Gas Management Patterns in Blockchain-enabled Process Execution

Hassan Atwi, Cesare Pautasso

Software Institute, Università della Svizzera italiana, Lugano, Switzerland
Hassan.Atwi@usi.ch, c.pautasso@ieee.org

Abstract. In this paper, we present a methodology for modeling efficient gas management in blockchain-enabled collaborative business processes. We introduce four patterns designed to optimize and manage gas fees throughout business process execution. These patterns are formally represented using BPMN to provide a clear visual framework for integration into business workflows. Furthermore, we translate these patterns into Solidity smart contracts and evaluate their performance within a real-world business scenario. Our approach aims to enhance the efficiency and cost-effectiveness of blockchain-based process execution.

Keywords: Blockchain · Gas · Solidity · Collaborative Processes · BPMN.

1 Introduction

Blockchain technology [11] has gained attention across many use cases due to key characteristics such as transparency, immutability, and tamper-resistance. One area where blockchain has shown strong potential is Business Process Management [1] (BPM). Smart contracts [2] allow collaborative business processes to be encoded as deployable programs that enforce process logic directly on the blockchain. This enables transparent and immutable execution of collaborative processes, helping ensure trust and compliance among all participants.

As a result, blockchain-based workflow engines [6,9] compile business processes into smart contracts to execute them on-chain. A key concern in writing smart contracts is their execution cost. Unlike traditional workflow engines that run on local infrastructure, blockchain-based execution requires the payment of gas fees, which are determined by code complexity. This makes process execution potentially expensive [12]. Therefore, making gas usage visible at the process modeling level can help reduce business operation costs for blockchain execution.

In this paper, we propose four design patterns aimed at managing and optimizing gas consumption in blockchain-based processes: event logging, guard checking, partial recovery, and gas sponsorship. These patterns (Fig. 1) are intended for both business practitioners and blockchain developers designing processes for on-chain execution. They are BPMN specific patterns and can be applied across various business scenarios in process modeling.

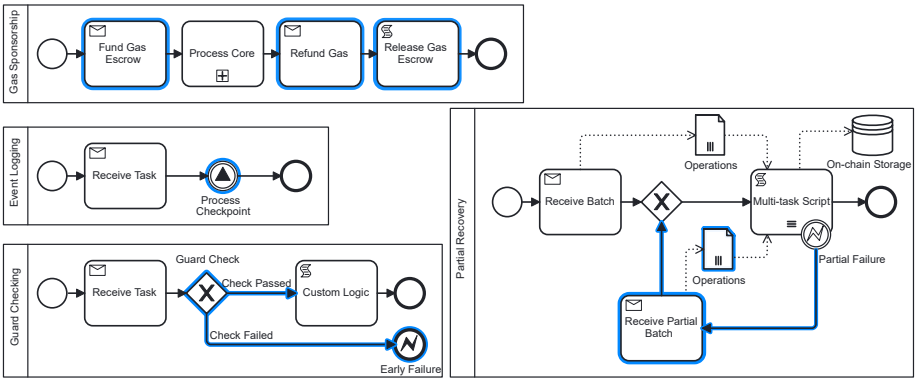


Fig. 1: Gas management design patterns modeled in BPMN (highlighted in blue).

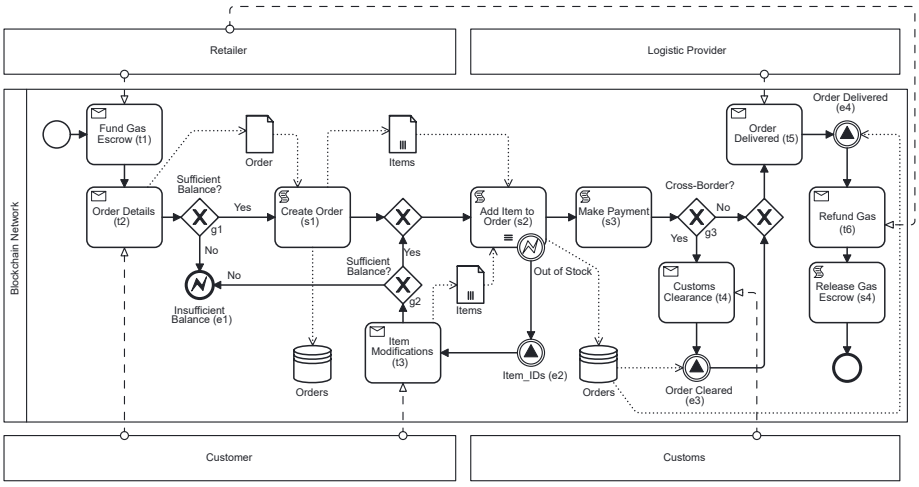


Fig. 2: BPMN diagram representing a supply chain collaborative process in which the four gas management patterns have been introduced.

To demonstrate these patterns, we apply them to a collaborative process scenario involving four participants (Fig. 2): Retailer, Logistics Provider, Customer, and Customs. These participants interact through a blockchain network that provides a shared view of the core supply chain process. In the scenario, the Customer places an order with multiple items. The order is validated based on stock availability and the Customer's balance. If the order is approved and it involves a cross-border transaction, Customs issues a clearance on-chain. The Logistics Provider then logs the delivery on the blockchain. The process is compiled into a Solidity smart contract and deploy on a local Ethereum network. We execute 500 traces of the process to evaluate the impact of each design pattern on both deployment and execution gas costs. Our results show which pattern combinations improve gas efficiency, while others may introduce additional costs.

The paper is organized as follows: In Section 2, we formalize the semantics of a blockchain-enabled collaborative process using BPMN so that its gas consumption can be estimated. In Section 3, we introduce the design patterns illustrating their application within the use case scenario. Section 4 presents an experimental evaluation of their benefits in terms of gas usage during deployment and execution. In Section 5, we discuss the results before concluding the paper in Section 6.

2 Background and Motivation

2.1 Blockchain-Enabled Business Process

To explain process execution costs, we introduce a set of assumptions on how various blockchain-related aspects are mapped to BPMN elements. In particular, we designate one pool to represent the on-chain logic. All other pools represent off-chain business processes, locally executed by each participant. Formally, the blockchain pool B is defined as:

$$B = (T, G, E, P, F, \iota)$$

where:

- T is the set of tasks, with $T_R \subseteq T$ as Receive Message Tasks (from off-chain participants), and $T_S \subseteq T$ as Script Tasks (deterministic on-chain logic),
- G, E: sets of gateways and events, respectively, where $E_R \subseteq E$ is the set of error events
- P: set of participants
- $F \subseteq (T \cup G \cup E) \times (T \cup G \cup E)$: control-flow edges (sequence flows),
- $\iota : T_R \to P$: maps each receive task to its initiating participant,

A blockchain transaction begins with a receive task and terminates at either another receive task or an end event (end event may be an error event). Formally, it is defined as a tuple:

$$\tau = (t_{\text{start}}, \sigma, \pi, \sigma', n_{\text{end}})$$

where:

- $t_{\text{start}} \in T_R$ is the initiating receive task,
- $n_{\text{end}} \in T_R \cup E_R$ is the terminating node, either a receive task T_R (success) or an error event E_R (failure),
- $\pi = \langle n_1, \ldots, n_k \rangle \in (T \cup G \cup E)^*$ is a valid transaction control-flow path such that:
$$(t_{\text{start}}, n_1),\ (n_1, n_2),\ \ldots,\ (n_{k-1}, n_k),\ (n_k, n_{\text{end}}) \in F$$
- $\sigma, \sigma' \in \Sigma$ are the initial and resulting blockchain states,
- Executing π transitions the state from σ to σ', denoted $(\sigma, \pi, \sigma') \in \theta$, where $\theta \subseteq \Sigma \times (T \cup G \cup E)^* \times \Sigma$ is the state transition relation,
- If $n_{\text{end}} \in E_R$, the transaction is reverted: the blockchain state remains unchanged ($\sigma' = \sigma$), and the control flow is rolled back to the initiating receive task t_{start}.

2.2 Gas Cost in Transaction Execution

In blockchain networks, the execution of logic comes with a cost because of the distributed nature of the network. For instance, in Ethereum blockchain networks, to prevent abuse through computationally intensive or potentially unbounded code execution, a fee called gas [16] is applied to all operations performed on-chain. Each transaction is assigned a gas limit and a gas price. The total fee is determined by multiplying the gas consumed during execution by the specified gas price. This fee is paid in ETH and deducted from the transaction sender's balance. If the transaction exceeds the specified gas limit, it is reverted, but the gas consumed up to the point of failure is still paid.

To map gas fees to business processes, we relate them to the BPMN elements executed during a blockchain transaction. Since each element corresponds to an on-chain operation, the total gas cost can be represented as the sum of the individual costs of these operations. Formally, the gas fee can be expressed as:

$$G_{\text{tx}}(\pi) = \sum_{i=1}^{|\pi|-1} G(n_i)$$

where $\pi = \langle t_{start}, \ldots, n_i, \ldots, n_{end} \rangle$ is a valid transaction execution control-flow path. $G(n_i)$ denotes the gas cost of executing each BPMN element n_i along the path. The total gas cost $G_{\text{tx}}(\pi)$ excludes the terminating node n_{end} and represents the sum of gas consumed by all preceding elements in the transaction. This total is paid by the sender of the transaction upon execution of the control flow path triggered by its receipt on the blockchain.

2.3 Related Work and Research Gap

In [10], the BPMN and CMMN notations are used to represent patterns covering blockchain concepts such as oracles, tokenization, and on-chain encryption. The study does not include any gas cost evaluation for the patterns. The pattern collections presented in [18,17] provide a systematic overview of design patterns for blockchain-based applications and serve as a foundation for integrating blockchain components into broader software architectures. In [7], a specific category of blockchain patterns is introduced, focusing on payment mechanisms and transfer of funds between participants. While the primary objective of the study is to explore payment patterns rather than transaction costs, it notes the high deployment cost of the patterns. The study does not use business process modeling to represent the patterns. [14] is a collection of Solidity design patterns and best practices derived from real-world decentralized applications. In our work, we adopt the Guard Check pattern from that collection and represent it using BPMN.

To optimize the cost of executing business processes on blockchain, the authors of [3] transform existing BPMN models into Petri nets and apply optimization rules to enhance execution efficiency on the blockchain. The results are empirically compared to a baseline by replaying execution logs and measuring

gas consumption. This work concentrates on low-level transformations of business processes, rather than addressing application and domain specific concerns at the business level. In [8], the authors explore a range of low-level optimization strategies for Ethereum smart contracts. They evaluate strategies for analyzing and minimizing gas consumption during both the contract generation and deployment phases. However, this work remains focused on the technical level and does not address higher-level business concepts as we do in this paper.

3 Gas Management Design Patterns

3.1 Partial Recovery

Context: In many blockchain applications, a participant may need to perform multiple operations that are submitted together in a single transaction, commonly referred to as batching. In BPMN, this concept is often represented using a multi-instance parallel task, where the same task is executed concurrently for multiple items or data elements.

Problem: While batching reduces the overhead of submitting multiple separate transactions, it also introduces dependencies between the batched operations which are executed atomically. If one operation within the batch fails, the entire transaction is reverted due to the atomic (all-or-nothing) nature of transactions. This forces the user to resubmit the whole transaction after resolving the issue, resulting in additional costs. How to avoid the participant having to resubmit the entire batch to reprocess their operations, leading to increased gas fees?

Solution: Operations within a batch transaction are separated and executed individually rather than atomically as a batch. Successful operations are persisted on-chain, while failed ones are flagged and skipped. A dedicated event is emitted containing references to the failed operations. Participants can then resubmit only the failed operations, i.e., performing a partial recovery of the batch, with corrected parameters, avoiding the need to resend the entire batch.

Use Case: In the use-case scenario (Fig. 2), the partial recovery pattern is applied during the process of adding items to a customer order, where multiple items are processed in a single batch. Assuming the customer has sufficient balance to place the order, the transaction τ_{order} may result in one of two execution paths depending on the availability of the requested items:

$$\pi_1 = \langle t_2, g_1, s_1, \{s_2^{(1)}, \ldots, s_2^{(N)}\}, s_3, g_3, n_f \rangle \quad \text{where } n_f \in \{t_4, t_5\},$$
$$\pi_2 = \langle t_2, g_1, s_1, \{s_2^{(1)}, \ldots, s_2^{(N)}\}, e_2, t_3 \rangle$$

Here, $\{s_2^{(1)}, s_2^{(2)}, \ldots, s_2^{(N)}\}$ represents the repeated execution of task s_2 for each item in the original order. N denotes the total number of items in the customer's order. Task s_2 is treated as a multi-instance parallel task, corresponding to the addition of each item individually.

Path π_1 is followed when all requested items are in stock and the transaction is completed successfully. Alternatively, path π_2 is taken when some or all items

are out of stock. In this case, the transaction emits event e_2, which carries references to the failed items. When the initial order contains out-of-stock items, the customer submits a follow-up transaction $\tau_{\text{order}'}$ to modify the order with the remaining items. This transaction may follow one of two paths:

$$\pi_3 = \langle t_3, g_2, \{s_2^{(1)}, \ldots, s_2^{(M)}\}, s_3, g_3, n_f \rangle \quad \text{where } n_f \in \{t_4, t_5\},$$
$$\pi_4 = \langle t_3, g_2, \{s_2^{(1)}, \ldots, s_2^{(M)}\}, e_2, t_3 \rangle$$

In these paths, $M \leq N$ refers to the number of items that are still pending and need to be added in the follow-up transaction(s). Each instance $s_2^{(i)}$ corresponds to the processing of a single item, whether it succeeds or fails.

The gas consumed by the customer can be calculated as follows: if all items are available in the initial transaction, the total gas usage is $G_{\text{customer}} = G_{\text{tx}}(\pi_1)$. If at least one item is unavailable, the total gas becomes:

$$G_{\text{customer}} = G_{\text{tx}}(\pi_2) + (\delta - 1) \cdot G_{\text{tx}}(\pi_4) + G_{\text{tx}}(\pi_3)$$

where δ represents the number of attempts required to complete the order.

Application: The partial recovery pattern can be implemented using the multicall [4] approach in blockchain systems. In a multicall, multiple user operations are bundled and executed within a single transaction to reduce overall execution fees. The partial recovery pattern complements this mechanism by allowing individual operations within the bundle to fail without reverting the entire transaction. This ensures that successful operations are preserved while isolating and handling failures independently, thereby saving additional cost, as there is no need to resubmit the entire set of operations in case of partial failure.

3.2 Gas Sponsorship

Context: In blockchain-enabled collaborative processes, multiple participants interact through the blockchain to achieve a shared business objective. Since executing tasks on-chain requires initiating blockchain transactions, participants are responsible for covering the gas fees. These fees can be substantial, especially in high-frequency scenarios.

Problem: Blockchain technology offers many advantages for business processes, such as traceability, decentralization, and immutability. However, these benefits come at the cost of transaction fees. Transaction costs can deter participants from engaging in blockchain-enabled processes, especially in environments where costs are prohibitive and unstable. This challenge may outweigh the advantages of blockchain-enabled processes, leading businesses to revert to centralized systems. How to avoid participants paying transaction fees as they interact with a process?

Solution: To address the drawback of transaction costs in blockchain-based business processes, participants in the collaboration agree to designate one or more gas sponsors responsible for covering the transaction fees of other participants. The agreed-upon process begins with an activity in which the sponsors

allocate funds within the smart contract, effectively acting as an escrow. Once the escrow is funded, the process execution continues. At the end of the process, the gas consumption of each participant is measured, and the escrow is released proportionally based on each participant's actual usage. With the help of the business process model, the sponsorship can be made selective based on two factors derived from the process itself. The first is *resource-selective*, identifying the eligible participants who may be refunded. The second is *path-selective*, referring to the specific control-flow path that a transaction has followed (π). Certain paths can be excluded or included in the refunding logic. Both factors can also be combined to enable a process-aware gas sponsorship strategy.

Use Case: In our use-case scenario 2, the participants agreed to adopt the gas sponsorship pattern within the process. The *retailer* was selected as the sponsor responsible for funding the escrow. Additionally, the participants agreed to apply both resource-selective and path-selective strategies. In the resource-selective strategy, only a defined set of participants are eligible for a refund, specifically: $\{customer, logistics\_provider\}$. In the path-selective strategy, the following control-flow paths are excluded from the refunding logic:

$$\begin{aligned}
\pi_1 &= \langle t_2, g_1, s_1, \{s_2^{(1)}, \ldots, s_2^{(M)}\}, e_2, t_3, g_2, e_1 \rangle, \\
\pi_2 &= \langle t_2, g_1, e_1 \rangle, \\
\pi_3 &= \langle t_4, e_3, t_5 \rangle
\end{aligned}$$

Paths π_1 and π_2 are excluded because they correspond to aborted transactions that revert due to insufficient customer balance. In these cases, the retailer explicitly agreed not to sponsor failed transactions. Path π_3 is excluded as it is initiated by the *customs* participant, who is not eligible for a refund under the agreed resource-selective policy. Therefore, the total gas sponsored by the retailer is the sum of the gas fees for all valid transaction paths, excluding the three excluded ones. Let Π be the set of all valid execution paths in the process, and let $\{\pi_1, \pi_2, \pi_3\} \subset \Pi$ be the set of paths excluded from refunding. The total gas cost G_{retailer} sponsored by the retailer is computed as:

$$G_{\text{retailer}} = \sum_{\pi \in \Pi \setminus \{\pi_1, \pi_2, \pi_3\}} G_{\text{tx}}(\pi)$$

Applications: The gas sponsorship pattern has gained attention in recent research aimed at abstracting users from the complexity and cost of interacting with blockchain systems, particularly in the context of Account Abstraction (AA) [15]. AA introduces a flexible transaction model that enables more user-friendly blockchain interactions. One key component of this model is the Paymaster, a smart contract responsible for sponsoring gas fees on behalf of users. This allows users to interact with decentralized applications (dApps) without needing to hold or manage native tokens for gas, thereby improving accessibility.

3.3 Guard Checking

Context: In blockchain-enabled business processes, task execution consumes gas, regardless of its success. Even when a transaction fails and is reverted, the participant still pays for the operations executed up to the failure point. Early input validation is essential to avoid incurring unnecessary gas costs during task execution.

Problem: A blockchain transaction is executed atomically, and while a failed transaction reverts all state changes, participants still incur the gas costs for the execution up to the point of failure. This is both counterproductive and costly for participants. When critical validations, e.g., balance checks, access permissions, are deferred to later stages of the process, earlier tasks may have already performed multiple expensive, state-changing operations. Even though the transaction is eventually reverted, the participant still incurs gas costs for computation and memory usage up to the point of failure. How to reduce the cost of failed transactions?

Solution: To avoid late failure in a blockchain process, early validation of critical conditions is applied in a process control-flow to ensure that faulty transactions are halted as early as possible, before any expensive state-changing logic is triggered. This pattern make use of decision gateways at the beginning of the execution path to validate preconditions. If the guard fails, the process reverts immediately with minimal gas consumption.

Use Case: In the process shown in Figure 2, the transaction for placing an order follows the path:

$$\pi = \langle t_2, g_1, s_1, \{s_2^{(1)}, \ldots, s_2^{(N)}\}, s_3, g_3, n_f \rangle \quad \text{where } n_f \in \{t_4, t_5\}$$

Let $\tau = (t_2, s, \pi, s', n_f)$ denote the transaction. If the customer's balance is insufficient, the control flow is redirected to a terminating event via the gateway g_1. However, if the balance check is deferred until s_3, the transaction proceeds through s_1 and s_2 before failing. These tasks incur significant gas costs due to their storage-intensive operations, i.e., order creation and item insertion, making early validation crucial for cost efficiency.

– Without early guard: If balance validation is deferred until s_3, the gas cost upon failure becomes:

$$G_{\text{fail-late}} = G(t_2) + G(s_1) + \sum_{i=1}^{N} G(s_2^{(i)}) + G(s_3)$$

– With early guard: Placing the check at g_1 ensures early termination if the balance is insufficient:

$$G_{\text{fail-early}} = G(t_2) + G(g_1)$$

The gas savings in case of failure are:

$$\Delta G = G_{\text{fail-late}} - G_{\text{fail-early}} = G(s_1) + \sum_{i=1}^{N} G(s_2^{(i)}) + G(s_3) - G(g_1)$$

This shows how guard checking reduces costs for failing transactions.

Applications: The guard checking pattern is widely adopted in many decentralized application (dApp) smart contract codebases. It typically appears at the beginning of function calls to enforce preconditions. Solidity offers a dedicated `"require()"` statement. In Uniswap [5] early checks are used to validate input conditions, e.g., ensuring sufficient output amounts or valid recipient addresses, before proceeding with more costly operations.

3.4 Event Logging

Context: Blockchain storage writes incur high gas costs, while event logging is cheaper but still suited to track the achievement of key process milestones.

Problem: Participants need visibility into the state of a blockchain-enabled process for monitoring, auditing, or coordination purposes. However, frequently persisting state information on-chain through storage operations is costly in blockchain environments, leading to increased gas consumption. How to reduce the cost for on-chain logging of the process instance state?

Solution: To reduce the cost associated with frequent state modifications in blockchain-enabled processes, we use blockchain events, which are modeled as signal events in BPMN. These events are emitted at specific checkpoints within the process and serve as lightweight markers that communicate progress without altering the contract's state. This approach offers a gas-efficient alternative for broadcasting and auditing the status of a business process on-chain.

Use Case: In our use-case scenario (Fig. 2), the transactions $\tau_{\text{customs\_clearance}}$ and $\tau_{\text{order\_delivered}}$, corresponding to the execution paths $\pi_1 = \langle t_4, e_3, t_5 \rangle$ and $\pi_2 = \langle t_5, e_4, t_6 \rangle$ respectively, demonstrate the application of gas-efficient checkpoint logging. Upon successful customs clearance or order delivery, a signal event is emitted to indicate the occurrence of a checkpoint within the process.

Since this status information, i.e., the order has been cleared or delivered, is not consumed by any on-chain logic or referenced by other smart contracts, persisting it via storage updates (e.g., `order.cleared = true`) incurs unnecessary gas costs. Instead, emitting a signal event offers a lightweight and cost-effective alternative for communicating progress to off-chain participants. Persisting such data on-chain when it serves no purpose in subsequent smart contract logic represents an anti-pattern. Gas-efficient checkpoint logging avoids redundant state changes while still sustaining transparency and traceability of process execution for external observers through the event log.

Applications: Event logging is a widely adopted pattern in decentralized applications (dApps) to minimize state-changing operations and reduce gas costs. Instead of persisting non-critical state updates on-chain, many dApps emit events to signal significant process milestones. A practical example is The Graph [13], a protocol that leverages blockchain-emitted events to index and query contract data efficiently. This approach enables transparency and auditability without incurring the high gas costs associated with on-chain storage operations.

4 Experimental Evaluation

We used the running example to quantitatively evaluate the patterns in terms of gas usage during both deployment and execution. To enable this evaluation, the collaborative process was compiled into a smart contract based on the state machine pattern [14]. The process state is represented by a token that marks the currently active task. The smart contract maintains a mapping from tasks to their corresponding state, i.e., process token. When a task becomes enabled, a token is placed on it. Upon the execution of that task, the token moves to the subsequent task according to the process flow. Public functions represent user-invoked tasks, while internal functions capture the behavior of script tasks, gateways, and events, i.e., callable only from within the contract itself.

During the evaluation, we distinguish between the base contract and the full contract. The base contract implements only the core process logic, i.e., the state machine, without incorporating any of the patterns. For instance, in the absence of the event logging pattern, the base contract persists order status using on-chain storage instead of emitting events. Guard checking is also omitted: control-flow elements such as gateways g_1 and g_2 are not present in the base model. Partial recovery is likewise not supported in the base contract. As a result, when some items are unavailable, the customer must resubmit a transaction with the entire item list. This leads to the execution path such as $\pi_{\text{full}} = \langle t_2, s_1, s_2, e_2 \rangle$, with $e_2 \in E_R$ being an error end event instead of a signal event. Lastly, the gas sponsorship pattern is not implemented in the base contract. Tasks responsible for funding and refunding participants, e.g., t_1 and t_6, are omitted. In contrast, the full contract includes all four patterns as specified in the process model.

4.1 Deployment Cost

The patterns discussed are designed to manage gas usage in a blockchain-enabled business process. In addition to their execution cost, the deployment cost is equally important. It is essential to evaluate the deployment cost to determine whether implementing these patterns is affordable, particularly in relation to their execution cost and overall profitability. We isolate the pattern from the base contract to inspect the deployment cost of each pattern. We then deploy the base contract along with the individual pattern and compare its deployment cost to that of the base contract. Understanding how much additional gas each pattern requires during deployment, relative to the base contract, is essential for evaluating the trade-offs involved. Table 1 presents the deployment costs for each pattern. The most expensive pattern to implement is the gas sponsorship pattern, with a deployment cost of 1,130,273 gas (26.22%), which incurs higher costs due to the continuous tracking of gas consumption in real-time across the contract's functions and the storage of gas usage by participants. The least expensive pattern to implement is event logging, with a deployment cost of 29,182 gas (0.68%), which is attributed to its simple implementation that involves emitting a native Solidity event when triggered within the control flow. In total, the

Table 1: Breakdown of full contract deployment cost.

Component	Deployment Cost (gas)	Deployment Cost Share (%)
Base Contract	2,692,867	62.46%
Patterns		
Event Logging	29,182	0.68%
Guard Checking	176,309	4.09%
Partial Recovery	282,067	6.54%
Gas Sponsorship	1,130,273	26.22%
Full Contract	4,310,698	100%

patterns represent 35.53% of the deployment cost of the base contract, which has a deployment cost of 2,692,867 gas (62.46%).

4.2 Execution Cost

To assess the execution cost of the patterns, we need to run the collaborative process by interacting with the deployed smart contracts. This interaction is simulated through a generated event log, consisting of 500 traces, where each trace represents an execution instance of the process. Each trace includes input parameters required to execute the process, such as the retailer's stock levels and item prices, the initial gas amount funded by the retailer, and the customer's Ether balance, which determines whether the purchase can be completed. The number of attempts (δ) made by the customer to successfully fulfill the order is generated as part of each trace, along with whether the order is domestic or cross-border. These parameters are randomly assigned across the traces to simulate different execution scenarios. The same event log is used to simulate each process trace on both the full and base contracts, with the execution path determined by the corresponding input parameters.

To analyze the gas cost impact of each pattern, we deploy contract variants that combine the base process with a single pattern. This setup enables us to isolate the cost contribution of each pattern and understand how it affects execution independently. By comparing these variants with the base contract, we observe the relative increase or decrease in total gas usage. In Table 2, the partial recovery pattern results in the highest gas reduction of 49.64% compared to the base contract, despite introducing the largest number of transactions. Conversely, the Refund Gas pattern results in a 13.36% increase in total gas usage. This is expected, as the pattern's goal is not cost optimization but refund management. It also introduces more transactions than the base contract, due to additional operations for funding and refunding participants. When all patterns are integrated in the full contract, we observe a 40.64% reduction in total gas consumption compared to the base contract, despite an increased number of transactions. This indicates that the combined use of patterns contributes to overall cost efficiency in the process execution.

Table 2: Transaction count, average cost, total gas used, and relative difference with respect to the Base Contract.

Component	Transactions		Total Gas Used	Relative Difference (%)
	Count	Avg. Cost (gas)		
Base Contract	3,475	272,667.57	947,519,800	–
Patterns				
Event Logging	3,475	256,525.79	891,427,135	-5.91%
Guard Check	3,475	268,767.43	933,966,831	-1.43%
Partial Recovery	4,594	103,902.50	477,328,105	-49.64%
Gas Sponsorhip	4,318	248,719.16	1,073,969,351	+13.36%
Full Contract	4,318	130,314.79	562,699,263	-40.64%

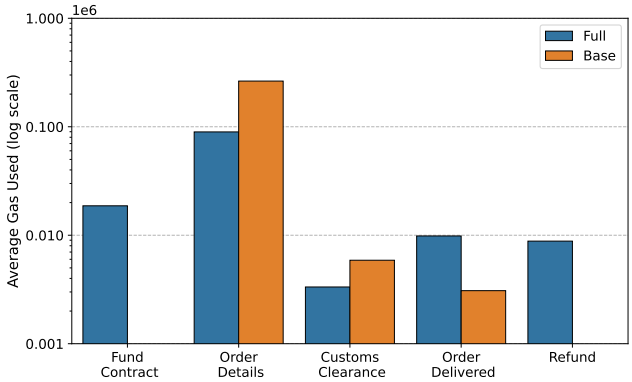
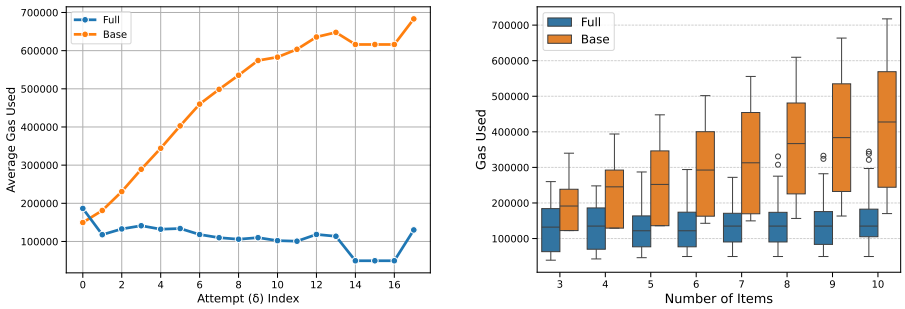


Fig. 3: Comparison of average gas usage by task in full vs base contract.

Looking at the impact of the patterns from a task execution perspective (Fig. 3), we can observe how each pattern affects different parts of the process in terms of gas usage. For instance, in the Order Details task, the gas consumption in the full contract is approximately 66% lower than that of the base contract. This reduction results from the combined effect of the partial recovery and Guard Check patterns integrated within the task's logic. It is also worth noting the gas usage in the Customs Clearance and Order Delivered tasks. Although both tasks incorporate the event logging pattern in the full contract, Order Delivered still consumes more gas than Customs Clearance. This discrepancy is due to the additional logic introduced by the gas sponsorship pattern in Order Delivered, i.e., tracking and storing gas usage on-chain. This observation highlights that certain pattern combinations may not be cost-effective. In this case, the gas sponsorship pattern introduces overhead that offsets the efficiency gains of event logging, nullifying its intended benefit.

Figure 4 illustrates gas usage trends in relation to the number of customer attempts. In the base contract, gas usage increases sharply as the number of



(a) Average gas usage by attempts. (b) Gas usage by number of items.

Fig. 4: Gas usage scaling across attempts (δ) and item counts.

Table 3: Distribution of funds among participants with their respective gas usage.

Participant	**Funded (ETH)**	**Gas Used (ETH)**	**Refund (ETH)**	**Refund (%)**
Retailer	295	0.11406309	294.72367810	99.91%
Customer	0	0.34435188	0.25267738	73.36%
Logistics	0	0.03658384	0.02364452	64.63%
Customs	0	0.01223939	0	0.00%

attempts grows ($G_{\text{tx}}(\pi_{\text{full}}) \cdot (\delta)$) (Fig. 4a). This behavior is due to the lack of partial recovery pattern. Each failed attempt requires resubmitting the entire order, resulting in redundant execution and higher costs. In contrast, the full contract shows a decreasing trend in gas usage across attempts. Thanks to partial recovery pattern, once certain items are successfully added, they are excluded from future attempts. Thus, only the failed items are retried ($M \leq N$), leading to reduced computational effort and lower cumulative gas costs. Similarly, as the number of items in an order increases (Fig. 4b), the base contract shows a steep rise in gas consumption. Since the full batch is reprocessed with every failure, the cost of each additional item compounds. The full contract maintains a relatively stable profile, as it isolates and reattempts only the necessary operations.

To demonstrate the utility of the gas sponsorship pattern, Table 3 presents the distribution of funds among participants. In this setup, the retailer acts as the sole sponsor, covering the gas costs of the entire process. Due to the adoption of a resource-selective refunding strategy, Customs is excluded from gas reimbursement and therefore receives no refund. Other participants, i.e., the Customer and Logistics, receive only partial refunds. This shortfall arises from certain gas-consuming operations that are not encompassed within the defined sponsorship boundaries, i.e., path-selective. Notably, Logistics consistently exhibits the lowest refund rate. As discussed in Fig. 3, this is attributed to the fact that the gas cost of the logistics task itself is lower than the overhead required to track and process gas usage on-chain, resulting in a refund gap for the participant.

5 Discussion

As observed in the evaluation, adding patterns to the base contract nearly doubles the deployment cost. This increase is expected due to the additional logic introduced by the patterns. However, this higher deployment cost represents an investment that will return significant gas savings during execution.

In some cases, combining patterns leads to a reduction in both execution costs and the number of transactions. Certain patterns may appear costly when applied in isolation, but become more efficient in a combined contract. This is due to the way patterns interact and complement each other, often reducing redundant operations. For example, the Partial Recovery pattern recorded fewer transactions in the full contract compared to its isolated implementation. This is largely due to the presence of a Guard Check, which filters out invalid transactions, e.g., due to low balance, early in the process. Another observation involves the gas sponsorship pattern, which shows significantly higher execution costs when applied alone (almost double) compared to when it is used in combination with other patterns. This is because it misses out on the optimizations introduced by the other patterns.

Just as some patterns work well together, others may have a negative impact when combined. For instance, gas sponsorship can increase the cost of otherwise lightweight patterns such as Event Logging. This overhead stems from the additional write operations needed to track gas usage. One way to address this is to introduce a fixed gas refund for such lightweight tasks when user refunds are necessary, thus avoiding excessive overhead for simple operations.

The presented patterns can act as a guide for process designers aiming to implement blockchain-based workflows while managing gas costs. By applying these patterns, designers can reduce gas usage and make blockchain adoption more attractive to participants who are concerned about transaction fees. Although the patterns are demonstrated within our own smart contract implementation, they should also be tested with other blockchain-enabled workflow engines to assess their general applicability and performance across different environments.

6 Conclusion

In this paper, we presented four BPMN-based gas management patterns for blockchain-enabled processes. These patterns were demonstrated through a use case scenario and evaluated under different settings to assess their effectiveness in reducing gas consumption. The results showed that, while the patterns introduce an increase in deployment cost, they offer significant gas savings during execution. These patterns are BPMN specific and can be applied across various business scenarios involving blockchain-based processes. They may serve as practical guidelines for designing gas-aware and efficient blockchain workflows.

Acknowledgement This work is supported by the Swiss National Science Foundation (SNSF) funded project "Flexible Choreographies in Multi-chain Environments" (196958).

References

1. Belchior, R., Guerreiro, S., Vasconcelos, A., Correia, M.: A survey on business process view integration: past, present and future applications to blockchain. Business Process Management Journal **28**(3), 713–739 (2022)
2. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper **3**(37), 2–1 (2014)
3. García-Bañuelos, L., Ponomarev, A., Dumas, M., Weber, I.: Optimized execution of business processes on blockchain. In: Proc. 15th International Conference on Business Process Management (BPM). pp. 130–146. Springer (2017)
4. Hughes, W., Russo, A., Schneider, G.: Multicall: A transaction-batching interpreter for ethereum. In: Proc. 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure. pp. 25–35 (2021)
5. Lo, Y.C., Medda, F.: Uniswap and the emergence of the decentralized exchange. Journal of financial market infrastructures **10**(2), 1–25 (2021)
6. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: Caterpillar: a business process execution engine on the ethereum blockchain. Software: Practice and Experience **49**(7), 1162–1193 (2019)
7. Lu, Q., Xu, X., Bandara, H.D., Chen, S., Zhu, L.: Patterns for blockchain-based payment applications. In: Proc. 26th European Conference on Pattern Languages of Programs. pp. 1–17 (2021)
8. Mandarino, V., Pappalardo, G., Tramontana, E.: Some blockchain design patterns for overcoming immutability, chain-boundedness, and gas fees. In: 2022 3rd Asia Conference on Computers and Communications (ACCC). pp. 65–71. IEEE (2022)
9. Mendling, J., et al.: Blockchains for business process management-challenges and opportunities. ACM Transactions on Management Information Systems (TMIS) **9**(1), 1–16 (2018)
10. Milani, F., Garcia-Banuelos, L., Filipova, S., Markovska, M.: Modelling blockchain-based business processes: a comparative analysis of bpmn vs cmmn. Business Process Management Journal **27**(2), 638–657 (2021)
11. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
12. Porkodi, S., Kesavaraja, D.: Escalating gas cost optimization in smart contract. Wireless Personal Communications **136**(1), 35–59 (2024)
13. Tal, Y., Ramirez, B., Pohlmann, J.: The graph: A decentralized query protocol for blockchains (2018), `https://raw.githubusercontent.com/graphprotocol/research/master/papers/whitepaper/the-graph-whitepaper.pdf`
14. Volland, F.: Solidity design patterns, `https://fravoll.github.io/solidity-patterns/`, accessed: 2025-05-21
15. Wang, Q., Chen, S.: Account abstraction, analysed. In: 2023 IEEE International Conference on Blockchain (Blockchain). pp. 323–331. IEEE (2023)
16. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)
17. Xu, X., Pautasso, C., Lo, S.K., Zhu, L., Lu, Q., Weber, I.: An extended pattern collection for blockchain-based applications. In: Transactions on Pattern Languages of Programming V, pp. 67–117. Springer (2025)
18. Xu, X., Pautasso, C., Zhu, L., Lu, Q., Weber, I.: A pattern collection for blockchain-based applications. In: Proc. 23rd European Conference on Pattern Languages of Programs. pp. 1–20 (2018)