the example model, Task 1 is connected by e2 to the *exclusive (XOR) gateway* xor1. This is drawn with a diamond marked with the × symbol and represents a conditional choice based on the value previously exchanged via message msg1. More specifically, this is a XOR-split gateway defining two execution branches to be selected on the basis of the conditions labelling the outgoing edges (i.e., e3 is selected if the value of the field x is greater than 0, otherwise e5 is selected); when executed, the gateway activates exactly one outgoing edge.

If x>0, the choreography execution continues with Task 2, which allows the participant B to reply to the participant A with a message of type msg2. After this message exchange, the choreography terminates by means of an *end* event (end1), which is drawn as a circle with an incoming edge and denotes an ending point of the choreography. If instead x<=0, the choreography execution directly terminates by means of the end2 event.

3. Flexible Execution of Choreographies on Algorand

In this section we show the general idea of how BPMN choreographies can be executed on the Algorand blockchain considering flexibility needs. We start by describing the main components and then we focus on the adaptation of choreographies and the different kind of changes that can be applied.

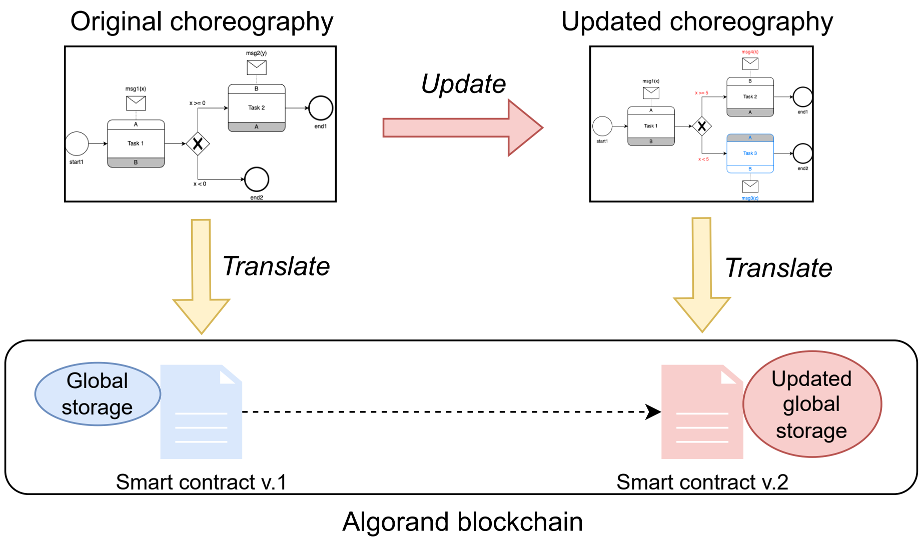

Figure 2: Overview of the main phases and components supporting flexible execution of choreographies.

Figure 2 depicts an overview of the different phases supporting the deployment and update of choreographies. During the **deployment**, a BPMN choreography model is created and translated into blockchain artifacts. In particular, a choreography translates into a smart contract, containing the logic of the interactions to be executed. Additionally, the Algorand global storage is instantiated, containing all relevant information to be stored in a permanent manner, such as data exchanged through choreography messages. At this point, the smart contract can be executed by interested parties sending data according to the deployed logic. During the execution, a choreography may need to adapt to certain changes in the business scenario or the general context. To manage such flexibility need, an **update** is performed on the choreography model, by applying changes to elements such as tasks, messages or participants. This leads to an update of its smart contract implementation, as the choreography is translated into a new version of the smart contract, which is deployed in the blockchain and replaces the existing one. In addition to the smart contract logic, it is also possible to update the global storage by operating on the relative information (e.g., adding or removing storage variables).

3.1. Runtime Adaptation of Choreographies

To ensure a flexible execution of choreographies, the update phase is crucial as it consists of the choreography and its underlying implementation. For this reason, here we focus on the high-level changes that can be applied to the choreography specification and used to derive the new version of the smart contract.

Choreography changes. When adapting a choreography at runtime, several changes can be made to the BPMN model categorized into different types [10, 13]. In this work, we focus on the (i) **addition**, (ii) **deletion**, and (iii) **modification** operations, as they represent atomic types of changes that can be applied to a choreography and that directly reflect on the smart contract and the global storage. These operations can affect the various BPMN choreography elements introduced in Section 2, specifically by adding, removing, and modifying (a) tasks, (b) messages, (c) participants, (d) control flow elements, and (e) events.

To provide an overview of the possible changes that can be applied, we rely on an updated version of the running example model in Fig. 1. Let us adapt it as shown in Fig. 3 to support two different replies from participant B: one directed to A and another one to C. More specifically, the changes in the adapted model are as follows (modified elements are highlighted in red color, while added ones are in blue):

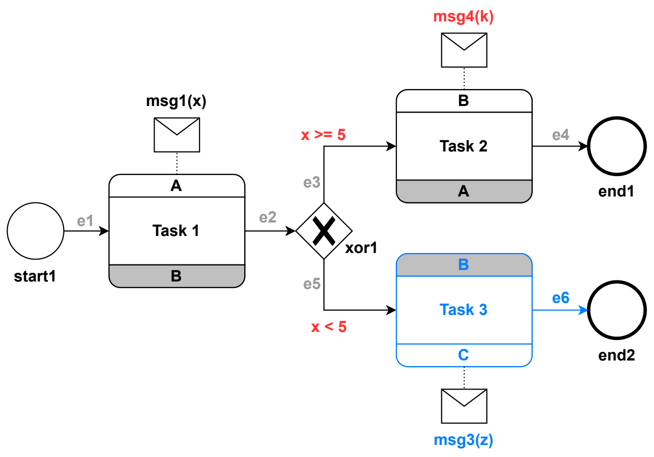

Figure 3: Running example: updated choreography.

(i) the XOR conditions are modified; (ii) message msg2 is replaced by msg4; (iii) Task 3, exchanging message msg3, is added; (iv) edge e6, connecting Task 3 to end2, is added. It is worth noticing that the shown types of changes represent base operations that can be combined or integrated to derive more complex patterns. For example, choreography elements can be moved, swapped, or replaced. Anyway, thanks to their compositional nature, these patterns do not pose any additional issue concerning the update of the underlying smart contract and global storage.

Controlling adaptation. While updating BPMN choreographies at runtime allows for dynamic adaptation to evolving requirements, some restrictions could be necessary. To this purpose, update policies can be adopted to regulate and constrain changes to the choreography.

A first possibility is the enablement or disablement of changes. In certain contexts, it may be desirable to disable updates for a certain period once the choreography is deployed. For instance, changes may be allowed only during specific reconfiguration periods or under defined conditions, such as when a quorum among participants is reached. Furthermore, a limited number of updates can be allowed for a given choreography; once this limit is reached all update requests will be denied. Another important aspect refers to participant permissions. Not all entities involved in the choreography should necessarily possess the authority to modify its structure. Policies can be used to assign update rights only to authorized participants, such as process owners. In addition, policies can be used to define element-specific restrictions. To safeguard essential parts of the process logic, certain elements may be declared immutable, or some elements act as a commit action that disables the update capability.

These various types of policies can be embedded in the smart contract layer associated with the choreography model. In this way, they will be a publicly visible part of the contract. By doing so, the execution infrastructure can automatically enforce the specified constraints, thereby mitigating the risk of undesired operations. The effectiveness and robustness of this controlled form of controlled update strongly rely on the native update mechanisms of Algorand.

4. Implementation

In this section we present the implementation of our proposal according to the different phases described in Section 3. We start by describing how an Algorand smart contract implements a BPMN choreography, then showing how it reflects runtime updates.

4.1. Choreography smart contract deployment

The first phase consists of the deployment of the Algorand smart contract encoding the logic of the BPMN choreography model. To this purpose, the smart contract is derived through a translation of the choreography model into smart contract code. As a translation mechanism, in this work we rely on the general approach presented in [18]. This is a *modular* translation, in the sense that the structure of the generated smart contract follows the structure of the choreography from which it originates, thus ensuring a one-to-one correspondence between BPMN choreography elements and smart contract functions. The translation is defined by induction on the syntax of choreographies and transforms each element of the BPMN choreography model into a corresponding function of the smart contract that faithfully implements the element's semantics. The general structure of the smart contract is composed of several functions related to: state initialization, message execution, control flow execution, and contract update. In the following, we show the main parts of the smart contract, focusing on the variables and functions affected by the successive update.

Listing 2 shows the `init` and `create` methods used to initialize the global storage of the smart contract. This state contains several variables. The `admin` variable (line 3) represents a privileged account used to manage sensitive operations such as updates or administrative controls in an Algorand smart contract. It is typically declared as `self.admin = GlobalState(Account)`, storing the authorized address directly in the global storage. This variable can be initialized at deployment using `self.admin.value = Txn.sender` (line 23), or assigned to a specific address with `self.admin.value = Account("ADDRESS")`. Access control is then enforced in the other methods through checks like `assert Txn.sender == self.admin.value`. Variables `party_a` and `party_b` (lines 4-5) store the addresses of the two choreography participants, i.e. A and B, respectively. The subsequent list of variables (lines 8-13) represents the marking of the edges e1, ..., e5 connecting the BPMN elements of the choreography (notably, e0 represents a spurious edge for denoting the enabled/disabled status of the start event). The values of these variables represent the choreography marking, which drives the execution flow. Message-specific variables (lines 16-17) keep track of the exchanged message data. The `locked` variable (line 20) is used to lock the smart contract methods (by means of assertion `assert self.locked.value == UInt64(0)` that causes method calls to fail) until the global storage is correctly updated. This variable acts as a safety switch: when it is set to false (i.e., value 0), the contract runs normally, instead, during updates it is set to true (i.e., value 1) to block all external activities, preventing inconsistencies; after the global storage updates, it is reset to false to resume execution safely. More details on the update procedure are provided in Section 4.2. Finally, the `create` method (lines 22-25) initializes the admin variable (line 23) and starts the process execution at the app creation time by placing one token on the edge e0 (line 24).

Listing 3 shows how the logic of a BPMN choreography model is rendered in terms of Algorand Python smart contract methods. In fact, the translation approach we rely on maps each choreography element (i.e., event, gateway, or task) into a method. Specifically, the reported code refers to the methods corresponding to the first three elements of the running example model, i.e., the event start1, the task Task 1, and the XOR-split gateway xor1; the other elements are translated in a similar way. The method corresponding to the start event start1 (lines 1-7) checks if the spurious edge of the event (i.e., e0) is marked; in the negative case it returns false (i.e., value 0), because the condition that triggers the element execution is not satisfied, otherwise the method moves one token from the spurious edge to the outgoing edge (i.e., e1) and returns true (i.e., value 1). Notably, `@subroutine` are methods to encapsulate internal logic, where no user interaction is admitted. These functions are not publicly exposed and can only be called internally by the contract. In contrast, for tasks involving user interaction, we defined `@arc4.abimethod` methods, which serve as the public, externally callable interface of the smart contract.

The method corresponding to the task Task 1 (lines 9-17) allows an external user to exchange the message msg1; this is indeed a directly callable method (see annotation `@arc4.abimethod`). The method accepts as input the payload to be sent, which is then stored inside the global storage in the variable `self.msg1_payload_x` (line 15). Before executing the method, three checks are performed: (i) the contract must be unlocked (line 11); (ii) the task must be enabled according to the control flow of the choreography (i.e., the edge e1 must be marked) (line 12); (iii) the caller account must corresponds to participant A (line 13). If all checks are positive, the method consumes the incoming token (line 14), stores the input data in the global variable (line 15), marks the outgoing edge to advance the control flow (line 16), and resumes the choreography execution (line 17). Finally, the method corresponding to the XOR-split gateway xor1 (lines 19-28) checks if the gateway is enabled (i.e., the incoming edge e2 is marked); in

```python
def __init__(self) -> None:
    # Account variables
    self.admin = GlobalState(Account)
    self.party_a = GlobalState(Account("ZLVEALW2YMEDRRXIUYHMGVTCCPQVJR6YBF22TVWFJE7DRKJEOL5JBBUDLA"))
    self.party_b = GlobalState(Account("6WHFXXVYMBFTZGRAVKGLNAGFUSPF7TYAXD5KL7H5MWB3VKJWJLRI7ZBTIY"))

    # Edge marking variables
    self.e0 = GlobalState(UInt64(0))
    self.e1 = GlobalState(UInt64(0))
    self.e2 = GlobalState(UInt64(0))
    self.e3 = GlobalState(UInt64(0))
    self.e4 = GlobalState(UInt64(0))
    self.e5 = GlobalState(UInt64(0))

    # Message-specific variables
    self.msg1_payload_x = GlobalState(UInt64(0))
    self.msg2_payload_y = GlobalState(UInt64(0))

    # Locking variable
    self.locked = GlobalState(UInt64(0))  # 0 = unlocked, 1 = locked

def create(self) -> None:
    self.admin.value = Txn.sender
    self.e0.value = UInt64(1)
    self.locked.value = UInt64(0)
```

Listing 2: Initialization of global state for the choreography contract.

the negative case it returns false, otherwise it executes the logic of the gateway and returns true. The gateway logic consists of consuming the incoming token (line 22) and checking the condition x>0 (line 23), on the basis of which one of the outgoing edges is marked (line 24 or line 26).

The execution of the choreography is carried out by the `execute()` method, which has the same code for any choreography (i.e., it is choreography agnostic). This is the public method that one has to invoke to activate the choreography once the smart contract has been deployed in the blockchain. It iteratively executes one choreography element method at a time until one element is successfully executed (denoted by the return value true); in our running example, the invoked methods are `start1()`, `xor1()`, `end1()`, and `end2()`. Notably, methods `task1()` and `task2()` are not invoked by `execute()`, because they have to be called by external users. If all elements of the choreography cannot be executed, the instance execution stops. In case there are enabled tasks, the execution is suspended and will be resumed when the waited message is delivered (see the call `execute()` as the last instruction of a task method's body). The complete code of the running example choreography is available in an online public repository [19].

Once the Algorand Python smart contract is derived from the choreography model, it has to be compiled to obtain the corresponding TEAL files, containing the approval and clear programs. Finally, these files can be deployed in the Algorand blockchain, creating an Algorand application, via the following command:

```
algokit goal app create
   --creator XYWE34VCZTEGZJ5PBTGN44PT75ZGUG7T2COJFAYU6I3TUHW5J7TNSYI3ZE
   --approval-prog original_running_example.approval.teal
   --clear-prog original_running_example.clear.teal
   --global-byteslices 16 --global-ints 35 --local-byteslices 8 --local-ints 8 --extra-pages 3
```

It is worth noticing that the above command specifies the maximum number of byte slices that may be stored in the global/local store (`global-byteslices/local-byteslices`), the maximum number of integer values that may be stored in the global/local store (`global-ints/local-ints`), and the additional program space for supporting larger TEAL programs (`extra-pages`). The specification of these limits is necessary in order to provide enough space for future extensions of the contract and related stores. However,

```python
@subroutine
def start1(self) -> UInt64:
    if self.e0.value == UInt64(1):
        self.e0.value -= UInt64(1)
        self.e1.value += UInt64(1)
        return UInt64(1)
    return UInt64(0)

@arc4.abimethod
def task1(self, msg1_param_x: UInt64) -> None:
    assert self.locked.value == UInt64(0), "Contract is locked"
    assert self.e1.value > UInt64(0), "Task 1 not active."
    assert Txn.sender == self.party_a.value, "Only A can send msg1"
    self.e1.value -= UInt64(1)
    self.msg1_payload_x.value = msg1_param_x
    self.e2.value += UInt64(1)
    self.execute()

@subroutine
def xor1(self) -> UInt64:
    if self.e2.value > UInt64(0):
        self.e2.value -= UInt64(1)
        if self.msg1_payload_x.value > UInt64(0):
            self.e3.value += UInt64(1)
        else:
            self.e5.value += UInt64(1)
        return UInt64(1)
    return UInt64(0)
```
Listing 3: Contract methods corresponding to BPMN choreography elements (an excerpt).

after the application creation, such values are immutable.

Remark 1. *When deploying the choreography's smart contract, the admin user must take into account the limits imposed by the Algorand platform on the occupancy of the contract and the associated storage. In particular, the user has to foresee how much additional space the contract and its variables may require in the future, as these space limits cannot be modified later. Notably, as shown in Section 4.2, variables unused in the updated contract can be removed from the global storage, freeing the space they occupy.*

4.2. Choreography smart contract update

The second implemented phase includes the update of the choreography smart contract due to an adaptation of the choreography model. This is achieved by translating the adapted BPMN model into a new version of the choreography smart contract and then updating it in the blockchain.

To this purpose, we rely on the adapted choreography model illustrated in Fig. 3. While the translation of the choreography's logic follows the approach previously described, its update is managed by means of two dedicated methods: one for updating the smart contract code and checking update rights, and another one for updating the global storage, initializing new variables, or deleting old ones. Therefore, the update procedure requires two separate method invocations.

Listing 4 shows the methods involved in the update. Differently from the previous shown methods, `update()` (line 1) is a `baremethod`, i.e., a low-level method for administrative operations that does not require data type encoding. In particular, it handles the raw action `UpdateApplication`. This method contains all the checks required to manage permissions for update and can trigger additional logic. In our running example, the update method checks if the user performing the contract update is authorized (line 3). Here, we assume that only the admin user can perform the update operation; of course, such privilege could be granted to more than one user, on the basis of the specific authorization policies required by the considered scenario. In addition, the method locks the contract by setting the variable

```
1  @baremethod(allow_actions=["UpdateApplication"])
2  def update(self) -> None:
3      assert Txn.sender == self.admin.value, "Only admin can update the contract"
4      self.locked.value = UInt64(1)  # Lock the contract
5  
6  @arc4.abimethod
7  def update_global_store(self) -> None:
8      assert Txn.sender == self.admin.value, "Only admin can update the global store"
9  
10     # Added variables
11     self.e6.value = UInt64(0)
12     self.msg4_payload_k.value = UInt64(0)
13     self.msg3_payload_z.value = UInt64(0)
14     self.party_c.value = Account("OWSIGW6NHIYMNMP7ZA53Z23GTRDXZYB3TGDPH5LH5TI3QBHDWIH775ZZLE")
15 
16     # Removed variables
17     del self.msg2_payload_y.value
18 
19     self.locked.value = UInt64(0)  # Unlock the contract
```

Listing 4: Methods managing the smart contract update.

`locked` accordingly (line 4). This prevents executing any methods implementing the logic of the updated choreography until the global storage is updated as well, in order to avoid any inconsistency.

The admin user can perform the update of the smart contract by means of the following command:

```
algokit goal app update
   --app-id 1001
   --from XYWE34VCZTEGZJ5PBTGN44PT75ZGUG7T2COJFAYU6I3TUHW5J7TNSYI3ZE
   --approval-prog updated_running_example.approval.teal
   --clear-prog updated_running_example.clear.teal
```

where `1001` is the identifier of the previously deployed Algorand application (generated and returned by the `create` command). The `update` command automatically invokes the `update()` method of the smart contract currently deployed; if it executes successfully, the current contract code is replaced by the new one.

The second method in Listing 4 (lines 6-19) permits updating the global storage to include new variables (representing, e.g., the marking of edges or the payload of exchanged messages) and/or delete variables no longer used in the updated contract. Similarly to the `update()` method, it can be successfully executed only by the admin user (line 8). The method performs additional actions that depend on the specific changes applied to the choreography. In this example, it adds variables representing the new edge e6 (line 11), messages msg3 and msg4 (lines 12, 13), and participant C (line 14). Moreover, the method deletes the variable representing the removed message msg2 (line 17). The variables that have to be added or removed can be automatically determined by comparing the original smart contract and the one generated from the choreography after it has been modified. Notably, the replacement of msg2 into msg4 is achieved by deleting the former variable and adding the latter. Finally, the method always unlocks the contract (line 19).

Remark 2. *Although, in principle, unused variables could be kept in the global storage, we decided to delete them in order to free the memory space they occupy. Indeed, we recall that the dimension of the global storage is limited and fixed once and for all at application creation time.*

It is also worth noticing that all variables referred into the the two methods in Listing 4 must be declared in the `init()` method. This also applies to deleted variables (`msg2_payload_y` in the example). Anyway, in case of further updates, the variable deleted in this code could be omitted by subsequent versions of the contract code.

Remark 3. *It is worth noticing that we cannot perform the update of both the smart contract and the global storage in a single step. Indeed, if the code performing the update of the global storage (within* `update_global_store()`*) would be moved inside the* `update()` *method, it would not be executed when the update of the smart contract is called by means of the* `algokit goal app update` *command. This is because, the command executes the* `update()` *method of the currently deployed contract, i.e., the initial one, and not the updated one.*

4.3. Cost analysis

To evaluate the feasibility of the proposed implementation, we present an evaluation of the cost analysis of the executed scenarios on the Algorand blockchain[1], expressed in both cryptocurrency and fiat currency.

More specifically, we have considered different execution scenarios concerning our running example. First, we executed the original choreography depicted in Figure 1 (scenarios 1 and 2), then we considered executions where the choreography is replaced by the one shown in Figure 3 (scenarios 3, 4, and 5). Due to Algorand's flat and low transaction cost structure, the expenses in *microAlgos* (μA) remain especially low (the fees for every transaction in our experiments are 1000 μA).

We detail below the five scenarios, whose results are reported in Table 1, also with the USD conversion of total costs[2].

1. **Scenario 1**: The choreography is deployed (1000 μA) and start1 is executed (1000 μA). Then, Task 1 is executed (1000 μA), providing an input value greater than 0, which directs the execution flow to Task 2. After the execution of Task 2 (1000 μA), the choreography terminates with end1.
2. **Scenario 2**: The choreography is deployed (1000 μA) and start1 is executed (1000 μA). Then, Task 1 is executed (1000 μA) providing an input value equal to 0, leading the choreography directly to terminate with end2.
3. **Scenario 3**: After deploying (1000 μA) the choreography and executing start1 (1000 μA), an update is performed via methods `update()` (1000 μA) and `update_global_store()` (1000 μA). Following this, Task 1 is executed (1000 μA) providing an input value greater than 5, which triggers Task 2. The execution of Task 2 (1000 μA) then leads the choreography to terminate with end1.
4. **Scenario 4**: After deploying (1000 μA) the choreography and executing start1 (1000 μA), an update is performed via methods `update()` (1000 μA) and `update_global_store()` (1000 μA). Following this, Task 1 is executed (1000 μA) providing an input value less than 5, which triggers Task 3. The execution of Task 3 (1000 μA) then leads the choreography to terminate with end2.
5. **Scenario 5**: The choreography is deployed (1000 μA) and start1 is executed (1000 μA). Then, Task 1 is executed (1000 μA) providing an input value greater than 5, which directs the execution flow to Task 2. An update is performed via methods `update()` (1000 μA) and `update_global_store()` (1000 μA). The execution of the updated Task 2 (1000 μA) then leads the choreography to terminate with end1.

Notably, if the update is performed after Task 1's execution, task Task 3 can never be executed. In fact, the execution of the choreography automatically progresses until it reaches a task, which requires the related task initiator user to send the message by calling the method corresponding to the task. Only during the period from the task enabling and its invocation, the `update()` method can be called to update the contract. Hence, once Task 1 is executed, the execution flow stops at Task 2 and therefore, after the update, Task 3 is unreachable (hence, the corresponding method is disabled).

If we consider choreography-based approaches in the literature, such as ChorChain [20] and FlexChain [21], we can draw a general comparison with our solution, despite the differences in their objectives and implementation architectures. Indeed, we observe that on a performant blockchain like Polygon[3], executing a choreography of 10 messages incurs approximately 0.10 USD for a supply chain scenario

[1]We used the Algorand local network for these experiments.
[2]Based on the ALGO/USD conversion at the time of writing (16/04/2025).
[3]Based on the POL/USD conversion at the time of writing (19/05/2025)

Table 1
Execution costs for the running example.

Scenario	Cost (μA)				Total (USD)
	Deploy	Update	Execution	Total	
1	1000	–	3000	4000	0.000717
2	1000	–	2000	3000	0.000537
3	1000	2000	3000	6000	0.0011
4	1000	2000	3000	6000	0.0011
5	1000	2000	3000	6000	0.0011

```python
1  def __init__(self) -> None:
2      ...
3      # Update policy
4      self.update_policy = GlobalState(UInt64(0))  # 0 = update disabled, 1 = update enabled
5
6  def create(self) -> None:
7      ...
8      self.update_policy.value = UInt64(1)
9
10 @arc4.abimethod
11 def task1(self, msg1_param_x: UInt64) -> None:
12     ...
13     self.update_policy.value = UInt64(0)  # Disable the update
14     self.execute()
15
16 @baremethod(allow_actions=["UpdateApplication"])
17 def update(self) -> None:
18     assert Txn.sender == self.admin.value, "Only admin can update the contract"
19     assert self.update_policy.value == UInt64(1), "Update is disabled"  # check the update policy
20     self.locked.value = UInt64(1)
```

Listing 5: Update policy example.

and 0.08 USD for an X-ray analysis scenario. In contrast, our implementation, for the same number of tasks, would require approximately 11,000 μA. This corresponds to an estimated total amount of about 0.0026 USD, demonstrating negligible costs and economic viability of our solution.

4.4. Update policies

The update mechanism of the smart contract can be governed through a policy-based control mechanism. A simple policy is illustrated in Listing 5. The purpose of this policy is to disable the update capability of the choreography once Task 1 has been executed. To achieve this, we rely on variable `update_policy` that acts as a flag: a value of 0 disables updates, while a value of 1 enables them. The variable is initialized to 1 at contract creation (line 8), indicating that updates are initially allowed. The update capability is disabled after the execution of Task 1 (line 13). The `update()` method (line 17) checks the value of `update_policy` (line 19); if updates are disabled, the assertion fails, causing the transaction that invokes `update()` to fail. Other simple update policies, based on the status of the choreography or the authorization rights of the user requesting the update, can be implemented similarly.

5. Related Works

The integration of blockchain technology with BPM has obtained attention in recent years, driven by the need for trustworthiness in multi-party business processes [22]. Blockchain, with its inherent immutability and decentralized nature, offers a promising solution for ensuring the integrity and transparency of such processes. Over the years, many works explored the creation of trustworthy blockchain-based systems for the execution of processes, mainly leveraging Ethereum [23, 24, 20] and tackling specific challenges such as confidentiality [25, 26, 27, 28], and flexibility [29, 21].

The latter is indeed a key limitation shared by Ethereum-based systems due to the lack of secure and modular upgradeability. Ethereum contracts are immutable by default, requiring complex proxy-based upgrade patterns [30] to allow logic evolution. These proxies delegate execution to separate logic contracts while maintaining state, but introduce significant risks, including misalignment of storage slots, selector clashes, and initialization bugs [31]. Tools like PROXYEX [11] and USCDetector [31] have revealed severe vulnerabilities in many real-world upgradeable contracts. Additionally, empirical studies show that a large portion of upgradeable contracts are never actually updated, raising concerns about the practicality and maintainability of such patterns [13].

These challenges are not unique and extend to BPM-specific platforms as well. The complexity of maintaining backward compatibility, redeploying stateful logic, and orchestrating transitions between contract versions can disrupt business workflows and undermine scalability [4]. For these reasons, researchers have emphasized the importance of flexibility in business process management, proposing solutions that answer the need to adapt process definitions at run-time. The authors in [21] encode BPMN choreography diagrams as a set of rules. These are then stored on-chain and evaluated off-chain for advancing the process state. In this way, runtime updates are enabled by updating the new rules corresponding to the new choreography version. However, introducing an off-chain execution engine represents a good trade-off between trust and flexibility, at the expense of potential vulnerabilities connected to the off-chain part. Authors in [32] focus on the flexibility of executing business processes on federated blockchains. The main goal is to provide interactions between different technologies, where the system is developed with on-chain and off-chain components. Authors in [29] propose a dynamic role binder for runtime choice of processes, focusing on controlled flexibility. This is done by binding and unbinding actors to a role in a dynamic way. The authors provide a model and corresponding binding policy specification language for the dynamic binding of actors to roles in collaborative processes. However, despite the possibility of selecting alternatives at runtime, there is no support for the introduction of brand-new elements in the choreography. In [33] the authors investigate the issue of upgradeability of smart contracts. They use static and dynamic contracts for the storage of logic and states, which incur high costs and a high level of complexity. The Algorand blockchain was considered in [34], where the authors proposed a translation from the Dynamic Condition Response (DCR) process modelling language to the Transaction Execution Approval Language (TEAL). However, the work focuses on the translation approach, without considering an explicit management of runtime updates.

Despite significant advancements, flexibility remains a persistent challenge in the execution of decentralized choreographies. Existing works typically rely on custom strategies that either introduce off-chain components or support a limited scope of changes. This limitation becomes even more important when considering BPMN choreographies, for which only few solutions are available. Our proposal leverages Algorand technology to natively support a fully flexible execution of choreographies considering a wide range of changes across all choreography elements. The choice of Algorand [17] was driven by its different execution model and native support for stateful smart contract upgrades, enabling the update of contract logic in place, without relying on proxies or external redeployment. This results in more secure, predictable, and maintainable upgrade paths. Moreover, a recent comparative study by Weber et al [35] demonstrates that Algorand significantly outperforms Ethereum in terms of runtime cost and process stability for BPM applications. Furthermore, Algorand's Pure Proof-of-Stake consensus protocol offers deterministic finality and low-latency transaction confirmation, crucial for maintaining the responsiveness of distributed BPMN choreographies [36].

6. Conclusions

Blockchain enables new forms of inter-organizational collaborations where trust and accountability are central concerns. Over the years, many solutions have been proposed to implement such collaborations on Ethereum-based platforms. Such solutions rely on a high-level specification, translated then into low-level code. In particular, BPMN choreographies have found widespread adoption for representing inter-organizational interactions, deployed and executed as smart contracts. Choreographies permit indeed to model only interactions between participants, without exposing internal details.

Despite blockchain supporting the implementation of choreographies, flexibility remains a major challenge. Indeed, a choreography may need to be adapted at runtime to react to unplanned situations, thus updating its implementation. However, despite changes to choreography that can be applied, the inherited immutability of blockchain hinders smart contract code updates. Furthermore, structured patterns or off-chain techniques lead to additional costs, increased complexity, or security limitations requiring novel approaches. In this direction, the introduction of the Algorand blockchain and its characteristics opens up novel research directions. Among the others, Algorand offers native features supporting the update of smart contracts and related storage, without the need for complex or gas-consuming patterns.

For this reason in this work we leverage Algorand to support flexible execution of BPMN choreographies. Specifically, we show how smart contracts encode choregraphy logic, and how they can be updated at runtime. To this purpose, we also consider different adaptation operations that can be done on a choreography and consequently reflected in the related smart contract.

As future work, we plan to implement more structured policies enabling fine-grained access control on update operations. Furthermore, we will systematically evaluate the proposed Algorand implementation in comparison with Ethereum-based upgrade patterns (e.g., proxy, diamond) by analyzing how different choreography scenarios behave across various implementations. Finally, we aim to develop an application with a user-friendly graphical interface that supports the shown phases, from the modeling of the choreography to the automatic generation of smart contracts.

Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU and partially funded by Ministero dell'Università e della Ricerca (MUR), issue D. M. 351/2022 "Borse di Dottorato" - Dottorato di Ricerca di Interesse Nazionale in "Blockchain & Distributed Ledger Technology", under the National Recovery and Resilience Plan.

Declaration on Generative AI

ChatGPT was used to assist with grammar, spelling, and phrasing. The authors reviewed and approved all content and take full responsibility for the final manuscript.

References

[1] J. Mendling, I. Weber, W. V. D. Aalst, J. V. Brocke, C. Cabanillas, F. Daniel, S. Debois, C. D. Ciccio, M. Dumas, S. Dustdar, et al., Blockchains for business process management-challenges and opportunities, ACM Transactions on Management Information Systems (TMIS) 9 (2018) 1–16.

[2] C. D. Ciccio, G. Meroni, P. Plebani, Business process monitoring on blockchains: Potentials and challenges, in: Enterprise, Business-Process and Information Systems Modeling, volume 387 of *LNBIP*, Springer, 2020, pp. 36–51.

[3] C. D. Ciccio, G. Meroni, P. Plebani, On the adoption of blockchain for business process monitoring, Software and Systems Modeling 21 (2022) 915–937. doi:10.1007/S10270-021-00959-X.

[4] F. Stiehle, I. Weber, Blockchain for business process enactment: a taxonomy and systematic literature review, in: International Conference on Business Process Management, Springer, 2022, pp. 5–20.

[5] S. Curty, F. Härer, H.-G. Fill, Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: a structured literature review, Software and Systems Modeling 22 (2023) 1857–1895.

[6] OMG, Business Process Model and Notation (BPMN V 2.0), 2011.

[7] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, et al., Chorchain: A model-driven framework for choreography-based systems using blockchain., in: ITBPM@ BPM, 2021, pp. 26–32.

[8] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, A choreography-driven approach for blockchain-based iot applications, in: 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops), IEEE, 2022, pp. 255–260.

[9] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, Flexible execution of multi-party business processes on blockchain, in: WETSEB, 2022, pp. 25–32.

[10] M. Reichert, B. Weber, Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies, volume 54, Springer, 2012.

[11] M. Zhang, P. Shukla, W. Zhang, Z. Zhang, P. Agrawal, Z. Lin, X. Zhang, X. Zhang, An Empirical Study of Proxy Smart Contracts at the Ethereum Ecosystem Scale , in: ICSE, IEEE Computer Society, 2025, pp. 620–620.

[12] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, A flexible approach to multi-party business process execution on blockchain, Future Gener. Comput. Syst. 147 (2023) 219–234.

[13] S. Malik, H. D. Bandara, N. R. van Beest, X. Xu, Smart contracts' upgradability for flexible business processes, in: BPM Blockchain, RPA, CEE, Educators and Industry Forum, Springer, 2024, pp. 55–70.

[14] J. Chen, S. Micali, Algorand: A secure and efficient distributed ledger, Theoretical Computer Science 777 (2019) 155–183.

[15] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, B. Xu, Smart contract development: Challenges and opportunities, IEEE transactions on software engineering 47 (2019) 2084–2106.

[16] Algorand, smart contract update, Algorand Developer, 2019.

[17] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich, Algorand: Scaling byzantine agreements for cryptocurrencies, in: Proceedings of the 26th symposium on operating systems principles, 2017, pp. 51–68.

[18] N. A. Malla, A. Marcelletti, A. Morichetta, F. Tiezzi, A blockchain-based, semantics-driven, modular implementation of bpmn choreographies, in: Society 5.0, CCIS, Springer, 2025. To appear. Available at https://github.com/nawaz-malla/Flexible-Business-Processes.

[19] N. Malla, Github repository 'Flexible-Business-Processes', https://github.com/nawaz-malla/Flexible-Business-Processes, 2025.

[20] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, Engineering trustable and auditable choreography-based systems using blockchain, ACM Trans. Manag. Inf. Syst. 13 (2022) 31:1–31:53.

[21] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, A flexible approach to multi-party business process execution on blockchain, Future Gener. Comput. Syst. 147 (2023) 219–234.

[22] Jan Mendling et al., Blockchains for business process management - challenges and opportunities, ACM Trans. Manag. Inf. Syst. 9 (2018) 4:1–4:16.

[23] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, A. Ponomarev, Caterpillar: A business process execution engine on the ethereum blockchain, Softw. Pract. Exp. 49 (2019) 1162–1193.

[24] A. B. Tran, Q. Lu, I. Weber, Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management, in: BPM Demo and Industrial Track, volume 2196 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 56–60.

[25] E. Marangone, M. Spina, C. D. Ciccio, I. Weber, CAKE: sharing slices of confidential data on blockchain, in: CAiSE Forum, volume 520 of *LNBIP*, Springer, 2024, pp. 138–147.

[26] E. Marangone, C. Di Ciccio, D. Friolo, E. N. Nemmi, D. Venturi, I. Weber, MARTSIA: enabling data confidentiality for blockchain-based process execution, in: EDOC, volume 14367 of *LNCS*, Springer, 2023, pp. 58–76.

[27] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, E. Scala, F. Tiezzi, Model-driven engineering for multi-party business processes on multiple blockchains, Blockchain: Research and Applications 2 (2021) 100018.

[28] E. Marangone, C. Di Ciccio, D. Friolo, E. N. Nemmi, D. Venturi, I. Weber, Enabling data confidentiality with public blockchains, 2023. URL: https://arxiv.org/abs/2308.03791. arXiv:2308.03791.

[29] O. López-Pintado, M. Dumas, L. García-Bañuelos, I. Weber, Controlled flexibility in blockchain-based collaborative business processes, Information Systems 104 (2022) 101622.

[30] V. Buterin, et al., A next-generation smart contract and decentralized application platform, white paper 3 (2014) 2–1.

[31] X. Li, J. Yang, J. Chen, Y. Tang, X. Gao, Characterizing ethereum upgradable smart contracts and their security implications, in: Proceedings of the ACM Web Conference 2024, 2024, pp. 1847–1858.

[32] M. Adams, S. Suriadi, A. Kumar, A. H. M. ter Hofstede, Flexible integration of blockchain with business process automation: A federated architecture, in: CAiSE Forum, volume 386 of *LNBIP*, Springer, 2020, pp. 1–13.

[33] P. Klinger, L. Nguyen, F. Bodendorf, Upgradeability concept for collaborative blockchain-based business process execution framework, in: ICBC, volume 12404 of *LNCS*, Springer, 2020, pp. 127–141.

[34] Y. Xu, T. Slaats, B. Düdder, S. Debois, H. Wu, Distributed and adversarial resistant workflow execution on the algorand blockchain, in: FC, volume 13412 of *LNCS*, Springer, 2022, pp. 583–597.

[35] F. Stiehle, I. Weber, The cost of executing business processes on next-generation blockchains: The case of algorand, in: BPM: Blockchain, Robotic Process Automation, Central and Eastern European, Educators and Industry Forum, Springer, 2024, pp. 89–105.

[36] J. Chen, S. Micali, Algorand, arXiv preprint arXiv:1607.01341 (2016).