

Supplementary Material

for

“DORI: A Distributed Observation and Reconciliation Interface for Finality-Aware Event Synchronization in Hybrid Web2–Web3 Applications”

Anonymous¹

¹Anonymous Institution

Submitted to *Springer Nature*

Manuscript ID: XXXX-XXXX-YYYY

August 6, 2026

Overview. This document provides supplementary material accompanying the main manuscript titled “*Distributed Observation and Reconciliation Interface (DORI)*”. It provides deployment details, interoperability mappings, an operational lifecycle walkthrough, extended experimental setup and results, related-work comparisons, and supporting figures and tables omitted from the main manuscript for space.

Contents

S.1Appendix: Unified Frontend Integration for Self-Signed Transactions	3
S.2Appendix: Web 3.0 Proxy Cluster for extended design	3
S.2.1 User Management Cluster Integration	4
S.3Appendix: End-to-End Observability and Traceability	4
S.3.1 Event-Driven vs REST Communication	5
S.3.2 Pseudonymized Log	6
S.4Appendix: Architectural Security Controls	6
S.4.1 Siloed Containerization	6
S.4.2 Unified Ingress Control	6
S.4.3 Encrypted Interservice Communication	7
S.4.4 State Synchronization and Tracing	7
S.5Appendix: Security Observability and Forensics	7
S.5.1 Distributed Tracing for Security Context:	7
S.5.2 Immutable Audit Logs:	7
S.5.3 Pseudonymization and Log Masking:	7
S.5.4 Strict Access Policies (CORS & CSRF)	7
S.6Appendix: Event-Driven Transaction Flow	7
S.6.1 Publish-Subscribe Architecture Using RabbitMQ	8
S.7Appendix: Messaging and Naming Standards	8
S.7.1 Exchange Naming Convention	8
S.7.2 Service Queue Naming Convention	9
S.7.3 Fanout Exchange in Proof of Concept	9
S.7.4 Topic Exchange Pattern	9

S.7.5 Event Naming Convention	9
S.7.6 Appendix: Cross-Platform Event Interoperability Profile Envelope	10
S.8 Detailed Operational Walkthrough of the DORI Lifecycle	11
S.8.0.1 Stage 1 - Blockchain Event Origin	11
S.8.0.2 Stage 2 - Event Stream Delivery	11
S.8.0.3 Stage 3 - Web 2.0 Event Synchronization Service	13
S.8.0.4 Stage 4 - Message Broker Routing and Distribution	13
S.8.0.5 Stage 5 and 6 - Event Consumption and State Synchronization	13
S.9 Appendix: Performance Test Setup Details	14
S.9.1 Resource Utilization	14
S.9.2 Scenario 1: Observation-Source Performance Comparison	14
S.9.3 Scenario 2: Local Concurrent Transactions	14
S.9.4 Scenario 3: Remote Server Stress Test	15
S.9.5 System Resource Constraints Under Stress	16
S.10 Appendix: Related Works	17
S.10.1 Enterprise dApps	17
S.10.2 Hybrid Architecture for dApps	18
S.10.3 Logging	18
S.10.4 Conventional Event Sourcing Pattern	18
S.10.5 Traceability	19
S.11 Appendix: Future Work	19

S.1 Appendix: Unified Frontend Integration for Self-Signed Transactions

In enterprise-grade applications, where user actions often involve both substantial off-chain data and critical on-chain references, managing integration and consistency across diverse services and teams, working with different technology stacks, is a significant challenge. Our approach specifically addresses workflows involving self-signed transactions: when a user submits a form, the off-chain data is included in the API payload, while the user’s wallet generates a transaction hash to serve as an on-chain reference, encapsulated within the payload (typically under an `onChainData` object). To ensure that all relevant services can access and synchronize with this on-chain reference, without requiring developers to orchestrate complex API sequences, this proposed solution employs a dedicated event extraction and distribution mechanism. In our on-premise deployments, this is realized via a Kong plugin, which intercepts API traffic, extracts on-chain references, and publishes them to a public event bus for downstream consumption.

For cloud-native solutions, this logic could be equivalently implemented with a Lambda or serverless function which is illustrated at Figure 2 (the main manuscript). This architecture enforces a clear separation of concerns: each team need only implement their own API endpoints, while synchronization and event dissemination are handled automatically via a public event bus. As a result, the integration process is streamlined, discipline is enforced across teams, and external services can reliably subscribe to public events to stay in sync with critical on-chain actions.

Figure 3 (the main manuscript) showcases this workflow. The process begins when the user interacts with the application interface and submits a transaction through the frontend client. The request (HTTP POST/PUT/PATCH) contains both the `dto` payload (off-chain data) and the `onChainData` object that holds the transaction reference generated by the wallet.

The request first passes through the access phase of **Kong Gateway**, which proxies the API call to the target service. During the access phase, the **Kong Plugin (rabbitmq-publisher)** script validates the `onChainData` fields. If the data is valid, the plugin publishes the JSON message containing the transaction metadata to the RabbitMQ fanout exchange. The **RabbitMQ Fanout Exchange** then broadcasts this event to all subscribed services. Each subscriber receives the same event message and processes it independently according to its internal business logic.

If the `onChainData` is missing or invalid, the plugin logs the attempt and skips publication, allowing the gateway to proceed with the proxied upstream response.

This interaction model demonstrates how the proposed architecture enables synchronous API handling while maintaining asynchronous event propagation. By decoupling message distribution from direct API dependencies, the system ensures reliable and scalable synchronization between Web 2.0 microservices and on-chain events, while preserving fault tolerance and observability through the gateway layer.

S.2 Appendix: Web 3.0 Proxy Cluster for extended design

The proxy backend is designed as a dedicated service cluster, which acts as an abstraction of an integrated smart contract functions exposed via REST API. It also works as an adapter to facilitate inter-service communication between Web 2.0 and Web 3.0 layers.

Though, it’s a common practice to integrate the frontend directly with smart contracts, often facilitated by tools like RainbowKit, which enables clients to sign their own transactions. However, when enterprises or organizations migrate to the Web 3.0 ecosystem, they may utilize an organizational wallet from which employees or stakeholders can perform Web 3.0 transactions on behalf of the organization. In such cases, the wallet’s private key should be securely stored on a Web 2.0 backend server, either as a Key Store or within a Cloud-Based Vault. Users or administrators can then perform actions from the frontend under the standard Web 2.0 authorization process, while the REST API request is handled by the proxy backend service, which is configured with the organizational wallet for delegated server-signed transactions. Essentially, this service acts as an entry point for enterprise administrators to manage their paid transactions via a concept similar to Account Abstraction [1].

S.2.1 User Management Cluster Integration

High availability of this service is necessary. It is designed to integrate seamlessly with enterprise identity providers such as Entra ID (Azure AD), Auth0, and Amazon Cognito, enabling dApps to meet enterprise-grade requirements such as single sign-on (SSO) and multi-factor authentication (MFA). This integration is necessary for an organization, particularly when they need specific users to use custodian wallet or a group of officials who needs to perform on-chain action through the organization wallet address. This type of wallet often fits for those dApps who interacts with Ownable Smart Contracts with specific roles [2]. If any role or group of users needs to perform self-signed actions on-chain, these Identity Provider can be helpful for retrieving their KYC data mapped to their public wallet address.

S.3 Appendix: End-to-End Observability and Traceability

Given the scale and complexity of the system, comprehensive traceability is essential for efficient debugging and system reliability. The proposed observability framework empowers decentralized application (dApp) developers with enhanced visibility into system behavior, thereby facilitating quicker identification and resolution of errors.

As shown in Figure 2 (Main Manuscript) Each service has the distributed tracing and logging sidecars. The sidecars submit the logs and traces to the trace and log collector tempo and loki, running at port 3201 and 3100 respectively. They can be visualized using the Grafana UI running at port 3200.

Since we are running OpenTelemetry, Loki, Tempo, and Grafana on our on-premise servers, we have customized their configurations for high-throughput load testing. Specifically, the OpenTelemetry Collector is tuned with a batch size of 8192, a timeout of 1 second, and an 80% memory usage limit with a 10% spike threshold. In Tempo, the ingestion limits have been increased to handle high trace volumes — allowing up to 1 GB/sec ingestion rate, a 1.5 GB burst size, and a maximum of 100 MB per trace. Similarly, Loki is configured with a 1-hour chunk retention and idle periods. These aggressive configurations were applied to simulate real-world stress scenarios, and developers using on-premise logging and tracing setups are advised to fine-tune these parameters based on available CPU, memory, and disk I/O resources.

In our architecture, events originate on-chain and are propagated into the system either through web-socket streams or cron jobs that periodically query pending transactions from The Graph. To visualize this synchronization lifecycle, we integrated additional instrumentation layers alongside HTTPInstrumentation. For real-time socket communication, we employed WSInstrumentation and SocketIOInstrumentation. However, for cron jobs and scheduled background tasks, we identified the absence of standardized instrumentation mechanisms in OpenTelemetry. As a workaround, we manually instantiate a new trace each time a transaction is fetched from The Graph, ensuring full visibility across asynchronous operations. The lack of built-in instrumentation support for job schedulers remains a known limitation and an open challenge within the current OpenTelemetry ecosystem.

By default, upon receiving an HTTP request at any service endpoint, a unique traceID and corresponding spanID are generated for the request lifecycle. These identifiers are then propagated throughout the application to maintain end-to-end traceability. The traceID persists for the duration of the HTTP request. Meanwhile, the spanID is scoped to each individual layer or function call, expiring once the specific execution context concludes.

As illustrated in Figure S.1, an example trace demonstrates the system’s observability capabilities. A HTTP/POST request is monitored using OpenTelemetry and visualized through Grafana, revealing the full execution path—from NestJS service-layer calls down to database operations. Each trace and span is also associated with corresponding logs, enabling deep contextual insights into each operation. The tracing mechanism in the state-transition diagram in Figure S.1. The transition overview is as follows:

- `/start system` → Program Start
- `/start OTel` → OpenTelemetry Instrumentations
 - `/collect trace` → Tempo Trace Collection
 - `/submit traces` → Grafana Visualizer

- `/start logger` → WinstonLogger
 - `/collect logs` → Loki Log Collection
 - `/submit logs` → Grafana Visualizer
 - `/check logs` → MorganMiddleware
 - * `/submit maskedLogs` → Loki Log Collection
- Grafana Visualizer
 - `/connect tracesToLogs`
 - `/connect logsToTraces`
 - `/stop` → End state

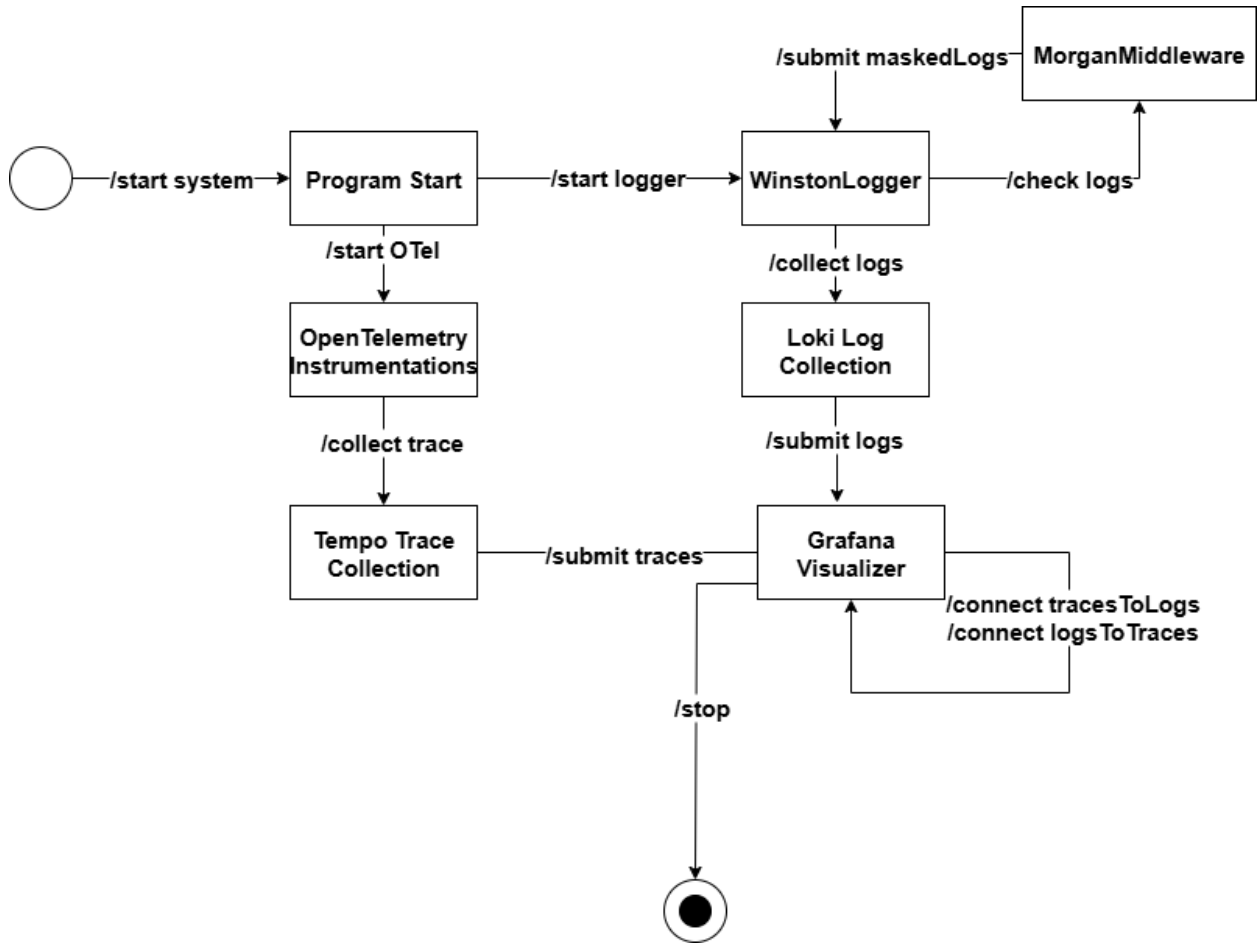


Figure S.1: Distributed Log Tracing Model

S.3.1 Event-Driven vs REST Communication

While there is no strict limitation on using REST APIs for inter-service communication, it may not be the most efficient choice for certain tasks. One core principle of REST is the client-server pattern, designed for handling larger payloads efficiently. However, in event-driven architectures focused on transmitting small data like status changes, a message broker server such as RabbitMQ is typically preferred.

Using a dedicated queue for authorization is a good trade-off, as the payload is just a fixed-length JWT token. Additionally, REST calls (e.g., POST, PUT, PATCH) hold a thread during HTTP transactions, which can strain system resources under load. In contrast, message brokers use the TCP protocol to queue and process messages asynchronously.

Message queues also offer delivery guarantees with retries, unlike REST, which is stateless and more prone to synchronization issues on failure. Services in REST architectures tend to be tightly coupled, whereas messaging systems enable decoupled, event-based microservices. This model improves system resilience and scalability, especially when blockchain-based event data is involved.

External tools can monitor emitted events, analyze logs, and offer off-chain insights—making the event-driven model particularly advantageous in Web3 applications.

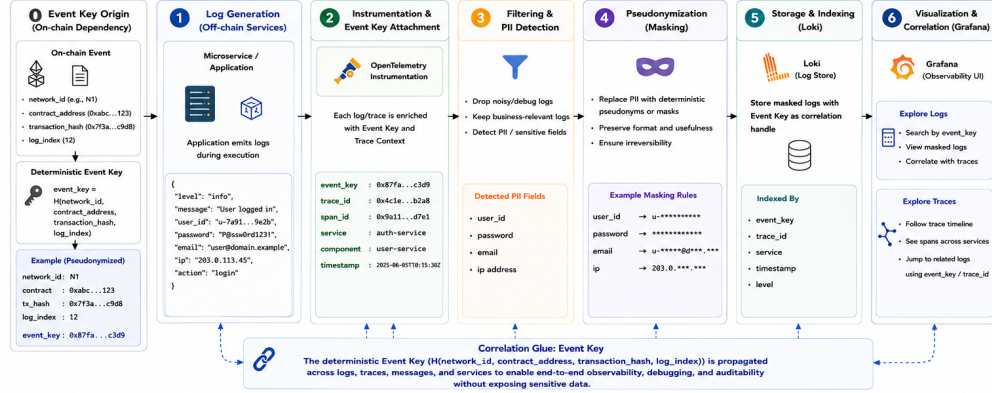


Figure S.2: Proposed Life-cycle of Pseudonymized Off-chain Log Data

S.3.2 Pseudonymized Log

This study recommends implementing *Log Masking* tools to safeguard user’s Personal Identifiable Information (PII) at the off-chain level [3] which is illustrated at Figure S.2. For public chain’s event emitted log, encryption of sensitive information and securing PII-related data with Web 3.0 digital signature standards, such as EIP712, is advised to facilitate future auditing verification. These measures address both dApp interactive Web 2.0 application logs and smart contract event logs.

S.4 Appendix: Architectural Security Controls

Our decentralized application (dApp) operates in a hybrid environment where on-chain smart contracts execute autonomous logic, while off-chain services manage sensitive private keys. Recognizing the inherent risks of this dual paradigm, the architecture adopts a **Zero Trust security model**, guided by the principle: “*Never trust, always verify*”. To uphold this philosophy, the system integrates multiple security controls, outlined below.

S.4.1 Siloed Containerization

All microservices operate inside isolated Docker containers. This ensures that the compromise of any individual component does not grant implicit access to the host machine or to adjacent services.

S.4.2 Unified Ingress Control

The system exposes no direct service endpoints to the public internet. Instead, a centralized API Gateway functions as the sole entry point, abstracting internal infrastructure and serving as the unified **Policy Enforcement Point (PEP)**.

S.4.3 Encrypted Interservice Communication

Implicit trust within the network is eliminated. All service-to-service communication uses **Mutual TLS (mTLS)**, ensuring both encryption and strong authentication between communicating components. No plain HTTP/HTTPS channels are exposed internally.

S.4.4 State Synchronization and Tracing

All on-chain states are replicated and synchronized with the microservices. Additionally, conventional tracing through LGTM, coupled with the use of transaction hashes on-chain, enables not only synchronization of off-chain states but also their verification against on-chain records. This approach empowers the system to both trust the off-chain database and independently verify its consistency with on-chain state.

S.5 Appendix: Security Observability and Forensics

In a distributed system, security relies on visibility. The architecture leverages the Audit Trail Service not just for business logic, but as a primary tool for security forensics and anomaly detection

S.5.1 Distributed Tracing for Security Context:

The system implements OpenTelemetry to ensure end-to-end visibility across all components. Every incoming request to the API Gateway is assigned a unique trace ID, which allows for complete tracking of the request's lifecycle. This trace ID can be followed throughout the entire system, providing transparency and accountability. All activities related to the request are logged and monitored via our audit trail, enabling efficient detection of anomalies or security breaches in real-time.

S.5.2 Immutable Audit Logs:

The system employs an immutable audit trail to create a secure checkpoint for all critical system events. Periodic state updates are captured and securely logged, ensuring that no changes can be made to the logs retroactively. This audit trail serves as a comprehensive record of the system's state over time, providing a robust mechanism for compliance, forensic analysis, and troubleshooting. The full lifecycle of each event is preserved, ensuring that all actions are auditable and traceable, thereby enhancing system integrity and security.

S.5.3 Pseudonymization and Log Masking:

Our log server aggregates logs across different microservices. These logs can often contain sensitive information that may lead to re-identification.

So, we have implemented a log masking middleware to prevent any PII leak. The middleware searches for common PII patterns using regex and redacts them if necessary. Moreover, our pseudoanonymization technique sensitive fields replaced with consistent hashes that the user can use to debug. This has been shown in Figure [S.2](#).

S.5.4 Strict Access Policies (CORS & CSRF)

To reduce exposure to client-side attacks, the API Gateway enforces restrictive Cross-Origin Resource Sharing (CORS) and Cross-Site Request Forgery (CSRF) policies. These protections prevent unauthorized or malicious requests from interacting with backend microservices.

S.6 Appendix: Event-Driven Transaction Flow

This Figure [S.3](#) shows a service-level sequence diagram and narrows the scope to demonstrate how each node within this subset adheres to the same communication strategy, providing a clearer understanding of

the system's behavior at a micro level. This complements the publish-subscribe architecture in the main manuscript (Figure 7), which is an event-driven execution. The services are segregated, with external services (purple and blue). The frontend interacts directly with smart contracts, but maintaining a backend is essential for enterprise-level logging and efficient state management within Web 2.0 services.

Figure S.3 provides a detailed sequence diagram that complements the high-level EDA view in Figure 5 of the main manuscript. It focuses on a specific communication scenario between the *Audit Trail Service*, external services such as *The Graph* and *Alchemy*, and internal microservices consuming these events.

The diagram illustrates how the Audit Trail Service queries transaction statuses and listens for real-time event topics. Upon receiving updates, it publishes events to the RabbitMQ bus using fanout or topic exchange strategies. This enables standardized propagation across subscribed off-chain services.

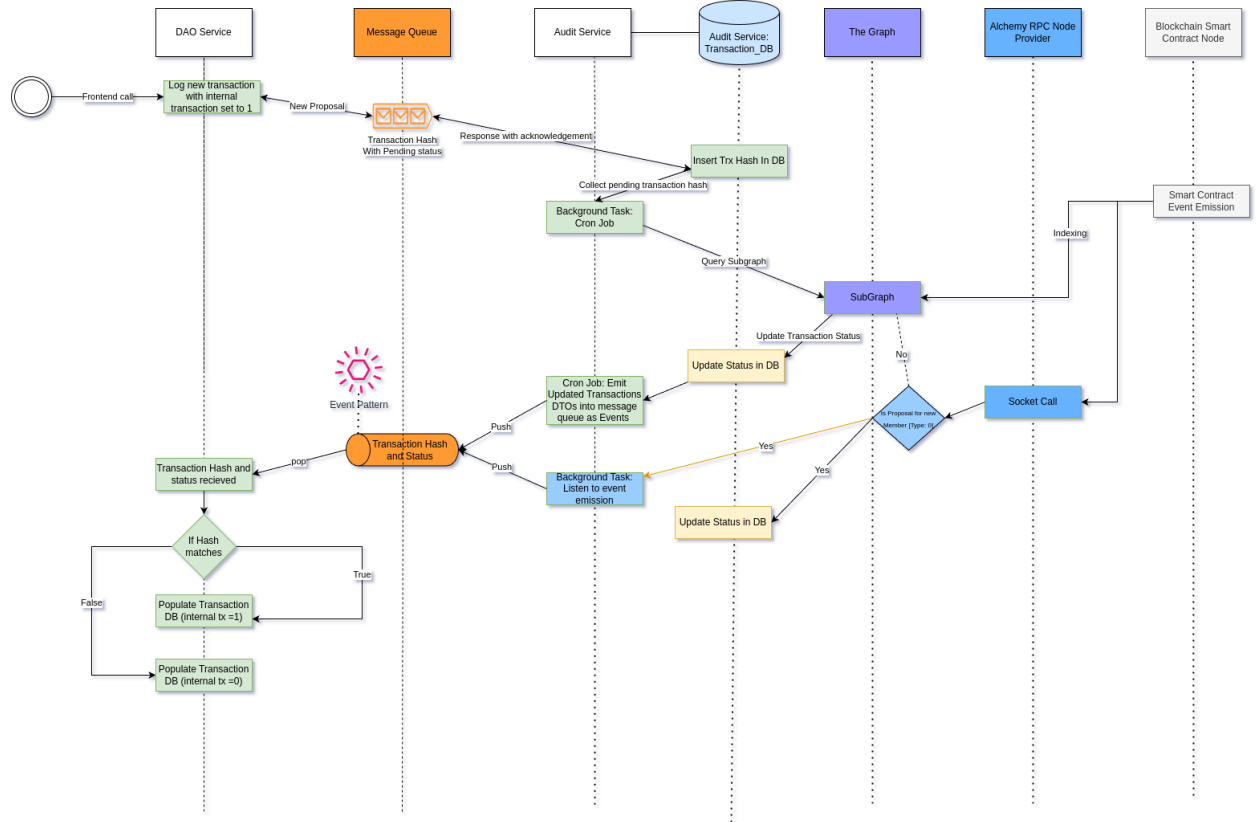


Figure S.3: Service-level sequence diagram for event-driven transaction flow between Web3 enabled services.

S.6.1 Publish-Subscribe Architecture Using RabbitMQ

To standardize inter-service communication across decentralized microservices, we adopt a publish-subscribe architecture using RabbitMQ. This appendix provides an in-depth view of the message broker integration, naming conventions, and queue structures omitted from the main manuscript for brevity.

S.7 Appendix: Messaging and Naming Standards

S.7.1 Exchange Naming Convention

We propose the following format as the standard naming convention for dApp exchanges:

<exchange_type>.<module_name>.<exchange_name>

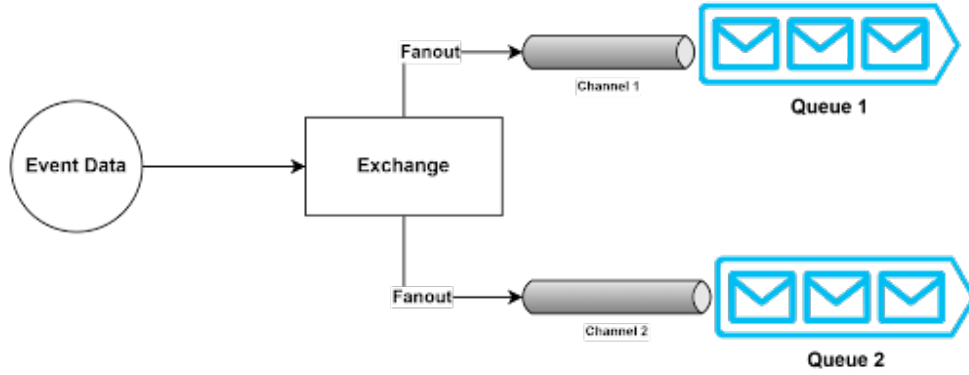


Figure S.4: Publish-Subscribe Model using Rabbitmq

This ensures that Web 2.0 services correctly interpret the event context, identifying its origin, the contract that was invoked, the function whose execution caused the event, and the specific intent or condition that downstream Web 2.0 services should address.

S.7.2 Service Queue Naming Convention

As illustrated in Figure S.4 (Fig. 6 in the main manuscript), each subscribing service attaches its own queue using the following convention:

```
<service_name>.web3_events.queue
```

S.7.3 Fanout Exchange in Proof of Concept

RabbitMQ supports multiple exchange types. In our PoC, we adopted a general-purpose fanout exchange to broadcast all events to all listening services:

```
exchange.web3_event_hub.fanout
```

This pattern is suitable when multiple services depend on the same event. For more selective routing, topic exchanges are preferred.

S.7.4 Topic Exchange Pattern

When selective event delivery is needed, topic exchanges allow services to subscribe to a filtered subset of events, improving message routing precision in complex domains.

S.7.5 Event Naming Convention

To align Web 2.0 with Solidity conventions, we transform event names from PascalCase to snake_case using the following pattern:

```
<network>.<contract_name>.<event_name>
```

Transformation Rules:

- Convert PascalCase to snake_case
- Prefix with blockchain network identifier
- Include contract name for namespace clarity

Solidity event name mapping with Web 2.0 event name:

TokensMinted
↓
optimism.market_operator.tokens_minted

This provides the important description that service developers developing downstream services urgently need.

S.7.6 Appendix: Cross-Platform Event Interoperability Profile Envelope

No single existing platform covers the complete lifecycle from Solidity event design and cross-platform observation to canonical identity, finality management, observability, and downstream routing. DORI therefore defines a layered interoperability profile that unifies these concerns without replacing existing standards or platforms. Solidity ABI event semantics provide the upstream filtering model through event signatures and indexed parameters; a provider-neutral subscription descriptor maps these properties to FireFly, Kaleido, Circle, and Coinbase monitoring interfaces [4, 5, 6, 7]. After observation, DORI represents the event using a CloudEvents-aligned canonical envelope, where the deterministic event key provides portable identity across heterogeneous sources and transports. AsyncAPI documents the resulting message contract independently of RabbitMQ, Kafka, webhooks, or other delivery technologies, while W3C Trace Context preserves vendor-neutral trace correlation across the Web3-to-Web2 boundary [8]. Together, these layers provide a unified semantic model for selecting, representing, tracing, and delivering blockchain-originated events across different enterprise integration platforms.

DORI is proposing an abstraction layer, while FireFly, Kaleido, Circle, and Coinbase are merely implementations of that abstraction

Table S.1: Mapping of DORI’s Provider-Neutral Subscription Descriptor to Existing Enterprise Platforms

DORI Field	Hyperledger FireFly	Kaleido Event Streams	Circle Event Monitoring	Coinbase Webhooks
chainId	Blockchain namespace	Target blockchain/network	blockchain	network
contract	Listener location / contract address	Contract subscription	contractAddress	contract_address
signature	Event definition / signature	Event signature	eventSignature	event_name or event_signature
indexedFilter	Indexed parameter filter	Indexed argument selection	Event topics	Event parameter filters
startBlock	Listener checkpoint / options	Stream checkpoint	Historical retrieval (where supported)	Platform-specific starting behavior

Optional ISO 20022 Financial Interoperability Profile For financial dApps, DORI may additionally support an ISO 20022-aligned payload profile. ISO 20022 provides a common methodology and business dictionary for modelling financial transactions, while the CloudEvents envelope provides protocol-neutral event metadata [9]. Accordingly, DORI retains the deterministic event key, lifecycle state, routing metadata, and trace context in the canonical envelope, while the **data** field may contain or reference an applicable ISO 20022 payment, cash-management, or securities message. This separation allows blockchain-originated settlement events to interoperate with established financial infrastructures without making ISO 20022 mandatory for non-financial dApps. The same approach is relevant to emerging tokenised-asset and cross-border settlement initiatives that combine distributed-ledger execution with standardised financial messaging.

Listing 1: DORI envelope with an optional ISO 20022 financial payload.

```

1 {
2   "specversion": "1.0",
3   "id": "<EventKey>",
4   "source": "urn:dori:eip155:10:0xContract",
5   "type": "finance.payment.settled",
6   "datacontenttype": "application/xml",
7   "dataschema":
8     "urn:iso:std:iso:20022:tech:xsd:pacs.XXX",
9   "dori_status": "FINAL",
10  "dori_routingkey":
11    "optimism.payments.settlement_completed",
12  "traceparent": "<W3C-Trace-Context>",
13  "data": "<ISO20022-Document>...</ISO20022-Document>"
14 }

```

Listing 2: Example DORI canonical event envelope.

```

1 {
2   "specversion": "1.0",
3   "id": "<EventKey>",
4   "source": "urn:dori:eip155:10:0xContract",
5   "type": "nft.tokens_minted",
6   "subject": "inventory",
7   "time": "<RFC3339-Timestamp>",
8   "datacontenttype": "application/json",
9   "chainId": 10,
10  "txHash": "<TransactionHash>",
11  "logIndex": 2,
12  "blockNumber": 123456,
13  "blockHash": "<BlockHash>",
14  "status": "FINAL",
15  "routingKey":
16    "optimism.inventory.tokens_minted",
17  "traceparent": "<W3C-Trace-Context>",
18  "data": {
19    "assetRef": "<OpaqueAssetReference>",
20    "amount": 100
21  }
22 }

```

S.8 Detailed Operational Walkthrough of the DORI Lifecycle

This compact lifecycle keeps the synchronization model focused on the core research problem: converting uncertain blockchain observations into safe, canonical, and finality-aware events for downstream Web 2.0 microservices.

S.8.0.1 Stage 1 - Blockchain Event Origin

At this stage, dApps deploy smart contracts across one or more blockchain networks. Each contract may conform to any ERC/EIP standard (such as ERC-20, ERC-721, or ERC-1155) or be a custom contract tailored to specific business logic. In the figure, at Stage 1, dApps with smart contracts that inherit ERC1155 are deployed on two different chains and have committed two different transactions. On the Optimism chain, a new batch of NFTs was minted, while on Polygon, a number of NFTs were sold.

Hence, each event record is uniquely identified by its **origin blockchain, contract address, and event specification**, which define the context and scope of the event for downstream systems.

S.8.0.2 Stage 2 - Event Stream Delivery

Our system relies on a fault-tolerant strategy to capture event streams from Web 3.0. Event is synchronized either by Node RPC providers like Alchemy, Infura or QuickNode, or picked up through the cron job of

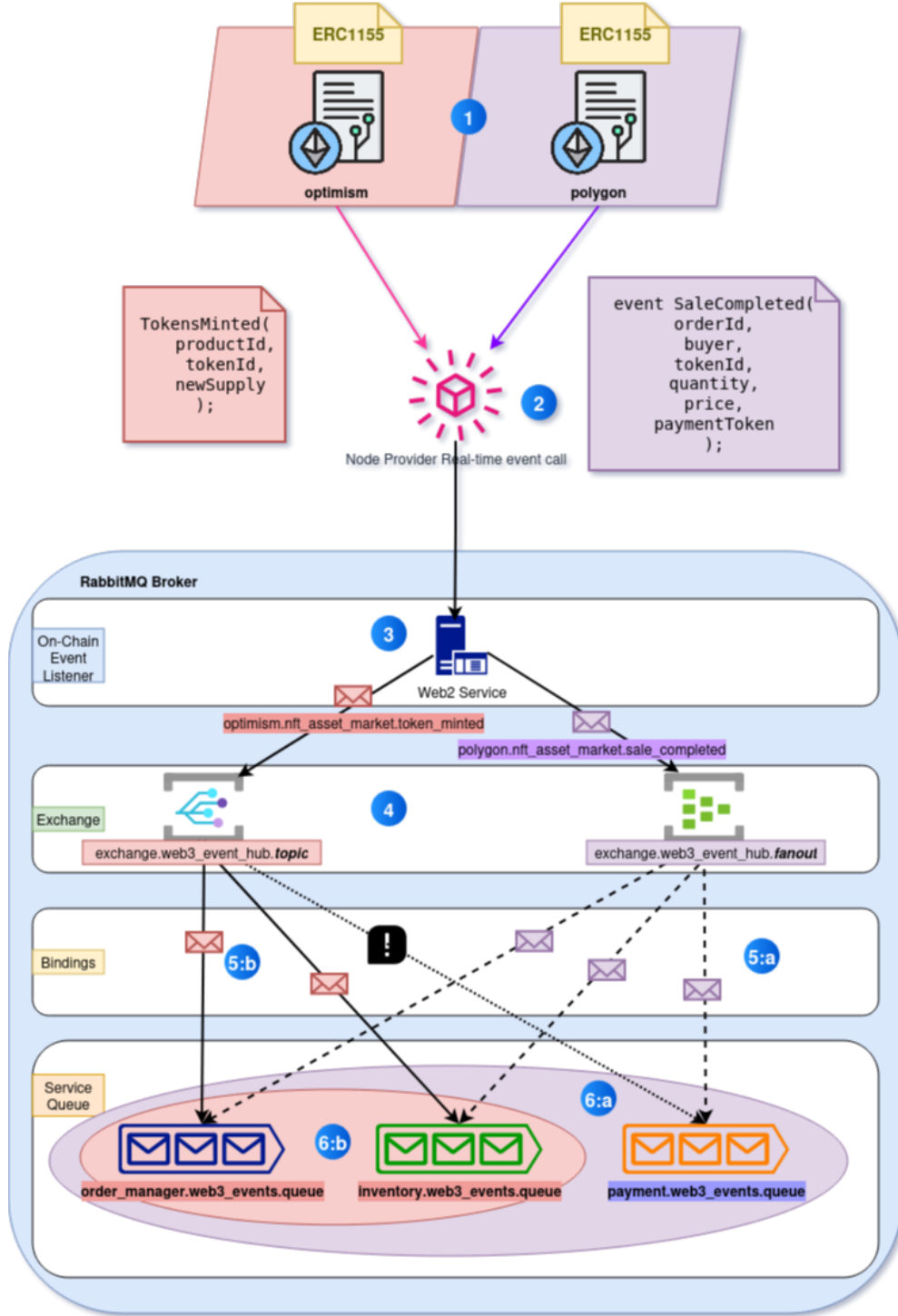


Figure S.5: DORI's finality-aware cross-domain event lifecycle. The Synchronization Coordinator derives a deterministic event identity, reconciles multi-source evidence, admits finality-qualified lifecycle updates, and projects them through targeted or broadcast delivery scopes to durable downstream queues.

GraphQL. These ensure serve as intermediaries that deliver real-time event data to subscribed off-chain services without requiring direct blockchain polling from each microservice.

In the figure, two different events are emitted, and at Stage 2, it can be seen that node providers are sending the live socket call. At Stage 3, a Web 2.0 backend application is listening to this event. In this work, the **Audit Trail Service** holds this role.

S.8.0.3 Stage 3 - Web 2.0 Event Synchronization Service

A background synchronization service within the Web 2.0 environment maintains live subscriptions to these event streams. Acting as an adapter, it converts Web 3.0 event payloads into standardized Web 2.0 message formats following defined naming conventions and schema definitions. In our diagram, this service is converting Web 3.0 events into Web 2.0 events and re-transmitting those into other Web 2.0 services by naming these events according to the proposed naming conventions.

S.8.0.4 Stage 4 - Message Broker Routing and Distribution

Once published, events are routed through appropriate exchange types depending on business requirements. It can be observed in Stage 4 that two different events are being published to two different types of exchanges, which can be determined based on business requirements.

According to the scenario illustrated in Figure S.5, when a new digital asset is minted, it may need to synchronize with multiple off chain services such as the inventory service and the order management service. In this case, a **Topic Exchange** facilitates the routing of the *token_minted* event to these specific services.

Although the payment service is also subscribed to the same topic exchange, it will not receive this event since it does not match the routing key associated with the subscription. As shown in Figure S.5 5:b, this architecture enables the formation of dynamic subset channels through topic based event routing, allowing event delivery only to the relevant services.

Since all three services are subscribers of the Topic Exchange, this system can have different routing subsets, as shown in 6:b, involving any two or any three services based on the event name. According to our convention, we utilize these event names for routing. This selective routing mechanism, guided by standardized event naming conventions, ensures that each service processes only the events relevant to its domain logic.

On the other hand, another event, **sale_completed**, from the Polygon chain represents a purchase event that requires all services to be notified. To meet such general synchronization requirements, this PoC includes the implementation of a **Fanout Exchange**, which broadcasts event without any routing conditions.

S.8.0.5 Stage 5 and 6 - Event Consumption and State Synchronization

Subscriber services consume events from their respective queues asynchronously. Each service processes messages at its own pace, ensuring decoupled and scalable event-driven workflows.

For this purchase event at the smart contract level, the off-chain payment service updates the payment status, the inventory service synchronizes stock levels, and the order management service updates the order status—all triggered by the same event. The overall process is illustrated in 5:a, while at the service queue layer, 6:b represents the superset in which all services are synchronized with the same event according to their respective data requirements. 6:b is the superset where all services are getting synced with the same event based on their required fields.

In summary, this methodology enables distributed Web 2.0 systems to maintain eventual consistency with blockchain state without requiring direct RPC connections from each microservice. The standardized message format ensures interoperability across different service implementations, while the publish-subscribe pattern provides scalable event distribution. The proposed lifecycle further strengthens this architecture by adding canonical event identity, multi-source reconciliation, finality-aware publishing, idempotent consumption, and compensation support for reverted events.

To ensure reliability and coordination, we propose maintaining strict **Service Level Agreements (SLAs)** for this architecture. This ensures seamless collaboration and alignment between frontend, backend, and Web 3.0 teams, promoting consistency and trust across the entire event-driven system.

S.9 Appendix: Performance Test Setup Details

S.9.1 Resource Utilization

Local Machine (Proxy Execution Environment):

- **RAM:** 32 GB
- **CPU:** AMD Ryzen 7 7840U 3.3 GHz, 8 Cores, 16 Threads
- **SSD:** 512 GB

Cloud Server (Placeholder Service Deployment):

- **RAM:** 8 GB
- **CPU:** Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz, 4 Core
- **SSD:** 120 GB

S.9.2 Scenario 1: Observation-Source Performance Comparison

1. Request Endpoint:

Alchemy RPC: https://eth-sepolia.g.alchemy.com/v2/{API_KEY}

The Graph: https://api.studio.thegraph.com/query/<subgraph_ID>/fincube-proxy-supgraph/0.0.1

Testing Criteria:

- Performance of Alchemy RPC under load.
- Performance of the Graph Under load.
- Comparison of The Graph and Alchemy.

k6 Load Tool Setup:

- Effective load = 15 concurrent users with ramping.
- Total Requests = 2,073 Alchemy RPC requests and 2,172 The Graph query requests

Response Time:

- **Mean:** 0.146s (Alchemy), 0.523s (The Graph).
- **Median:** 0.120s (Alchemy), 0.355s (The Graph).

Observed Outcome: Based on this performance analysis, Alchemy appears to be faster in terms of average and 95th percentile response times, while both providers demonstrate high reliability with a 100% success rate in this specific test. The Graph has a slightly higher throughput. The choice between the two may depend on the specific requirements of the application, prioritizing lower latency (Alchemy) or higher throughput (The Graph).

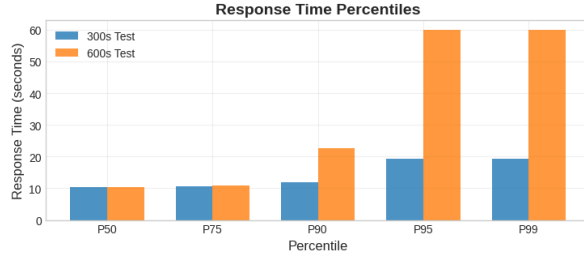
S.9.3 Scenario 2: Local Concurrent Transactions

1. Proxy Request Endpoint: [POST] host:3000/proxy/service/web3-dao-proxy/place-proposal

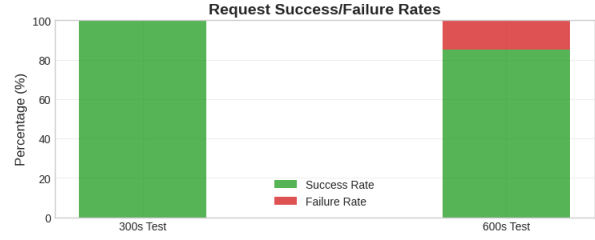
Testing Criteria:

- Performance of client-end proxy under load.

Load Parameters:



(a) Response time percentiles of proxy.



(b) Success and failure rate of proxy request.

Figure S.6: Performance analysis of the proxy service under concurrent load.

- 15 distinct users (real crypto wallets).
- Each user submits one proposal request per second.
- Effective load = 15 concurrent requests per second.
- Total requests = 300 (300s test), 600 (600s test).

Execution Duration: 8 minutes (300s test), 14 minutes (600s test).

Observed Outcome: A total of 300 on-chain transactions were successfully executed on the Sepolia testnet during the 300s run. Beyond this point, the success rate dropped and response times increased sharply. The difference between the 95th percentile response times for the 300s and 600s tests indicates testnet throttling effects. Reference can be found in Figure S.6a.

Latency: Average: 14.03 seconds (300s) and 34.56 (600s).

Throughput: Steady at approximately 0.0667 on-chain requests per second.

Success Rate: 300/300 transactions succeeded in the 300s test (100%), while only 510/600 (85%) succeeded in the extended 600s test. This is referenced in Figure S.6b.

S.9.4 Scenario 3: Remote Server Stress Test

1. Request Endpoint:

DAO Proposal Service Endpoint:

[POST] <IP>:3000/dao-service/proposal-service

Payload:

```
{
  "proposalType": "...",
  "metaData": "...",
  "proposerAddress": "...",
  "tx_hash": "..."
}
```

Testing Criteria:

- Ratio tests under load.
- Total Latency of the stress-test.

Load Parameters:

- Test-1:
 - Concurrent Users = 15
 - Total Requests = 1200 HTTP requests
 - Total Test Duration = 16 minutes. (Total time to process all transactions and synchronize all services)
- Test-2:
 - Concurrent Users = 15
 - Total Requests = 1400 HTTP requests
 - Total Test Duration = 21 minutes. (Total time to process all transactions and synchronize all services)
- Test-3:
 - Concurrent Users = 15
 - Total Requests = 1600 HTTP requests
 - Total Test Duration = 22 minutes. (Total time to process all transactions and synchronize all services)
- Test-4:
 - Concurrent Users = 15
 - Total Requests = 1800 HTTP requests
 - Total Test Duration = 25 minutes. (Total time to process all transactions and synchronize all services)

Observed Outcome: The total end-to-end performance relies of the message queue service and two independent micro-service, DAO-service and Audit-Trail.

Success Rate:

- **Test-1:** 100% success rate under the tested load.
- **Test-2:** Almost 100% success rate under the tested load.
- **Test-3:** 100% success rate under the tested load.
- **Test-4:** 100% success rate of transaction confirmation and 97.1% for downstream service synchronization under the tested load.

We recommend that teams deploying similar systems consider using managed observability services offered by Grafana or AWS to handle the load more efficiently. Alternatively, implement rate-limiting strategies such as batching operations and configuring job schedulers with less aggressive intervals to reduce system strain.

S.9.5 System Resource Constraints Under Stress

Despite the high success rate the test, our stress test revealed a **significant performance impact** caused by the 4-core CPU in our on-premise server, which became heavily throttled due to the high volume of traces and logs generated during testing. The throttling issue is shown in Figure S.7 taken from the management console of our on-premise server during stress test. The spike occurs during each batch update, and is due to a culmination of factors from cron job transaction updates to aggressive traceability and logging settings. Hence, our service shows increased times to synchronize and leads to failures at 1800 transactions.

Table S.2: Synchronization Success Rate under Different Transaction Loads

Transactions Tested	Confirmed Transactions from On-chain (%)	On-chain Confirmation Synced (%)
1200	100.0	100.0
1400	100.0	99.9
1600	100.0	100.0
1800	100.0	97.1

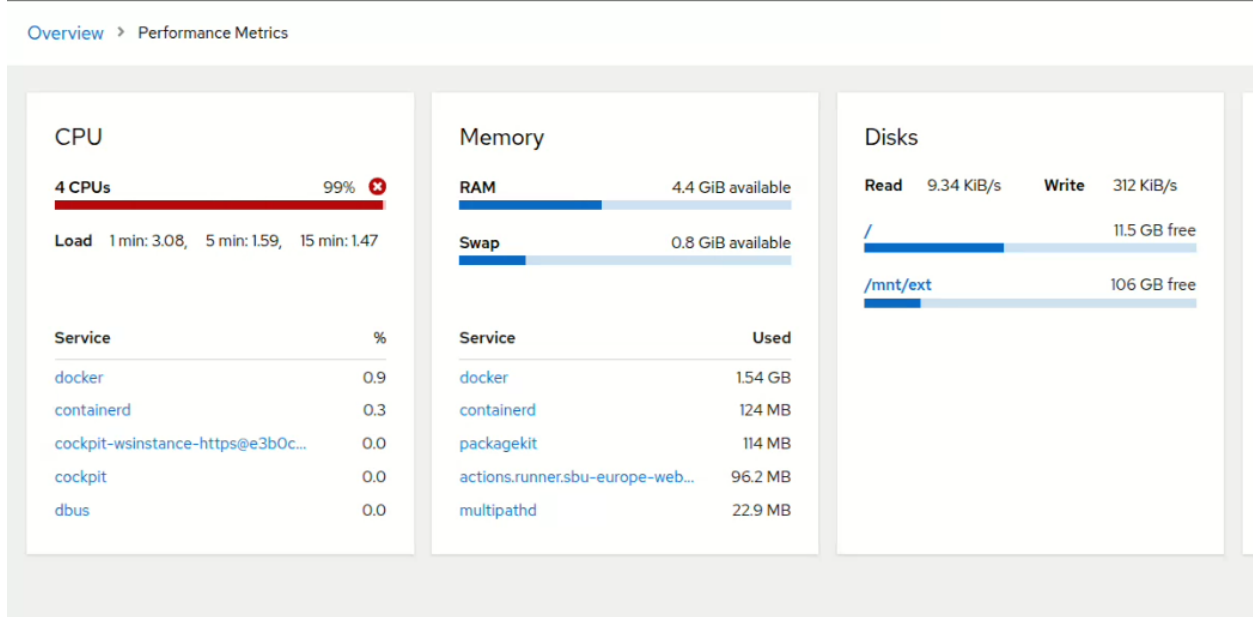


Figure S.7: CPU, RAM and Disk Usage during Stress Testing

S.10 Appendix: Related Works

The rapid growth of Web 3.0 and blockchain technology has greatly impacted the development of dApps and cloud systems. The related work section highlights recent advances and innovations relevant to the solutions discussed in the paper.

S.10.1 Enterprise dApps

Blockchain-based solutions, as discussed in the literature, improve accessibility, scalability, security, and auditability, effectively bridging the gap between Web 2.0 and Web 3.0 developers. One study highlights the need for improved security, specialized testing, and the emergence of new professional roles within blockchain, suggesting important directions for future research [10]. In certain systems, blockchain is used to maintain data that must be kept up-to-date and well synchronized to ensure enhanced tracing and auditing capabilities [11]. Enterprise-grade products with B2B collaboration, such as IBM's Supply Chain [12] and Vakt [13], rely on improved reliability, efficiency, and traceability as key selling points for state-of-the-art technology. This work emphasizes the importance of developing tools to ensure the reliability and scalability of blockchain systems, thereby advancing the adoption of Web 3.0.

Table S.3: Comparison between different existing works

Ref.	Web 2.0 to Web 3.0 Transition	Pluggable Architecture	GDPR Compliance	CIA Triad			Scalability	Observability	Framework
				Confid.	Integr.	Avail.			
Our paper	✓	✓	✓	✓	✓	✓	✓	✓	✓
[10]	✓	✓	✗	✗	✗	✓	✓	✓	✓
[14]	✓	✗	✗	✓	✓	✓	✓	✓	✓
[15]	✓	✗	✓	✓	✓	✓	✓	✓	✓
[16]	✓	✓	✓	✓	✓	✓	✓	✗	✓
[17]	✗	✓	✗	✗	✓	✓	✓	✗	✗
[18]	✗	✗	✓	✓	✓	✓	✗	✗	✗
[19]	✗	✗	✓	✓	✓	✓	✗	✗	✗
[20]	✓	✗	✓	✓	✓	✓	✓	✓	✓
[21]	✗	✗	✓	✓	✓	✓	✓	✓	✗
[22]	✗	✗	✓	✓	✓	✓	✓	✓	✗
[23]	✗	✗	✓	✓	✓	✓	✓	✗	✓

S.10.2 Hybrid Architecture for dApps

New architectures and best practices have also been proposed. For example, [14] discusses event-driven architecture for real-time auditing and off-chain storage to maintain data confidentiality. Similarly, [15] offers a structured approach to dApp development, focusing on security assessments and gas optimization. Several studies also address scalability. [16] proposes a hybrid decentralized architecture, highlighting the role of Web 2.0 developers in lowering transaction costs, while we introduce a hybrid pipeline to facilitate access to decentralized environments. Additionally, [17] explores decentralized load balancing for containerized microservices, which our system expands by integrating it with an orchestration system for both Web 2.0 and Web 3.0 services.

S.10.3 Logging

In terms of security, various studies suggest methods for encrypting sensitive log data. [18] develops an immutable audit trail using blockchain, relevant to our hybrid auditing system combining Web 2.0 and Web 3.0. [19] emphasizes tamper-proof audit logs secured within a peer-to-peer network. Although most studies recommend Hyperledger Fabric for blockchain, our system supports both public and private blockchains. For instance, [21] addresses blockchain-based secure log management, but current research is limited to log management with up to 30 peers, leading to performance issues. Our system, however, handles scalability through load balancing. Additionally, anonymizing sensitive log data can enhance privacy and storage efficiency. A method in the literature shows a 50% reduction in storage needs and a 25% improvement in performance [22], although privacy concerns remain when sharing logs across institutions.

S.10.4 Conventional Event Sourcing Pattern

DORI is related to the conventional Event Sourcing pattern, where application state changes are stored as a sequence of events that can be replayed to reconstruct system state. However, DORI addresses a different boundary condition. In conventional Event Sourcing, events are usually produced by the application and treated as authoritative once appended to the event store. In hybrid Web3-to-Web2 systems, blockchain events are external observations that may arrive from multiple sources, appear out of order, be duplicated, be delayed, or be invalidated by chain reorganization. DORI therefore introduces a finality-aware event admission layer before events are published to downstream services or retained for audit. This layer derives a deterministic event key, reconciles RPC/WebSocket, subgraph, and backfill observations, tracks confirmation depth, and emits finalized or compensating Web2-compatible events. Thus, DORI does not replace Event Sourcing; it adapts event-sourcing principles for uncertain, multi-source, finality-sensitive blockchain event streams.

Compared with conventional Event Sourcing, DORI’s novelty lies in the event admission process: blockchain observations are not appended as authoritative events until they are canonicalized, reconciled across sources, and evaluated against a finality policy.

S.10.5 Traceability

One key contribution of our paper is the introduction of enhanced traceability in decentralized applications (dApps). A dApp must not only monitor off-chain performance but also track on-chain events. Sentio [24] provides an on-chain monitoring service which can index events and show insights. Dune Analytics is a paid service which allows users to query blockchain data in real-time using SQL, and create customizable dashboards to visualize on-chain data [25]. Tenderly [26] is a blockchain monitoring and alerting platform that offers real-time monitoring of smart contracts, transactions, and blockchain performance. Additionally, blockchain explorers like Blockscout [27] index transactions and display live dApp interactions, as well as overall network traffic.

Given the scale and complexity of the system, comprehensive traceability is essential for efficient debugging and system reliability. The proposed observability framework empowers decentralized application (dApp) developers with enhanced visibility into system behavior, thereby facilitating quicker identification and resolution of errors.

Our system adopts a similar approach to these solutions in terms of traceability but also adds unique features, such as log masking, modular distribution of nodes, and fault-tolerant event tracing. Table S.3, presents a comparative analysis of the different works discussed in our related works, highlighting the key concerns of each study.

S.11 Appendix: Future Work

The rise of Deep Learning and advanced Artificial Intelligence (AI) models, such as transformers, has unlocked powerful new ways to generate actionable analytics from system data. These analytics can be especially valuable for dApps, where even small improvements in efficiency can lead to significant cost savings. Having demonstrated how Web 3.0 logs can be synchronized with Web 2.0 logs, we see an opportunity to develop a foundational AI model capable of predicting transaction traffic, generating alerts for suspicious activity, and detecting fraudulent transactions. Such capabilities would be highly beneficial for enterprises aiming to expand into Web 3.0 ecosystems.

Stock markets have long leveraged similar AI and machine learning models to analyze vast amounts of financial data, predict market trends, detect anomalies, and inform trading strategies [28]. These models help investors make data-driven decisions by forecasting price movements, identifying unusual trading patterns, and managing risk effectively.

While cloud providers like Amazon AWS already offer analytics services, their explainability and insights are largely limited to traditional Web 2.0 logs. In contrast, our foundational model will focus on explaining and predicting critical information derived specifically from Web 3.0 transactions. For instance, it will enable the prediction of ERC-20 token transfers, NFT mints, and even forecast future NFT prices ahead of time, providing enterprises with a valuable strategic advantage.

References

- [1] OpenZeppelin, “Account abstraction — openzeppelin docs,” <https://docs.openzeppelin.com/community-contracts/0.0.1/account-abstraction>, 2023, accessed: 2025-05-28.
- [2] OpenZeppelin, “Contract module which provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions.” *OpenZeppelin security product documentation*, 2025. [Online]. Available: <https://docs.openzeppelin.com/contracts/5.x/api/access>
- [3] European Union Agency for Cybersecurity (ENISA), “Data pseudonymisation: Advanced techniques and use cases,” 2021, accessed: 2024-10-10. [Online]. Available: <https://www.enisa.europa.eu/publications/data-pseudonymisation-advanced-techniques-and-use-cases>
- [4] Hyperledger FireFly, “Contractlistener reference,” <https://hyperledger-firefly.github.io/firefly/head/reference/types/contractlistener/>, accessed: 2026-07-12.

- [5] Kaleido, “Event streams: Automate blockchain operations,” <https://www.kaleido.io/blockchain-platform/event-streams>, accessed: 2026-07-12.
- [6] Circle, “Quickstart: Event monitoring for smart contracts,” <https://developers.circle.com/contracts/scp-event-monitoring>, accessed: 2026-07-12.
- [7] Coinbase Developer Platform, “Webhooks quickstart: Onchain activity subscriptions,” <https://docs.cdp.coinbase.com/webhooks/quickstart>, accessed: 2026-07-12.
- [8] S. Kanzhelev, M. McLean, A. Reitbauer, B. Drutu, N. Molnar, and Y. Shkuro, “Trace context,” World Wide Web Consortium, W3C Recommendation, Nov. 2021, accessed: 2026-07-12. [Online]. Available: <https://www.w3.org/TR/2021/REC-trace-context-1-20211123/>
- [9] ISO 20022 Registration Authority, “About iso 20022,” <https://www.iso20022.org/about-iso-20022>, 2026, accessed: 2026-07-12.
- [10] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, “Blockchain-oriented software engineering: Challenges and new directions,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 169–171.
- [11] P. Dudczyk, J. Dunston, and G. Crosby, “Blockchain technology for global supply chain management: A survey of applications, challenges, opportunities & implications (march 2024),” *IEEE Access*, vol. PP, pp. 1–1, 01 2024.
- [12] IBM, “Blockchain for supply chain,” 2025, accessed: 2025-05-20. [Online]. Available: <https://www.ibm.com/blockchain-supply-chain>
- [13] Vakt, “June 2018: Vakt is launched to re-imagine global commodities through the use of blockchain,” 2018, accessed: 2025-05-20. [Online]. Available: <https://www.vakt.com/press/june-2018-vakt-is-launched-to-re-imagine-global-commodities-through-the-use-of-blockchain>
- [14] M. Wöhrer, U. Zdun, and S. Rinderle-Ma, “Architecture design of blockchain-based applications,” in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2021, pp. 173–180.
- [15] L. Marchesi, M. Marchesi, and R. Tonelli, “Abcde –agile block chain dapp engineering,” *Blockchain: Research and Applications*, vol. 1, p. 100002, 12 2020.
- [16] F. Wessling, C. Ehmke, O. Meyer, and V. Gruhn, “Towards blockchain tactics: Building hybrid decentralized software architectures,” in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 234–237.
- [17] M. Rusek, G. Dwornicki, and A. Orłowski, “A decentralized system for load balancing of containerized microservices in the cloud,” vol. 539, 11 2017, pp. 142–152.
- [18] C. Regueiro, I. Seco, I. Gutiérrez-Agüero, B. Urquizu, and J. Mansell, “A blockchain-based audit trail mechanism: Design and implementation,” *Algorithms*, vol. 14, no. 12, 2021. [Online]. Available: <https://www.mdpi.com/1999-4893/14/12/341>
- [19] A. Ahmad, M. Saad, M. Bassiouni, and D. Mohaisen, “Towards blockchain-driven, secure and transparent audit logs,” 11 2018.
- [20] M. A. Islam, M. A. Islam, M. A. H. Jacky, M. Al-Amin, M. S. U. Miah, M. M. I. Khan, and M. I. Hossain, “Distributed ledger technology based integrated healthcare solution for bangladesh,” *IEEE Access*, vol. 11, pp. 51 527–51 556, 2023.
- [21] Ali, A. Khan, M. Ahmed, and G. Jeon, “Bcals: Blockchain-based secure log management system for cloud computing,” *Transactions on Emerging Telecommunications Technologies*, vol. 33, 04 2021.

- [22] S. Ghiasvand and F. Ciorba, *Anonymization of System Logs for Preserving Privacy and Reducing Storage: Proceedings of the 2018 Future of Information and Communication Conference (FICC)*, Vol. 2, 01 2019, pp. 162–179.
- [23] J. Wang, F. Peng, H. Tian, W. Chen, and J. Lu, *Public Auditing of Log Integrity for Cloud Storage Systems via Blockchain*, 06 2019, pp. 378–387.
- [24] Sentio, “Sentio: Development platform for decentralized applications,” 2025, accessed: 2025-05-23. [Online]. Available: <https://www.sentio.xyz/>
- [25] D. Analytics, “Dune analytics web app overview,” 2025, accessed: 2025-05-23. [Online]. Available: <https://docs.dune.com/web-app/overview>
- [26] Tenderly, “Tenderly monitoring,” 2025, accessed: 2025-05-23. [Online]. Available: <https://tenderly.co/monitoring>
- [27] Blockscout, “Blockscout: Open source blockchain explorer,” 2025, accessed: 2025-05-23. [Online]. Available: <https://blockscout.com/>
- [28] M. Vijh, D. Chandola, V. A. Tikkiwal, and A. Kumar, “Stock closing price prediction using machine learning techniques,” *Procedia Computer Science*, vol. 167, pp. 599–606, 2020, international Conference on Computational Intelligence and Data Science. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050920307924>