Decentralized Credit Verification System

Florien Lazaro
Blockchains Group 1
i6367261

Pedro Vaz Serrão Santos
Blockchains Group 1
i6341247

Francisco Costa
Blockchains Group 1
i6349732

Luuk Janssens
Blockchains Group 1
i6332099

***Abstract*—Centralized credit reporting systems expose users to significant privacy risks through opaque data access, frequent breaches, and lack of user control. We present a blockchain-based decentralized credit verification platform that separates verifiability from disclosure. Users are registered on-chain with hashed attributes while sensitive financial data remains off-chain. The system enables granular, time-limited consent management where users control exactly who can access their creditworthiness data and for how long. All access events are immutably logged, creating a transparent audit trail. We implement this hybrid architecture using Solidity smart contracts and demonstrate through comprehensive testing that the system scales efficiently: consent operations maintain constant gas costs (~192k gas per grant), while revocation remains practical (~51k gas). Our approach addresses the Oracle Problem by delegating identity verification to trusted banking partners who perform KYC/AML checks, thus bridging real-world identity with on-chain pseudonymous credentials. The platform provides regulatory compliance advantages through verifiable proof of proper data handling while preserving user sovereignty over personal financial information.**

0 Contents

1 Introduction

1.1 Problem Statement

Centralized credit reporting systems create significant privacy risks and lack user control. Users are often unaware of who accesses their financial data, and massive data breaches frequently compromise sensitive personal information. Furthermore, users cannot easily revoke access once granted, and there is no immutable, transparent record of data access. This project aims to build a blockchain-based decentralized identity and data sharing platform for the financial domain, specifically for creditworthiness verification, where users retain ownership of their data and control access through revocable, auditable consents.

1.2 Other Solutions

There already exist some solutions to the introduced problem.

One of them is a W3C-backed decentralized identity model where users control cryptographic identifiers and verifiable credentials issued by trusted parties [1]. Data sharing happens through signed credentials, not a centralized database. Storage can be off-chain with anchors on blockchains. It shows a decentralized alternative to centralized identity; demonstrates selective disclosure and user ownership, but lacks fine-grained, revocable on-chain consent per data access event.

Another interesting solution is Sovrin, a public-permissioned identity ledger designed specifically for self-sovereign identity [2]. It provides credential issuance, revocation registries and proof verification over a blockchain backend.

Above solutions validate the feasibility of decentralized identity, but they do not solve the more domain-specific challenges of decentralized and user-controlled revocable, auditable, scoped access to sensitive credit-related information. This gap motivates the design of our system, which integrates identity, consent and auditability into a single on-chain model.

1.3 System Overview

We implemented a hybrid model separating verification from data storage. Users are registered on-chain with hashed attributes while keeping sensitive financial data off-chain. When a Lender needs verification, the User grants on-chain consent. The Lender then requests data from the off-chain store, which verifies the consent against the blockchain before serving data. All access events are immutably logged.

2 Architecture

An overview of the system's architecture can be found in Fig. 1.

2.1 Methodology

The system was developed using a local Hardhat node. The contracts were deployed through Ignition modules. All validation was performed using Solidity-based tests (Forge-style) covering the `CreditRegistry`, `ConsentManager` and end-to-end interaction flows. The front-end user interface demo was built using Next.js. Further information can be found in the associated Github repository.

2.2 Model

2.2.1 Proposed Solution

Our decentralized credit system separates verifiability from disclosure. Broadcasting exact credit scores on-chain, even under pseudonyms, compromises financial privacy, enabling global correlation, de-anonymization, and profiling by adversaries with access to auxiliary datasets.

The proposed solution is not an **anonymous credit score system** in the conventional sense: it is a selectively disclosable, consent-driven credit verification protocol. Users maintain control over when and to whom their exact score (or a bracketed proof of score) is revealed.

The chain anchors:

- Existence of a valid score
- consent to disclose it
- Proof that a particular requester accessed it

The score itself remains off-chain, or is revealed only to a requesting lender through zero-knowledge proofs or verifiable access events.

2.2.2 Users and Roles

The system defines the following roles:

User (borrower / identity owner). This is the individual who owns the data and the digital identity. They cannot directly initiate registration, as they are registered by a Partner Bank after KYC/patrimony vetting. The User has the ability to manage consent settings (granting and revoking access to their data).

Partner bank (onboarding and requester authority). An established financial institution acting as both the initial identity validator and potential data requester. The Partner Bank:

- performs the initial customer identity verification and patrimony assessment (KYC/AML),
- uses a dedicated account to register the User's unique (but pseudonymous) ID onto the platform,
- may act as a Lender that requests access to a User's data to verify creditworthiness, which requires obtaining valid consent.

Requester (third-party lender). Any other entity (e.g., a non-partner financial service, small lender) that requests access to the User's data. Requesters must obtain valid consent before accessing any information and have no onboarding authority.

Administrator. Responsible for initial deployment and configuration of smart contracts and technical setup of all partner bank / requester accounts.

2.2.3 Functional Requirements

To address the problem statement, the system implements the following core functions:

- **User Registration**: Users can register a digital identity using hashed attributes (e.g., email hash) to ensure privacy while maintaining uniqueness.
- **Consent Management**: Users can grant granular, time-limited consent to specific Requesters for specific data types / identity attributes (scopes).
- **Consent Revocation**: Users have the ability to revoke previously granted consents at any time, immediately invalidating further access.
- **Audit Logging**: Every access attempt to the off-chain data is logged on-chain. This includes both successful accesses (authorized) and failed attempts (unauthorized), creating an immutable audit trail.
- **Incentivization**: The system itself creates an incentive for the user to grant consent for data sharing, which is to get their loan approved. For the user to achieve that, he must share his data with the lender.

2.3 Assumptions

To successfully use the system, we assume the following:

2.3.1 Banking Partnership and Value Proposition

Our foundational premise: society operates on trust in established banking institutions to accurately evaluate patrimony and track asset ownership. Banks are financially incentivized to maintain this trust; their credibility directly translates to their competitive position and profitability. This creates a natural alignment of interests.

Our partnership model: Rather than competing with banks, we position ourselves as an infrastructure provider. We would partner with established, well-regarded banking institutions to onboard their existing customer base onto our platform.

2.3.2 Why Banks Would Adopt Our System

The system offers complete mechanical transparency. Every consent grant, revocation, and access attempt is immutably logged and auditable, providing enhanced customer trust. In an era of increasing data breach concerns and privacy regulations, this transparency becomes a competitive differentiator for partner banks.

By offering verifiable proof that customer data is accessed only with explicit, time-bound consent, banks can strengthen their relationship with privacy-conscious customers, thus enhancing customer trust.

We provide credit verification infrastructure, not banking services. Banks retain their core business (lending, accounts, financial products), while outsourcing the consent management and audit trail to our decentralized platform. This means our service does not introduce competition.

An immutable audit trail of all data access events simplifies compliance with data protection regulations (GDPR, CCPA, etc.) and provides verifiable proof of proper data handling in case of audits. Thus, there is a regulatory compliance advantage.

2.3.3 The Delegation Model: Why Banks Should Be the Onboarding Authority

Making a trusted bank the initial registration authority leverages their existing, legally mandated processes and solves a critical problem for a decentralized system: the Oracle Problem (how to get trusted real-world data/identity onto the blockchain).

1) **Identity verification (KYC/AML)** Banks are legally required to perform KYC (Know Your Customer) and AML (Anti-Money Laundering) checks to establish a verified, real-world identity for every account holder. By delegating onboarding to the bank, we automatically inherit a vetted identity.
2) **Patrimony assessment & credit context** The bank is the source of the high-quality, verified financial data (patrimony, income, account history) that forms the basis for lending decisions.

   The act of the bank registering the user signals that the user has a verified financial relationship and a pre-existing level of financial trust. This is the trusted context needed for our credit verification platform.
3) **Regulatory compliance and risk mitigation** Banks already follow strict rules for verifying customers and protecting data. By letting a bank handle the first identity check, our system automatically benefits from these existing controls. This reduces the chance that someone registers with a false identity and lowers overall security risks. In addition, the platform keeps an immutable record of all consents and data-access events. Such a clear audit trail makes it easier for organizations to show that they handle user data correctly and follow legal requirements.

2.4 Front-End

The associated front-end demo allows users to connect their wallet to access identity- and consent management features and the audit logging.

3 Implementation

3.1 Data Model

3.1.1 On vs. Off-Chain Allocation of Attributes

Opaque on-chain data (hashes) exists to ensure integrity, not readability. A hash on-chain proves the off-chain value existed, was unaltered, and was bound to a specific identity and consent state at a specific time. It enables verifiable access control and auditability without exposing the underlying financial details.

Chain stores verifiable commitments; the off-chain system stores the data itself.

TABLE I
ON-CHAIN VS OFF-CHAIN ATTRIBUTE ALLOCATION

Attribute	On-Chain	Off-Chain	Hashed
`UserId`	Yes	No	No
`Email`	Yes	Yes	Yes
`AccountReference`	No	Yes	Yes
`CreditTier`	Yes	No	No
`IncomeBracket`	Yes	No	No
`DebtRatioBracket`	Yes	No	No
`AuditLogEvents`	Yes	No	No
`Raw Financial Data`	No	Yes	Yes

3.2 Consent Model

In the context of this section, **“Call”** implies: an externally owned account or contract executes a message call targeting a contract function.

3.2.1 Workflow: Bank-Verified User Registration

1) Bank verifies identity and patrimony The Partner Bank completes the required legal (KYC/ AML) and financial (patrimony assessment) verification for the user.
2) Bank prepares ID data The Partner Bank’s approved system account prepares a transaction containing the User’s pseudonymous ID (e.g., a hash of a unique bank ID) and a timestamp.
3) Bank calls registration The Partner Bank signs a transaction calling the platform’s `IdentityManager` contract: `registerUser(bankID, pseudonym)`.
4) `IdentityManager` creates record The `IdentityManager` contract creates an immutable `UserIdentity` struct containing `{ pseudonym, bankID, registrationTimestamp, onboardingAuthority: bankAddress }`.
5) `IdentityManager` stores ID The contract stores this struct and maps the pseudonym to the User’s address, effectively creating the verified digital identity on the platform.

6) User profile activation The User is notified, and their wallet address is now recognized by the platform as a verified identity, ready to manage consent.

   The associated Fig. 2 provides an overview of the process.

3.2.2 Workflow: Consent Grant

1) Borrower signs a transaction calling `grantConsent(lender, scopes, durationSeconds)`.
2) `ConsentManager` creates a consent struct `{ borrower, lender, scopes[], startBlockTime, expiryBlockTime, isRevoked = false }`.
3) `ConsentManager` stores this struct under `consentId`, the hash of the set of fields.

The associated Fig. 3 provides an overview of the process.

3.2.3 Workflow: Data Fetch with Consent Check

1) Lender sends a data access request to `OffChainStore`.
2) `OffChainStore` queries `ConsentManager` with `checkConsent(borrower, scope)`.
3) `ConsentManager` returns `valid` or `invalid`.
4) `OffChainStore` serves encrypted financial data only if valid.

The associated Fig. 4 provides an overview of the process.

3.2.4 Workflow: Revocation

1) Borrower signs a transaction calling `revokeAllConsentsLender()`.
2) `ConsentManager` resolves instance with `consentId` and sets the `revoked` field to `true`.
3) Subsequent checks for `consents[consentId]` resolve to invalid.

The associated Fig. 5 provides an overview of the process.

3.2.5 Token Incentives and Consent Duration

Our system purposefully excludes a token-based reward mechanism for data sharing. The primary value proposition is privacy preservation and transparency of the mechanism for credit verification, not the monetization of personal data. Introducing financial rewards for granting consent could create perverse incentives, encouraging users to compromise their privacy for short-term gain, which undermines the platform's core mission of sovereign identity and controlled, minimal disclosure.

The consent duration is set by the owner of the data.

3.3 Design Reasoning

3.3.1 Choices of Storage Types

`storage`: persistent blockchain data:

Used for state variables that persist across function calls and transactions.

State variables example:

```
// contracts/ConsentManager.sol:20-21
mapping(bytes32 => Consent) public consents;  //
consentId => Consent
mapping(address => bytes32[]) public
borrowerConsents;
```

These mappings store consent records permanently on-chain. The `consents` mapping maintains all consent agreements indexed by ID, while `borrowerConsents` tracks which consents each borrower has granted.

Storage references in functions:

```
// contracts/ConsentManager.sol:101-102
Consent storage consent = consents[consentId];

if (consent.borrower == address(0)) {
    ...
}
```

In `isConsentValid()`, we use a `storage` reference to read the persistent `Consent` struct from the blockchain without copying it to memory. This is gas-efficient when multiple fields of the same struct are accessed.

```
// contracts/ConsentManager.sol:74-82
consents[consentId] = Consent({
    borrower: msg.sender,
    lender: lender,
    scope: scope,
    expiryTime: expiryTime,
    revoked: false
});

borrowerConsents[msg.sender].push(consentId);
```

In `grantConsent()`, we store consent data permanently using storage variables and maintain the borrower's consent index.

`memory`: temporary function data:

Used for local variables and return values that exist only during function execution.

Local variable example:

```
// contracts/ConsentManager.sol:65
uint256 expiryTime = block.timestamp +
durationSeconds;

consentId = keccak256(
    abi.encodePacked(
        msg.sender,
        lender,
        scope,
        block.timestamp,
        block.number
    )
);
```

Both `expiryTime` and `consentId` are temporary variables computed during function execution. They do not persist after the function completes.

Return values as memory:

```
// contracts/ConsentManager.sol:177-179
function getBorrowerConsents(address borrower)
    external
    view
    returns (bytes32[] memory)
{
    return borrowerConsents[borrower];
}
```

The return type `bytes32[] memory` creates a copy of the storage array in memory before returning it to the caller. This is necessary for external function returns.

`calldata`: function arguments:

Used implicitly for external function parameters, providing gas-efficient, read-only access to arguments.

Function parameters example:

```
// contracts/ConsentManager.sol:125-129
function checkAuthorization(
    address borrower,
    address lender,
    bytes32 scope
) external returns (bool) {
    ...
}
```

Parameters `borrower`, `lender`, and `scope` are in `calldata`. They are passed from external callers and cannot be modified within the function.

Optimization with Storage References:

```
// contracts/ConsentManager.sol:131
bytes32[] storage userConsents =
borrowerConsents[borrower];

for (uint256 i = 0; i < userConsents.length; i++) {
    bytes32 consentId = userConsents[i];
    Consent storage consent = consents[consentId];
    ...
}
```

Using `storage` references to `userConsents` and `consent` avoids copying large data structures to memory. This is more gas-efficient than:

```
bytes32[] memory userConsents =
borrowerConsents[borrower];
```

especially when only reading data or when iterating over many entries.

3.3.2 Event Indexing

The `indexed` keyword on event parameters makes them searchable and filterable without scanning all event data.

```
// contracts/ConsentManager.sol:25-31
event ConsentGranted(
    bytes32 indexed consentId,
    address indexed borrower,
    address indexed lender,
    bytes32 scope,
    uint256 expiryTime
);
```

Why we indexed these three parameters:

- `consentId` (indexed): allows efficient queries for a specific consent grant by ID.
- `borrower` (indexed): enables filtering all consents granted by a specific borrower.
- `lender` (indexed): enables filtering all consents granted to a specific lender.
- `scope` and `expiryTime` (not indexed): remain available in event data but are not placed in log topics; they are less critical for real-time filtering and would consume additional topic slots.

3.3.3 `mappings`

A `mapping` is a hash table keyed by the declared type. The expression `consents[consentId]` performs deterministic storage slot derivation based on the key, rather than positional indexing.

```
mapping(bytes32 => Consent) public consents;

consentId = keccak256(
    abi.encodePacked(
        msg.sender,
        lender,
        scope,
        block.timestamp,
        block.number
    )
);

consents[consentId] = Consent({...});
```

This approach provides:

- Uniqueness
- Deterministic lookup
- Collision safety

3.3.4 Optimization Process

Storage Pointer Caching vs. Repeated Storage Access:

Naive version (more gas):

```
return
    consents[consentId].borrower != address(0) &&
    !consents[consentId].revoked &&
    block.timestamp <=
consents[consentId].expiryTime;
```

Optimized form (less gas):

```
Consent storage consent = consents[consentId];

if (consent.borrower == address(0)) return false;
if (consent.revoked) return false;
if (block.timestamp > consent.expiryTime) return
false;

return true;
```

Avoiding Redundant Public Field Getters:

Solidity auto-generates getters for public state variables, so manual getters increase bytecode size unnecessarily.

In the end, we failed to find an approach which avoids declaring public `external view` getter functions for `public` state variables, such as `mappings`.

Identity Registration Check and Sybil Resistance:

```
require(
    consents[consentId].lender == address(0),
    "Consent already exists"
);
```

This prevents duplicate registrations and mitigates certain Sybil attack patterns.

3.4 Audit Log

The `AuditLog` contract maintains an append-only record of events emitted by authorized loggers. Each `AuditEntry` stores the `accessorUserId`, `subjectUserId`, `hashedScope`, `unixTimestamp` and an `eventType` from the defined enum. Only addresses marked as `authorizedLoggers` by the admin may call `logEvent`, preventing unauthorized actors from injecting entries. This design creates a tamper-resistant audit trail that reflects all operations the platform's contracts choose to record.

4 Experimental Results

4.1 Testing

The test suite validates all core platform functionality using Foundry's Solidity testing framework. Tests are organized into three files covering unit tests, integration tests, and end-to-end workflows.

4.2 Test Summary

TABLE II
Test suite summary

Test Suite	Passing	Failing	Total
`ConsentManager.t.sol`	9	0	9
`CreditRegistry.t.sol`	24	0	24
`EndToEnd.sol`	8	0	8
Total	**41**	**0**	**41**

4.3 Tested Functionality and Design Rationale

4.3.1 Identity Registration Tests

TABLE III
Identity registration tests

Test	Description	Why Critical
RegisterIdentityAttributesSuccessfully	Verifies users can register with hashed email, credit tier, income/debt brackets, and account reference	Core requirement: users must establish on-chain identity before participating in the consent system
RevertDuplicateRegistration	Prevents same address from registering twice	Ensures identity uniqueness; prevents Sybil attacks and duplicate identity claims
RevertRegistrationWithInvalidEmailHash	Rejects zero-hash email	Enforces data integrity; email hash is required for off-chain contact verification
RevertRegistrationWithEmptyCreditTier	Rejects empty credit tier	Ensures minimum viable profile; lenders need categorical tier for basic eligibility checks
RevertRegistrationWithInvalidAccountReferenceHash	Rejects zero-hash account reference	Account reference links on-chain identity to off-chain data store; essential for hybrid architecture

4.3.2 Consent Management Tests

TABLE IV
Consent management tests

Test	Description	Why Critical
WorkflowConsentGrant	Verifies consent creation with borrower, lender, scopes, start/expiry times	Core workflow: borrowers must be able to delegate time-limited access to specific lenders
WorkflowDataFetchWithConsentCheck	Validates `checkConsent` returns `true` for granted scopes, `false` otherwise	Enforces access control; off-chain store relies on this to gate data release
WorkflowRevocation	Confirms `revokeAllConsents` invalidates all active consents to a lender	Essential for user control: immediate revocation is a key privacy guarantee
DuplicatePrevention	Ensures identical borrower→lender consents cannot be created twice	Prevents ID collisions and enforces one active consent per scope-duration pair
FullWorkflowSingleLender	Runs full lifecycle: grant, validate, authorize, revoke, post-revocation	End-to-end guarantee that all core consent operations behave correctly in sequence
ConsentExpiration	Verifies consents become invalid after expiry time	Time-bounded access prevents indefinite exposure; aligns with data minimization principles
MultipleLendersIndependentConsents	Confirms revoking one lender's consent doesn't affect others	Granular control: users manage each lender relationship independently
MultipleConsentsToSameLender	Validates multiple scope-specific consents to same lender	Supports fine-grained permissions; lender may need different scopes at different times
LargeScaleGrantCreation	Stress test with 50 users × 50 lenders (7,350 consents)	Validates scalability and gas predictability under realistic multi-party load

4.3.3 Data Fetch and Access Control Tests

TABLE V
Data fetch and access control tests

Test	Description	Why Critical
Workflow_DataFetchWithConsentCheck	Full flow: register → grant → check → retrieve reference hash	Validates complete data access workflow from registration to off-chain data pointer retrieval
Workflow_DataFetchWithoutConsent	Verifies `checkConsent` returns `false` when no consent exists	Prevents unauthorized access; default-deny security model
Workflow_EnforceScopeRestrictions	Confirms lenders can only access explicitly granted scopes	Enforces principle of least privilege; scope-level access control
ReadPublicAttributesWithoutConsent	Public categorical tiers readable without consent	Design decision: coarse-grained tiers (A/B/C) are intentionally public for basic eligibility queries

4.3.4 Identity Update Tests

TABLE VI
Identity update tests

Test	Description	Why Critical
UpdateCreditTierSuccessfully	Verifies selective field updates	Users must update financial status as circumstances change
UpdateMultipleFields	Confirms batch updates to multiple attributes	Efficiency: single transaction for multiple changes
PartialUpdatePreservesOtherFields	Empty strings preserve existing values	UX: users update only what changed without resubmitting entire profile
RevertUpdateForNonExistentIdentity	Prevents updates to unregistered addresses	Data integrity: cannot modify non-existent records
MaintainDataIntegrityAfterMultipleUpdates	Verifies data consistency through sequential updates	Ensures idempotent, predictable state transitions

4.3.5 Audit Logging Tests

TABLE VII
Audit logging tests

Test	Description	Why Critical
EndToEndFlow	Verifies audit entries created for registration and updates	Immutable audit trail is core value proposition; proves data lineage
FailedAccessLogging	Logs unauthorized access attempts	Security: detects and records suspicious activity for forensics
AuditLogAuthorizationRequired	Prevents unauthorized contracts from writing logs	Log integrity: only authorized contracts can append audit entries

4.3.6 End-to-End Workflow Tests

TABLE VIII
End-to-end workflow tests

Test	Description	Why Critical
EndToEndFlow	Complete flow: deploy → register → consent → access → update → audit	Validates all components integrate correctly as a system
RevocationFlow	Full revocation lifecycle including `revokeConsentById` and `revokeAllConsents`	Confirms both revocation methods work end-to-end
ConsentExpirationFlow	Time-based consent invalidation via `vm.warp`	Validates temporal access control enforcement

4.4 Gas Cost Summary

4.4.1 Deployment Costs

TABLE IX
Contract deployment costs

Contract	Gas	Cost (30 gwei, $3,500 ETH)
`AuditLog`	1,535,406	~$161
`ConsentManager`	1,419,830	~$149
`CreditRegistry`	1,827,755	~$192
Total	**4,782,991**	**~$495**

4.4.2 Average Gas per Function

TABLE X
Average gas cost per function

Function	Avg Gas	Est. Cost (USD)
`registerIdentityAttributes`	244,734	~$25.70
`grantConsent`	192,833	~$20.25
`checkConsent`	405,830	~$42.61
`revokeConsentById`	50,853	~$5.34
`revokeAllConsents`	64,618	~$6.78
`updateIdentityAttributes`	96,689	~$10.15
`getIdentityAttributes`	17,260	~$1.81
`isConsentValid`	7,219	~$0.76

4.4.3 Deployment

The below metrics were collected from profiling tests using:

```
bunx hardhat test --gas-stats
```

The large-scale test simulates 50 users interacting with 50 lenders, creating approximately 7,350 consent grants, 2,450 consent checks and 7,350 revocations.

Contract Deployment Costs:

TABLE XI
Contract deployment costs with bytecode size

Contract	Deployment Gas	Size (bytes)
`AuditLog`	1,535,406	6,832
`ConsentManager`	1,419,830	6,368
`CreditRegistry`	1,827,755	8,619
Total	**4,719,677**	**21,524**

Per-Operation Gas Costs — ConsentManager Operations:

TABLE XII
ConsentManager per-operation gas costs

Function	Min	Avg	Median	Max / Calls
`checkConsent`	26,731	405,830	410,168	766,064 / 2,479
`consents(mapping)`	11,840	11,840	11,840	11,840 / 3
`getBorrowerConsents`	10,649	345,480	358,873	358,873 / 52
`getScopes`	5,697	6,906	6,906	8,114 / 2
`grantConsent`	192,433	192,833	192,673	232,646 / 7,374
`isConsentValid`	5,098	7,219	7,220	7,229 / 7,361
`revokeAllConsents`	52,982	64,618	55,461	110,950 / 6
`revokeConsentById`	50,830	50,853	50,854	50,854 / 7,351

Per-Operation Gas Costs — CreditRegistry Operations:

TABLE XIII
CreditRegistry per-operation gas costs

Function	Min	Avg	Median	Max / Calls
`consentManager`	2,748	2,748	2,748	2,748 / 1
`getAccountReferenceHash`	2,851	2,851	2,851	2,851 / 4
`getIdentityAttributes`	17,260	17,260	17,260	17,260 / 11
`hasIdentityAttributes`	2,961	2,961	2,961	2,961 / 5
`registerIdentityAttributes`	186,442	244,734	186,454	361,300 / 18
`setAuditLog`	44,125	44,125	44,125	44,125 / 8
`updateIdentityAttributes`	37,499	96,689	46,099	212,104 / 6

Per-Operation Gas Costs — AuditLog Operations:

TABLE XIV
AuditLog per-operation gas costs

Function	Min	Avg	Median	Max / Calls
`authorizeLogger`	47,587	47,587	47,587	47,587 / 8
`getAccessHistory`	8,233	8,233	8,233	8,233 / 1
`getAuditEntry`	14,614	14,614	14,614	14,614 / 2
`getLogsCount`	2,440	2,440	2,440	2,440 / 2
`getRecentLogs`	28,361	28,361	28,361	28,361 / 1

4.4.4 Scalability Analysis (50 Users × 50 Lenders)

TABLE XV
Scalability analysis metrics

Metric	Value
Total User–Lender Pairs	2,450
Consent Grants Created	7,350
Consent Checks Performed	2,450
Consent Revocations	7,350
Validity Checks	7,350

4.4.5 Cost Estimation at Scale

Assuming 30 gwei gas price and ETH at $3,500:

TABLE XVI
Cost estimation at scale

Operation	Gas Used	Cost (ETH)	Cost (USD)
Full System Deployment	4,782,991	0.1435	~$502.25
Register Identity (avg)	244,734	0.0073	~$25.70
Grant Consent (avg)	192,833	0.0058	~$20.25
Check Consent (avg)	405,830	0.0122	~$42.61
Revoke Consent (avg)	50,853	0.0015	~$5.34

Observations After 7,350+ Consent Operations:

Consent granting scales linearly with constant gas per operation (~192k gas).

Consent checking has dynamic cost based on the number of active consents to iterate through (26k–766k gas).

Revocation at ~51k gas per consent is efficient and practical for users to manage access.

Read operations (getIdentityAttributes, hasIdentityAttributes) are inexpensive (~3k–17k gas), suitable for frequent verification queries.

5 Discussion and Conclusion

The system demonstrates that a hybrid on/off-chain architecture can provide verifiable creditworthiness checks without exposing raw financial data. On-chain consent artifacts enforce strict, revocable, and time-bounded access, while the AuditLog contract offers an immutable trace of all operations that the platform's components choose to log. This ensures that verification and data access remain transparent, auditable, and user-controlled.

Experimental evaluation shows that the model scales efficiently across thousands of lender–borrower relationships, with predictable gas usage for granting, checking, and revoking consent. The architecture maintains linear scalability while preserving strict access-control guarantees, confirming that a decentralized consent layer can operate effectively under realistic load.

However, the broader context must be acknowledged. Even with an on-chain consent system and transparent auditability, banks remain in possession of the original financial data and may continue storing or processing it outside the system's visibility. The platform cannot prevent traditional institutions from retaining information already obtained through existing legal and regulatory frameworks. Instead, its contribution lies in constraining verification workflows, enabling users to control what additional disclosures occur, and providing verifiable proof of how data access was mediated through the decentralized layer.

In this sense, the system does not eliminate institutional data asymmetry, but it introduces a verifiable, user-centric mechanism for governing future data sharing. This model offers a practical path toward improving privacy guarantees in credit verification without requiring banks to abandon their regulatory obligations or operational practices.

6 Who Did What

- **Florien Lazaro**: UI and Off-Chain software development.
- **Francisco Costa**: Project structuring, testing and benchmarking.
- **Luuk Janssens**: Report and presentation.
- **Pedro Santos**: Lead research, architecture and On-Chain software development.

7 Appendices

7.1 High-Level System Architecture

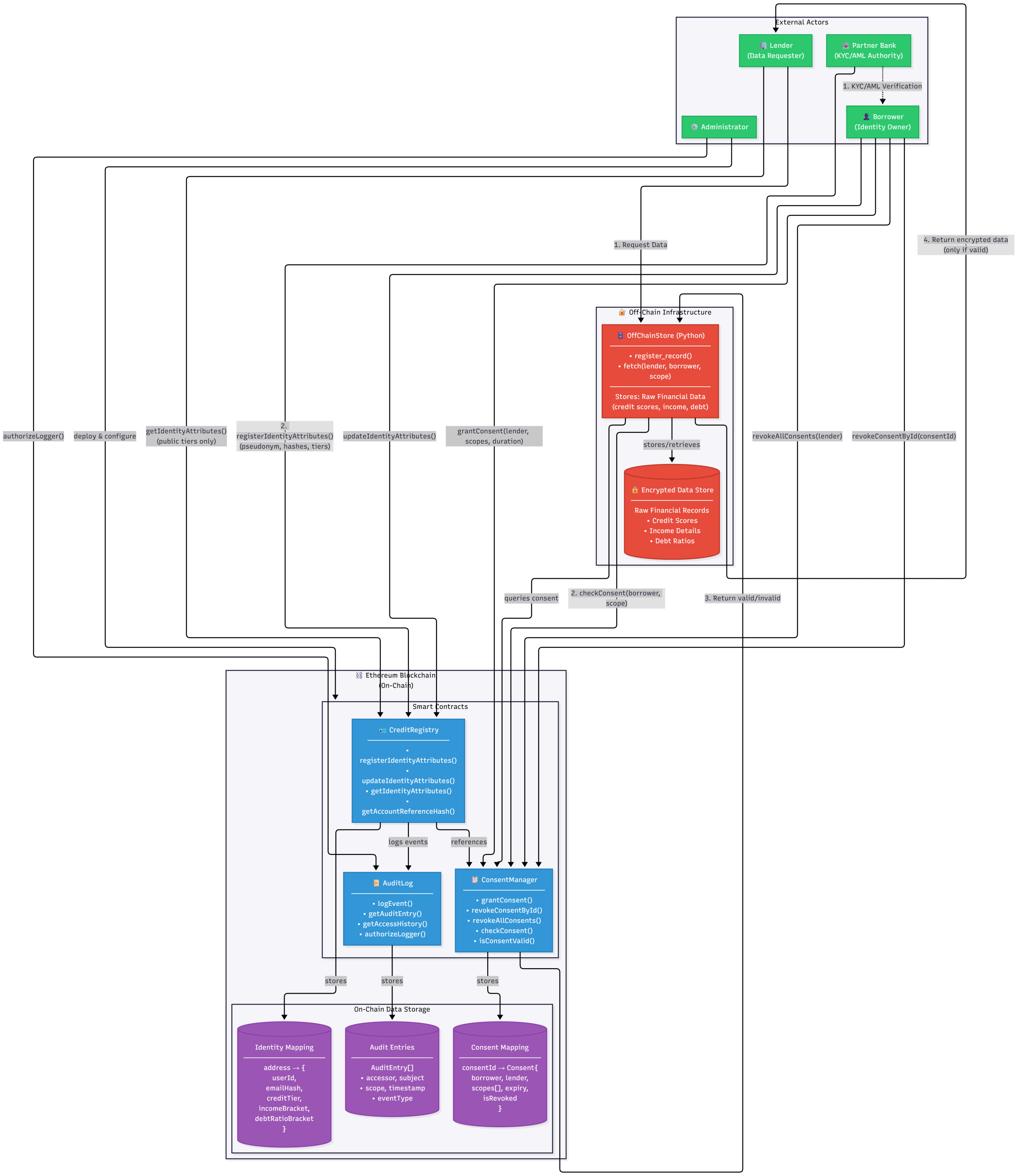


Fig. 1. High-level system architecture showing all components and trust boundaries

7.2 User Registration

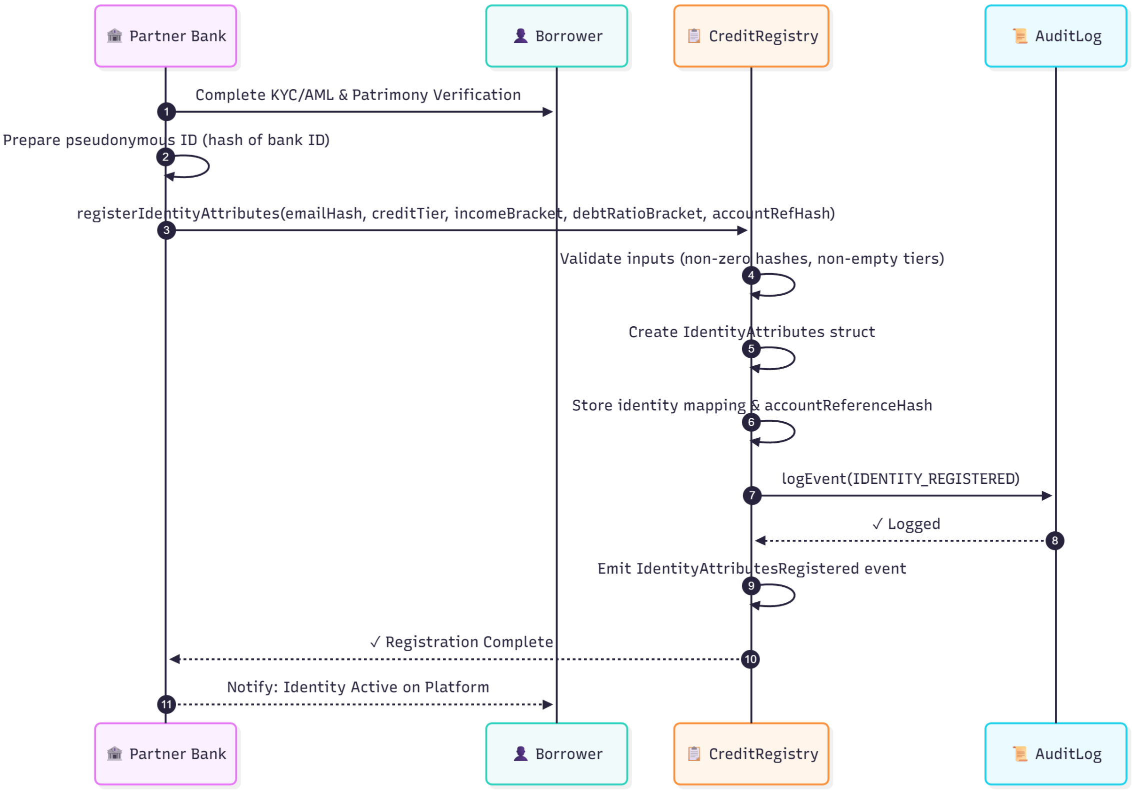

Partner Bank
Borrower
CreditRegistry
AuditLog
1 Complete KYC/AML & Patrimony Verification
2 Prepare pseudonymous ID (hash of bank ID)
3 registerIdentityAttributes(emailHash, creditTier, incomeBracket, debtRatioBracket, accountRefHash)
4 Validate inputs (non-zero hashes, non-empty tiers)
5 Create IdentityAttributes struct
6 Store identity mapping & accountReferenceHash
7 logEvent(IDENTITY_REGISTERED)
8 ✓ Logged
9 Emit IdentityAttributesRegistered event
10 ✓ Registration Complete
11 Notify: Identity Active on Platform
Partner Bank
Borrower
CreditRegistry
AuditLog

Fig. 2. Bank-verified user registration flow – KYC and account linking process

7.3 Consent Grant Workflow

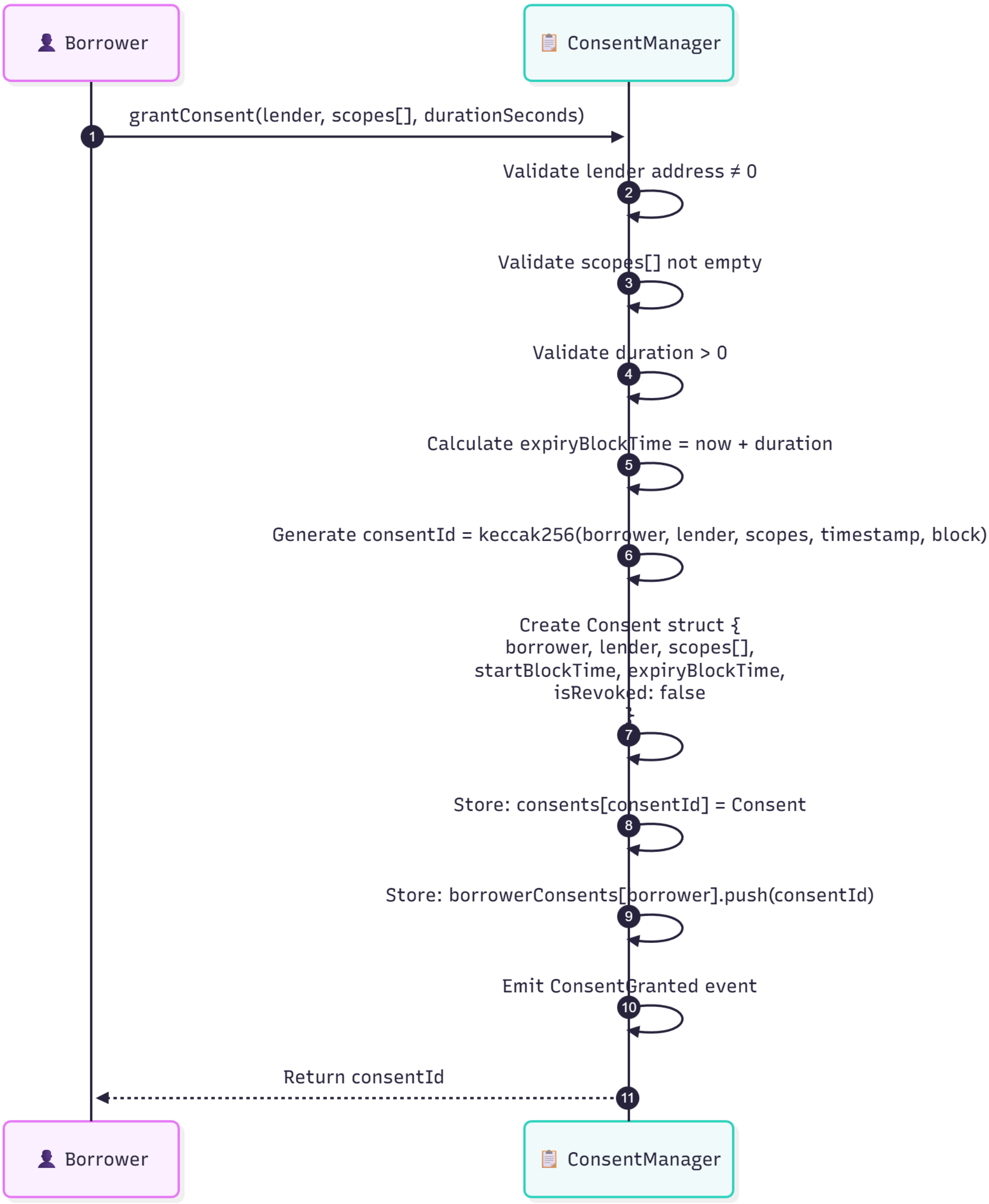

Borrower
ConsentManager
1
grantConsent(lender, scopes[], durationSeconds)
2
Validate lender address ≠ 0
3
Validate scopes[] not empty
4
Validate duration > 0
5
Calculate expiryBlockTime = now + duration
6
Generate consentId = keccak256(borrower, lender, scopes, timestamp, block)
7
Create Consent struct {
borrower, lender, scopes[],
startBlockTime, expiryBlockTime,
isRevoked: false
}
8
Store: consents[consentId] = Consent
9
Store: borrowerConsents[borrower].push(consentId)
10
Emit ConsentGranted event
11
Return consentId
Borrower
ConsentManager

Fig. 3. Consent grant workflow: user authorizes data sharing with explicit scope and duration

7.4 Data Fetch with Consent Check

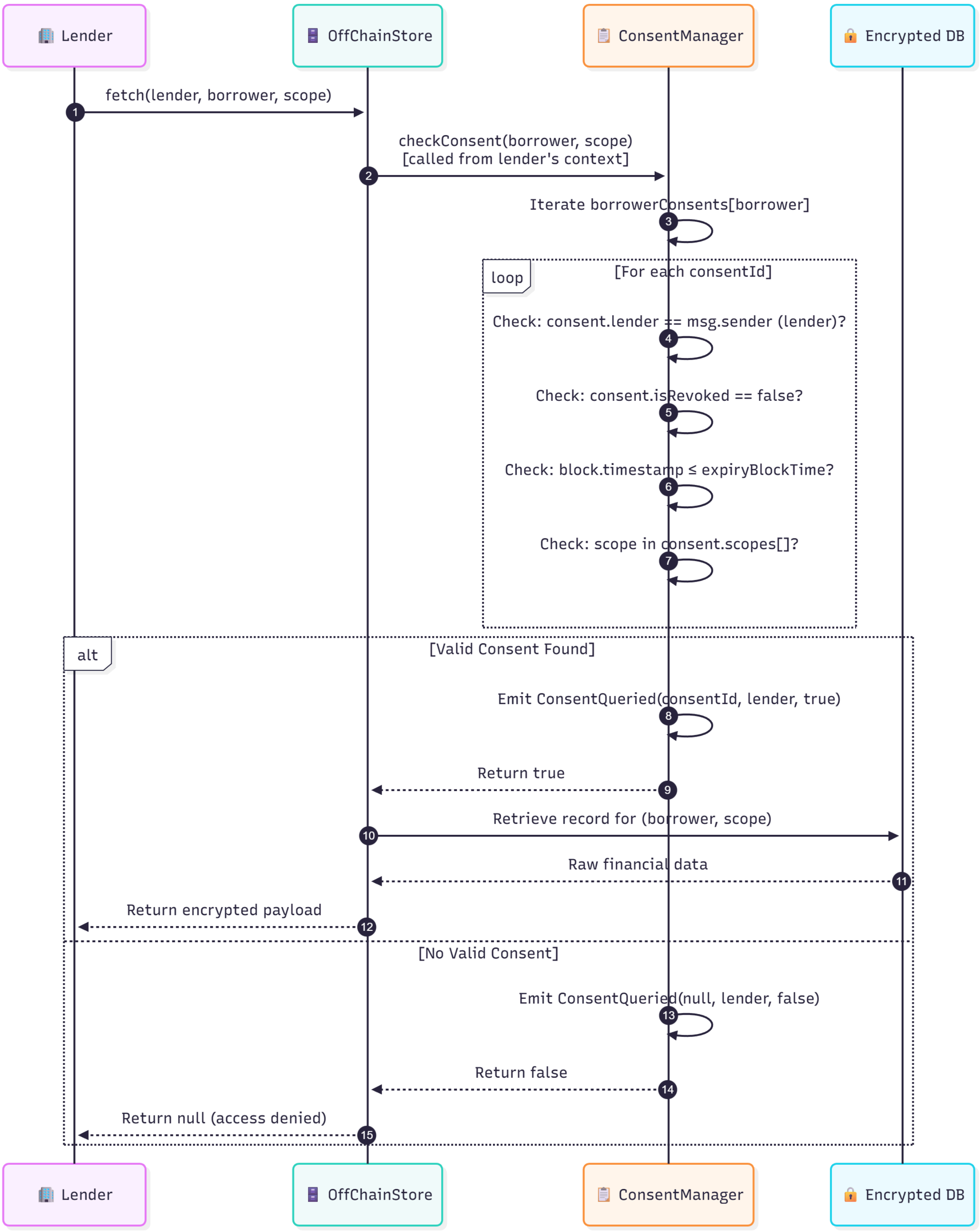
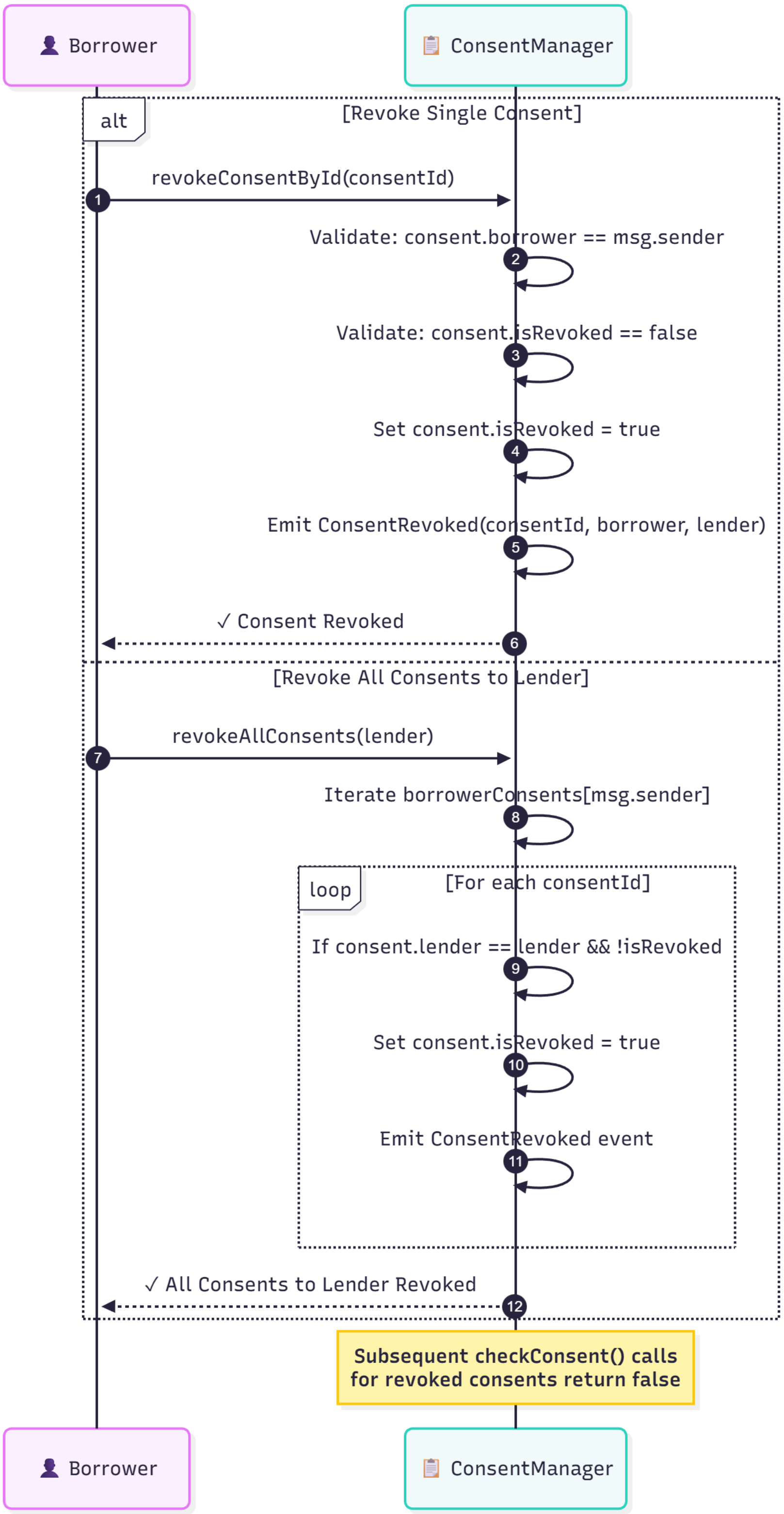

Lender
OffChainStore
ConsentManager
Encrypted DB
fetch(lender, borrower, scope)
1
checkConsent(borrower, scope)
[called from lender's context]
2
Iterate borrowerConsents[borrower]
3
loop
[For each consentId]
Check: consent.lender == msg.sender (lender)?
4
Check: consent.isRevoked == false?
5
Check: block.timestamp ≤ expiryBlockTime?
6
Check: scope in consent.scopes[]?
7
alt
[Valid Consent Found]
Emit ConsentQueried(consentId, lender, true)
8
Return true
9
Retrieve record for (borrower, scope)
10
Raw financial data
11
Return encrypted payload
12
[No Valid Consent]
Emit ConsentQueried(null, lender, false)
13
Return false
14
Return null (access denied)
15
Lender
OffChainStore
ConsentManager
Encrypted DB

Fig. 4. Real-time consent validation during every data fetch request

7.5 Consent Revocation Workflow



Borrower
ConsentManager
alt
[Revoke Single Consent]
1
revokeConsentById(consentId)
2
Validate: consent.borrower == msg.sender
3
Validate: consent.isRevoked == false
4
Set consent.isRevoked = true
5
Emit ConsentRevoked(consentId, borrower, lender)
6
✓ Consent Revoked
[Revoke All Consents to Lender]
7
revokeAllConsents(lender)
8
Iterate borrowerConsents[msg.sender]
loop
[For each consentId]
9
If consent.lender == lender && !isRevoked
10
Set consent.isRevoked = true
11
Emit ConsentRevoked event
12
✓ All Consents to Lender Revoked
Subsequent checkConsent() calls for revoked consents return false
Borrower
ConsentManager

Fig. 5. Consent revocation workflow: immediate revocation propagation to all data consumers

7.6 Overall Data Flow Summary

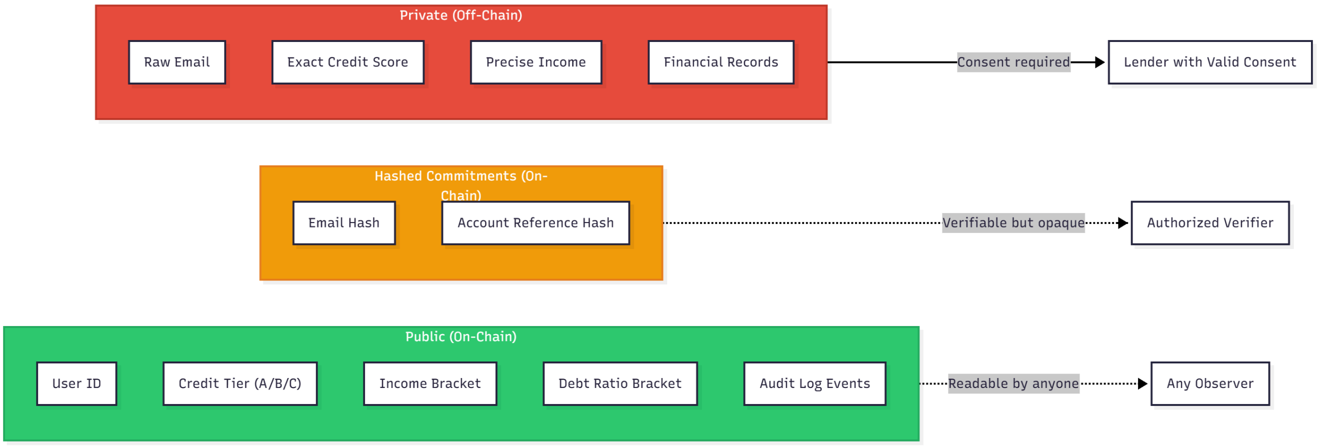


Fig. 6. End-to-end data flow summary from consent creation to consumption

References

[1] W3C Working Group, "Decentralized Identifiers (DIDs) v1.0." [Online]. Available: https://www.w3.org/TR/did-core/

[2] A. Tobin and D. Reed, "The Sovrin Identity Network," in *Sovrin Foundation Whitepaper*, 2016. [Online]. Available: https://sovrin.org/wp-content/uploads/Sovrin-Protocol-and-Token-White-Paper.pdf