Introducing Event-Driven Properties Management in *-chain

Stefano Bistarelli[1], Ivan Mercanti[1], Paolo Mori[2] and Carlo Taticchi[1,*]

[1]*Università degli Studi di Perugia - Dipartimento di Matematica e Informatica, Italy*
[2]*Consiglio Nazionale delle Ricerche - Istituto di Informatica e Telematica, Pisa, Italy*

Abstract

Blockchain technology has improved supply chain management by increasing transparency, traceability and automation. This paper introduces an updated version of *-chain, an automated framework that enables the creation of supply chain management systems from high-level designs through a user-friendly web interface. The proposed improvements integrate events into Ethereum smart contracts to enable asset property notarisation throughout the supply chain, without modifying on-chain states. This approach optimises efficiency and reduces costs while preserving decentralisation and data immutability.

Keywords

Ethereum Events, Supply Chain Management, Domain Specific Graphical Language

1. Introduction

The globalisation of production and distribution networks has made supply chains increasingly complex, involving multiple stakeholders across different regions. Among the challenges for suppliers and end customers is the issue of counterfeiting. Counterfeit products can undermine the trust and integrity of brands, as they often imitate the quality and branding of authentic goods. This not only poses risks to consumers who may unknowingly purchase inferior products but also creates significant complications for businesses that face potential revenue losses. For instance, counterfeit wine or olive oil, produced under fraudulent claims of origin, can harm consumers by providing substandard products and also undermine the reputation of legitimate producers. According to the European Union Intellectual Property Office [1], counterfeiting costs the European economy billions annually, especially in sectors such as clothing and cosmetics.

In this context, ensuring the authenticity, provenance, and integrity of goods has become a critical challenge. One of the most effective ways to safeguard these aspects is through supply chain traceability, which allows stakeholders to track the movement of goods from origin to final destination. This practice is particularly crucial in industries where product integrity is tied to geographical origin, traditional production methods, and consumer trust. Protected Designations of Origin (PDO) play a vital role in ensuring the traceability of products and their production methods. PDOs are a form of intellectual property protection granted to products that are deeply linked to a specific geographical area and are produced according to traditional methods. In Italy, renowned examples of PDOs include *Parmigiano Reggiano*, *Prosciutto di Parma*, and *Olio Extra Vergine di Oliva Chianti Classico*. These products are carefully regulated and certified by local authorities to ensure they are only produced in their designated regions using specific processes, thus safeguarding their authenticity. Besides protecting against counterfeiting, PDOs also promote local economies, ensuring that the benefits of authentic production stay within the communities that have historically cultivated the skills and knowledge to produce these high-quality goods.

DLT2025: 7th Distributed Ledger Technology Workshop, June, 12-14 2025 - Pizzo, Italy

*Corresponding author.

stefano.bistarelli@unipg.it (S. Bistarelli); ivan.mercanti@unipg.it (I. Mercanti); paolo.mori@iit.cnr.it (P. Mori); carlo.taticchi@unipg.it (C. Taticchi)
0000-0001-7411-9678 (S. Bistarelli); 0000-0002-9774-1600 (I. Mercanti); 0000-0002-6618-0388 (P. Mori); 0000-0003-1260-4672 (C. Taticchi)

Traditional supply chain systems for these products often rely on centralised databases, which are vulnerable to manipulation and lack transparency across the entire product lifecycle. These limitations hinder the ability to reliably track products from origin to destination, weakening both consumer trust and regulatory compliance. Blockchain technology offers a promising solution by providing an immutable, distributed ledger that can record product data in a transparent and tamper-evident manner. When combined with smart contracts, it enables automated and verifiable traceability throughout the supply chain. For instance, counterfeiting can be prevented by recording every stage of a product's journey, from production to distribution, on a blockchain that can be consulted by consumers and stakeholders.

In previous work, the *-chain [2] framework was proposed to assist producers, whose expertise is usually outside blockchain technologies, in designing and implementing blockchain-based Supply Chain Management Systems (SCMS) for tracking specific production processes. The *-chain framework includes a Domain-Specific Graphical Language (DSGL) and a set of tools to facilitate SCMS creation. Users can design a graphical representation of a supply chain by combining predefined constructs for operations, assets, containers, and roles. The framework then generates Solidity smart contracts required to implement the SCMS and web interfaces for interacting with the blockchain.

In this paper, we present an enhanced version of *-chain that incorporates the use of events [3] to notarise properties of an asset as it progresses through different stages of a supply chain. The language used for the smart contracts is Solidity, which natively handles the emission of events in response to operations called by users. Events are a Solidity logging mechanism that enables the tracking of specific actions within a contract without using blockchain storage. These events are stored in the blockchain's log, which is separate from the contract's state and can be accessed off-chain by external applications or other contracts. Within *-chain, each time an asset transitions to a new state, an event is emitted that records changes in its related properties. These events are immutable and can be read by stakeholders like producers, consumers and other supply chain participants, to verify the asset's properties at each stage of its life cycle.

An initial motivation for adopting an event-driven approach in our framework is the significant reduction in gas costs. Indeed, while a storage update can cost around 20,000 gas, emitting an event with similar data may only require 2,000–3,000 gas, reducing costs by up to ten times. Events also make it possible to create reactive interfaces where significant changes are notified without additional storage operations. This allows external components with asynchronous workflows, such as payment confirmations or shipment tracking, to react passively to state changes, avoiding constant polling of smart contracts and thus reducing network load. Moreover, since events are immutable and verifiable, they prove to be useful for auditing, debugging and tracking contractual interactions. Popular libraries such as Web3.js and Ethers.js provide native support for event management, further simplifying the integration with decentralised applications.[1] Finally, the use of events promotes a clear separation of concerns: while the smart contract manages the logic and state of the application, events are used exclusively to signal changes.

Therefore, this work is a study to evaluate whether events can be effectively used to manage SCMS, particularly using *-chain as a case study to explore the practical implications of this design choice.

The remainder of this paper is organised as follows. Section 2 provides an overview of the Solidity language, with a focus on events, and briefly describes the DSGL introduced in [4]. Section 3 outlines the use of events within the *-chain framework, illustrating the registration and retrieval of asset properties via the blockchain log. Section 4 highlights the main advantages we have identified for the use of events, also exemplifying practical scenarios through the *-chain web user interfaces. Section 5 reviews significant literature related to our work, and Section 6 concludes the paper with a discussion of potential future directions for this research.

[1]The Web3.js and Ethers.js documentation are respectively available at https://docs.web3js.org and https://docs.ethers.org.

2. Background

In this section, we provide background information on the technologies used in our work, including Solidity, Ethereum events, and the *-chain DSGL.

2.1. Solidity

Smart contracts are programs based on rule sets and deployed on the Ethereum blockchain, which allows functions to be executed if certain conditions are met. Ethereum smart contracts can be written using several programming Turing-equivalent languages, but the most popular is Solidity. These contracts can execute transactions automatically, so there is no need for a third-party entity to take action. Their use will thus ensure that transactions are executed safely, quickly, autonomously and transparently. Solidity is a high-level, statically typed programming language specifically designed for writing smart contracts on Ethereum and Ethereum-compatible blockchains. It includes features such as inheritance, libraries, and complex user-defined types, which make it well suited for creating robust and modular contract logic. Solidity compiles down to the Ethereum Virtual Machine bytecode, allowing developers to define both state variables and executable functions within contracts.

Smart contracts written in Solidity are powerful tools that facilitate automated transactions and interactions on the blockchain. They enable users to execute a series of operations that maintain the blockchain's state and handle persistent data. These contracts can respond to external transactions or function calls, making them highly versatile. Due to the immutable and decentralised nature of the blockchain, it is critical for developers to focus on both efficiency and security when designing smart contracts. Efficiently written contracts minimise the computational resources required to execute operations, which not only improves performance but also reduces costs for users. Given that each operation within a smart contract incurs a gas cost, optimizing code can lead to considerable savings for those interacting with it.

Gas serves as a measure of the computational effort needed to execute the operations on the Ethereum network. Users are charged gas fees based on the complexity and intensity of the resources of the operations they wish to perform. Every transaction initiated and every function call made within a smart contract consumes gas, and users pay for this gas in Ether, the native cryptocurrency of the Ethereum network.

2.2. Events

Events in Solidity act as a communication mechanism between smart contracts and components outside the blockchain. When an event is emitted, it writes data into the blockchain's log, which is not directly accessible by other smart contracts but can be retrieved efficiently by external applications like monitoring tools. These logs, although not part of the contract status, are permanently stored on the chain and cryptographically linked to the transaction that issued them, thus guaranteeing immutability and verifiability.

When declaring an event, a template is defined for the log entries, specifying the parameters that will be included when the event is emitted. Some of these parameters may be marked as indexed, which means that they will be stored as searchable entries separated by arguments to facilitate efficient filtering and querying. During contract execution, when the emit keyword is used to trigger an event, the Ethereum Virtual Machine creates a log entry consisting of the contract address, an array of arguments, and a data load. The first argument is always the hash of the event signature, while the other arguments correspond to the indexed parameter values. Non-indexed parameters are encoded using the Application Binary Interface (ABI) specification and stored in the data field.

Events are particularly useful for signalling state changes, enabling asynchronous communication and reducing computation on the chain by moving logic to external observers. A typical use case of events concerns transaction tracking, real-time updates of user interfaces, and verification. For example, an event could be issued when a load of olives is washed, and users subscribed to this event could react immediately to the change of state, without having to continuously query the blockchain.

2.3. *-chain DSGL

*-chain employs a DSGL specifically designed to allow domain experts, such as producers, to intuitively model and represent any supply chain. This language uses a set of predefined constructs that correspond to the various operations of the production process, each with its own graphical representation. Below, we outline the key elements of the system, i.e. assets, containers, operations and roles, referring the reader to [2] for an in-depth description.

Assets represent the objects involved in the production process described by the supply chain. For example, in olive oil production, a batch of olives is considered an asset. Each asset is associated with a set of properties that reflect the effects of operations on it during production. Two special properties are common to all assets: the *owner*, i.e. the participant in the supply chain who owns the asset, and the *controller*, which is the participant who has physical control of the asset.

Containers are used to store and move *assets*. Uncountable assets must always be placed in containers to be tracked, while *countable* assets can be stored in containers for convenience. When an asset is placed in a container, some of its properties become unchangeable, while others, associated with those of the container itself, may change. For example, the weight of a batch of olives cannot change when they are placed inside a container, but their position changes along with that of the container when it is moved.

The **operations** in *-chain DSGL represent the activities performed on assets during the production process. These operations allow the creation, transformation, tracking, and management of assets within the supply chain. Let us now list some of the various types of operations defined, using the olive oil supply chain to illustrate their functioning.

- `asset_create` is employed to create new assets in the system. In the olive oil process, this operation can be used during the olive harvesting stage to create the olive asset.
- `asset_update`: modifies asset properties such as position or temperature. For example, moving a batch of olives produces an update of their position property.
- `asset_transform` represents significant transformations that create new assets starting from existing ones, such as turning olives into olive paste.
- `asset_destroy` represents the destruction of an asset, such as when the remaining pomace is discarded.

Each operation can be paired with *constraints*, e.g. conditions based on asset properties, to ensure its validity within the system. Additionally, operations can only be executed by participants in the supply chain who play the appropriate role and possess the corresponding rights.

The **role** system regulates the execution of operations by supply chain participants. The idea is that operations can only be performed by authorised participants with specific capabilities certified by their role. *-chain adopts a *role-based access control* (RBAC) [5] model, in which distinct roles are defined and the right to execute an operation is linked to one or more of these roles. When designing a supply chain using, the domain expert first creates a set of roles to assign to supply chain participants upon registration, then associates an authorisation rule with each operation requiring protection.

2.4. *-chain Tools

The *-chain framework comprises several core components that collectively enable the design, implementation, and interaction with blockchain-based SCMSs. The **Graphical Editor** supports supply chain domain experts in modelling their SCMSs by means of the *-chain DSGL. This tool generates two parallel representations of the modelled supply chain: a graphical representation and a JSON-based textual representation.

The **Model Translator** processes the latter representation to produce a skeleton of the SCMS in the Solidity language. This skeleton defines the structure and signatures of the smart contracts corresponding to the elements specified in the supply chain model. The resulting Solidity code can then be refined by a domain expert, who incorporates the application-specific logic required to fulfil the operational needs of the SCMS. The expert is also responsible for deploying the finalised smart contracts onto the designated blockchain platform.

In parallel, the **Model Interface Builder** enables the automatic generation of user interfaces tailored to different actors within the SCMS. These include:

- the *Administrator Interface*, which allows administrators to configure the SCMS by assigning roles and permissions to participating entities;
- the *Participant Interface*, through which users can record their actions on supply chain assets, according to their assigned roles;
- the *Viewer Interface*, which provides visibility over the historical operations performed on a specific asset, as recorded on the blockchain.

3. Event-Driven Optimisation in *-chain

In the smart contracts generated by *-chain, we incorporate events to track state changes of blockchain-based assets without increasing the storage requirements of the contract. For each asset type defined in the supply chain model, the Model Translator produces a dedicated event to emit snapshots of the asset's state whenever an operation is performed. These events capture essential metadata, including the asset's unique identifier, ownership and control addresses, off-chain references (e.g. properties hash) and the current lifecycle state.

Unlike earlier versions of our system [2] that relied on additional on-chain variables, events now provide a more efficient mechanism for recording asset state transitions. The insertion of events is automated at the points in the contract where asset state mutations occur. Specifically, the generator adds an event emission in all *-chain operations discussed in Section 2.3.

Listing 1 shows the code of a generic *-chain event to log the full snapshot of an asset's state. `productId` is the unique ID of a product, which refer to an instance of an asset. `owner` is the address of the asset's owner. `controller` is the current controller responsible for the asset. `hash` is the hash of asset's properties at this specific state, with the full data stored off-chain in a database. `state` is the current state of the asset, represented as an enum.

```
event asset_history(
    uint256 indexed productId,    // Unique identifier for this product
    address owner,                // Permanent owner address
    address controller,           // Entity currently controlling the product
    bytes32 hash,                 // Off-chain data reference to properties
    asset_state state             // Enum: lifecycle state of the asset
);
```

Listing 1: Example of an auto-generated event for asset tracking.

In our framework, we represent the properties of each asset through a single cryptographic hash rather than emitting each property individually within the event. This design choice is motivated by two primary factors: gas efficiency and technical limitations in the Solidity event system. First, storing and emitting multiple properties directly in an event incurs significant gas costs, especially when dealing with complex or high-dimensional data structures. Each additional parameter increases the size of the event log and, consequently, the cost of executing transactions that trigger those events. By aggregating all relevant asset attributes into a single off-chain data structure and computing a hash of its contents, we can drastically reduce the on-chain computation and make the event emission more cost-effective.

Second, Solidity enforces a strict limit on the number of parameters that can be passed to an event, currently capped at 17, with at most 3 of them being indexed.[2] For asset models with many attributes

[2]Solidity ABI specification: https://docs.soliditylang.org.
Solidity compiler implementation: https://github.com/ethereum/solidity.

this limit becomes a bottleneck. Using a hash sidesteps this constraint entirely, allowing us to include an arbitrary number of properties in the off-chain record, while still maintaining verifiability through the on-chain hash. Moreover, this approach guarantees data integrity and tamper detection, as the hash acts as a compact fingerprint of the off-chain asset properties. Any unauthorised modification to the underlying data will result in a hash mismatch, signalling that the record has been altered.

Consequently, trust in the off-chain application is not required for data verification. Since the hash of the asset state is immutably recorded on-chain, any user can independently validate the integrity of the clear-text data retrieved from the database by recomputing the hash and comparing it to the on-chain reference.

However, this mechanism comes with an inherent limitation: the hash is non-reversible by design. If the original off-chain data is lost or corrupted, it cannot be reconstructed from the hash alone. This implies that while the hash ensures that data has not been tampered with, it also makes the system dependent on reliable and persistent off-chain storage for data availability and recovery. Therefore, ensuring secure and redundant storage of asset records is essential to complement the use of hashing in event logging.

The proposed architecture can be further refined to mitigate the risk of a single point of failure, such as through the adoption of distributed storage, including traditional database replication and decentralised technologies such as IPFS.[3] Replication strategies [6] also offer valuable insights for improving data availability and resilience in similar decentralised settings. Moreover, given that the data in question are used to certify the integrity of supply chain processes, the stakeholders involved have a direct incentive to ensure their availability over time, as this directly impacts their ability to prove the authenticity and quality of their products.

The asset contract emits events every time a function that alters the asset's state is invoked. In particular, when an asset is created, an event is triggered to log its initial state. This occurs in the function that initialises the asset, ensuring that all relevant parameters in Listing 1 are recorded as soon as the asset is created in the contract. Capturing this initial state allows for tracking the creation of each asset. Once a specific operation authorised by the contract is successfully executed, for example for processing an asset or performing an irreversible action via a user-triggered function, the contract emits an event to signal its completion. For instance, when an asset changes storage location or undergoes a significant transformation, the emitted event marks the finalisation of that action, recording the performed operation. Finally, when an asset is destroyed or reaches its final state (e.g. after it has been processed and its usefulness in the chain has ended), an event is emitted to record this finalisation. This serves as the closing point of the asset's lifecycle and certifies that the destruction of the asset is completed and cannot be altered.

3.1. Accessing the Event Log

Our system must be capable of deploying decentralised applications that interact with the Ethereum blockchain. To this end, the most widely used tools from an implementation perspective are the Web3.js and Ethers.js libraries.[4] Both libraries provide similar core functionalities: they enable connections to Ethereum providers, interaction with smart contracts via ABI definitions, message and transaction signing, and wallet management.

In *-chain, we adopt Ethers.js, currently one of the most lightweight solutions for managing smart contracts, accounts, providers, and cryptographic operations. While Web3.js has historically been the most popular choice, in recent years Ethers.js has emerged as the preferred alternative among developers and researchers due to a range of architectural and practical advantages.

From a technical standpoint, Ethers.js offers a modular and security-oriented design. Unlike Web3.js, which includes numerous legacy dependencies and a more cumbersome API, Ethers.js was built to be easily integrable, and better suited to contexts where code size and transparency are critical requirements.

[3]IPFS website: https://ipfs.tech.
[4]Web3.js documentation: https://web3js.readthedocs.io.
Ethers.js documentation: https://docs.ethers.org.

This makes it particularly appropriate for our needs, where we deploy user interfaces as distributed applications and need to seamlessly integrate decentralised components (smart contracts) with off-chain modules. Another key advantage of Ethers.js is its native compatibility with development tools like Hardhat, which greatly simplifies the creation of local testing environments.[5] This is increasingly valuable as public testnets become less reliable for routine development and debugging of blockchain-based applications.

In our implementation, blockchain events are read and processed with the aim of retrieving two key pieces of information: the hashes of the properties associated with a given product, and the current state of that product. This information is crucial for two tasks. The first is to allow the complete history of a product to be displayed in the interface of participants and viewers. The second is to correctly determine which operations are permitted on a certain product, based on its state and the user's role. To this end, we implement two separate functions that interact with the Ethereum blockchain via the Ethers.js library. These functions read and process relevant events from the blockchain log.

The first function, `queryBlockchainEventHash`, queries the blockchain for a specific type of event associated with a `productId`. Each of these events represents a recorded state transition for that product and includes the hash of the properties characterising the asset in that particular state. The function extracts these hash values from the emitted events and returns them as an array. The hashes are then cross-referenced with off-chain database entries to retrieve the corresponding human-readable property names and values. As a result, for each individual product, it is possible to reconstruct the complete sequence of states it has gone through, together with the associated properties at each state. This historical view is then displayed in the web interface, allowing users to inspect how the product's characteristics evolved over time.

To execute the query, three elements are required: the ABI of the smart contract, the address of the contract on the blockchain, and the name of the event to be filtered, which is derived dynamically according to the type of product selected. When executed, the function loads the appropriate ABI, connects to the Ethereum network via MetaMask and instantiates the smart contract interface. It then filters the blockchain logs for all matching events based on the `productId`. From each event thus retrieved, the hash is extracted, which will then be used to trace back the properties associated with the product in the database. This mechanism allows the system to reconstruct and visualise the evolution of the state of any product in terms of its associated properties in the various states.

In addition to retrieving the properties of the product, we also need to identify its current state, since, similarly to the role system, this determines which operations are permitted on it. In fact, each operation can only be performed if the product is in a specific state. To determine the current state, the function `queryBlockchainEventLastState` adopts the same event query procedure used for property hashes. However, unlike the latter, which requires processing all the retrieved events, this function extracts the state only from the last event emitted. Since the events returned by filtering the logs with Ether.js are ordered chronologically by block number, from oldest to newest, the last event returned will also contain the most recent status associated with the selected product. Based on the retrieved state, the participant interface dynamically enables the operations available for the product, in accordance with the user's role and SCMS logic.

A key advantage offered by public blockchains, such as Ethereum, is the total transparency of recorded information. All smart contract transactions, as well as issued events, are visible and can be consulted by anyone. *-chain intentionally exploits this feature to guarantee maximum traceability and ensure that the state of an asset can be fully reconstructed simply by reading the sequence of events associated with its identifier. This allows any observer, both internal and external to the supply chain, to independently verify the correctness of the process, thus fostering auditability and trust.

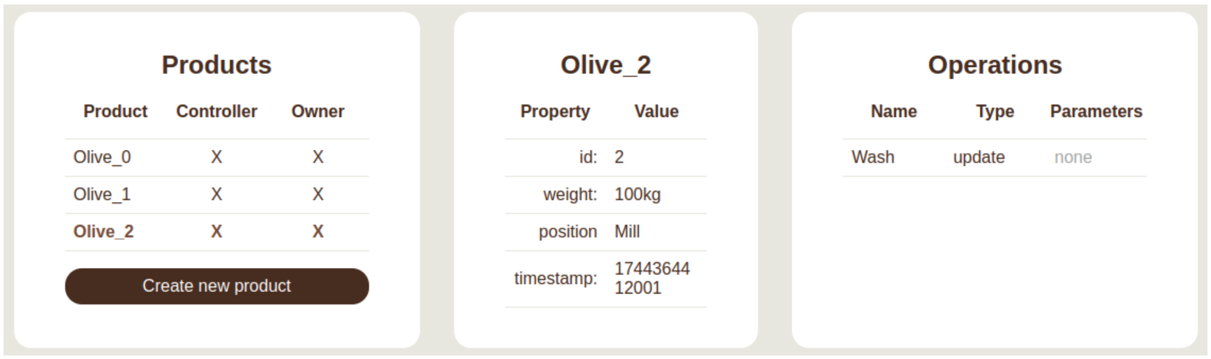
We reiterate that the privilege to consult the status of an asset does not imply the possibility of modifying it. In fact, the operations that enable the life cycle of an asset to be advanced are protected by the access control mechanism embedded in our smart contracts: only participants officially registered in the chain and endowed with the appropriate role can invoke operations that alter the state of the asset.

[5]Hardhat website: https://hardhat.org.

3.2. User Interfaces

We now demonstrate how the properties and states read from the events modify the participant interface to show updates on products. Each participant will have access to functionalities based on their role. Although our goal is to provide a general overview of the system through the lens of the newly introduced events, without focusing on a specific supply chain, in this section we will use the oil supply chain, previously studied in [4], for illustrative purposes.

In Figure 1, we see three panels of the the participant interface displaying summary information about products and the operations executable on them by a participant with the "miller" role. The leftmost panel shows all registered products, along with their respective identifiers, owners, and controllers. These details are all retrieved from the blockchain using functions automatically generated by *-chain. When one of these products is selected, the functionalities of the other two panels are activated.



Products

Product	Controller	Owner
Olive_0	X	X
Olive_1	X	X
Olive_2	**X**	**X**

Olive_2

Property	Value
id:	2
weight:	100kg
position	Mill
timestamp:	17443644 12001

Operations

Name	Type	Parameters
Wash	update	none

Figure 1: Example of participant interface displaying product details, property history, and valid operations based on product state.

The central panel displays properties related to the asset selected from the left panel, including the name and associated value. If the product has undergone state changes, the properties from each state will be shown sequentially, so that the user can reconstruct the product's history. The properties are read as hashes from the events in the blockchain logs via the function `queryBlockchainEventHash`, as described above. The off-chain database then allows us to map each hash to the name-value pairs of the associated properties.

Finally, the rightmost panel lists the valid operations allowed on the selected product, i.e. those for which the product is in the required state, the user holds the required role, and constraints on the owner and controller are met. In addition to the operation name, its type and input parameters are also displayed. These parameters will set the properties the product will assume in the next state. Operations of the type `asset_create` are the only ones handled differently, as they can be executed without selecting an existing product. As such, we have chosen to display them in the first panel, making the consultation and creation of individual products more straightforward. When an operation is invoked, by clicking its name in the third panel or using the "Create new product" button in the first panel, a form appears where parameters can be set and execution confirmed. If executed successfully, the operation registers the necessary information on the blockchain and emits an event with the hash of the properties. At this point, the interface is updated, displaying the properties and operations for the selected product corresponding to its current state.

The participant interface also displays a visual representation of the supply chain generated using the *-chain DSGL. This visual aid provides an immediate overview of the product's progress through the chain, as illustrated in Figure 2. When a product is selected and after each operation has been performed, the interface highlights the following components of the representation:

- the current state of the asset
- the valid operations available from the current state
- the history of previous states and operations the product has passed through
- the properties associated to current and previous asset states

To enhance their visibility, the operations that can be performed on the product are further highlighted with a shadow in the background. All other elements of the representation that have not yet been reached in the production cycle, that is, potential future states, properties, and operations that cannot yet be executed, are displayed with transparency.

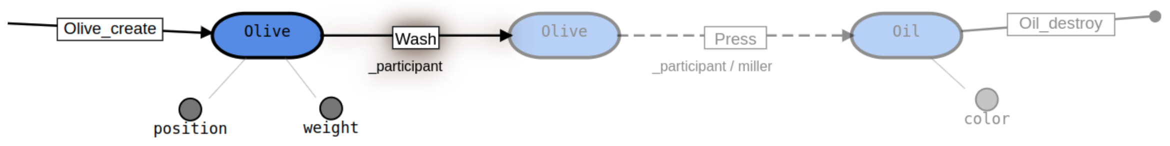


Figure 2: Example of participant interface with a dynamic visualisation of the product's state, valid operations, and history within the supply chain.

4. The Practical Role of Events in Asset Lifecycle Management

Events in Ethereum smart contracts play a central role in linking on-chain processes with off-chain systems and provide a powerful abstraction for tracking asset lifecycles. This section explores how events can be leveraged to improve the usability, transparency and performance of our system, using practical examples from supply chain contexts to illustrate the benefits in concrete terms.

4.1. Event Logging and Advantages for Auditability and Tracking

Events are stored on transaction receipts instead of being stored in contract storage, providing a gas-efficient way to track important changes without increasing on-chain data costs. Although events are not part of the contract's permanent state, they are permanently recorded in the blockchain logs and cannot be altered or removed. This immutability makes them highly reliable for purposes such as compliance auditing, debugging, and tracking the lifecycle of contractual operations.

From an architectural point of view, this approach supports an event-driven model that decouples on-chain logic from off-chain processing. External systems, such as monitoring tools, dashboards, or analytics platforms, can subscribe to our event streams and respond to changes in near-real time. This enables asynchronous integration, allowing external user-facing applications to reflect changes in contract states without the need to repeatedly query the blockchain. Furthermore, event logs provide a transparent and verifiable history of every significant action taken within the smart contract. For auditors and external developers, this allows for a detailed inspection of asset flows and role-based interactions. For end users, events enhance trust and usability by powering transaction histories, notifications, and activity timelines.

Importantly, this event-driven design significantly enhances both the development and usability of our participant and viewer interfaces. By allowing the front end to listen to emitted events and reflect state changes in real-time, it removes the need for manual data fetching or complex state synchronisation. As a result, the architecture ensures that contract activities are communicated efficiently and transparently to all users.

4.2. Separation of Responsibilities and Security

A key design principle in smart contracts is the separation of responsibilities between core logic and external data handling. In our framework, this separation is maintained: the smart contract enforces the rules and governs asset state transitions, where the state represents the current phase or condition of the asset within the supply chain, while the event system is used to notify external systems and store verifiable summaries of those states.

Although we use events to persist a hash of the asset properties, this does not compromise the contract's role or responsibilities. The contract does not rely on event data for its own execution or correctness. Instead, it emits a cryptographic fingerprint of the asset state, allowing external systems to store and interpret the data without affecting the contract's logic.

This approach keeps the contract simple and modular, making it easier to maintain or extend. From a security point of view, it also ensures that any issues in off-chain processing do not impact the contract, since the on-chain behaviour is fully self-contained [3]. Additionally, avoiding on-chain storage in favour of events reduces gas costs while still supporting transparency and auditability, as the emitted hashes can be used to verify the integrity of off-chain data.

4.3. Real-Time Interaction and System Efficiency

Smart contract events play a central role in enabling reactive architectures within blockchain-based applications. Unlike traditional storage updates, events do not alter the contract's internal state but are instead recorded in the transaction logs, which are permanently accessible and efficiently indexable by client applications. In *-chain, events are emitted to capture every meaningful state transition in the lifecycle of supply chain assets. This mechanism allows external applications, such as dashboards, web interfaces, and monitoring tools, to subscribe to these logs and maintain an up-to-date view of the system without directly querying contract storage. In practice, this translates into significant architectural and performance advantages, of which we provide an overview below.

Without events, applications would need to periodically query smart contracts to check for state changes, which is both inefficient and costly. In contrast, event-driven systems can listen passively and only react when relevant events are emitted, greatly reducing network load and resource consumption. Events also enable the creation of user interfaces that automatically react to changes occurring on-chain. For example, in an olive oil supply chain, when a batch is packaged, the smart contract can emit an event that triggers a front-end interface to automatically update the status of the corresponding asset. This update happens in real-time, without the need for manual refreshes or periodic polling. Moreover, events are particularly suited for handling asynchronous workflows. They can confirm, for instance, when a payment has been received or when a delivery has been initiated. These transitions often depend on external interactions, such as user wallets or oracles, and events provide an efficient mechanism to observe and respond to such changes. Additionally, events help separate concerns by decoupling state updates from notification logic. Since events are primarily designed for off-chain consumption, they allow state changes to be signalled to listening systems without requiring an additional write to the blockchain. This further reduces gas consumption and enhances system efficiency.

To better understand the practical advantages of using events in blockchain applications, it is helpful to compare two different approaches to handling state updates in a supply chain context. A concrete example helps illustrate the difference: imagine a crate of olives that needs to be tracked as it moves through various stages, being transferred from the warehouse to the mill, washed, and eventually pressed into oil.

In an event-driven system, smart contracts emit events whenever something relevant happens along this chain. When the crate is moved to the mill, an event is emitted. Later, when the olives are washed, another event is triggered, and the same occurs once pressing is completed. These events can be detected in real-time by external applications, such as dashboards, quality control systems, or notification services. The key benefit is that the front-end automatically reflects each change as soon as it is recorded on-chain, without requiring any manual action. This ensures that all stakeholders (producers, processors, inspectors, or logistics operators) have an up-to-date view of the crate's status, improving coordination and reducing uncertainty.

By contrast, a system that does not rely on events would need to use manual checks or polling. In the manual case, users must refresh the interface or actively request updates to see if the crate has progressed to the next stage. If they forget or delay doing so, the displayed information may be outdated, which can lead to poor decisions or miscommunication. Polling automates this process to some extent by regularly querying the blockchain to check if a change has occurred. However, this approach increases network load, introduces delays between state changes and their visibility in the interface, and requires extra effort to manage the polling logic across many assets. This difference becomes even more relevant as the supply chain grows in scale and complexity. With hundreds of crates being processed simultaneously and multiple actors interacting with the system, continuously checking the status of each item quickly becomes inefficient. Events provide a more scalable and elegant solution, since applications simply listen for the specific changes they care about, and react only when those changes occur.

4.4. Reducing Gas Costs Through Event-Based Logging

One of the most practical advantages of using events in Ethereum smart contracts is the significant reduction in gas costs. Writing data to the blockchain's persistent storage is among the most expensive operations in terms of gas consumption, while emitting an event, although also recorded on-chain in the transaction logs, is considerably cheaper. In a typical supply chain scenario, every time an asset (e.g. a crate of olives) progresses in its lifecycle, the system must record the associated state change. If each of these transitions, such as "transferred to mill", "washed", or "pressed", were stored directly in the contract's storage, the cost would grow rapidly, especially when scaled to hundreds or thousands of assets.

A straightforward optimisation is to store only a global cryptographic hash representing all asset properties, rather than the properties themselves. This already offers a significant gain: writing a single `uint256` hash to storage costs roughly $20,000$ gas, compared to storing multiple fields, which could result in several times that amount. However, our proposed solution further reduces this cost by emitting the hash as part of an event instead of persisting it in storage. Emitting an event with a `uint256` hash typically costs around $1,500$–$2,000$ gas, an order of magnitude less than storing the same hash in the contract state. This difference becomes particularly important when updates are frequent or when the number of properties is large. Moreover, since events are designed for off-chain consumption, they provide a natural mechanism for transparent communication with user interfaces and external systems. In *-chain, only the minimal information required for validation and access control is stored directly in the contract, while all non-critical but verifiable data (e.g. asset properties) is emitted in structured events.

5. Related Work

In literature, various studies have addressed blockchain-based SCMS from multiple perspectives, often focusing on specific use cases or industries. The survey in [7] provides a theoretical overview of trends and future directions in blockchain-enabled supply chains, while [8] offers a comprehensive review of agrifood applications, highlighting the potential of blockchain to improve traceability, safety, and fraud prevention, as well as practical challenges such as scalability and regulation.

Other works focus on domain-specific models. [9] develops a blockchain-based economic exchange tracking system to enhance trust in financial transactions, though the system remains largely theoretical. In the manufacturing sector, [10] examines blockchain adoption strategies through a Stackelberg game model, considering both direct and retail distribution channels. [11] addresses the use of blockchain in vaccine distribution, suggesting verification mechanisms to ensure reliability, though without implementing a concrete supply chain schema.

From a decision-theoretic perspective, [12] analyses the trade-off between traceability and environmental sustainability in a global supply chain, showing that blockchain adoption is feasible only under specific economic and consumer behaviour conditions. Finally, [13] presents a decentralised system for pharmaceutical distribution that leverages smart contracts to eliminate intermediaries, improving traceability and reducing transaction times.

While these studies provide valuable insights into the potential of blockchain in SCMS, they tend to either remain conceptual or focus on storing data directly in the contract state. Our work takes a different direction by investigating the feasibility of using events, rather than persistent storage, as the main mechanism for recording supply chain interactions. This approach is evaluated within the context of *-chain, an asset-oriented framework designed to be domain-agnostic and highly customisable.

6. Conclusion and Future Work

In this paper, we present an updated version of *-chain, a framework that helps experts design and implement decentralised SCMS via web interfaces. The main contribution is the integration of events into the generated Ethereum smart contracts, which allow the properties of assets to be recorded in the blockchain log as they evolve along the supply chain. The advantage lies in the fact that writing into the blockchain log requires less gas than writing into storage, while the decentralisation and immutability of the data are maintained. This approach also opens up opportunities for system integrations in which external components can listen for changes without continuously querying the blockchain. Although this strategy does not replace all uses of on-chain states, it offers a practical compromise in contexts where notarisation and verifiability are important, but full storage is not strictly necessary.

In future developments, we will focus on extending *-chain in several directions. One of our primary goals is to introduce mechanisms that allow users to choose whether to record the properties of an asset directly in the blockchain storage or via hashes in the logs as is the default in the current implementation. At the same time, we plan to implement a type of operation that, by design, does not involve writing into the blockchain, but uses exclusively an off-chain database. These adjustments aim to lower the cost for operations that do not require strong guarantees. In order to improve cost efficiency, we also would like to conduct an in-depth study of the cost differences resulting from the use of events and the management of properties and other parameters via hashes.

Another line of research concerns the formal validation of smart contracts. Immediately after the generation of the ABI, we could introduce automatic checks to ensure that the contract meets certain key structural requirements. For example, it should be possible to derive a valid sequence of operations like `asset_create` and subsequent transformations that logically link an asset at the beginning of the supply chain to its final output. If no valid trace exists, the implementation of the contract must be interrupted to avoid logical inconsistencies. Finally, we intend to incorporate a hierarchical RBAC model, allowing for more expressive and secure role delegation. This will also result in a better alignment between organisational structure and on-chain permissions.

Acknowledgements

The authors would like to thank Francesco Faloci for the work carried out during his doctoral thesis, which contributed in part to the development of this paper. The authors are member of the INdAM Research group GNCS and of Consorzio CINI. This work has been partially supported by: INdAM - GNCS Project, codice CUP_E53C24001950001 MUR project PRIN 2022TXPK39 - PNRR M4.C2.1.1. “Empowering Public Interest Communication with Argumentation (EPICA)” CUP H53D23003660006, funded by the European Union - Next Generation EU, Missione 4 Componente 1; MUR PNRR project SERICS (PE00000014 AQuSDIT: CUP_H73C22000880001, COVERT: CUP_J93C23002310006), funded by the European Union – Next Generation EU; EU MUR PNRR project VITALITY (J97G22000170005), funded by the European Union – Next Generation EU; University of Perugia - Fondo Ricerca di Ateneo (2020, 2022) – Projects BLOCKCHAIN4FOODCHAIN, FICO, RATIONALISTS, “Civil Safety and Security for Society”; Piano Sviluppo e Coesione Salute PSC 2014-2020 - Project I83C22001350001 LIFE: “the itaLian system wIde Frailty nEtwork” Linea di azione 2.1 “Creazione di una rete nazionale per le malattie ad alto impatto” - Traiettoria 2 “E-Health, diagnostica avanzata, medical devices e mini invasività” Codice locale progetto T2-AN-12 CUP J93C22001080001.

Declaration on Generative AI. During the preparation of this work, the authors used ChatGPT in order to: Grammar and spelling check, Paraphrase and reword. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

[1] E. U. I. P. Office, Economic impact of counterfeiting in the clothing, cosmetics, and toy sectors in the EU, Technical Report TB-02-23-317-EN-N, European Union Intellectual Property Office, 2024. URL: https://doi.org/10.2814/053613. doi:10.2814/053613.

[2] S. Bistarelli, F. Faloci, P. Mori, *-chain: A framework for automating the modeling of blockchain based supply chain tracing systems, Future Gener. Comput. Syst. 149 (2023) 679–700. URL: https://doi.org/10.1016/j.future.2023.07.012. doi:10.1016/J.FUTURE.2023.07.012.

[3] L. Li, Y. Liang, Z. Liu, Z. Yu, Understanding solidity event logging practices in the wild, in: S. Chandra, K. Blincoe, P. Tonella (Eds.), Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, ACM, 2023, pp. 300–312. URL: https://doi.org/10.1145/3611643.3616342. doi:10.1145/3611643.3616342.

[4] S. Bistarelli, F. Faloci, P. Mori, C. Taticchi, Olive oil as case study for the *-chain platform, in: M. Pizzonia, A. Vitaletti (Eds.), Proceedings of the 4th Workshop on Distributed Ledger Technology co-located with the Italian Conference on Cybersecurity 2022 (ITASEC 2022), Rome, Italy, June 20, 2022, volume 3166 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 94–102. URL: https://ceur-ws.org/Vol-3166/paper07.pdf.

[5] D. F. Ferraiolo, D. R. Kuhn, R. Chandramouli, Role-Based Access Control, 2nd ed., Artech House, Inc., USA, 2007.

[6] T. Amjad, M. Sher, A. Daud, A survey of dynamic replication strategies for improving data availability in data grids, Future Gener. Comput. Syst. 28 (2012) 337–349. URL: https://doi.org/10.1016/j.future.2011.06.009. doi:10.1016/J.FUTURE.2011.06.009.

[7] M. Pournader, Blockchain applications in supply chains, transport and logistics: a systematic review of the literature, International Journal of Production Research 58 (2019). doi:10.1080/00207543.2019.1650976.

[8] A. Mohammed, V. M. Potdar, M. Quaddus, W. Hui, Blockchain adoption in food supply chains: A systematic literature review on enablers, benefits, and barriers, IEEE Access 11 (2023) 14236–14255. URL: https://doi.org/10.1109/ACCESS.2023.3236666. doi:10.1109/ACCESS.2023.3236666.

[9] C. Tsai, Supply chain financing scheme based on blockchain technology from a business application perspective, Ann. Oper. Res. 320 (2023) 441–472. URL: https://doi.org/10.1007/s10479-022-05033-3. doi:10.1007/s10479-022-05033-3.

[10] B. Gong, H. Zhang, Y. Gao, Z. Liu, Blockchain adoption and channel selection strategies in a competitive remanufacturing supply chain, Comput. Ind. Eng. 175 (2023) 108829. URL: https://doi.org/10.1016/j.cie.2022.108829. doi:10.1016/j.cie.2022.108829.

[11] Y. Gao, H. Gao, H. Xiao, F. Yao, Vaccine supply chain coordination using blockchain and artificial intelligence technologies, Comput. Ind. Eng. 175 (2023) 108885. URL: https://doi.org/10.1016/j.cie.2022.108885. doi:10.1016/j.cie.2022.108885.

[12] D. Biswas, H. Jalali, A. H. Ansaripoor, P. D. Giovanni, Traceability vs. sustainability in supply chains: The implications of blockchain, Eur. J. Oper. Res. 305 (2023) 128–147. URL: https://doi.org/10.1016/j.ejor.2022.05.034. doi:10.1016/j.ejor.2022.05.034.

[13] K. C. Bandhu, R. Litoriya, P. Lowanshi, M. Jindal, L. Chouhan, S. Jain, Making drug supply chain secure traceable and efficient: a blockchain and smart contract based implementation, Multim. Tools Appl. 82 (2023) 23541–23568. URL: https://doi.org/10.1007/s11042-022-14238-4. doi:10.1007/S11042-022-14238-4.