algorithm [13]. Unlike cloud platforms, open and permissionless blockchains such as Ethereum or Cardano [14] offer transparent execution that is verifiable by the distributed nodes and is particularly suited for decentralized applications [2]. Smart contracts can be utilized for verifiable computation to achieve (a) execution not dependent on centralized technical infrastructures and (b) non-centralized coordination verified by distributed nodes [3]. Cloud platforms, such as Amazon Web Services (AWS) or Microsoft Azure, provide infrastructure services, software platforms, or application services [15]. Infrastructure concerns specific physical or virtual servers, while it is hidden in services for platforms and applications to provide abstraction, especially through lambda functions and serverless computing [16]. Here, execution is specified modularly and with a functional paradigm, permitting per-module or per-function distribution for availability, reliability, and scalability in terms of parallel execution on distributed servers.

Over the past years, Transport Layer Security (TLS) has become the de-facto standard for providing authentication and encryption on the web [17]. Established through Public Key Infrastructure (PKI), X.509 certificates are applied for certificate signing, asymmetric encryption, and the preparation of symmetric encryption of end-to-end connections. With security assumptions being predicated on the presence of trusted authorities, the provided security and authentication are outside the scope of blockchain transactions. Even though blockchain-based authentication has been suggested before, including applications such as IoT [18], Voting [19], Single Sign On [20], as well as PKI supported by blockchains [21], [22], authentication is not provided for execution aspects on blockchains and web architectures. For this purpose, blockchains could be involved in key generation and signatures combined with X.509 certificates for web-based authentication as suggested by the proposed architecture.

On blockchains, executable models can be applied. Related work in this area suggests execution and tracking by monitoring run-time behaviour or tracing it back in time. Models are used primarily in the execution of business processes and workflows [23]–[26], ontologies [27], and state machines [28], [29], in addition to attestation [30], [31], and instance tracking [3], [8]. The attestation concept with instance tracking is utilized in this paper for capturing states of models over time and for metadata distribution, allowing distributed parties to monitor and verify. Decentralized application execution is achieved by the proposed architecture, based on prior work [3], with cloud platforms and a blockchain extended by web servers and certificates.

III. Proposed System Architecture

The following subsections describe the system structure, concepts, and procedures for execution and instance tracking.

A. System Structure

In Figure 1, the architecture of the proposed system is shown. Components of the architecture are clients, controlling and tracking the execution, cloud platforms, web servers, and blockchain platforms with a smart contract to distribute and register execution states with their respective clients based on public keys and certificates.

A `Client` represents one of the distributed parties involved in executing or tracking decentralized applications. For initiating an execution, an *Executable Model* is prepared within the client application. Specific to the cloud platform, the model defines the execution logic, e.g., in terms of AWS Step Function models written in a custom Amazon States Language [32] or executable processes and workflows defined by the Business Process Model and Notation 2.0 standard [33]. For each instance at run-time, the `Instance Tracking` component logs individual states entered during execution within the cloud platform and verifies states using data from a blockchain node.

The `Cloud Platform` executes the platform-specific model within an engine such as AWS Step Functions or AWS Lambda. Control flow and data are controlled by the client and reflected in each instance state reached during execution. The platform is required to execute the control flow and functional specifications with data provided by the model such that discrete state data can be derived. For example, individual lambda functions with their input and output data, state machines, or other state-based representations might be used. Furthermore, the platform is required to provide distribution and scalability as well as interfaces for interacting with a blockchain API such as web3.js [34]. For this purpose, additional platform services or virtual machines might be used.

The `Blockchain Platform` hosts a smart contract for registering execution states and clients with certificates as well as for notifying clients when entering a state. It is required for the smart contract to operate on an open and permissionless blockchain network autonomously. The selected network must support smart contracts with the capability to issue and store client accounts based on public-private key pairs, e.g., Ethereum or Avalanche [35].

The `Web Server` publishes states in the form of an instance protocol. Over time, the protocol is appended by clients controlling the execution. In parallel, clients track the execution of an instance and subscribe in order to observe execution behaviour. For authentication and encryption, the server is required to support X.509 certificates and TLS connections as outlined in the following processes.

B. Execution and Instance Tracking

The procedure and overall lifecycle is given corresponding to Figure 1 by the steps 1. deployment, 2. running instances with key generation and the creation and signing of certificates, 3. entering and publishing states, 4. tracking and verifying states, and 5. the termination of instances.

1. Deploy: The executable model is deployed by a client on the cloud platform's execution engine, resulting in data object $m = (model\_id, model\_data)$. The platform-specific $model\_data$ contains the definition of the execution. For example, a state machine in JSON syntax [32].

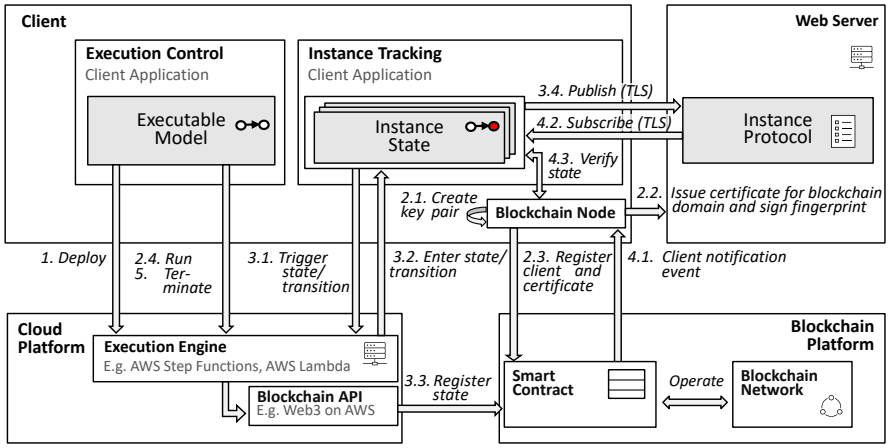

Fig. 1. Architecture for the model-based execution and instance tracking of decentralized applications, secured by certificates and blockchain-based authentication. Clients 1. deploy executable models, 2. run instances with creation of public-private key pairs for the blockchain and certificate issuance, 3. enter and publish states in a web-hosted instance protocol as well as a blockchain smart contract, 4. track and verify states by receiving notification events for obtaining states, subscribing to new states, and verifying states through public keys, and 5. terminate instances.

The $model\_id$ is generated as a content-based identifier by $model\_id = h(model\_data)$ with hash function h, e.g., SHA-256 [36] for verification (step 4.).

2. Running instances:

2.1. Create key pair: The client invokes a key generation function in the local blockchain node, capable of creating a public-private key pair $k = (k_{pub}, k_{prv})$ with client account $a_c = address(k_{pub})$, where $address$ is a derivation function obtaining an address from a public key (see, e.g., [34, pp. 69-70, 73-74]). By including the blockchain node in the client component, key generation is initiated and performed locally at the client without any server interaction. In case of Ethereum, a key derivation function for use with ECDSA is applied for a 256 bit key and the secp256k1 curve [34].

2.2. Issue certificate for blockchain domain and sign fingerprint: The client issues an X.509 certificate for use with TLS at the web server through a state-of-the-art ECC or RSA implementation compatible with the client. The certificate crt is created with common name CN matching the domain of the web server's uri. In turn, uri is linked to the client address a_c through a blockchain domain registrar, e.g., the Ethereum Name Service (ENS) [37]. Finally, a fingerprint fp is created for crt using a state-of-the-art hash function such as SHA-256, i.e., $fp = h(crt)$, and signed with the private key of the client as in $f_{sig} = sign(fp, k_{prv})$. The blockchain domain registration must have been prepared by (a) registering a domain d as in $register\_name(d, a_c)$ such that d can be resolved to a_c, (b) setting d as primary name for a_c as in $set\_primary\_name(a_c, d)$ such that a_c allows looking up d, and (c) adding one record with the url under d as in $add\_record(d, url)$ such that the web server and instance protocol will be reachable at url when given d. The $add\_record(d, url)$ function stores url as a text record under domain d at the blockchain domain registrar[1]. Within a governmental or other organizational structure, a higher-level certificate might be used in addition to signing the certificate, e.g., initiated by a certificate signing request (CSR).

2.3. Register client and certificate: The client registers an account address, fingerprint, and signature with the smart contract through the blockchain node. For this purpose, registration function $register\_certificate(a_c, fp, fp_{sig})$ is invoked by the client in a blockchain transaction storing the transmitted information with the sender's account address a_s if $a_s == a_c$. The function stores fp and fp_{sig} in mapping data structures of the smart contract under the key a_c. In addition, the smart contract provides a retrieval function $(fp, fp_{sig}) = get\_certificate(a_c)$ that returns the fingerprint and signature. The function looks up fp and fp_{sig} in the data structures of the smart contract with key a_c.

2.4. Run: Execution is started from the client side. In the execution engine, an instance is then created, resulting in instance data object $i = (model\_id, instance\_id, instance\_data)$ to be received by the client. Depending on the cloud platform, $instance\_data$ is composed out of further identifiers and metadata for the geographic region or other execution parameters. The $instance\_id$ is generated as a content-based identifier by $instance\_id = h(instance\_data)$ for verification.

[1]For ENS see, e.g.: app.ens.domains/c-acd398d9f25c40b1d292bff2190a08d7d907c568.eth?tab=records

3. Entering and publishing states:
   3.1. Trigger state / transition: The client triggers entering a new state or transition at the cloud platform. At this point, the execution engine is invoked and the client awaits the state change.
   3.2. Enter state / transition: The execution engine has changed the state, resulting in client-side state data $s = (instance\_id, state\_id, state\_data)$ for the platform-specific model instance. The $state\_id$ is generated as a content-based identifier using $state\_id = h(state\_data)$ for verification at a later point in time. State representations of the platform in $state\_data$ can include a set of executed functions with input and output data, a state of a state machine, or similar state-based representations (see, e.g., Amazon State Language [32]).
   3.3. Register state: The client registers the entered state with identifiers of the model, instance, and state by invoking the smart contract function $register\_state(model\_id, instance\_id, state\_id)$. In $register\_state$, the IDs in the form of hash values are stored with the transaction sender a_s in the smart contract if the transaction sender matches the client-provided address, i.e., if $a_s == a_c$. In addition, the smart contract provides retrieval functions $instance\_id = get\_instance(state\_id)$ returning an instance ID for a state ID and $model\_id = get\_model(instance\_id)$ which is returning a model ID for an instance ID. Further, the client's address can be retrieved through $a_c = get\_client(state\_id)$
   3.4. Publish (TLS): The client appends the server-side instance protocol available under domain d. The newly reached state is stored with the state, instance, and model data as $published\_state = (s, i, m)$ such that any client can obtain it. TLS is applied for authentication and encryption of the TCP client-server connection.
4. Tracking and verifying states:
   4.1. Client notification event: For each entered state, any client involved in the execution listens and receives notification event $e = (state\_id, a_c)$ through the blockchain node from the smart contract. Given a_c, the blockchain-based domain registrar is invoked to obtain $uri = lookup(a_c)$, where the web server is available and allows for subscriptions. The event data $(state\_id, a_c)$ is stored locally.
   4.2. Subscribe (TLS): With event e, the client subscribes to the instance protocol available online at uri. By subscription, the instance protocol is obtained as a set of published states $\{S \mid published\_state \in S\}$ that is updated over time. TLS is applied for authentication and encryption of the TCP client-server connection. The connection is created if the client successfully validates the certificate.
   4.3. Verify state: Given the published states subsequently received through the subscription, the client verifies the prior existence of each state with the state data received from client notification events through the blockchain node. For each state $state\_id$ of the instance protocol, the smart contract is called using the blockchain node. In particular, corresponding IDs are obtained with the smart contract functions $instance\_id = get\_instance(state\_id)$ and $model\_id = get\_model(instance\_id)$. A state is verified if the following conditions are true.
      4.3.1. The web server url is present in a record under the address registered with the blockchain domain registrar, i.e., $uri == lookup(a_c)$.
      4.3.2. The certificate fingerprint registered with the smart contract is signed by a_c, obtained with $(fp, fp_{sig}) = get\_certificate(a_c)$, and matches the server-provided certificate crt in terms of the fingerprint, i.e., if $verify\_signature(fp_{sig}, a_c)$ and $fp == h(crt)$ are true.
      4.3.3. The $published\_state = (s, i, m)$ allows to establish the prior existence of the state with the corresponding instance and model, i.e., if $h(s) == state\_id$ and $h(i) == instance\_id$ and $h(m) == model\_id$ are true.
      4.3.4. The state has been registered by the client, i.e., if the smart contract function $a_c == get\_client(s)$ is true.

      The instance protocol is verified if all states are verified.
5. Terminate: The client identified by a_c terminates the execution at the cloud platform. At this point, the execution engine and blockchain API no longer register states for $instance\_id$ with the smart contract.

IV. Implementation

This section demonstrates the feasibility of implementation using the AWS cloud platform and the Ethereum blockchain. A prototype application has been created for the client in addition to a web server, AWS deployments of Step Functions models, and an Ethereum smart contract. At the client side, execution control and instance tracking are realized in Python 3.9 and a PostgreSQL 15 database implementation for storing instance states. The smart contract is implemented in Solidity 0.8.18 for registering and distributing events. All components are available online together with deployment information on the Ethereum Sepolia testnet, the client account for smart contract interactions and signing of certificates, the ENS name and domain name registrations, the configured web server, and the model, instance, and state data of executions[2].

A. Executable Models on AWS

An executable model for the use case of parallel data processing in a distributed scenario is depicted in Figure 2, both as a State Machine Diagram (a) and in JSON format (b).

[2]See github.com/fhaer/Itrex-Engine-Event-Processing

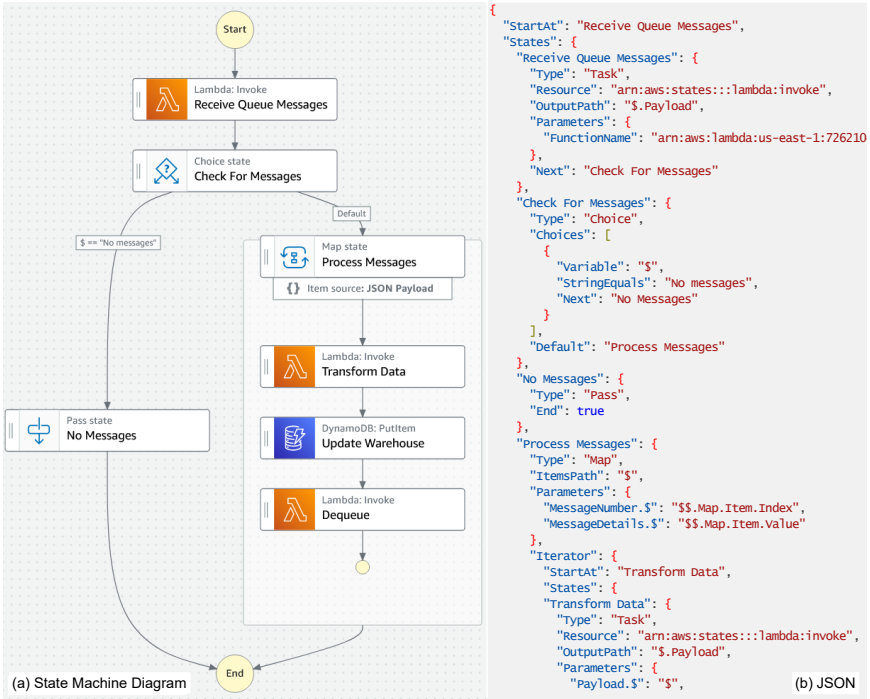

Fig. 2. AWS Step Functions model for parallel data processing in a distributed scenario, where data is received through a message queue, transformed, and stored in a data warehouse. The model is shown as (a) state machine diagram and (b) JSON representation. A hash value h is used for identification based on the model content and for verification.

Starting with a message queue, data is received, transformed with parallel processing, and stored in a data warehouse. Oftentimes, this pattern is found in distributed applications, e.g., involving compute- and data-intensive processing of IoT data, business transactions, industrial system data, or analytics and machine learning. On the cloud platform, services composed as executable models here utilize Step Function models. Model elements are task states with service interactions, initial and final states, and additional elements for mapping and control flow. Using AWS services, messages are received via AWS Lambda with an SQS message queue and processed concurrently with data transformation in Lambda, updating data warehouse records in DynamoDB, and dequeuing in Lambda. Additional tests incorporated further step function models for machine learning training and IoT device data processing.

B. Prototype

Two operating modes are distinguished in the client. (a) Execution control with model deployment and instantiation as well as triggering and tracking of AWS execution events for Step Function states and their smart contract registration. (b) Instance tracking and analysis for publishing, listening, and subscribing to execution states at the client side as well as the server-side instance protocol.

1) Execution Control on AWS and Smart Contract Calls: The client application first deploys executable models of the aforementioned use case. At this point, the client creates a public-private key pair with the blockchain node and a certificate for the previously provisioned domain and web server. In testing, a standard browser-compatible certificate was issued using RSA with 2048 bit in the form of a crt file. The file was applied to SHA-256 for the fingerprint and signed with the client's private key based on ECDSA with a 256 bit key. The blockchain domain registration was carried out with ENS for the domain c-acd398d9f25c40b1d292bff2190a08d7d907c568.eth through transactions signed with the client's private key[3].

Subsequent state transitions are triggered and observed in AWS CloudWatch event logs. The observed events include model deployments, running instances, entering states, and terminating instances. While observing events, each state is stored with corresponding state, instance, and model data, processed in JSON format, and normalized regarding the formatting. This data is the input of the SHA-256 function that calculates the content-based identifiers.

Figure 2 shows one of the deployed models with JSON data and the calculated hash value h, indicated at the bottom of the Figure. The JSON data of instances encompasses the Amazon Resource Names (ARN) of the execution and state machine, including the AWS region, with additional metadata such as timestamps of the instance start and termination, name, and status. States are represented with JSON data containing identifiers, names, timestamps, and state types, as well as the input data and output data of each state in the execution trace. States are dependent on pre-states and uniquely represented.

[3]See app.ens.domains/c-acd398d9f25c40b1d292bff2190a08d7d907c568.eth

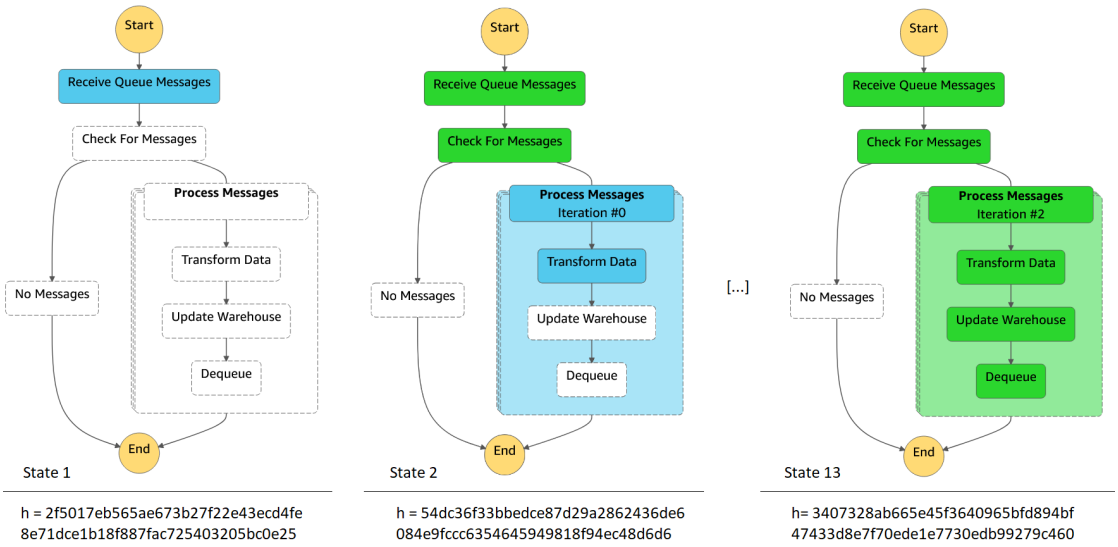

Fig. 3. AWS Step Function instance created from the model shown in Fig. 2. The instance consists of 13 states for parallel data processing, executed in multiple concurrent iterations. By chance, the last state of the instance belongs to "Iteration #2".

Calculating hash values for the instance protocol involves multiple states, thus, individually calculated state hash values are summarized in a root hash value of a Merkle tree [34]. In terms of performance, the client-side hash function calculations resulted in a negligible delay in the millisecond range.

For model, instance, and state data of one of the instances, Figure 3 visualizes the state machine on the AWS platform. When the execution engine enters a new state of an instance, the client captures the corresponding event and calculates h, as indicated in the figure. When entering a state, the client calls the smart contract's registration function for storing the identifier data based on the hash values. By this action, clients are notified through a contract event. Finally, the hash values are published together with model, instance, and state JSON data by sending HTTP PUT requests to the server.

2) Instance Tracking and Instance Protocols: From the client point-of-view, execution events are observed in order to store instance states locally with state, instance, and model data such that an instance protocol can be published for other clients. Upon the notification by an event, the client stores the ENS domain, client account, and state identifier locally in PostgreSQL and sends an HTTP GET request to subscribe to the server-side instance protocol. When sending the request, certificate validation is carried out before the server retrieves and responds with the registered model, instance, or state JSON data. For verification with the blockchain node, hash values are calculated, stored as identifiers, and compared to the locally stored model, instance, and state identifiers originating from the smart contract. In this way, the verification follows the concept of blockchain-based attestation [31], where validity is established through (a) integrity, by re-calculation and comparison of hash values on the client side, (b) the existence of the data at a prior point in time, and (c) the client address, requiring it to be present as transaction sender for the state registration function of the smart contract.

An example for the execution states of an instance h = *a681[...]779d* and model h = *c632[...]81e1* is shown in Figure 4. It lists 13 locally captured instance states that are stored in a client-side database as well as the server-side instance protocol using JSON. The initial state machine model is given in Figure 2 and results in the 13 states corresponding to Figure 3. These states of the type "task state" perform interactions with services for computation or data operations. State 1 executes the "Receive Queue Message" lambda function for receiving data messages, registered at time 1678443257 (row 1, Figure 4). The conditional "Check For Messages" is evaluated "true" and starts the parallel processing element "Process Messages". Tasks within this element are executed for each message. In this case, four messages started concurrent iterations of the tasks "Transform Data", "Update Warehouse", and "Dequeue". In particular, a lambda function converts data types and formats in "Transform Data", a "put" operation stores data in a data warehouse using the DynamoDB database, and "Dequeue" finally removes the message from the queue. Each task execution over the four iterations has been captured in a state. E.g., state 2 shows the "Transform Data" task in "Iteration #0" occurred first and was registered 11 s after the initial state (row 2). State 13 reveals that the final task of "Iteration #2" occurred last, 81 s after the first state (row 13). The corresponding blockchain transactions are stored in the Ethereum Sepolia testnet under the client account[4].

V. Discussion

This paper investigates how blockchain and cloud-based execution can be combined for decentralized applications while providing scalability and authentication through blockchains

[4]See sepolia.etherscan.io/address/0xAcD398d9F25C40b1d292bfF2190A08D7D907c568

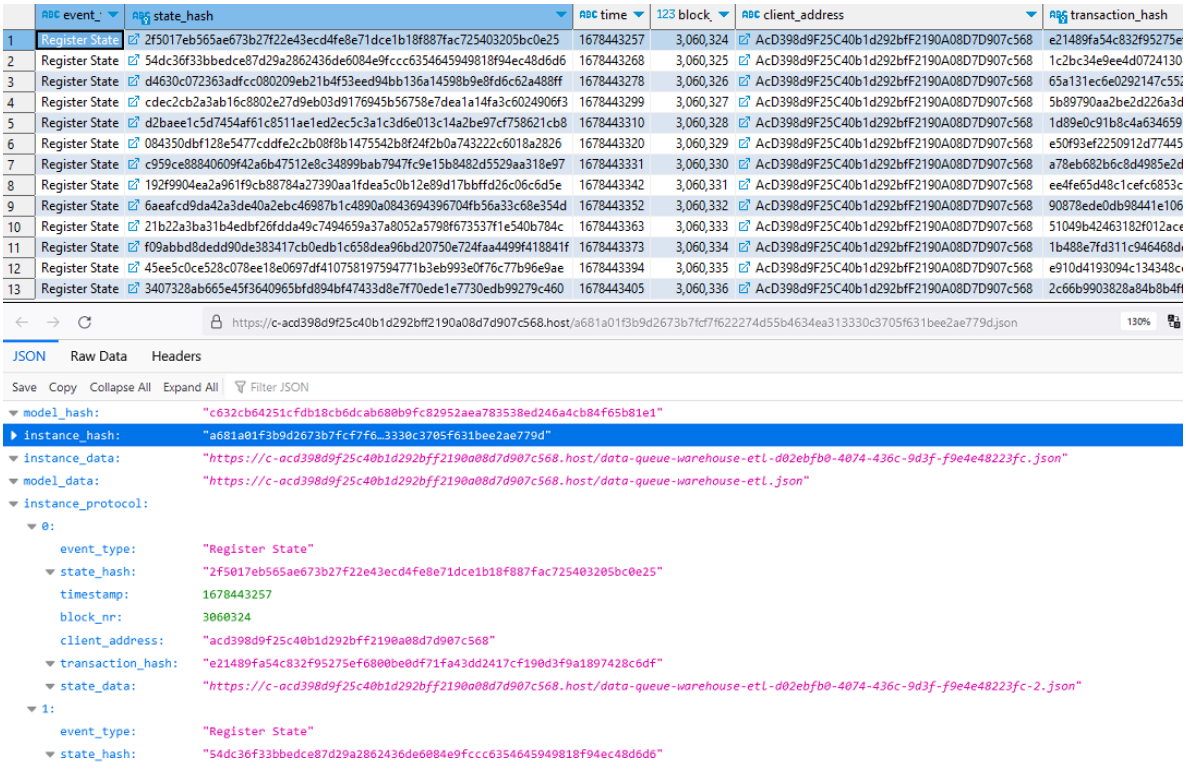

Fig. 4. Execution states of the instance shown in Fig. 3. The table displays local instance states, stored in a PostgreSQL 15 database at the client. The web browser below shows the instance protocol stored at the web server in JSON data. With completed verification of the local states, the client- and server-side data matches. The server-side data is obtained through the domain c-acd398d9f25c40b1d292bff2190a08d7d907c568.eth registered in ENS with client account address 0xAcD398d9F25C40b1d292bfF2190A08D7D907c568 and URI record https://c-acd398d9f25c40b1d292bff2190a08d7d907c568.host.

and certificates. In order to address this research objective, instance tracking [3], [8] and attestation [31] approaches are applied with an extended architecture and prototype implementation.

As a first result, the architecture describes the components required for executing models on cloud platforms together with recording execution states using a smart contract and instance protocols on a web server.

Secondly, the architecture shows the possibility of combining blockchain- and certificate-based validation and authentication by (a) linking blockchain account addresses with their public keys to domain names, e.g., using ENS, (b) linking the account addresses to server-side data referenced by hash values in a smart contract, (c) and verifying the published data at distributed clients using the hash values from the blockchain in combination with authentication established through the account address of the domain, the certificate, and the certificate signature registered with the smart contract.

Thirdly, regarding decentralized applications, the architecture permits (a) the recording of instance protocols locally for each distributed party in a verifiable manner, (b) the tracking of states, instances, and models, and (c) the analysis of past executions. In these use cases, the feasibility of implementation could be positively evaluated on the Ethereum blockchain and AWS with Step Function models.

In summary, the architecture combines cloud platforms and blockchains towards enabling scalable and secure applications. Regarding the support of decentralized applications, cloud platforms carry out the execution as centralized entities, however, they do provide execution control for distributed parties through role concepts, geographic distribution, and redundancy through serverless computing models such as lambda functions and state machines. The execution and instance tracking approach supports these concepts and might be extended to address further aspects related to decentralization such as control and feedback mechanisms, or governance. An implication of the demonstrated architecture is that decentralized applications can utilize blockchains and smart contracts for decentralization aspects in combination with cloud platforms and serverless computing for secure and scalable applications.

VI. Conclusion and Outlook

Overall, the proposed architecture suggests decentralized applications can be supported through model-based execution and instance tracking on blockchains and cloud platforms with scalability and authentication. The enhanced performance and security properties obtained through this approach could be beneficial for data- and compute-intensive applications such as machine learning, IoT operations, workflow automation, or for collaborations and decentralized organizations. Future research will focus on evaluating further decentralization properties for the coordination of data- and compute-intensive executions.

References

[1] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The Serverless Computing Survey: A Technical Primer for Design Architecture," *ACM Computing Surveys*, vol. 54, no. 10s, 2022, 10.1145/3508360.

[2] K. Wu, Y. Ma, G. Huang, and X. Liu, "A first look at blockchain-based decentralized applications," *Software: Practice and Experience*, vol. 51, no. 10, 2021, 10.1002/spe.2751.

[3] F. Härer, "Executable Models and Instance Tracking for Decentralized Applications - Towards an Architecture Based on Blockchains and Cloud Platforms," in *Proceedings of the PoEM 2022 Workshops and Models at Work Co-Located with Practice of Enterprise Modelling 2022*, ser. CEUR, vol. 3298, 2022.

[4] S. Wang, W. Ding, J. Li, Y. Yuan, L. Ouyang, and F.-Y. Wang, "Decentralized Autonomous Organizations: Concept, Model, and Applications," *IEEE Transactions on Computational Social Systems*, vol. 6, no. 5, 2019, 10.1109/TCSS.2019.2938190.

[5] F. Härer, *Integrierte Entwicklung Und Ausführung von Prozessen in Dezentralen Organisationen. Ein Vorschlag Auf Basis Der Blockchain. Dissertation*. University of Bamberg Press, 2019, 10.20378/irbo-55721.

[6] ——, "Process Modeling in Decentralized Organizations Utilizing Blockchain Consensus," *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 15, 2020, 10.18417/emisa.15.13.

[7] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung, "Decentralized Applications: The Blockchain-Empowered Software System," *IEEE Access*, vol. 6, 2018, 10.1109/ACCESS.2018.2870644.

[8] F. Härer, "Decentralized Business Process Modeling and Instance Tracking Secured By a Blockchain," in *Proceedings of the 26th European Conference on Information Systems (ECIS)*. AIS Electronic Library (AISeL), 2018.

[9] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008, accessed on 2023-03-12. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[10] V. Buterin, "Ethereum: The Ultimate Smart Contract and Decentralized Application Platform," 2013, accessed on 2023-03-12. [Online]. Available: http://web.archive.org/web/20131228111141/http://vbuterin.com/ethereum.html

[11] Q. Wang, J. Yu, S. Chen, and Y. Xiang, "SoK: DAG-based Blockchain Systems," *ACM Computing Surveys*, vol. 55, no. 12, 2023, 10.1145/3576899.

[12] S. Bouraga, "A taxonomy of blockchain consensus protocols: A survey and classification framework," *Expert Systems with Applications*, vol. 168, 2021, 10.1016/j.eswa.2020.114384.

[13] M. Dotan, Y.-A. Pignolet, S. Schmid, S. Tochner, and A. Zohar, "Survey on Blockchain Networking: Context, State-of-the-Art, Challenges," *ACM Computing Surveys*, vol. 54, no. 5, 2021, 10.1145/3453161.

[14] F. Härer, "Towards Interoperability of Open and Permissionless Blockchains: A Cross-Chain Query Language," in *2022 IEEE International Conference on E-Business Engineering (ICEBE)*, 2022, 10.1109/ICEBE55470.2022.00041.

[15] F. Nadeem, "Evaluating and Ranking Cloud IaaS, PaaS and SaaS Models Based on Functional and Non-Functional Key Performance Indicators," *IEEE Access*, vol. 10, 2022, 10.1109/ACCESS.2022.3182688.

[16] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *Journal of Cloud Computing*, vol. 10, no. 1, 2021, 10.1186/s13677-021-00253-7.

[17] R. Holz, J. Hiller, J. Amann, A. Razaghpanah, T. Jost, N. Vallina-Rodriguez, and O. Hohlfeld, "Tracking the deployment of TLS 1.3 on the web: a story of experimentation and centralization," *ACM SIGCOMM Computer Communication Review*, vol. 50, no. 3, pp. 3–15, Jul. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3411740.3411742 10.1145/3411740.3411742.

[18] A. Garba, D. Khoury, P. Balian, S. Haddad, J. Sayah, Z. Chen, Z. Guan, H. Hamdan, J. Charafeddine, and K. Al-Mutib, "LightCert4IoTs: Blockchain-Based Lightweight Certificates Authentication for IoT Applications," *IEEE Access*, vol. 11, 2023, 10.1109/ACCESS.2023.3259068.

[19] D. Khoury, E. F. Kfoury, A. Kassem, and H. Harb, "Decentralized Voting Platform Based on Ethereum Blockchain," in *2018 IEEE International Multidisciplinary Conference on Engineering Technology (IMCET)*, 2018, 10.1109/IMCET.2018.8603050.

[20] S. Patel, A. Sahoo, B. K. Mohanta, S. S. Panda, and D. Jena, "DAuth: A Decentralized Web Authentication System using Ethereum based Blockchain," in *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, 2019, 10.1109/ViTECoN.2019.8899393.

[21] D. Moussaoui, B. Kadri, M. Feham, and B. Ammar Bensaber, "A Distributed Blockchain Based PKI (BCPKI) architecture to enhance privacy in VANET," in *2nd International Workshop on Human-Centric Smart Environments for Health and Well-being (IHSH)*, 2021, 10.1109/IHSH51661.2021.9378727.

[22] F. Li, Z. Liu, T. Li, H. Ju, H. Wang, and H. Zhou, "Privacy-aware PKI model with strong forward security," *International Journal of Intelligent Systems*, vol. 37, no. 12, 2022, 10.1002/int.22283.

[23] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling, "Untrusted Business Process Monitoring and Execution Using Blockchain," in *14th International Conference, Business Process Management (BPM 2016)*. Springer, 2016, 10.1007/978-3-319-45348-4_19.

[24] J. A. Garcia-Garcia, N. Sánchez-Gómez, D. Lizcano, M. J. Escalona, and T. Wojdyński, "Using Blockchain to Improve Collaborative Business Process Management: Systematic Literature Review," *IEEE Access*, vol. 8, 2020, 10.1109/ACCESS.2020.3013911.

[25] O. López-Pintado, L. Garćıa-Bañuelos, M. Dumas, I. Weber, and A. Ponomarev, "Caterpillar: A business process execution engine on the Ethereum blockchain," *Software: Practice and Experience*, vol. 49, no. 7, 2019, 10.1002/spe.2702.

[26] S. Curty, F. Härer, and H.-G. Fill, "Blockchain application development using model-driven engineering and low-code platforms: A survey," in *Enterprise, Business-Process and Information Systems Modeling, EMMSAD 2022*. Springer, 2022, 10.1007/978-3-031-07475-2_14.

[27] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, and A. Das, "Auto-Generation of Smart Contracts from Domain-Specific Ontologies and Semantic Rules," in *2018 IEEE International Conferences on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data*, 2018, 10.1109/Cybermatics_2018.2018.00183.

[28] H. Nakamura, K. Miyamoto, and M. Kudo, "Inter-organizational Business Processes Managed by Blockchain," in *Web Information Systems Engineering – WISE 2018*. Springer, 2018, 10.1007/978-3-030-02922-7_1.

[29] A. Mavridou and A. Laszka, "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach," in *Financial Cryptography and Data Security: 22nd International Conference, FC 2018*. Springer, 2018, 10.1007/978-3-662-58387-6_28.

[30] F. Härer and H.-G. Fill, "Decentralized Attestation of Conceptual Models Using the Ethereum Blockchain," in *2019 IEEE 21st Conference on Business Informatics (CBI)*. IEEE, 2019, 10.1109/CBI.2019.00019.

[31] ——, "Decentralized attestation and distribution of information using blockchains and multi-protocol storage," *IEEE Access*, vol. 10, 2022, 10.1109/ACCESS.2022.3150356.

[32] Amazon, "Aws step functions developer guide," 2023, accessed on 2023-03-12. [Online]. Available: https://docs.aws.amazon.com/step-functions/dg/

[33] Object Management Group, "Business Process Model and Notation (BPMN) 2.0.2," 2014, accessed on 2023-03-12. [Online]. Available: https://www.omg.org/spec/BPMN/2.0.2

[34] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building smart contracts and DApps*. O'Reilly Media, 2019.

[35] Härer, "Towards interoperability of open and permissionless blockchains: A cross-chain query language," in *IEEE International Conference on E-business Engineering (ICEBE) 2022*. IEEE, 2022, 10.1109/ICEBE55470.2022.00041.

[36] NIST, "Secure Hash Standard (SHS)," U.S. Department of Commerce, Tech. Rep. Federal Information Processing Standard (FIPS) 180-4, 2015, 10.6028/NIST.FIPS.180-4.

[37] Ethereum Name Service (ENS), "Ens documentation," 2023, accessed on 2023-04-14. [Online]. Available: https://docs.ens.domains/