Object-Centric Process Mining for Blockchain Applications

Extracting and Representing Ethereum Execution Data in OCEL 2.0

Richard Hobeck[*,a], Alessandro Berti[b], Ingo Weber[c], Wil M.P. van der Aalst[d]

[a] Chair of Service-centric Networking, Technische Universitaet Berlin, Germany.
[b] RWTH Aachen University, Aachen, Germany & Fraunhofer FIT, Sankt-Augustin, Germany.
[c] Technical University of Munich, School of CIT & Fraunhofer-Gesellschaft, Munich, Germany
[d] RWTH Aachen University, Aachen, Germany & Celonis, Munich, Germany & Fraunhofer FIT, Sankt-Augustin, Germany.

Abstract. *Analyzing the execution behavior of Ethereum Decentralized Applications (DApps) with process mining presents significant challenges due to the multi-object nature of DApp data. Traditional event logs, like XES, struggle to capture the respective structures and interactions effectively. This paper proposes a method for extracting DApp execution data from Ethereum and representing it in the Object-Centric Event Log (OCEL) 2.0 format. We address central challenges in this process, including dynamic contract deployments, preserving the order of operations within transaction traces, and accurately representing object types and their evolving roles. Our findings demonstrate that, while OCEL 2.0 offers some advantages for capturing the rich interactions within DApps, certain limitations regarding hierarchical object types and event granularity require workarounds. We evaluate the practicality of our approach with a case study of the prediction market platform* Augur*, highlighting how object-centric process mining can provide insights into DApp behavior. This work contributes to a better understanding of object-centric process mining in the context of blockchain data.*

Keywords. Process Mining • Blockchain • OCEL 2.0 • Event Log Extraction

Communicated by Julius Köpke. Received 2024-05-01. Accepted on 2025-01-07.

1 Introduction

Blockchain is a distributed ledger system that enables transactions among parties without the need for a central authority (Nakamoto 2008). Second-generation blockchains like Ethereum extend these capabilities by allowing the execution of *smart contracts* – self-executing code that runs on the blockchain network (Buterin et al. 2014). Relying on smart contracts are Decentralized Applications (DApps). DApps are consequently deployed to the blockchain network and interact with other smart contracts and users, generating extensive runtime data (Wood 2014). The logged data can be analyzed with process mining techniques, e. g., to verify that the code execution was performed as expected (Hobeck et al. 2024).

Process mining bridges the gap between modeled processes and actual process execution by analyzing event logs. Traditionally, event logs are stored in formats like the eXtensible Event Stream (XES) (Acampora et al. 2017). In XES, each event is associated with *a single* case (or process instance), and events are ordered within cases. While effective for linear and simple processes, this structure can lead to issues when dealing with complex systems like blockchain applications, where multiple interacting entities and concurrent activities are common. Here, one event may relate to multiple process instances, instead of a single one.

To address such limitations, the Object-Centric Event Log (OCEL) format was developed (Ghahfarokhi et al. 2021). OCEL allows events to

be related to multiple objects of different types, capturing more complex and realistic process behaviors. The latest iteration, OCEL 2.0 (Berti et al. 2024), introduces capabilities to document dynamic object roles and qualified relations between objects and events, enhancing the ability of logs to capture the nuances of blockchain operations.

Various methods have been proposed to generate event logs from blockchain data (Moctar-M'Baba et al. 2022b), including approaches that simplify the logs by focusing on particular aspects of smart contracts (Alzhrani et al. 2024; Beck et al. 2021; Corradini et al. 2019; Klinkmüller et al. 2019; Mühlberger et al. 2019). However, these methods often fall short in capturing the full behavior of DApps due to limitations in handling dynamic and distributed data structures common in blockchain environments (Moctar-M'Baba et al. 2022a).

In this paper, we propose a comprehensive data extraction technique and use the OCEL 2.0 format to better capture the dynamic behavior of Ethereum-based applications. By broadening the data sources and adapting an object-centric approach, we address the challenges associated with capturing and analyzing blockchain data using process mining techniques.

We investigate the following research questions:

RQ1 How can data from dynamically deployed Decentralized Applications on Ethereum be extracted with limited preliminary knowledge about the applications?

RQ2 Which technical and data integrity challenges arise when adopting object-centric event logs (OCEL 2.0) to capture the behavior of dynamically deployed Decentralized Applications on Ethereum, and how can these challenges be addressed?

RQ3 To what extent does OCEL 2.0 accurately represent the behavior of dynamically deployed Decentralized Applications on Ethereum, and in which scenarios does it face limitations?

RQ4 How can object-centric process mining enhance the analysis of event data from Ethereum DApps compared to a single-case notion approach?

The contributions of this paper are[1] :

1. *Data extraction method:* We introduce a data extraction method for Ethereum DApps that *adapts to evolving DApp structures*, code updates, and distributed logging practices, ensuring data capture throughout a DApp's lifecycle. The extraction method incorporates *a range of blockchain data sources*, including function calls and their input parameters, values of logged variables, contract creations, and the flow of digital assets (cryptocurrency and tokens).
2. *Addressing OCEL 2.0 conversion challenges:* We identify and address the challenges encountered when transforming complex blockchain data into the object-centric event log (OCEL 2.0) format, offering solutions for accurately capturing DApp interactions.
3. *Mapping blockchain data to OCEL 2.0:* We provide a mapping of Ethereum data structures to the elements of the OCEL 2.0 meta-model, showing how OCEL 2.0 can capture multi-object interactions and dynamic relationships within a DApp.
4. *Object-centric process mining analysis:* We analyze DApp data formatted as OCEL 2.0, examining object lifecycles, object-to-object relations, and process visualizations with multiple objects – data aspects that are difficult to explore using traditional event log formats or raw blockchain data.

The remainder of the paper is structured as follows. In Section 2, we describe the XES and OCEL logging formats. Section 3 provides an

[1] This work extends prior research by Hobeck and Weber (2023). The extension affected the following contributions: In 1. we extend the data extraction capabilities to additional Ethereum operations. In 2. we address challenges of converting the extracted blockchain data into the more expressive OCEL 2.0 format (in contrast to OCEL), and added the underlying discussion of blockchain data characteristics. Consequently, 3. is an entirely new contribution. In 4. we added life-cycle and object-to-object analyses and focused the process visualization and interpretation on a subprocess of the examined DApp.

overview of key blockchain concepts necessary to understand our approach. We discuss related literature in Section 4. Section 5 outlines data attributes relevant to object-centric event logs derived from the Ethereum blockchain. Our methodology for data extraction from the Ethereum blockchain is described in Section 6. In Section 7, we examine the challenges encountered during data extraction and data transformation. Section 8 evaluates the suitability of the Object-Centric Event Log (OCEL) 2.0 format in capturing blockchain data, with a discussion of practical implications in Section 9. Finally, Section 10 concludes the paper and suggests directions for future research.

2 Logging Formats in Process Mining

In process mining, event logs are used to store process data. The structure and content of these logs can influence the quality of the process mining analysis. There are two primary paradigms in process mining: *case-centric* and *object-centric*.

Case-centric process mining is the traditional approach in which each event in an event log is associated with a single *case*, representing a unique instance of a process. The *eXtensible Event Stream (XES)* standard (Aalst 2022; Acampora et al. 2017) is designed for case-centric process mining and widely adopted. Consider an online retail process in which customers place orders consisting of multiple items, which are then shipped and invoiced separately. The example process has the following activities: *Create Order*, *Add Item*, *Ship Item*, *Generate Invoice*, and *Receive Payment*. Here, events may relate to multiple objects, such as orders, items, and invoices. In case-centric process mining, we must choose a case notion to flatten the log, such as *Order ID*. Table 1 shows how events are recorded. Choosing a single case notion introduces several issues:

1. *Deficiency*: If an event does not have the chosen case notion, it may be omitted or misrepresented. E.g., events not directly referencing orders like *Receive Payment* might not appear in the log.
2. *Convergence*: If an event relates to multiple instances of the same object type, it may be duplicated. E.g., *Generate Invoice* relates to different orders but is duplicated due to the selected case notion. Replicating events may result in misleading diagnostics involving costs and time.
3. *Divergence*: Events in a case having the same activity may be related to different objects (next to the object selected as the case), creating apparent causalities. E.g., the *Ship Item* events for different items belonging to the same order may appear sequentially in the same case, suggesting a false dependency.

In contrast, an *object-centric event log* allows events to be associated with multiple objects (Aalst 2019, 2022). Table 2 presents the same online order process using an object-centric approach. The object-centric view was formalized in the *Object-Centric Event Log (OCEL)* format (Ghahfarokhi et al. 2021). OCEL extends the traditional event log model by allowing events to reference multiple objects, of potentially different types. Our object-centric example event log in Table 2 aligns with the OCEL format. Each event can relate to all relevant objects – orders, items, invoices – providing a more accurate representation of the process. By supporting multiple object references per event, OCEL overcomes the issues of the XES format in complex scenarios: All events are included without forcing a single case notion; Relationships between objects are inferred through shared events, allowing for better understanding of interactions; By capturing the correct associations between events and objects, OCEL prevents misleading dependencies.

Building upon the foundation of OCEL, the release of *OCEL 2.0* (Berti et al. 2024) introduced extensions to better model relationships within event data. Key improvements in OCEL 2.0 include:

- *Qualified event-to-object relationships*: Events can have relationships to objects with specific roles or qualifiers, allowing for dynamic role

Table 1: Traditional case-centric event log (flattened on order ID)

Case ID	Timestamp	Activity	Attributes
Order_1	2024-11-01 10:00	Create Order	
Order_1	2024-11-01 10:05	Add Item	Item ID: Item_A
Order_1	2024-11-01 10:10	Add Item	Item ID: Item_B
Order_1	2024-11-02 09:00	Ship Item	Item ID: Item_A
Order_1	2024-11-03 08:30	Ship Item	Item ID: Item_B
Order_1	2024-11-04 12:00	Generate Invoice	Invoice ID: Inv_1
Order_0	2024-11-04 12:00	Generate Invoice	Invoice ID: Inv_1

Table 2: Object-centric event log capturing multiple objects per event

Event ID	Timestamp	Activity	Object
E1	2024-11-01 10:00	Create Order	Order_1
E2	2024-11-01 10:05	Add Item	Order_1,Item_A
E3	2024-11-01 10:10	Add Item	Order_1,Item_B
E4	2024-11-02 09:00	Ship Item	Item_A
E5	2024-11-03 08:30	Ship Item	Item_B
E6	2024-11-04 12:00	Generate Invoice	Order_1,Order_0,Inv_1
E7	2024-11-05 15:00	Receive Payment	Inv_1

The scenario assumes the existence of an *Order_0* that was created outside of the event log snippets as well as a bulk invoice creation for orders.

assignments during the process. E.g., the event *Add Item* involves *Order* and *Item*. With OCEL 2.0, we can qualify these relationships to specify that *Order* acts as a *container* and *Item* as *content* being added, enhancing the semantics of the log.

- *Qualified object-to-object relationships*: Relationships between objects are explicitly modeled rather than inferred through shared events. E.g., the relationship between *Order_1* and *Item_A* can be represented as a *contains* relationship. E.g., knowing that *Item_A* is part of *Order_1* allows analysts to trace the lifecycle of items within orders.
- *Evolution of object attributes over time*: OCEL 2.0 captures changes in object attributes in the process lifecycle, providing a temporal view of object states. E.g., an *Item*'s status may change from *Pending* to *Shipped* to *Delivered*. By recording these attribute changes over time, analysts can identify patterns, delays, or bottlenecks in the process.

Formally, an OCEL 2.0 log is described as a tuple L (Berti et al. 2024):

$$L = \big(E,\ O,\ EA,\ OA,\ evtype,\ time,\ objtype,\ eatype,\ oatype,\ eaval,\ oaval,\ E2O,\ O2O\big)$$

where:

- $E \subseteq \mathbb{U}_{\text{ev}}$ is the set of events.
- $O \subseteq \mathbb{U}_{\text{obj}}$ is the set of objects.
- $evtype : E \to \mathbb{U}_{\text{etype}}$ assigns types to events.
- $time : E \to \mathbb{U}_{\text{time}}$ assigns timestamps to events.
- $EA \subseteq \mathbb{U}_{\text{attr}}$ is the set of event attributes.
- $eatype : EA \to \mathbb{U}_{\text{etype}}$ assigns event types to event attributes.
- $eaval : (E \times EA) \nrightarrow \mathbb{U}_{\text{val}}$ assigns values to event attributes.
- $objtype : O \to \mathbb{U}_{\text{otype}}$ assigns object types to objects.
- $OA \subseteq \mathbb{U}_{\text{attr}}$ is the set of object attributes.
- $oatype : OA \to \mathbb{U}_{\text{otype}}$ assigns object types to object attributes.
- $oaval : (O \times OA \times \mathbb{U}_{\text{time}}) \nrightarrow \mathbb{U}_{\text{val}}$ assigns values to object attributes over time.
- $E2O \subseteq E \times \mathbb{U}_{\text{qual}} \times O$ are the qualified event-to-object relations.
- $O2O \subseteq O \times \mathbb{U}_{\text{qual}} \times O$ are the qualified object-to-object relations.

Here, $\mathbb{U}_{\text{ev}}$, $\mathbb{U}_{\text{obj}}$, $\mathbb{U}_{\text{etype}}$, $\mathbb{U}_{\text{otype}}$, $\mathbb{U}_{\text{attr}}$, $\mathbb{U}_{\text{val}}$, $\mathbb{U}_{\text{time}}$, and $\mathbb{U}_{\text{qual}}$ are sets of strings called universes. They represent the universes of events, objects, event types, object types, attribute names, attribute values, timestamps, and qualifiers, respectively. Figure 1 contains the conceptualization of the OCEL 2.0 metamodel. Implementations of OCEL 2.0 are available in various formats.[2]

Goossens et al. (2024) compared OCEL 2.0 to other object-centric event log formats. They made the following assessments: *eXtensible Object-Centric* logs (XOC) (Li et al. 2018) suffer from data quality issues due to data duplication during object attribute updates; *Data-aware Object-Centric Event Log* (DOCEL) (Goossens et al. 2023) does not support central object-to-object characteristics; *Artifact-Centric Event Log* (ACEL) (Moctar

[2] https://www.ocel-standard.org/, accessed 20-03-2025

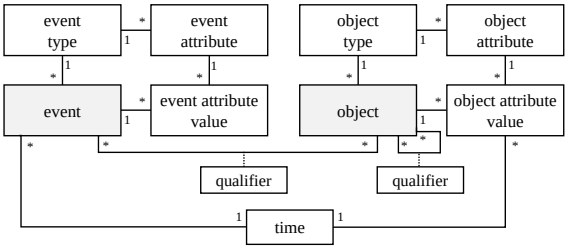


Figure 1: UML diagram conceptualizing OCEL 2.0 (adapted from (Berti et al. 2024)).

M'Baba et al. 2023), although tailored to artifact-centric processes prevalent in blockchain environments, faces scalability issues; and *Event Knowledge Graphs*'s (EKG) (Fahland 2022) low-level of storage formalisms limits analysis tool support. OCEL 2.0 does not fully support object-to-object relations but otherwise offers a wide range of desired object-centric logging format characteristics according to Goossens et al. (2024). Therefore, we opt for OCEL 2.0 as the target logging format for this paper.

3 Key Blockchain Concepts

This section outlines fundamental technical concepts relevant to understanding the methods and analyses presented in the paper.

Blockchain is an append-only store of transactions distributed across computational nodes and structured as a linked list of blocks, each containing a set of transactions. (Xu et al. 2019, Chapter 1). The blocks in the data structure, which is also called ledger, are linked sequentially through cryptographic hashes. Each block contains a list of transactions that are verified and agreed upon by network participants through a consensus mechanism. This architecture ensures the integrity and immutability of data once entered in the blockchain (Nakamoto 2008).

Ethereum is a second-generation blockchain platform that extends the basic blockchain functionality to support executing programs, called *smart contracts*. Smart contracts are written in high-level programming languages like Solidity and are compiled into bytecode executed by the *Ethereum Virtual Machine* (EVM). The interfaces of smart contracts are documented in *Application Binary Interfaces* (ABIs). ABIs are created at compile time and document a smart contract's interface functions (which can be called) as well as a set of log entries that are defined in the smart contract or inherited from other contracts.[3] Ethereum enhances the basic concept of blockchain by incorporating a built-in Turing-complete programming language, enabling the execution of complex programs. The native currency of the Ethereum blockchain is *Ether (ETH)* (Buterin et al. 2014).

Accounts on the Ethereum blockchains can be one of two types. *Externally Owned Accounts* (EOA) are controlled by the private key holder (e. g., a user). *Contract Accounts* (CA) capture smart contract code that can be executed on the Ethereum network and the program's state. Both types of accounts get assigned a 42-character address that they can be addressed with. Accounts have an Ether balance and can call other smart contracts via their CAs (Xu et al. 2019, p.37f).[4]

Decentralized Applications (DApps) are applications whose core functionality is implemented as a set of smart contracts. DApp source code is publicly available on the blockchain network (as byte code and often published open source), so users can perform code reviews. Also, the execution of DApp code is guaranteed to be deterministic (Xu et al. 2019, p. 39f), and Turing complete so that logic can be executed reliably. Among the best-known Ethereum DApps are the game *CryptoKitties*, the betting platform *Augur*, and the cryptocurrency exchange *Uniswap*.

Ethereum Virtual Machine (EVM) is a runtime environment for deterministic smart contract execution. The EVM is included in every node of the Ethereum blockchain, and executions do not affect the network until they are validated and completed (Buterin et al. 2014).

Transactions on a blockchain refer to the actions initiated by EOAs to alter the state of the blockchain, such as transferring cryptocurrency or

[3] https://docs.soliditylang.org/en/develop/abi-spec.html, accessed 20-03-2025

[4] https://ethereum.org/en/developers/docs/accounts/, accessed 20-03-2025

executing a function in a smart contract (Buterin et al. 2014).

Transaction Traces are detailed step-by-step executions of Ethereum transactions that show how the state of the blockchain changes with each operation within a transaction. These traces offer information for understanding interactions within and across smart contracts[5] (also called *internal transactions*).

4 Related Work

Prior research proposed frameworks to transform blockchain data to event logs in various formats, but these often exhibit limitations in capturing the dynamic, multi-entity nature of blockchain applications.

Klinkmüller et al. (2019) introduced the Ethereum Logging Framework (ELF) for transforming Ethereum on-chain data into event logs. Beck et al. (2021) advanced this work with the Blockchain Logging Framework (BLF), a versatile extract-transform-load (ETL) tool designed for various blockchain platforms. ELF and BLF facilitate systematic logging but rely on user-defined queries with predefined target contracts, potentially overlooking DApp behavior changes that fall outside the scope of initial specifications.

Mühlberger et al. (2019) presented an approach for extracting event logs from replayed Ethereum transactions. They were the first ones to use smart contract function signatures as a basis for event identification. However, their approach left available blockchain data attributes unconsidered, such as log entries and contract creations.

Alzhrani et al. (2024) developed an Event Log Generator (ELG) as part of their framework for rule-based DApp classifications to automate log generation. Their ELG is limited to retrieving log entries from Ethereum and decoding them using the emitting contract's ABI.

Further extending these concepts, Corradini et al. (2024) emphasized the need for capturing additional data from Ethereum smart contract executions, such as state transitions and internal transactions. The approach requires little preliminary knowledge about the smart contract. Morichetta et al. (2024)'s EveLog operates in a similar fashion. Their approach also avoids the need for predefined configurations, allowing it to operate across a broader range of applications by systematically extracting and structuring logs into an XES format. Both approaches offer a comprehensive view by considering internal transactions and state changes. They still face limitations in going beyond a predefined set of smart contracts, and handling DApps consisting of multiple dynamically deployed smart contracts.

In addition to the above approaches, we broaden the data scope from individual smart contracts to entire DApps by considering contract creations. Additionally, our approach leverages the Object-Centric Event Log (OCEL) 2.0 to capture the data with interactions and multi-object dependencies within Ethereum-based DApps in a scalable way.

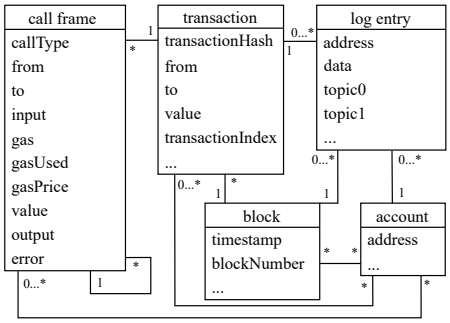


Figure 2: Class diagram showing relations between contracts, transactions, events, and call frames.

5 Data Attributes for Object-Centric Event Logs from the Ethereum Blockchain

The Ethereum blockchain contains various data attributes that can be analyzed with process mining. Figure 2 is a class diagram depicting the relations between blockchain data structures. Transactions are included in blocks. Transactions capture transitions from one state of the blockchain to another (Wood 2014). The result of executing a

[5] https://geth.ethereum.org/docs/developers/evm-tracing, accessed 20-03-2025

transaction is hashed, and the hash is stored in the respective transaction receipt.

On the Ethereum blockchain, transactions may contain invocations of smart contract functions, which result in computational operations being executed on the Ethereum Virtual Machine (EVM). These computational operations can be captured in transaction traces. Transaction traces exist on different levels of granularity. They can include assembly-level operations, e. g., comparisons and bit-wise logic operations, or push operations (within the EVM's computational stack), but they also comprise log entries and system operations (e. g., creations of CAs and message calls between CAs) (Wood 2014). Transaction traces exist temporarily and are not stored permanently in blocks of the Ethereum blockchain. In order to retrieve traces for historical transactions, transactions have to be replayed on the EVM (replaying the transition from one state of the blockchain to the next), and the computational steps have to be stored separately from the blockchain. For that purpose, different tracers exist for the most widely used Ethereum execution client Geth.[6] [7]

A transaction trace is hierarchically structured. In the transaction depicted in Listing 1, multiple CAs were invoked. Each CA invocation appears on subsequent hierarchical levels within *call frames* (sometimes called *internal transactions*). Call frames include additional attributes, such as *from*: initiating address; *to*: target address; *gas*: budget available for the operation; *gasUsed*: budget consumed for the operation; *input*: data payload; *output*: result of the operation; *value*: Ether value sent with the operation; *type*: operation type; and *logs*: log entries emitted during the operation. In the transaction in Listing 1, a call was issued from address 0xd...d6 to CA 0x75...99. To compute the call, 0x75...99 issued calls to other contracts and emitted a log entry. In transaction traces, the following operation types are defined:

Listing 1: Example transaction trace

```
{
  'from': '0xd82369aaec27c7a749afdb4eb71add9e64154cd6',
  'gas': '0xc578c',
  'gasUsed': '0xb74f3',
  'to': '0x75228dce4d82566d93068a8d5d49435216551599',
  'input': '0x9684da1a',
  'output': '0x00...77',
  'calls': [
    {
      'from': '0x75228dce4d82566d93068a8d5d4943521655159
          ↪ 9',
      'gas': '0xc1b19',
      'gasUsed': '0x3a6',
      'to': '0xb3337164e91b9f05c87c7662c7ac684e8e0ff3e7'
          ↪ ,
      'input': '0xf3...00',
      'output': '0x00...95',
      'value': '0x0',
      'type': 'CALL'
    },
    {
      'from': '0x75228dce4d82566d93068a8d5d4943521655159
          ↪ 9',
      'gas': '0xc0f85',
      'gasUsed': '0xab0ae',
      'to': '0xe62e470c8fba49aea4e87779d536c5923d01bb95'
          ↪ ,
      'input': '0x48...00',
      'output': '0x00...77',
      'calls': [
        {
          'from': '0xe62e470c8fba49aea4e87779d536c5923d0
              ↪ 1bb95',
          'gas': '0xb6295',
          'gasUsed': '0x2a4c6',
          'to': '0xe991247b78f937d7b69cfc00f1a487a293557
              ↪ 677',
          'input': '0x60...00',
          'output': '0x60...29',
          'value': '0x0',
          'type': 'CREATE'
        }
        // ...
      ]
      'value': '0x0',
      'type': 'CALL'
    }
  ],
  // ...
  'logs': [
    {
      'address': '0x75228dce4d82566d93068a8d5d4943521655
          ↪ 1599',
      'topics': [
        '0x299eaafd0d27519eda3fe7195b73e5269e442b3d80928
            ↪ f19afa32b6db2f352b6',
        '0x0000000000000000000000000000000000000000000000000
            ↪ 0000000000000',
        '0x0000000000000000000000e991247b78f937d7b69
            ↪ cfc00f1a487a293557677'
      ],
      'data': '0x00...00'
    }
  ],
  'value': '0x0',
  'type': 'CALL'
}
```

Notes: Enabled log tracing; input and output are abbreviated; transaction hash: 0x44c09f8eeff886723b79890e14743192a8c8d8a8eac158ed-17600c94e502cce8.

[6] https://ethereum.org/en/developers/docs/nodes-and-clients/, accessed 20-03-2025

[7] https://geth.ethereum.org/docs/developers/evm-tracing/built-in-tracers, accessed 20-03-2025

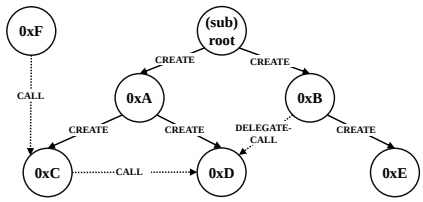


Figure 3: Schema of a single tree-like deployment structure of smart contracts and possible message calls between accounts.

Contract creations. Type mnemonic: `CREATE` or `CREATE2`. Contract creations are deployments of new smart contracts. When a smart contract creation is triggered, the EVM interprets the accompanying input data and attempts to deploy it as smart contract code in a newly created CA. A contract creation is depicted in Listing 1 between lines 27 and 36. Our approach assumes that when a DApp CA creates a new CA, the newly created CA also belongs to the DApp. CAs can be added to a DApp at initial deployment, but there are also mechanisms for updating or adding smart contracts to running DApps, e. g., the factory pattern, registry pattern (Xu et al. 2018), or diamond proxy.[8] A set of CAs belonging to one DApp that gets deployed can be seen as a tree structure (or a forest consisting of several trees or sub-trees). Every CA of a DApp has a single creator (parent) and can have several children that it can create. Additionally, the CAs of the DApp can send each other message calls laterally without traversing the creation branches. Contracts of the DApp can also be called from outside of the DApp (see Figure 3).

Calls. Type mnemonic: `CALL`. Two different types of calls exist. (1) *Ether transfers without function calls* are solely directed to value transfers without additional input data and without triggering the execution of smart contract logic. (2) *Function calls* appear when an account calls a CA's function. A function call depicted in Listing 1 between lines 9 an 18. Function calls to contracts preserve the context of the caller and callee as separate (in contrast to delegated calls).

Delegated calls. Type mnemonic: `DELEGATECALL`. `DELEGATECALL`s execute code in the context of the calling contract (in contrast to function calls). The `DELEGATECALL` function allows a contract to invoke a function in another contract. During a `DELEGATECALL`, the code of the called contract is executed with the storage, sender, and ETH value of the calling contract. This means that while the code is defined in the called contract, it manipulates the state variables of the calling contract. The primary application of `DELEGATECALL` is to enable updates of smart contract functionality. Since deployed smart contracts on Ethereum are immutable, `DELEGATECALL` provides an option to alter a contract's behavior post-deployment by delegating certain function calls to separate interchangeable contracts. [9]

Log entries. Log entries are used to communicate information about CAs' code execution to entities outside of the smart contract.[10] A log entry is depicted in Listing 1 between lines 44 and 54. This particular log entry has three topics. The first topic usually contains a hashed signature of the event. The remaining topics hold up to three indexed parameters specified in the event definition, usually encoded. Log entries have to be specified in smart contract code in order to be emitted. Within limits, developers can choose what information shall be exposed with log entries. If smart contracts operate with ERC-tokens, however, developers are advised to implement standard interfaces, including certain sets of events, e. g., to document token creation and transfers (ERC-20[11] , ERC-721[12]).

6 Data Extraction Method (RQ1)

The goal of our data extraction method is to gather as much data about a DApp as possible with minimum knowledge about the DApp. Therefore, the

[8] https://eips.ethereum.org/EIPS/eip-2535, accessed 20-03-2025

[9] https://eips.ethereum.org/EIPS/eip-7, accessed 20-03-2025

[10] https://ethereum.org/en/developers/docs/smart-contracts/anatomy/#events-and-logs, accessed 20-03-2025

[11] https://eips.ethereum.org/EIPS/eip-20#events, accessed 20-03-2025

[12] https://eips.ethereum.org/EIPS/eip-721#specification, accessed 20-03-2025

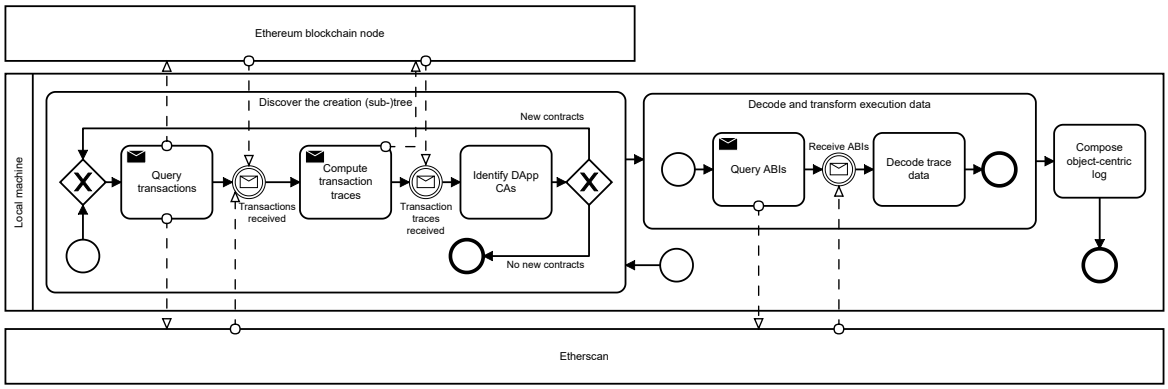


Figure 4: Extraction method for retrieving transaction trace-based event data.

extraction method has to be capable of identifying DApp information from a small amount of input data. Hence, the extraction method takes as input *a non-empty set of accounts of a DApp*, and *a block range*. In order to extract the execution data of the DApp, three steps follow: (1) discover the creation sub-tree(s) of the DApp, (2) decode and transform execution data, and (3) compose object-centric event log. Figure 4 visualizes the extraction method that is described in this section.

Discover the creation sub-tree(s) of a DApp. We begin with the *input set of accounts of the DApp* and *a specified block range*. All operations are confined to this block range. First, we retrieve transactions involving any of the input accounts as a sender or receiver. These transactions are queried both as regular transactions and internal transactions from Etherscan, as well as log entry transactions from an Ethereum node. Next, we identify transactions that contain `CALL`s from or to input accounts. The identified transactions are then replayed using an EVM on an Ethereum Archival node to obtain complete execution traces, including `CREATE` relationships between accounts. To achieve this, we use a build-in debug tracer with the call tracer setting configured to output log entries.[13] We then perform a forward and backward search for DApp accounts along the creation sub-tree to identify additional DApp accounts. Specifically, based on `CREATE`-relations between accounts we identify: a) Child CAs: If the trace data includes contract creations by known DApp accounts, these newly created CAs *(children)* are added to the DApp account set. b) Parent accounts: If the trace data shows that accounts already identified as part of the DApp were created by other accounts, these creator accounts *(parents)* are added to the DApp account set. Whenever previously unknown accounts are added to the DApp account set, the previous steps are repeated. This iterative process is repeated until the entire creation sub-tree of the DApp is identified within the specified block range. If the input set contains accounts from multiple creation sub-trees, multiple creation sub-trees may be discovered. The resulting transaction traces are further processed in the next step.

Decode and transform execution data. Apart from `CREATE`-relations, the previously replayed transaction traces contain `CALL`s, `DELEGATECALL`s, and log entries. The `CALL` / `DELEGATECALL` parameters, and log entry parameters within this data are encoded. We utilize the corresponding contract's ABI specifications to decode the raw data of log entries and function calls with existing libraries.[14] [15] ABIs can be inserted manually or if publicly available, they can be queried from Etherscan. Since some of the ERC standard events were not documented in

[13] https://geth.ethereum.org/docs/interacting-with-geth/rpc/ns-debug#debug_tracetransaction, accessed 20-03-2025

[14] https://github.com/iamdefinitelyahuman/eth-event/blob/master/eth_event/main.py, accessed 20-03-2025

[15] https://web3py.readthedocs.io/en/latest/web3.contract.html#web3.contract.Contract.decode_function_input, accessed 20-03-2025

the ABIs, we created custom ABIs and attempted to decode log entry values as a fallback option. The extracted data was organized into a tabular form, with each row representing a `CALL`, a `DELEGATECALL`, a `CREATE`, or a log entry. The extraction method makes use of the third-party service Etherscan,[16] which provides access to Ethereum data. We used Etherscan to retrieve the transactions and ABIs of identified DApp CAs. The implementation of the data extraction method is available.[17]

Compose object-centric event log

After decoding, the blockchain data needs to be transferred into an OCEL 2.0 event log. In Table 3 we mapped components of Ethereum data (see Figure 2) to elements of the OCEL 2.0 metamodel (see Figure 1). This mapping categorizes blockchain operations into the three primary types: `CREATE`, `CALL/DELEGATECALL`, and `LOG ENTRY`. Each of these operations is systematically mapped to corresponding OCEL 2.0 elements. E.g., the `CREATE` operation, which signifies contract creation, is mapped to an OCEL event with the event type "Contract creation" and includes event attributes derived from callframe (`CF-A`), transaction (`TX-A`), and block attributes (`B-A`). Similarly, `CALL` and `DELEGATECALL` operations are mapped based on decoded function names and their parameters, while log entry operations are associated with decoded event names and relevant log parameters. Object-to-object relationships are captured through qualifiers that describe the nature of interactions between entities, such as "creates" or "owned by". Algorithm 1 is an algorithmic description of how the mapping is applied to ensure the conversion from blockchain data to an OCEL 2.0 log. To convert the tabular blockchain data to OCEL 2.0, we use the PM4Py library's OCEL package[18] . Note that while little domain knowledge is needed to extract the event log data, a certain level of domain knowledge is still needed to format the data to OCEL 2.0 (e. g., for choosing data attributes as objects). The extracted object-centric event log is available on Zenodo.[19]

Algorithm 1 OCEL 2.0 event log composition

Require: Transaction traces T in blocks B, and mapping m
Ensure: OCEL 2.0 log L with events, objects, and relationships

Initialize empty OCEL 2.0 log L
for each transaction trace $t \in T$ **do**
  Extract operations O_t from t
  for each operation $op \in O_t$ **do**
    Create Event e:
      $e \leftarrow m(op)(evtype, eatype, eaval)$
      assign e block timeStamp $time_b$
    for each related object o **do**
      if Object $o.id$ exists in L **then**
        update o:
          $o \leftarrow m(op)(objtype, oatype, oaval)$
          Update ro history with $time_b$
      else
        create o:
          $o \leftarrow m(op)(objtype, oatype, oaval)$
          Initialize ro history with $time_b$
      end if
      Establish event-to-object relationship $e2o$:
        $e2o \leftarrow m(op)(E2O)$
    end for
    for each object-to-object relationship $o2o \in m(op)(O2O)$ **do**
      Establish object-to-object relationship:
        $o2o \leftarrow m(op)(O2O)$
    end for
    Add $e, o, e2o, o2o$ to L
  end for
end for

7 Distinct Characteristics of Blockchain Execution Data

In this section, we present distinct characteristics within the blockchain transaction traces. In a second step, we intend to check if these characteristics can be represented in OCEL 2.0 without loss or duplication of information. Therefore, we answer *RQ2*.

Ordering of Operations. Operations within transaction traces have no direct timestamps. Therefore, a challenge lies in preserving the temporal and hierarchical integrity of operations within the transaction traces. Temporal ordering of operations in Ethereum is only possible with the block timestamps. Each transaction included in

[16] https://etherscan.io/, accessed 20-03-2025
[17] https://github.com/rhobeck/trace_based_logging, accessed 20-03-2025
[18] https://pm4py-source.readthedocs.io/en/latest/pm4py.objects.ocel.html, accessed 20-03-2025
[19] https://zenodo.org/records/14228751, accessed 20-03-2025

Table 3: Mapping blockchain data to OCEL 2.0 elements

OCEL 2.0 Element	**CREATE**	**CALL / DELEGATECALL**	**LOG ENTRY**
time	Block `timestamp`	Block `timestamp`	Block `timestamp`
Event *E*	`CREATE` occurs	`CALL` or `DELEGATECALL` occurs	`LOG ENTRY` occurs
evtype	Contract creation	Decoded `CALL` / `DELEGATECALL` function name	Decoded `LOG ENTRY` name
eatype	`CF-A`, `TX-A`, `B-A`	Decoded `CALL` / `DELEGATECALL` parameters, `CF-A`, `TX-A`, `B-A`	Decoded `LOG ENTRY` parameters, `TX-A`, `B-A`
eaval	"create new contract", `CF-A` values, `TX-A` values, `B-A` values	Decoded `CALL` / `DELEGATECALL` parameter values, `CF-A` values, `TX-A` values, `B-A` values	Decoded `LOG ENTRY` parameter values, `TX-A` values, `B-A` values
E2O	"creator" (`from`), "created contract" (`to`)	"caller" (`from`), "callee" (`to`), qualifiers based on `CALL` / `DELEGATECALL` parameters	Qualifiers based on `LOG ENTRY` parameters
Object *O*	Entities involved during `CREATE`	Entities involved during `CALL` / `DELEGATECALL`	Entities involved in `LOG ENTRY`
objtype	Role in DApp (e. g., based on smart contract logic)	Domain-specific object types from `CALL` / `DELEGATECALL` parameters (e. g., `MARKET`, `ORDER`)	Domain-specific object types from `LOG ENTRY` parameters (e. g., `MARKET`, `ORDER`, `TOKEN`), log entry emitter
oatype	Address, type of account	Domain-specific object attributes from `CALL` / `DELEGATECALL` parameters (e. g., for `MARKET`: `MarketType`, `EndTime`, `Price`)	Domain-specific object attributes from `LOG ENTRY` parameters (e. g., for `TOKEN`: `TokenId`, `TokenType`), address
oaval	Address value, "EOA"/"CA"	Address values, domain-specific `CALL` / `DELEGATECALL` attribute values (e. g., for `MarketType`: "binary market")	Address values, domain-specific `LOG ENTRY` attribute values (e. g., for `TokenType`: type value), address value
O2O	Relation during creation ("creates", "created by")	domain-specific relations (e. g., for `MARKET`: "report submitted by")	domain-specific relations (e. g., for `TOKEN`: "owned by")

`B-A` ... `Block attributes`: `blockNumber`
`TX-A` ... `Transactions attributes`: `transactionHash`, `transactionIndex`
`CF-A` ... `Callframe attributes`: `from`, `to`, `gas`, `gasUsed`, `value`, `input`, `output`, and `error`

a block inherits the block's timestamp, causing a form-based event capture (Suriadi et al. 2017). Additionally, within a block, transactions have a strict order[20] that is documented as *transactionIndex* as part of the transaction receipt.[21] Operations within a transaction trace lack individual timestamps or pre-defined ordering attributes but follow a tree-structured execution logic. When transforming a transaction trace into an event log format, the order of operations can be preserved in attributes using two approaches:

Sequential enumeration (see *tracePos* in Listing 2): A straightforward option is to enumerate operations from the beginning to the end of the transaction trace. While preserving a basic order of execution and offering a simple secondary ordering attribute, this option does not capture the hierarchical structure of a transaction trace.

Hierarchical enumeration (see *tracePosDepth* in Listing 2): A more nuanced option is to enumerate operations based on their hierarchical depth, preserving the tree structure of nested calls within the trace. This method assigns identifiers that reflect both the sequential position and the depth within the trace hierarchy by assigning the position of the parent operation as a prefix.

In part, the representation in sub-processes is relevant to accurately represent Ethereum-specific implementations. E.g., `DELEGATECALL`s are not annotated with ETH transfers, although ETH transfers could happen. Instead, a `DELEGATECALL` "propagates the sender and value from the parent

[20] https://ethereum.org/en/developers/docs/blocks/, accessed 20-03-2025
[21] https://web3py.readthedocs.io/en/stable/web3.eth.html#web3.eth.Eth.get_transaction_receipt, accessed 20-03-2025

Listing 2: Schematic transaction trace with inserted position options

```
{
  'type': 'CALL', 'tracePosDepth': '1', 'tracePos': 1, '
      ↪ calls': [
    {'type': 'CALL', 'tracePosDepth': '1.1', 'tracePos':
        ↪  2, 'calls': [
      {'type': 'CALL', 'tracePosDepth': '1.1.1', '
          ↪ tracePos': 3},
      {'type': 'DELEGATECALL', 'tracePosDepth': '1.1.2
          ↪ ', 'tracePos': 4, 'calls': [
        {'type': 'CALL', 'tracePosDepth': '1.1.2.1',
            ↪  'tracePos': 5}
      ]
      }
    ],
    'logs': [
      {'topics': ['0xA', '0xB'], 'tracePosDepth': '1.1
          ↪ .3', 'tracePos': 6},
      {'topics': ['0xC'], 'tracePosDepth': '1.1.4', '
          ↪ tracePos': 7}
    ]
  },
  {'type': 'CALL', 'tracePosDepth': '1.2', 'tracePos':
      ↪  8}
],
'logs': [
  {'topics': ['0xD'], 'tracePosDepth': '1.3', '
      ↪ tracePos': 9}
]
}
```

Notes: Enabled log tracing.

scope [i. e., the call the delegated call is nested within] to the child scope [i. e., the delegated call]".[22] This association information is lost if the event log data does not explicitly capture delegated calls in the context of their parent calls.

Undecodable Data. The transaction traces contain encoded data that requires the ABIs of the corresponding CAs for decoding. However, challenges emerge from incomplete or unavailable ABIs. ABIs may not document every public function and log entry of a smart contract. E. g., before the recent introduction of Solidity v0.8.20, log entries according to the ERC standard, as well as log entries emitted through invoked code from imported libraries, were not included in an ABI. For other smart contracts, ABIs may not be publicly accessible altogether. As an example: Using the ABI of CA 0x75...99, the log entry in Listing 1 between lines 44 and 54 can be decoded to log entry name: "UniverseCreated" with the parameters "parentUniverse" and "childUniverse" giving the log entry semantic meaning. Without the ABI that decoding would not have been possible. To reduce the amount of undecodable data and to expand options for token tracking, ABIs of DApp CAs can be extended by adding standard events of ERC tokens.[23] This extension allows decoding ERC-token standard events irrespective of the availability and completeness of a CA's ABI. Nonetheless, without the ABI a number of function calls and log entries cannot be decoded. As a result, function names, parameters, or log entry details remain encoded. Yet, undecodable data contains information, incl. *type* of call, *gas*, *from* address, *to* address, and *value* of transferred ETH. That information can be included in the event log for further analysis.

Identification of Objects and Object Types. Identifying objects and object types in the Ethereum environment remains largely case-specific and poses challenges to an automated approach. However, certain object types are consistently applicable on Ethereum. For example, independent from the specific DApp, EOAs can serve as users, CAs serve as code executing units, or tokens are transferrable assets. In this context, distinguishing between EOAs, DApp CAs, and non-DApp CAs presents a challenge. Both EOAs and CAs are represented by a 42-character hexadecimal address that does not differentiate between the two types of entities. One way to identify an EOA is to recognize the account as the initiator of a transaction, a role that is reserved for EOAs. Another option is to query the bytecode for an account; for CAs, the query will return bytecode. For EOAs, the query will return *0x*.[24] Furthermore, knowing the `CREATE`-relations among the DApp CAs allows to differentiate between DApp CAs and a Non-DApp CAs. Differentiating between the types of CAs helps to diversify objects in an object-centric event log. Access to a CA's smart contract code can help to understand a CA's

[22] https://eips.ethereum.org/EIPS/eip-7, accessed 20-03-2025

[23] https://ethereum.org/en/developers/docs/standards/tokens/, accessed 20-03-2025

[24] https://web3py.readthedocs.io/en/v5/web3.eth.html#web3.eth.Eth.get_code, accessed 20-03-2025, note that the query also returns *0x* for self-destructed contracts and contracts of which the most recent creating transaction failed.

functionality and provide context to its operations. For example, function calls to a CA that is known to govern a type of token or a dispute process will be related to the token or dispute process. Code analysis can be particularly helpful in the context of factory CAs. Factory CAs deploy multiple CAs that each inherit the same functionality from the factory. Therefore, knowing the factory CA's functionality and the `CREATE`-relation helps to contextualize the factory's deployments and their operations and group deployed CAs by their functionality.

The implication is that an address, when chosen as an object type, can represent multiple object types. This can cause complications when annotating object types. An address could refer to the generic object type *DApp CA* and, more specifically, to the object sub-type *Dispute* if disputes are handled by dedicated CAs. Additionally, EOAs can assume dynamic roles throughout the process execution, introducing further complications in object-type annotation in the event log. For example, a user can function as *market creator* for one event (event A) and as a *reporter* for another event (event B). This transition poses a challenge in the object annotation when aiming to 1) accurately represent the EOA's involvement as an object type, 2) capture the EOA's role as a market creator in one event (event A), and 3) capture the EOA's subsequent role as a reporter in another event (event B), while preventing information loss or duplicating the EOA's address in the log for specification – which would introduce object convergence.

Handling Reverted Transactions. Transactions can fail for different reasons, such as insufficient gas or invalid execution instructions. Failed transactions revert, i. e., all changes made to the ledger by the execution of the failed transactions are set to the state prior to the transaction (Wood 2014). Despite reversion, failed transactions are processed (mined) and consequently are part of the blockchain (Xu et al. 2019, p.230). This inclusion enables the recomputation of traces for failed transactions. However, since changes to the blockchain by failed transactions are reverted, the operations of reverted transactions need to be excluded from subsequent analysis. Hence, such transactions should be filtered out before applying process mining techniques. The trace of a reverted transaction is depicted in Listing 3; the error was initially noted in line 16 and subsequently escalated back to line 7.

Listing 3: Reverted example transaction trace

```
1  {
2    'from': '0x51bf919cc3af947265b0a94820445e9322b375a3',
3    'gas': '0x15298',
4    'gasUsed': '0x70c4',
5    'to': '0x1985365e9f78359a9b6ad760e32412f4a445e862',
6    'input': '0xa9059cbb000000000000000000000000d4e63d63a4
         ↪ 18140636155b41ccae365f95a15c520000000000000000
         ↪ 0000000000000000000000000000000012dacafbb5e7c3
         ↪ 80',
7    'error': 'execution reverted',
8    'calls': [
9      {
10       // ...
11       'from': '0x1985365e9f78359a9b6ad760e32412f4a445e86
             ↪ 2',
12       'gas': '0x13d9c',
13       'gasUsed': '0x59d',
14       'to': '0x6c114b96b7a0e679c2594e3884f11526797e43d1'
             ↪ ,
15       'input': '0xa9059cbb000000000000000000000000d4e63d
             ↪ 63a418140636155b41ccae365f95a15c5200000000
             ↪ 0000000000000000000000000000000000000012
             ↪ dacafbb5e7c380',
16       'error': 'execution reverted',
17       'type': 'DELEGATECALL'
18     }
19   ],
20   'value': '0x0',
21   'type': 'CALL'
22 }
```

Notes: Transaction hash: 0x9ab3c2831caa81d58344b3fbbd74f3a90caf386eead08169-bdff61226da8afad.

8 Case Study: Augur Prediction Market

After exploring the extraction method and blockchain data characteristics, the following section moves towards the application of OCEL 2.0 to data of a real-world DApp: the betting market platform Augur (v1.0) (Peterson et al. 2018). This case study serves two purposes:

1. *Evaluate the practicality of the data extraction method* (*RQ3*): By applying our method to a complex, real-world DApp, we assess its ability to extract meaningful data from a dynamically evolving blockchain environment. We also investigate how the identified challenges in data extraction and representation (Section 7) manifest in a practical scenario.

2. *Investigate the analytical benefits of object-centric process mining* (*RQ4*): We explore how the object-centric perspective, enabled by OCEL 2.0, allows us to gain insights into DApp behavior that would not be readily accessible with traditional, case-centric approaches.

Augur was chosen for this case study due to its popularity, public data availability on the Ethereum Mainnet, and the existence of a prior process mining case study using a single-case notion event log (Hobeck et al. 2021). This previous work provides a partial basis for validating our extracted data and comparing the insights gained from different analytical perspectives.

Our data extraction covered the period from Augur's initial deployment on July 8, 2018 (block 5,926,229) to November 10, 2020 (block 11,229,577). Applying the method described in Section 6, we traversed the `CREATE` relations to discover 21,096 Augur-related addresses. We successfully retrieved valid ABIs for 241 of these addresses and decoded a significant portion of the extracted data. The extraction yielded 20,082,623 operations related to Augur, including interactions with external DApps. Focusing specifically on operations executed by Augur's core contracts, we decoded 1,514,608 log entries, 5,225,410 calls, and 4,313,674 delegated calls. Importantly, all 22,772 events from the previous case study (Hobeck et al. 2021) were present in our extracted data, providing a degree of validation for our method.

From the extracted data, we identified the following object types within the Augur ecosystem:

- *EOA (Externally Owned Account):* Represents users who interact with the platform.
- *ORDER:* Represents a buy or sell request for shares in a market.
- *CONTRACT:* Represents an Augur-related smart contract deployed on the blockchain.
- *TOKEN:* Represents a digital asset or a unit of value.
- *MARKET:* Represents a prediction market for betting on the outcome of events.

Representing blockchain data in OCEL 2.0. Addressing *RQ3* we evaluate the adequacy of OCEL 2.0 (Berti et al. 2024) for representing data extracted from Augur. OCEL 2.0 offers constructs like qualified object relationships and evolving object attributes, which are beneficial for object-centric process mining. However, we encountered representational challenges. One challenge is the hierarchical nature of the transaction traces (as illustrated in Listing 2). Transaction traces comprise events in a tree structure, rather than a linear sequence. OCEL 2.0 lacks a mechanism to explicitly represent this tree structure and the relationships between higher- and lower-level events. As a workaround, we introduced new objects within the OCEL 2.0 log to link higher-level events to their corresponding lower-level events. Another limitation relates to the hierarchical structure of object types in blockchain systems. A single object may belong to multiple (sub)object types. E.g., a "Market" in Augur is also a "Contract Account (CA)". Ideally, an object could possess both "Market" and "CA" types simultaneously. OCEL 2.0 currently lacks this feature. Our solution involved creating multiple objects of different types representing the same logical entity, linking them to the same set of events. Despite these limitations, we managed to represent dynamic object roles in the data using qualifications for event-to-object relationships. E.g., a single EOA (user) can act as a "market creator" in one event and as a "reporter" in another. OCEL 2.0's qualifications allow us to capture these evolving roles. It enables additional analysis compared to single-case notion event logs, e. g., to understand interactions and dynamic role changes of objects within DApps.

OCEL lifecycle analysis. We proceed to investigating the analytical benefits of object-centric process mining and analyze the lifecycle of individual object types within Augur using the OCEL 2.0 event log (*RQ4*). Figure 6 depicts histograms of the number of unique activities executed for each object type. The histograms show patterns for different object types. Most EOAs (representing users) perform only 1 or 2 unique activities, indicating consistent and often simple interaction

patterns with the platform. Markets exhibit a wider range of unique activity counts between 11 and 30 activities, with 28 being the most frequent. This suggests that it takes 11 activities to resolve a market free of disputes and complications. If complications occur, the number of activities involved jumps to 17-21 and 26-30 activities, respectively. Orders also show a varying number of unique activities with a leap between 3 and 13 activities, also likely influenced by exception mechanisms such as disputes. Figure 5 shows the lifecycle durations of EOAs and orders. Almost all EOAs stay active on Augur for longer than a week, with around half having a lifespan of more than a month. Most orders are processed within a week, but can take up to a year to be fulfilled.

Object-interaction analysis. Building upon the individual object lifecycle analysis, we explore *RQ4* and examine the interactions *between* different object types within Augur. Drawing from the object relationships captured in the OCEL 2.0 log, we can analyze the cardinality and frequency of interactions, providing a more comprehensive view of the DApp's behavior. As depicted in Figure 7, most EOAs interact with a limited number of markets (primarily one or two), suggesting that users focus their participation on selected markets. Changing the perspective, Figure 8 shows that a single market always involves multiple EOAs, with a peak between 5 and 7, suggesting active participation in most markets. Figure 8 also shows that most Augur CAs are created to serve in only a single market, suggesting that the developer encapsulated market functions using a factory pattern. Figure 9 shows that most orders involve a single

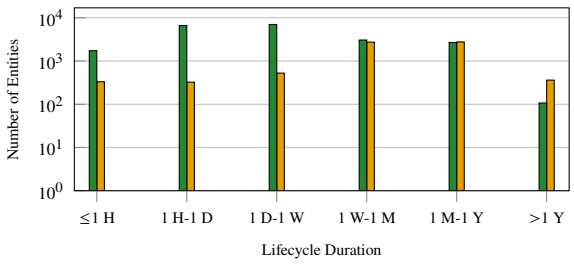


Figure 5: Lifecycle durations of EOAs (orange) and orders (green).

contract. This object-centric interaction analysis enables a more detailed understanding of DApp behavior compared to single-case notion event log analysis, which often focuses on individual process instances rather than the interplay between multiple objects.

The dispute resolution process. To further investigate the analytical capabilities of object-centric process mining, we focus on the dispute resolution process within Augur. This process is triggered when participants challenge the reported outcome of a market event. It provides an example of multi-object interactions.

A dispute resolution process in Augur typically involves the following steps:

- *Initial Reporting*: After a market event has occurred, the designated reporter submits a report on the outcome within a set time frame.
- *Dispute Round*: If the reported outcome is disputed, the market enters a dispute round. Participants stake tokens on what they believe is the correct outcome.
- *Outcome Shift*: If enough tokens are staked on an alternative outcome, the market outcome shifts to this new outcome.
- *Further Disputes*: If the new outcome is disputed, the process repeats, with each dispute round requiring more tokens to shift the outcome again.
- *Finalization*: If no disputes occur within a dispute round or if a dispute bond cannot be filled, the market finalizes with the current outcome. The winning tokens are then distributed to the correct reporters.

We selected a subset of activities and object types to represent the dispute resolution process. The selected activities include *delegate call to report*, *call to log initial report submission*, *create dispute*, *call to create dispute*, *contribute to dispute*, *redeem dispute*, *complete dispute*, *call to log market finalization*, *redeem as initial reporter*, and *redeem dispute crowdsourcer*. Furthermore, we focus on *MARKET* and *ORDER* as object types, considering their roles in the context of market

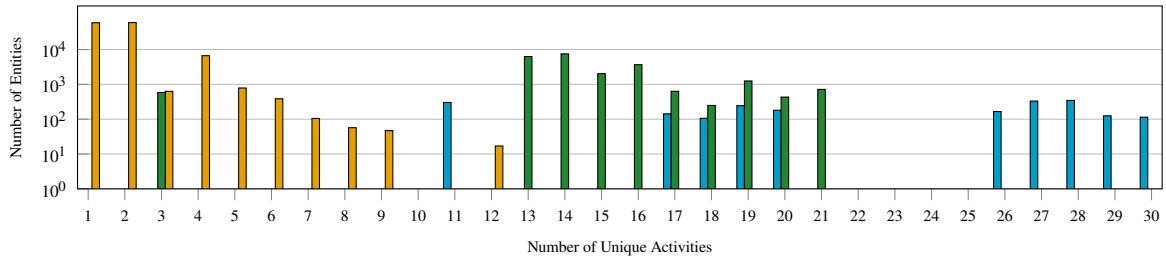


Figure 6: Unique activities performed by EOAs (orange), per order (green), and per market (blue).

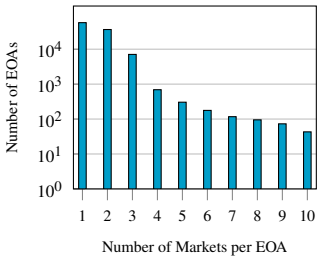


Figure 7: Distribution of markets based on the count of EOAs operating in each market.

Figure 8: Distribution of markets by the number of EOAs (orange) and contracts (red) involved.

Figure 9: Distribution of orders, categorized by the number of contracts associated with each order.

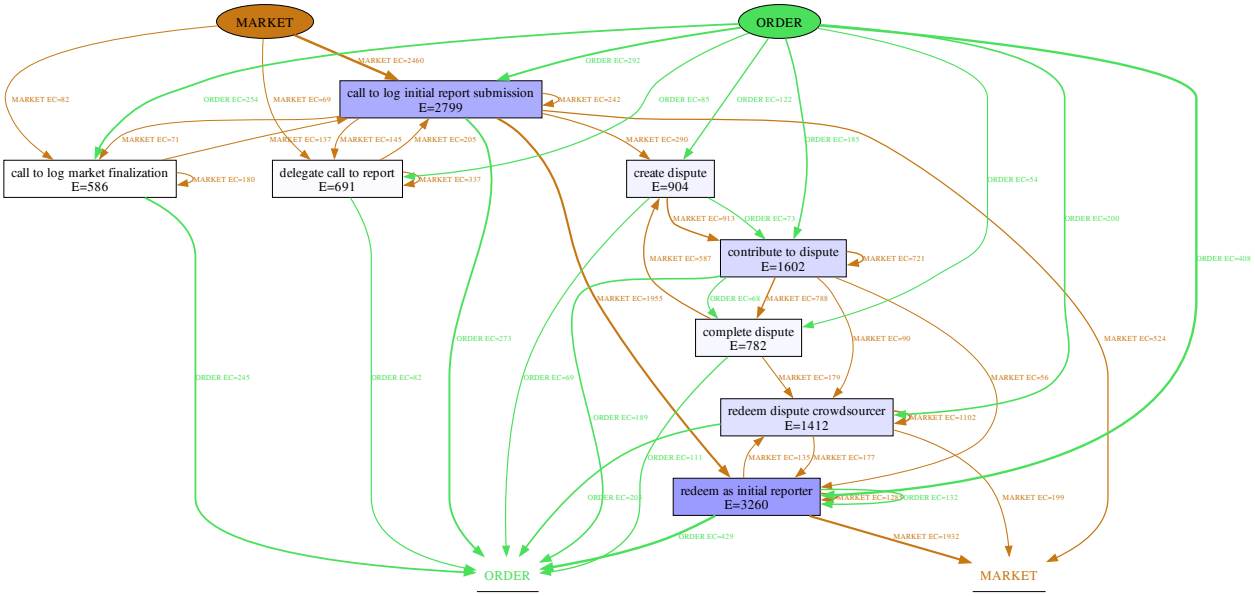
disputes. *MARKET* objects represent a link to the specific events being bet on, while *ORDER* objects are involved in the creation and fulfillment of bets within these markets. Figure 10a and Figure 10b show two object-centric directly-follows graphs, respectively annotated with frequency and performance (median of the arcs' performance) information. An object-centric directly-follows graph is a collaboration of the directly-follows graphs discovered for the single object types (Berti and Aalst 2023). The (frequency-based) object-centric directly-follows graph proposed in Figure 10a shows that it is less frequent to have disputes than not. Also, there are frequent loops over all the activities involved in the market object type (e. g., many users may claim redemption or contribute to disputes). Also, we show that for 587 times a dispute has been opened for the same market after another dispute has just been closed. According to our extraction, the event with activity *complete dispute* may be skipped, indicating that other events could alternatively lead to the conclusion of disputes. The figures also show that orders can be made at any time during a dispute so users can place tokens on either outcome.

9 Discussion

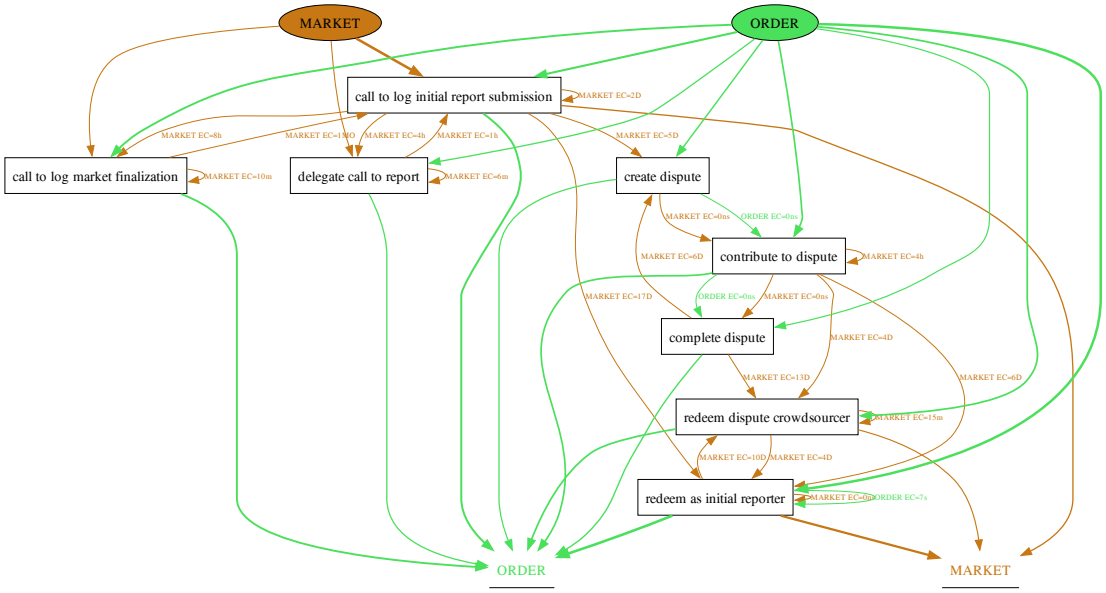
During the implementation of the data extraction method and the data handling with OCEL 2.0 we encountered observations and challenges we reflect on in this section.

Data dependency and extraction. The current implementation of the data extraction method (presented in Section 6 and responding to *RQ1*) relies on the third-party service Etherscan. The Etherscan source code is not fully public, so its use for data retrieval reduces the transparency of the research process. However, that dependence was accepted for two reasons:

- *Etherscan is an indexed data store of the Ethereum blockchain*. In order to reduce the size of the Ethereum blockchain, only the results of transactions are stored on-chain, whereas the operations leading up to the transactions' outcomes (transaction traces) are not. For querying data from a blockchain archival node, that means that for a CA 0xA, only those transactions that have 0xA as a sender or receiver of a transaction in a certain block range can be queried. It is also possible that CA 0xA received a function call during the execution of



(a) Frequency object-centric directly-follows graph of the dispute resolution process.



(b) Performance object-centric directly-follows graph of the dispute resolution process.

Figure 10: Object-centric directly-follows graphs

a transaction from the sender EOA 0xB to the receiver CA 0xC. In that case, the function call 0xC → 0xA is not stored on-chain. Indexing services such as Etherscan provide data stores that save (a subset of) such function calls and make them easily queriable. We tried to mitigate the dependence and validate the Etherscan data by querying transactions of log entries (which are emitted upon function call, not upon reception of a transaction) from an Ethereum node. However, the dependence on an indexed data store remains. It is possible to circumvent using a third-party service by creating a local database containing message calls between all active accounts within a block range.

- *Retrieving DApp CA's ABIs*. Log entry data, as well as function names and inputs, are stored and encoded on the blockchain. Decoding the data requires information from the CA's ABI. Etherscan provides a number of verified ABIs for CAs, which we used. If the smart contract code of the CA is known, the ABI can also be computed by a Solidity compiler.

Client-specific implementations. The data extraction was implemented with the Geth-based Ethereum client Erigon so that the rich features of Geth were available for re-computations of transaction traces. We cannot make a statement about reproducibility with other clients except Geth and Erigon. However, both clients have a combined market share of about 46% of all Ethereum clients, making the approach accessible for a large share of node operators.[25]

Transferability to other blockchains. The presentation of the paper's approach is based on the technology of Ethereum blockchains. Some of the concepts we used are transferable to other blockchains. For one, executable code is also deployed as smart contracts in other second-generation blockchains, e. g., Hyperledger Fabric.[26] Also, the notion of transactions of assets is a concept shared by all blockchains (Xu et al. 2019, p.5) and can be exploited to retrieve object-centric process mining logs across platforms.

Challenges in data handling and data analysis. Section 7 explores the challenges related to the handling of blockchain data in process mining, particularly the ordering of operations and the integration of comprehensive smart contract data. Alternative methods are required to maintain temporal accuracy, such as sequential and hierarchical enumeration. Additionally, the dynamic roles of accounts and the complexity of smart contract interactions present challenges in applying the OCEL 2.0 standard. Despite these challenges, OCEL 2.0 provides a structured format that supports the nuanced requirements of blockchain data. The Augur case study investigated the practical application of the data extraction method and the analytical benefits of using OCEL 2.0 for object-centric process mining for blockchain applications. The analysis of object lifecycles, interactions, and the dispute resolution process provided insights into the behavior and implementation of the Augur DApp, which would be difficult to obtain using single-case notion event logs.

10 Conclusion and Future Work

This paper introduced a novel methodology for extracting and analyzing execution data from Ethereum-based DApps using object-centric process mining. We addressed the challenges of dynamic contract deployments, operation ordering within transaction traces, handling undecodable data due to ABI limitations, and representing evolving object types and roles within the OCEL 2.0 framework. While OCEL 2.0 effectively captures the multi-object interaction characteristics of DApps, limitations regarding hierarchical object typing and event granularity give reason for further development of the standard. Our proposed workarounds provide practical solutions to these limitations, enabling more comprehensive DApp analysis. The Augur case study showed the practical application of our method and the analytical benefits of an object-centric perspective.

[25] https://clientdiversity.org/#distribution, accessed 20-03-2025

[26] https://hyperledger-fabric.readthedocs.io/en/latest/smartcontract/smartcontract.html, accessed 20-03-2025

By analyzing object lifecycles, interactions, and the dispute resolution process within Augur, we gained additional insights offered by the object-centric approach – insights not readily attainable with traditional, case-centric process mining techniques. While some of the technical details of our work are specific to Ethereum DApps, we estimate that many insights and aspects apply to the broader class of dynamic software applications in general.

Future work can improve the transparency and autonomy of data extraction processes, e. g., by using or creating open-source blockchain indexing databases. The discovery of DApp contracts is currently limited to traversing the `CREATE` relations between accounts, and can be improved to handle DApps consisting of multiple address sub-trees that interact only via `CALL` / `DELEGATECALL` with no linking `CREATE` relations. Future work may also automate object discovery in the raw extraction data to reduce the manual effort necessary to construct the OCEL 2.0 log. Additionally, the capabilities of OCEL 2.0 can be extended to better handle the specific requirements of blockchain data, such as the ordering of operations and role dynamics.

References

van der Aalst W. M. P. (2019) Object-Centric Process Mining: Dealing with Divergence and Convergence in Event Data. In: Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings. Lecture Notes in Computer Science Vol. 11724. Springer, pp. 3–25

van der Aalst W. M. P. (2022) Process Mining: A 360 Degree Overview. In: Process Mining Handbook. Lecture Notes in Business Information Processing Vol. 448. Springer, pp. 3–34

Acampora G., Vitiello A., Stefano B. N. D., van der Aalst W. M. P., Günther C. W., Verbeek E. (2017) IEEE 1849: The XES Standard: The Second IEEE Standard Sponsored by IEEE Computational Intelligence Society [Society Briefs]. In: IEEE Comput. Intell. Mag. 12(2), pp. 4–8

Alzhrani F. E., Saeedi K., Zhao L. (2024) A process-aware framework to support Process Mining from blockchain applications. In: J. King Saud Univ. Comput. Inf. Sci. 36(2), p. 101956

Beck P., Bockrath H., Knoche T., Digtiar M., Petrich T., Romanchenko D., Hobeck R., Pufahl L., Klinkmüller C., Weber I. (2021) BLF: A Blockchain Logging Framework for Mining Blockchain Data. In: Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track at BPM 2021 co-located with 19th International Conference on Business Process Management (BPM 2021), Rome, Italy, September 6th - to - 10th, 2021. CEUR Workshop Proceedings Vol. 2973. CEUR-WS.org, pp. 111–115

Berti A., van der Aalst W. M. P. (2023) OC-PM: analyzing object-centric event logs and process models. In: Int. J. Softw. Tools Technol. Transf. 25(1), pp. 1–17

Berti A., Koren I., Adams J. N., Park G., Knopp B., Graves N., Rafiei M., Liß L., Unterberg L. T. G., Zhang Y., Schwanen C., Pegoraro M., van der Aalst W. M. P. (2024) OCEL (Object-Centric Event Log) 2.0 Specification

Buterin V. et al. (2014) A next-generation smart contract and decentralized application platform. In: white paper 3(37), pp. 1–37

Corradini F., Marcantoni F., Morichetta A., Polini A., Re B., Sampaolo M. (2019) Enabling Auditing of Smart Contracts Through Process Mining. In: From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday. Lecture Notes in Computer Science Vol. 11865. Springer, pp. 467–480

Corradini F., Marcelletti A., Morichetta A., Re B. (2024) A Data Extraction Methodology for Ethereum Smart Contracts. In: 2024 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops), pp. 524–529

Fahland D. (2022) Process Mining over Multiple Behavioral Dimensions with Event Knowledge Graphs In: Process Mining Handbook Springer International Publishing, pp. 274–319

Ghahfarokhi A. F., Park G., Berti A., van der Aalst W. M. P. (2021) OCEL: A Standard for Object-Centric Event Logs. In: New Trends in Database and Information Systems - ADBIS 2021 Short Papers, Doctoral Consortium and Workshops: DOING, SIMPDA, MADEISD, MegaData, CAoNS, Tartu, Estonia, August 24-26, 2021, Proceedings. Communications in Computer and Information Science Vol. 1450. Springer, pp. 169–175

Goossens A., De Smedt J., Vanthienen J. (2024). In: Business Process Management Forum. Springer Nature Switzerland, pp. 37–54

Goossens A., De Smedt J., Vanthienen J., van der Aalst W. M. P. (2023) Enhancing Data-Awareness of Object-Centric Event Logs. In: Process Mining Workshops. Springer Nature Switzerland, pp. 18–30

Hobeck R., Klinkmüller C., Bandara H. M. N. D., Weber I., van der Aalst W. (2024) On the Suitability of Process Mining for Enhancing Transparency of Blockchain Applications. In: Business & Information Systems Engineering

Hobeck R., Klinkmüller C., Bandara H. M. N. D., Weber I., van der Aalst W. M. P. (2021) Process Mining on Blockchain Data: A Case Study of Augur. In: Business Process Management - 19th International Conference, BPM 2021, Rome, Italy, September 06-10, 2021, Proceedings. Lecture Notes in Computer Science Vol. 12875. Springer, pp. 306–323

Hobeck R., Weber I. (2023) Towards Object-Centric Process Mining for Blockchain Applications. In: Business Process Management: Blockchain, Robotic Process Automation and Educators Forum. Springer Nature Switzerland, pp. 51–65

Klinkmüller C., Ponomarev A., Tran A. B., Weber I., van der Aalst W. M. P. (2019) Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications. In: Business Process Management: Blockchain and Central and Eastern Europe Forum - BPM 2019 Blockchain and CEE Forum, Vienna, Austria, September 1-6, 2019, Proceedings. Lecture Notes in Business Information Processing Vol. 361. Springer, pp. 71–86

Li G., de Murillas E. G. L., de Carvalho R. M., van der Aalst W. M. P. (2018) Extracting Object-Centric Event Logs to Support Process Mining on Databases. In: Information Systems in the Big Data Era. Springer International Publishing, pp. 182–199

Moctar M'Baba L., Assy N., Sellami M., Gaaloul W., Farouk Nanne M. (2023) Process mining for artifact-centric blockchain applications. In: Simulation Modelling Practice and Theory 127, p. 102779

Moctar-M'Baba L., Assy N., Sellami M., Gaaloul W., Nanne M. F. (2022a) Extracting Artifact-Centric Event Logs From Blockchain Applications. In: IEEE International Conference on Services Computing, SCC 2022, Barcelona, Spain, July 10-16, 2022. IEEE, pp. 274–283

Moctar-M'Baba L., Sellami M., Gaaloul W., Nanne M. F. (2022b) Blockchain logging for process mining: a systematic review. In: 55th Hawaii International Conference on System Sciences, HICSS 2022, Virtual Event / Maui, Hawaii, USA, January 4-7, 2022. ScholarSpace, pp. 1–10

Morichetta A., Paoloni Y., Re B. (2024) Event log extraction methodology for ethereum applications. In: Future Generation Computer Systems, p. 107566

Mühlberger R., Bachhofner S., Ciccio C. D., García-Bañuelos L., López-Pintado O. (2019) Extracting Event Logs for Process Mining from Data Stored on the Blockchain. In: Business Process Management Workshops - BPM 2019 International Workshops, Vienna, Austria, September

1-6, 2019, Revised Selected Papers. Lecture Notes in Business Information Processing Vol. 362. Springer, pp. 690–703

Nakamoto S. (2008) Bitcoin: A peer-to-peer electronic cash system. In: Satoshi Nakamoto

Peterson J., Krug J., Zoltu M., Williams A. K., Alexander S. (July 2018) Augur: A Decentralized Oracle and Prediction Market Platform.. Forecast Foundation. Last Access: accessed 2021-01-05

Suriadi S., Andrews R., ter Hofstede A. H. M., Wynn M. T. (2017) Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. In: Inf. Syst. 64, pp. 132–150

Wood G. (2014) Ethereum: A secure decentralised generalised transaction ledger. In: Ethereum project yellow paper 151(2014), pp. 1–32

Xu X., Pautasso C., Zhu L., Lu Q., Weber I. (2018) A Pattern Collection for Blockchain-based Applications. In: Proceedings of the 23rd European Conference on Pattern Languages of Programs, EuroPLoP 2018, Irsee, Germany, July 04-08, 2018. ACM, 3:1–3:20

Xu X., Weber I., Staples M. (2019) Architecture for Blockchain Applications. Springer