Consensix Labs

Decentralized Professional Credentials

A Chain-Agnostic Protocol for W3C Verifiable Credentials with Proof-of-Concept Implementations on Ethereum and IOTA

Table of Contents

Executive Summary

Professional credentials – employment history, certifications, peer endorsements – are among the most frequently exchanged and least reliably verified pieces of information in the working world. Today, these credentials live in disconnected silos: a LinkedIn profile here, a university transcript there, a certification badge on yet another platform. Verification is manual, slow, and often impossible for the party that needs it most. Fraud is common. Portability is an afterthought.

This paper presents a protocol and working proof of concept for decentralized professional credentials. Issuers (employers, certification bodies, peers) create cryptographically signed credentials following the W3C Verifiable Credentials Data Model v2.0, the current web standard for expressing credentials in a machine-verifiable format. The credential content stays with the holder – no central database, no platform lock-in. A SHA-256 hash of each signed credential is anchored on a blockchain, creating a tamper-proof record of its existence and enabling transparent revocation. Verifiers check a credential's authenticity by validating the issuer's digital signature (which requires no network call) and confirming its on-chain status (registered, not revoked, not expired).

The proof of concept implements the protocol on two architecturally different blockchains – Ethereum (using Solidity) and IOTA Rebased (using Move) – to demonstrate that the protocol is genuinely chain-agnostic. It includes a command-line tool for the full credential lifecycle, a web interface for interactive demonstration, automated end-to-end test scenarios on both chains, and comparative metrics covering cost, developer experience, and architectural tradeoffs. Three credential types are supported: employment credentials, professional certifications, and peer endorsements.

1. Introduction

The Problem

Every hiring manager has experienced it: a candidate's resume lists impressive credentials, but verifying them requires phone calls, email chains, and weeks of waiting. A claimed certification might have expired. An employment record might be exaggerated. A peer endorsement might be fabricated entirely. The systems that issue these credentials – universities, employers, certification bodies, professional networks – operate in isolation, with no common mechanism for a third party to independently check their validity.

This fragmentation creates real costs. Background checks are a multi-billion-dollar industry built on the difficulty of verification. Credential fraud affects hiring decisions, professional licensing, and institutional trust. And even when credentials are legitimate, moving them between platforms is rarely possible. Your employment history on one platform is invisible to another. Your professional certifications exist in the issuing body's database and nowhere else.

The core issue is structural. Each credential issuer operates its own database, defines its own format, and provides its own (often limited) verification mechanism. There is no common standard for how credentials are represented, no universal way to check their authenticity, and no mechanism for the holder to control where and how their credentials are shared.

The Opportunity

Two developments have converged to make a better approach feasible.

The first is the **W3C Verifiable Credentials Data Model**, which reached version 2.0 as a W3C Recommendation in May 2025. This is an open standard that defines how credentials can be expressed in a format that is cryptographically verifiable by anyone. An issuer signs a credential with their private key. A verifier checks the signature using the issuer's public key – no need to contact the issuer, no API to call, no permission to request. The standard defines a three-role model (issuer, holder, verifier) and a JSON-based format that any system can implement.

The second is **blockchain technology** – not for storing credentials (which would be expensive and a privacy concern), but as an accountability layer. By anchoring a cryptographic hash of each credential on a public blockchain, we create a tamper-proof record of its existence at a specific point in time. Revocation becomes transparent and auditable: when an issuer revokes a credential, the on-chain record reflects this immediately, and any verifier can see it. No central authority controls the registry. No single point of failure can take it down.

The combination of these two technologies yields a system where credentials follow an open standard, holders control their own data, issuers can be anyone, and verification is independent, instant, and free.

Scope and Approach

This paper presents a protocol for decentralized professional credentials and a working proof of concept that implements it. The PoC covers three credential types (employment, certification, peer endorsement), anchoring on two blockchains (Ethereum and IOTA), a command-line interface, a web interface, and automated end-to-end scenarios.

The approach reflects a specific philosophy: blockchain is an audit and accountability layer for off-chain systems, not a data store. Credential content stays off-chain with the holder. Only hashes go on-chain. This keeps costs low, preserves privacy, and avoids the scalability constraints of on-chain data storage.

Several features are explicitly deferred to keep the PoC focused. Selective disclosure – the ability to reveal specific claims within a credential without exposing the full document – is designed for (the credential structure supports it) but not implemented. Production key management, DID methods beyond `did:key`, and credential storage wallets are also out of scope. These are discussed in Section 9 as future directions.

2. Background

Verifiable Credentials

A verifiable credential is a set of claims made by an issuer about a subject. In everyday terms: a university (issuer) states that a person (subject) holds a degree (claim). What makes it *verifiable* is that the issuer cryptographically signs the credential, and anyone with the issuer's public key can confirm the signature is authentic.

The W3C Verifiable Credentials Data Model v2.0 standardizes this concept. A credential contains a context (identifying the format), a type (what kind of credential it is), an issuer (who made the claims), a subject (who the claims are about), and the claims themselves. It also supports validity periods – a credential can declare when it becomes valid and when it expires.

There are three roles in the system. The **issuer** creates and signs the credential. The **holder** receives and stores it – importantly, the holder controls when and with whom they share it. The **verifier** checks its authenticity by validating the signature and, in our protocol, checking its on-chain status.

The key insight of the VC model is that verification is *independent*. A verifier does not need to contact the issuer, query a database, or request permission. They only need the credential itself and the issuer's public key. This makes verification instant, free, and resistant to the issuer going offline.

Decentralized Identifiers

A Decentralized Identifier (DID) is a globally unique identifier that is controlled by its owner rather than a central authority. Unlike a username on a platform, a DID is not issued by any organization and cannot be revoked by one.

This proof of concept uses `did:key`, a DID method that encodes the public key directly in the identifier itself. A `did:key` looks like `did:key:z6Mkh...` where the string after `z6Mk` is the base58-encoded Ed25519 public key. Resolving a `did:key` to its public key requires no network call – the key is embedded in the identifier. This makes it simple to implement, fast to resolve, and suitable for a proof of concept where production key management is out of scope.

The limitation of `did:key` is that it does not support key rotation. If a private key is compromised, the corresponding DID must be abandoned. Production systems would use DID methods with key rotation support (such as `did:web` or `did:iota`), but `did:key` is sufficient for demonstrating the credential protocol.

Existing Approaches

The problem of digital credentials is not new, and several approaches exist. The following table summarizes the landscape, including both blockchain-based solutions and centralized platforms, to provide context for where this work fits.

Approach	Standard	Credential Format	Verification	Revocation	Portability	Status
Blockcerts	Open (MIT/Hyland)	JSON-LD	On-chain hash anchoring	Issuer-managed	Portable	Active
Hyperledger AnonCreds	Hyperledger	ZKP-based	Ledger-verified	Accumulator-based	Within ecosystem	Active
Ceramic / ComposeDB	Decentralized data	JSON streams	Stream-based	Application-level	Portable	Active
EAS (Ethereum Attestation Service)	EVM-specific	Schema-based	On-chain or off-chain	On-chain revocation	EVM networks only	Active
Credly / Accredible	Proprietary	Platform-specific	Platform-verified	Platform-controlled	Limited	Commercial
LinkedIn Endorsements	Proprietary	Unstructured	Not independently verifiable	Not revocable	Platform-locked	Commercial

Blockcerts is the closest existing work to the protocol presented here – it also anchors credential hashes on-chain for verification. Our approach differs in three respects: we use the newer W3C VC v2.0 standard (Blockcerts uses v1.1), we implement on two architecturally different blockchains to demonstrate chain-agnosticism, and we use JWT encoding for broad compatibility with existing web infrastructure.

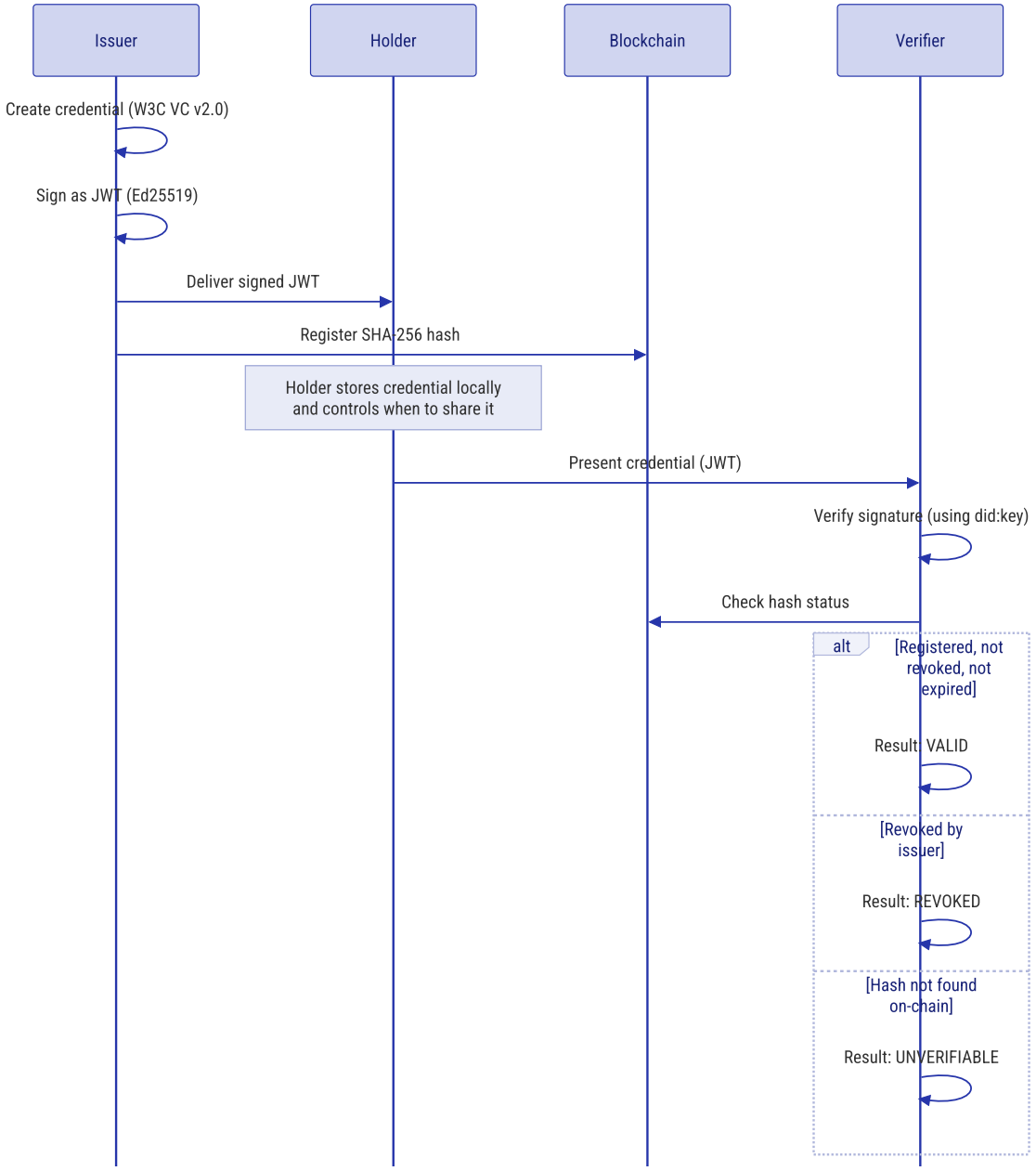
Hyperledger AnonCreds takes a fundamentally different approach, using zero-knowledge proofs to enable selective disclosure natively. This is more privacy-preserving but significantly more complex to implement and currently tied to the Hyperledger ecosystem. Our architecture is designed so that selective disclosure can be added later (see Section 9) without reworking the foundation.

EAS is EVM-specific and uses its own schema system rather than W3C VCs. It is well-suited for EVM-native applications but does not address multi-chain portability.

The centralized platforms (Credly, LinkedIn) are the most widely used in practice, but they are proprietary silos. A credential on Credly exists only on Credly. A LinkedIn endorsement cannot be independently verified and carries no cryptographic guarantee.

3. The Protocol

How It Works



The protocol has four stages:

1. **Issuance.** The issuer creates a credential following the W3C VC v2.0 format, populating it with claims about the holder (e.g., job title, certification name, endorsed skills). The issuer signs it as a JWT using their Ed25519 private key and delivers the signed JWT to the holder.

2. **Anchoring.** The issuer computes the SHA-256 hash of the complete signed JWT and registers this hash on a blockchain. The on-chain record includes the hash, the issuer's DID, and an optional expiration timestamp. This step establishes that the credential existed at a specific point in time and provides a revocation channel.

3. **Presentation.** When the holder wants to prove a credential, they present the signed JWT to a verifier. The holder controls this step entirely – no platform mediates the sharing, and the issuer is not involved.

4. **Verification.** The verifier performs two independent checks. First, they validate the JWT signature using the issuer's public key (extracted from the `did:key` in the JWT header – no network call needed). Second, they query the blockchain for the credential's hash to confirm it is registered, not revoked, and not expired. If both checks pass, the credential is valid.

Note that hashing the complete signed JWT (not just the payload) means that two issuers creating identical claims about the same subject will produce different on-chain hashes, because their different signatures produce different JWTs. Every credential has a globally unique hash.

Credential Types

The proof of concept supports three credential types, chosen to represent the breadth of professional credentials:

Type	Issued By	Key Fields	Use Case
EmploymentCredential	Employer	employerName, role, startDate, endDate	Verifiable employment history
CertificationCredential	Certification body	certificationName, certifyingBody, dateAwarded	Professional certifications with expiration
PeerEndorsement	Individual peer	endorserName, relationship, skills, statement	Skill endorsements from colleagues

EmploymentCredential represents verified employment history. An employer attests that a person held a specific role during a specific period. This is the most common credential type and the most frequently falsified on resumes.

CertificationCredential represents professional certifications issued by recognized bodies. It introduces expiration dates – a certification might be valid for three years. The on-chain record includes the expiration timestamp, and verifiers check it during validation.

PeerEndorsement demonstrates that issuers are not limited to organizations. Any individual can endorse another's skills. This is the decentralized equivalent of a LinkedIn recommendation, but cryptographically signed and independently verifiable. Three endorsers are used in the test scenarios to show that a single holder can accumulate multiple endorsements from different issuers.

Trust Model

The protocol uses an **open issuer model**: any address can register a credential hash on the blockchain. There is no gatekeeper, no approval process, and no issuer registry. Anyone can claim to be an issuer.

This is a deliberate design choice, not a missing feature. Trust is the verifier's responsibility, not the blockchain's. A verifier decides which issuers they trust, just as in the physical world: a university degree is trusted because the verifier trusts the university, not because a central authority certified the university as a valid issuer. The blockchain

provides the verification infrastructure (did this credential exist? has it been revoked?), but the trust decision remains with the verifier.

Revocation, however, is access-controlled. Only the address that originally registered a credential hash can revoke it. This prevents anyone from revoking someone else's credentials while ensuring issuers retain the ability to withdraw their attestations.

A permissioned alternative – where only pre-approved issuers can register credentials – is a valid design for specific use cases (e.g., a government-run credential registry). The open model was chosen for this PoC because it is simpler, more general, and better demonstrates the protocol's flexibility.

Advantages

Holder control. Credentials are stored by the holder, not locked in a platform. The holder decides when, with whom, and which credentials to share.

Independent verification. A verifier needs only the signed JWT and blockchain access. No API call to the issuer, no platform account, no permission required.

Transparent revocation. Revocation is recorded on a public blockchain. It cannot be done secretly, and any verifier can check the current status at any time.

Standards-based. The protocol uses W3C VC v2.0 and `did:key` – open standards that any organization can implement. No proprietary formats or vendor dependencies.

Chain-agnostic. The same credential, signed the same way, producing the same hash, can be anchored on any blockchain. The PoC demonstrates this on Ethereum and IOTA.

Privacy-preserving foundation. Credential content stays off-chain. The blockchain sees only hashes. The credential structure uses flat fields that can support selective disclosure in a future extension.

Limitations

No selective disclosure. In the current implementation, presenting a credential means presenting all of it. You cannot prove you worked at a specific company without also revealing your job title and dates. The architecture is designed so that selective disclosure can be added later (see Section 9), but it is not implemented.

No key rotation. The `did:key` method used in this PoC does not support key rotation. If an issuer's private key is compromised, all credentials signed with it are permanently tainted. Production systems would use DID methods with rotation support.

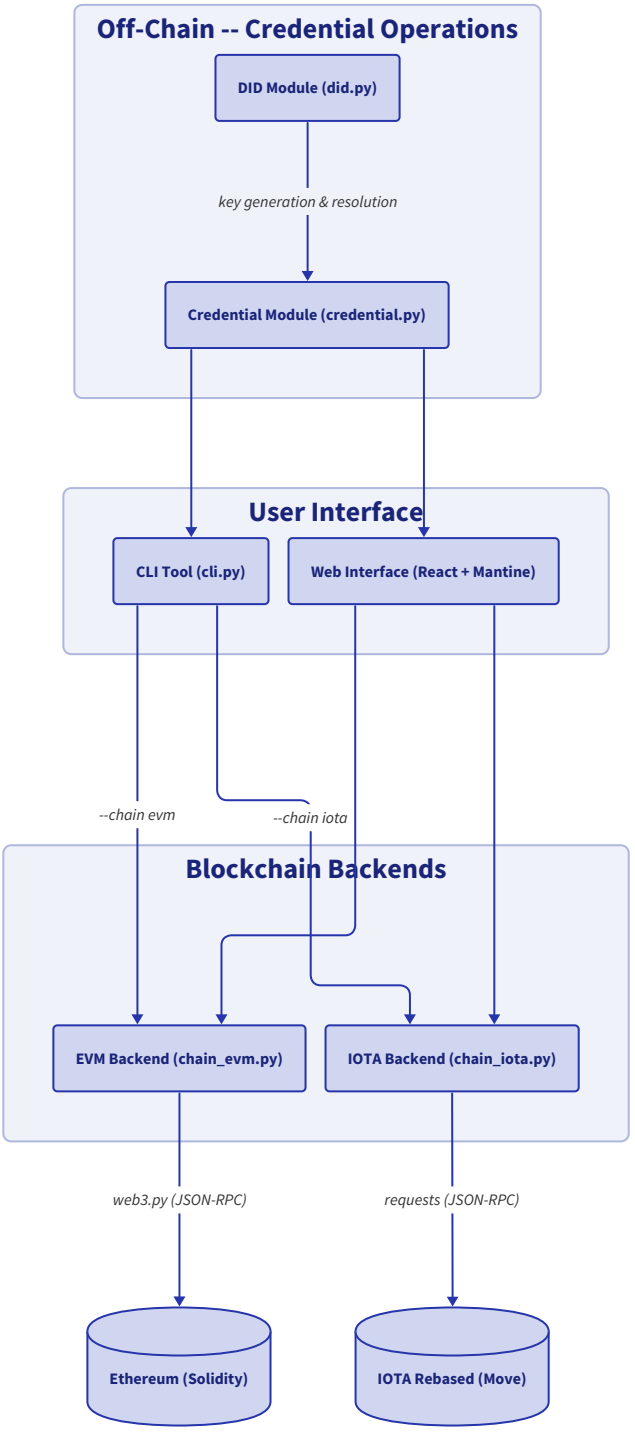
Trust bootstrapping. The protocol does not address how a verifier discovers which issuers to trust. In practice, this would involve issuer directories, reputation systems, or institutional trust frameworks – all out of scope for this PoC.

Credential recovery. If the holder loses their copy of a signed JWT, the on-chain hash alone is not useful – it proves a credential existed but cannot reconstruct its content. Credential backup and recovery is a wallet-level concern not addressed here.

On-chain cost at scale. Registering each credential individually incurs a per-transaction cost. For high-volume issuers (e.g., a university issuing thousands of diplomas), batch anchoring via Merkle trees would be more efficient. This is discussed in the paper but not implemented.

4. Architecture

System Overview



The system has three layers. The **credential operations layer** handles DID generation, credential creation, JWT signing, and signature verification – all purely off-chain, requiring no blockchain interaction. The **user interface layer** provides both a CLI and a web interface, with a chain selector that routes operations to the appropriate backend. The **blockchain**

backend layer implements the same logical interface (register, revoke, get status, check validity) for each supported chain.

This separation means that adding a new blockchain requires only a new backend module that satisfies the same interface. The credential operations, user interface, and verification logic remain unchanged.

Credential Data Model

The credential follows the W3C VC Data Model v2.0, secured as a JWT per the VC-JOSE-COSE specification. Key design decisions:

The **full credential is the JWT payload**. In the earlier v1.1 specification, the credential was wrapped in a `vc` claim within the JWT. Version 2.0 removes this wrapping – the credential fields (`@context` , `type` , `issuer` , `credentialSubject` , etc.) are top-level JWT claims.

`validFrom` and `validUntil` replace the v1.1 fields `issuanceDate` and `expirationDate` . These describe the credential's validity period. The JWT's own `iat` and `exp` fields (if present) describe the *signature's* validity, which is a distinct concept.

Flat `credentialSubject` fields enable future selective disclosure. Rather than nesting claims in complex structures, each claim is a top-level field within `credentialSubject` . This means individual fields can be disclosed or withheld independently when selective disclosure is added.

The **JWT header** includes `typ: "vc+ld+jwt"` (per the VC-JOSE-COSE spec), `alg: "EdDSA"` (Ed25519), and a `kid` pointing to the issuer's `did:key` . A verifier extracts the public key from the `kid` to validate the signature.

The **on-chain anchor** is the SHA-256 hash of the complete signed JWT string. This is a 32-byte value that fits naturally as a `bytes32` on EVM or a `vector<u8>` on IOTA.

Smart Contract Design

Both blockchain implementations satisfy the same logical interface:

- `registerCredential(hash, issuer, expiration)` – anchor a credential hash with metadata
- `revokeCredential(hash)` – permanently revoke (original registrant only)
- `getCredentialStatus(hash)` – read the full on-chain record
- `isCredentialValid(hash)` – convenience check (registered, not revoked, not expired)

On EVM, the contract uses Solidity's `mapping(bytes32 => CredentialRecord)` for storage. A separate `mapping(bytes32 => address)` tracks the registrant address for revocation access control, kept separate from the public record because blockchain addresses are chain-specific and should not leak into the credential data model. Read operations are standard Solidity `view` functions with zero gas cost.

On IOTA, the contract uses a shared `Registry` object containing a `Table<vector<u8>, CredentialRecord>` – the closest analog to Solidity's mapping pattern and the design that enables a direct cross-chain comparison. The `Registry` is a shared object created during module initialization, allowing any address to interact with it. Timestamps come from IOTA's `Clock` object (millisecond precision, compared to EVM's second-precision `block.timestamp`).

An alternative IOTA design (Option B) would create individual owned objects per credential rather than entries in a shared Table. This is more idiomatic for Move's object model – each credential would be an independently owned and transferable object. The tradeoff is that querying the registry becomes harder (no single object to read from) while composability improves (credentials can be transferred, wrapped, or used in programmable transactions). Option A was chosen for this PoC because it provides the most direct comparison with the EVM implementation.

Cross-Chain Considerations

Implementing the same protocol on two architecturally different blockchains revealed several differences that affect both the implementation approach and the developer experience:

State model. EVM uses an account-based model where contract state is accessed through function calls. IOTA uses an object-based model where state lives in objects that can be read directly. For read operations, this meant EVM uses `view` function calls (zero gas cost) while IOTA uses direct object reads via the `iotax_getDynamicFieldObject` RPC method (also zero cost, but a fundamentally different mechanism).

Timestamp resolution. EVM's `block.timestamp` is in seconds. IOTA's `Clock` is in milliseconds. The CLI normalizes both to seconds for consistent output, but the contracts store timestamps in their native resolution.

Transaction signing. EVM uses the well-established secp256k1 ECDSA scheme. IOTA uses Ed25519 with an intent-signing scheme: the transaction data is prefixed with a 3-byte intent marker, the result is hashed with Blake2b-256, and the hash is signed. The serialized signature includes a scheme flag byte and the public key alongside the signature itself. This is straightforward to implement but not documented in a Python-friendly way.

Cost model. EVM charges gas (computation units multiplied by a gas price). IOTA charges directly in NANOS (the smallest IOTA denomination), broken down into computation cost, storage cost, and storage rebate. The two models are not directly comparable in absolute terms, but the relative costs of different operations are informative.

5. Technical Implementation

This section is intended for developers. It assumes familiarity with Solidity, Move, Python, Docker, and blockchain development concepts. Web applications are also referenced.

Smart Contracts

CredentialRegistry.sol (Solidity 0.8.28) is a 170-line contract that manages the credential registry on EVM chains. It uses custom errors (`CredentialAlreadyRegistered` , `CredentialNotFound` , `NotOriginalRegistrant` , `CredentialAlreadyRevoked` , `EmptyIssuer`) rather than require strings for gas efficiency. Events are emitted for both registration and revocation, enabling off-chain indexing. The `isCredentialValid` function performs all three checks (exists, not revoked, not expired) in a single call, comparing `expiration` against `block.timestamp` .

credential_registry.move (IOTA Move, 223 lines) implements the same interface using a shared `Registry` object. The module's `init` function creates the Registry and shares it via `transfer::share_object` , making it accessible to all network participants. Table operations use IOTA's v1.1.0 Table API, where keys are passed by value.
The `is_credential_valid` function takes a `Clock` reference for timestamp comparison, which is a required pattern in Move since there is no global block timestamp equivalent. Error codes are defined as module-level constants. A complete test suite (6 tests) covers registration, revocation, expiration, duplicate registration prevention, and access control.

CLI Tool

The CLI is built with Python and Click. It provides eight commands: `generate-key` , `deploy` , `issue` , `register` , `revoke` , `verify` , `status` , and `list` .

`credential.py` handles W3C VC v2.0 creation, JWT signing, signature verification, and hashing. It uses PyJWT with Ed25519 keys from the `cryptography` library. The credential types and their required fields are defined in a simple dictionary, making it easy to add new types.

`did.py` implements `did:key` generation and resolution. Generating a key produces a DID and a private key. Resolving a DID extracts the Ed25519 public key from the multicodec-encoded identifier. No network call is required in either direction.

`chain_evm.py` (219 lines) uses web3.py for all blockchain interaction. Contract compilation happens inline via py-solc-x. Transaction building, signing, and submission follow standard web3.py patterns. Read operations are simple view calls.

`chain_iota.py` (620 lines) uses pure HTTP requests against the IOTA JSON-RPC API. There is no Python SDK for IOTA Rebased – this is a key developer experience finding. Transaction building uses the `unsafe_moveCall` and `unsafe_publish` RPC methods, which construct unsigned transactions server-side. Signing is implemented locally following IOTA's intent-signing scheme (3-byte intent prefix, Blake2b-256 hash, Ed25519 signature, serialized as `flag || signature || public_key`). Read operations use `iotax_getDynamicFieldObject` to read credential records directly from the Registry's Table – a more natural approach for IOTA's object model than trying to call view functions via devInspect, which proved problematic on v1.1.0 (see Section 5.3).

Both chain backends expose the same interface: `deploy_contract` , `register_credential` , `revoke_credential` , `get_credential_status` , `is_credential_valid` , and `get_gas_used` (the name follows IOTA's own RPC terminology; the CLI displays "Cost (NANOS)" for IOTA output). The CLI's `cli.py` selects the backend based on the `--chain` flag, and each command resolves its private key from chain-specific environment variables (`EVM_DEPLOYER_PRIVATE_KEY` / `IOTA_DEPLOYER_PRIVATE_KEY` , etc.).

Developer Experience Comparison

Implementing the same protocol on both chains provides a direct comparison of the developer experience. This is one of the more practically useful findings of this research.

Python ecosystem maturity. EVM has web3.py, a comprehensive Python SDK with years of production use. It handles ABI encoding, transaction building, signing, gas estimation, and receipt parsing. IOTA Rebased's primary SDK is TypeScript. Developing in Python required constructing raw JSON-RPC calls, implementing the signing scheme manually, and parsing BCS-encoded responses. The IOTA backend is three times the size of the EVM backend (620 vs 219 lines) largely because of this gap.

Contract compilation. Solidity can be compiled inline from Python using py-solc-x. Move requires an external toolchain (`iota move build` running in a Docker container). This adds a build step to the deployment pipeline that doesn't exist for EVM.

Read operations. On EVM, reading contract state is a simple view function call – web3.py handles the ABI encoding and decoding transparently. On IOTA, the initial approach (calling view functions via `devInspectTransactionBlock`) failed due to transaction format incompatibilities on v1.1.0. The working solution reads credential records directly from the on-chain Table using dynamic field lookups (`iotax_getDynamicFieldObject`). This is actually a more natural approach for IOTA's object model, but discovering it required significant trial and error.

Terminology. IOTA's JSON-RPC API uses "gas" terminology (inherited from its Sui codebase) while IOTA's own documentation refers to "transaction fees" and "costs" measured in NANOS. This mismatch adds cognitive overhead when working with the API.

Aspect	EVM (Ethereum)	IOTA Rebased
Python SDK	web3.py (mature, comprehensive)	None (raw JSON-RPC)
Contract compilation	Inline (py-solc-x)	External (iota move build)
Backend code size	219 lines	620 lines
Transaction signing	Handled by web3.py	Manual implementation (intent-signing)
Read operations	View function calls	Direct object/dynamic field reads
Key dependencies	web3.py, py-solc-x	requests, bech32, cryptography
API documentation	Extensive, Python examples	TypeScript-focused, Python gaps

The IOTA experience is not negative – it is simply less mature for Python developers. The object model is powerful (direct state reads are elegant), Move's type safety caught issues at compile time that Solidity would only surface at runtime, and the feeless transaction model on IOTA mainnet makes high-volume credential anchoring economically attractive. But a Python developer working with IOTA today should expect to do more manual work than they would with EVM.

6. Running the Proof of Concept

Prerequisites

- Docker and Docker Compose
- The Ethereum Local Testing Starter Pack
- The IOTA Local Testing Starter Pack

Configuring the Environment

```
cd decentralized-professional-credentials-protocol
cp .env.example .env
```

Edit `.env` to set the IOTA starter pack path:

```
IOTA_STARTER_PACK_DIR=/path/to/IOTA-Local-Testing-Starter-Pack
```

The remaining defaults (RPC URLs, pre-funded account keys) work with the starter packs out of the box.

Build the Docker image:

```
docker compose build
```

Running the EVM Scenarios

Step 1: Start the Ethereum network.

```
cd /path/to/Ethereum-Local-Testing-Starter-Pack
./eth.sh start
```

Step 2: Run the scenarios.

```
cd /path/to/decentralized-professional-credentials-protocol
./run_scenarios_evm.sh
```

The script generates fresh keys, deploys the contract, then runs all three scenarios (employment with revocation, certification with expiration, peer endorsements). Each scenario issues a credential, registers it on-chain, and verifies it. The employment scenario additionally revokes the credential and verifies the revocation. Expected output: all verifications show the correct status (VALID, REVOKED, or EXPIRED as applicable).

Running the IOTA Scenarios

Step 1: Start the IOTA network.

```
cd /path/to/IOTA-Local-Testing-Starter-Pack
./iota.sh start
```

Step 2: Run the scenarios.

```
cd /path/to/decentralized-professional-credentials-protocol
./run_scenarios_iota.sh
```

The script first compiles the Move contract using the starter pack's `iota-tools` container (this fetches the IOTA framework dependency from GitHub on the first run, which takes approximately one minute). It then runs the same three scenarios as the EVM script. Expected output is identical in structure, with IOTA-specific cost metrics shown in NANOS.

Running the Move Contract Tests

To run the Move unit test suite independently:

```
cd /path/to/IOTA-Local-Testing-Starter-Pack
docker compose run --rm \
    -v /path/to/decentralized-professional-credentials-protocol/contracts/iota:/tmp/contract \
    iota-tools \
    sh -c "cd /tmp/contract && iota move test"
```

All six tests should pass: registration and query, revocation, expiration, duplicate registration prevention, non-registrant revocation prevention, and nonexistent credential handling.

7. Web Interface

Overview

The proof of concept includes a web interface that demonstrates the full credential lifecycle interactively. While the CLI and automated scenarios prove the protocol works, the web interface makes it tangible – a user can issue a credential, see it in their wallet, register it on a blockchain, verify it, revoke it, and verify again, all through a browser.

The interface is built with React and Mantine 8, served via nginx, and communicates with the existing FastAPI backend. It runs as a separate Docker Compose profile alongside the API, so it is optional – the CLI and scenario scripts work independently.

Running the Web Interface

The web interface requires a deployed contract and at least one keypair. If you have already run the automated scenarios (Section 6), both are in place. Otherwise, deploy the contract and generate keys first using the scenario scripts or the CLI.

Start the API and web interface:

```
docker compose --profile ui up api web
```

The web interface is available at `http://localhost:3000`. The API runs at `http://localhost:8000` (proxied through nginx, so the frontend accesses it at the same origin).

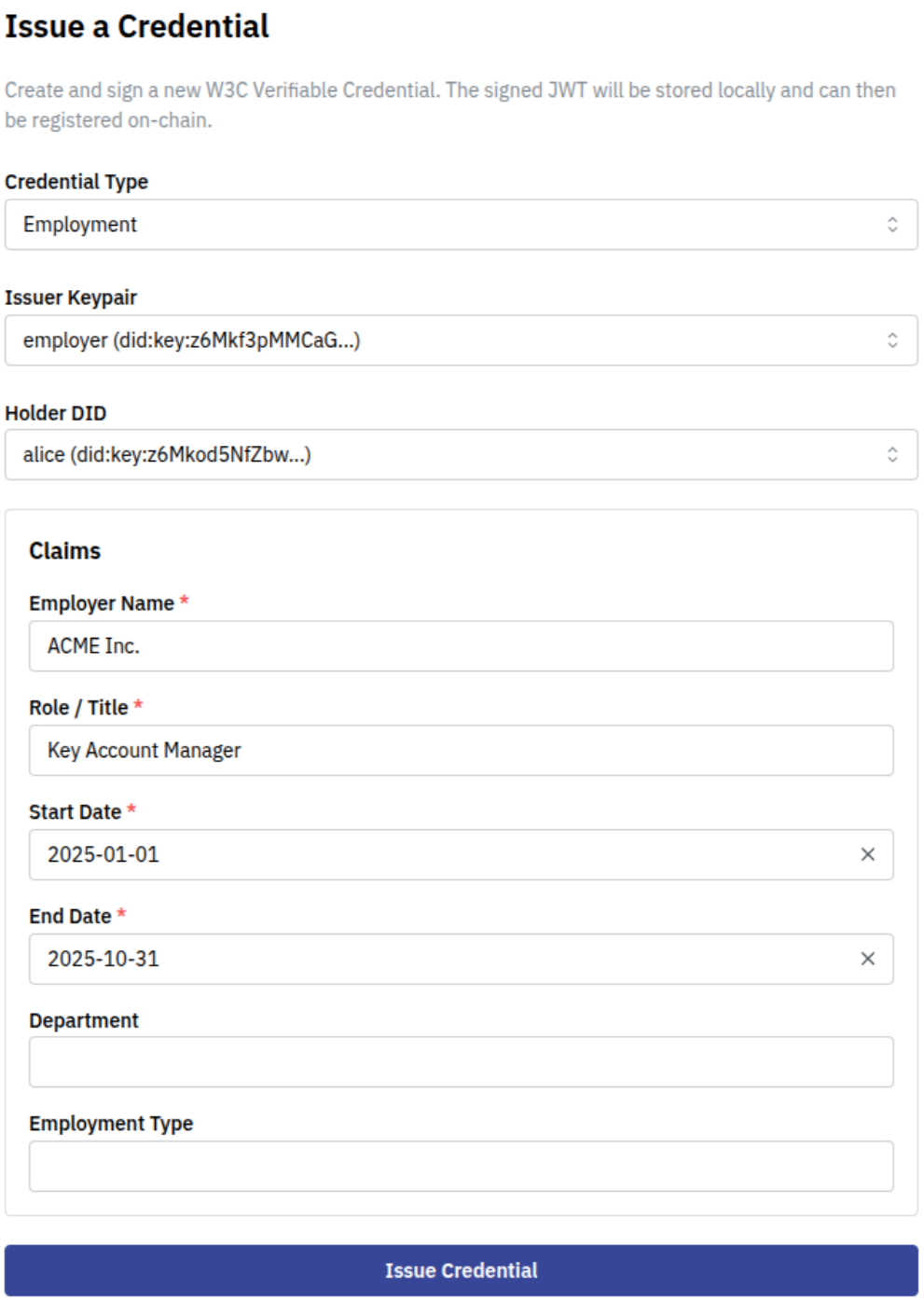
A chain selector in the header controls which blockchain is used for all on-chain operations. Select EVM or IOTA depending on which network is running and which contract is deployed.

Walkthrough

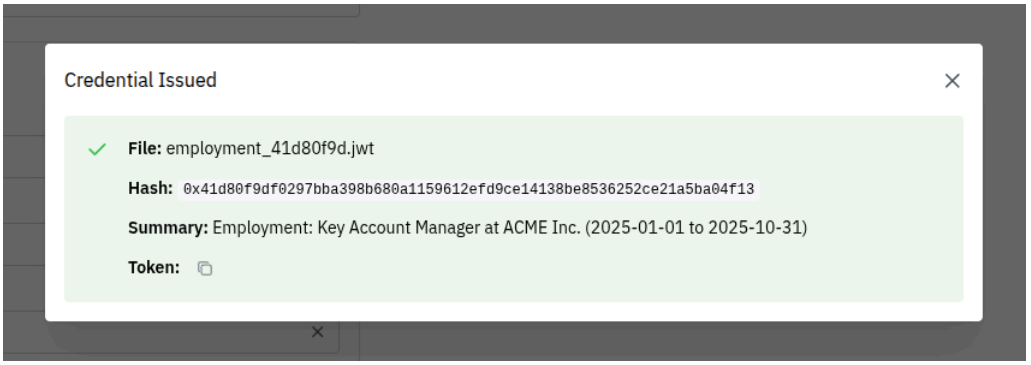
The following walkthrough demonstrates the complete credential lifecycle through the web interface.

Step 1: Issue an employment credential.

Select “Employment” as the credential type, choose an issuer keypair (e.g., “employer”), select or paste the holder’s DID (e.g., Alice’s DID), and fill in the claims: employer name, role, start date, and end date. Click “Issue Credential.”

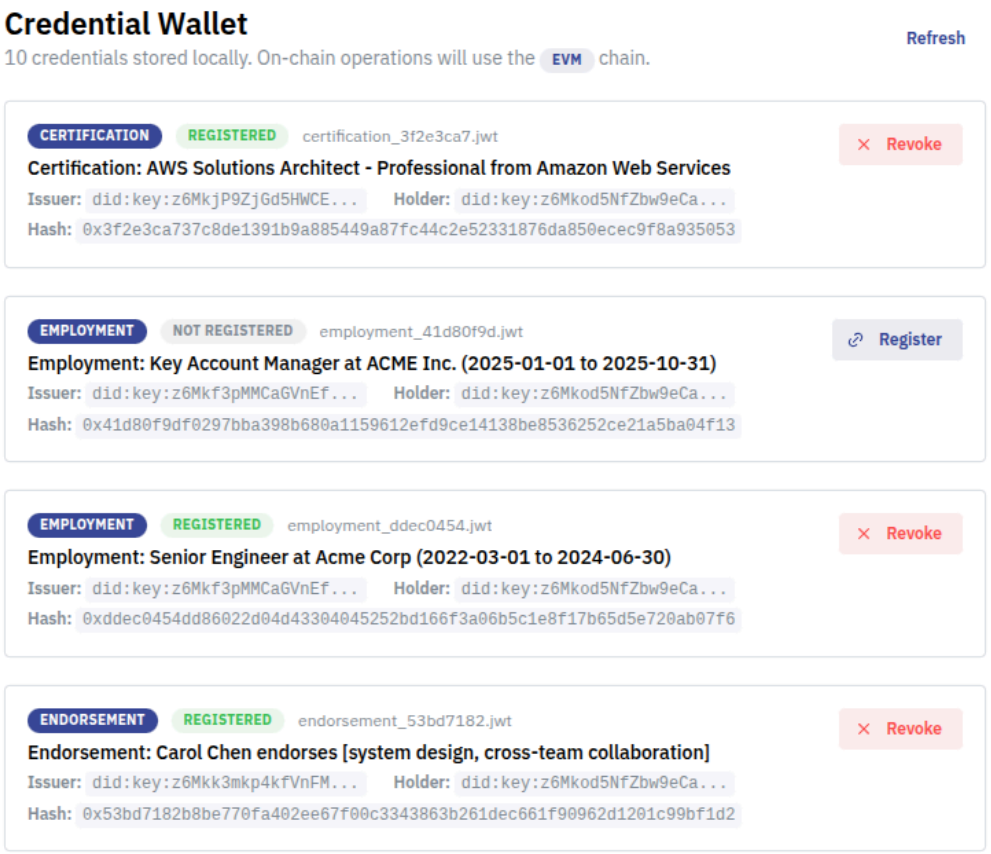


A modal appears showing the result: the credential filename, its SHA-256 hash, a summary, and a copy button for the JWT token. Copy the token – you will need it for verification.



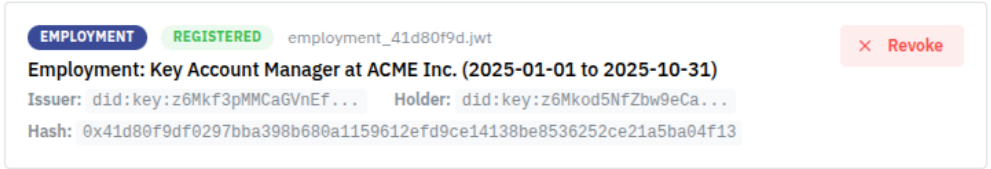
Step 2: View the credential in the wallet.

Switch to the Credentials tab. The newly issued credential appears in the list with its type, summary, issuer and holder DIDs, and hash. The on-chain status shows “Not Registered” since it has not been anchored yet.



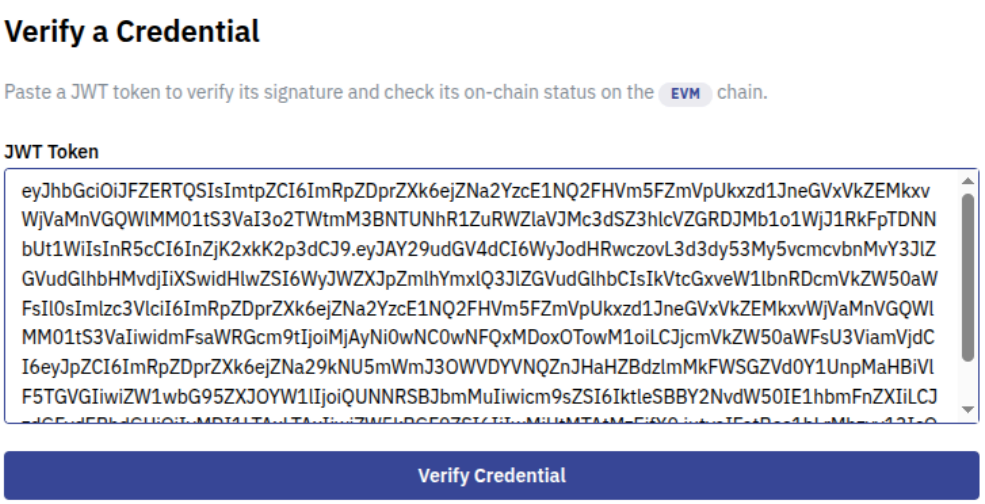
Step 3: Register the credential on-chain.

Click the “Register” button on the credential card. The interface sends the credential hash to the selected blockchain. After confirmation, the status badge updates to “Registered” and the transaction cost is displayed.



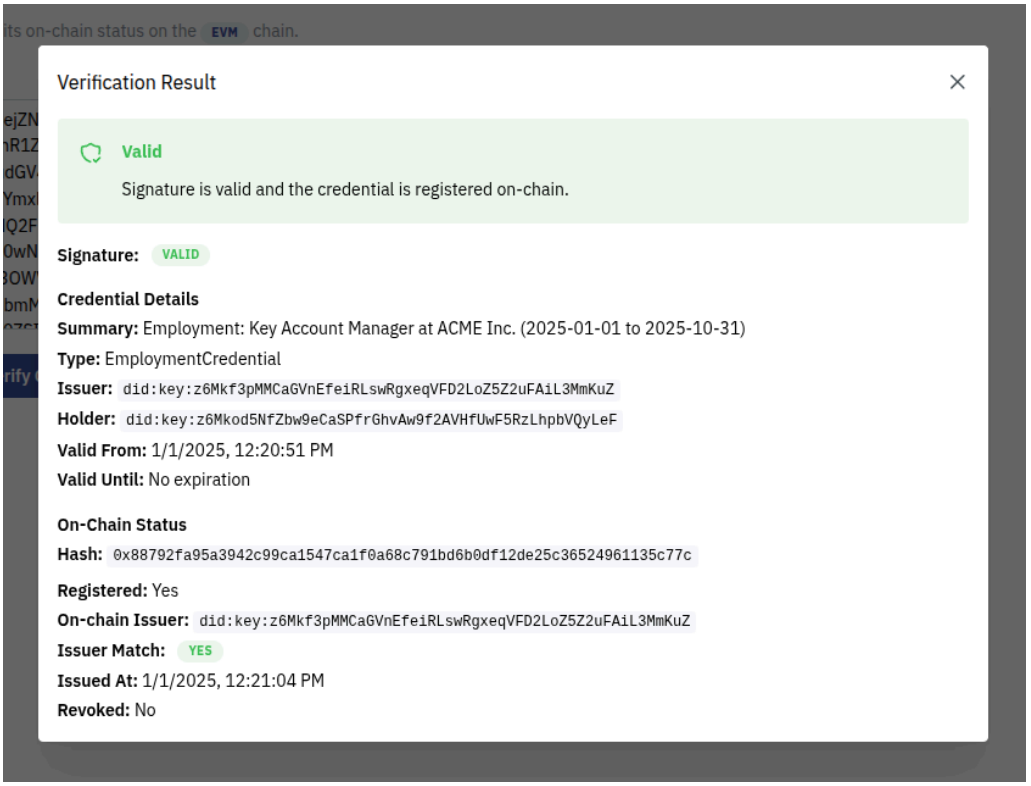
Step 4: Verify the credential.

Switch to the Verify tab. Paste the JWT token copied in Step 1 and click “Verify.”



The interface checks the JWT signature (off-chain, using the issuer’s public key from the `did:key`) and queries the blockchain for the hash status.

A result modal appears showing the verification outcome. The credential is VALID: the signature is authentic, the hash is registered on-chain, it has not been revoked, and it has not expired.

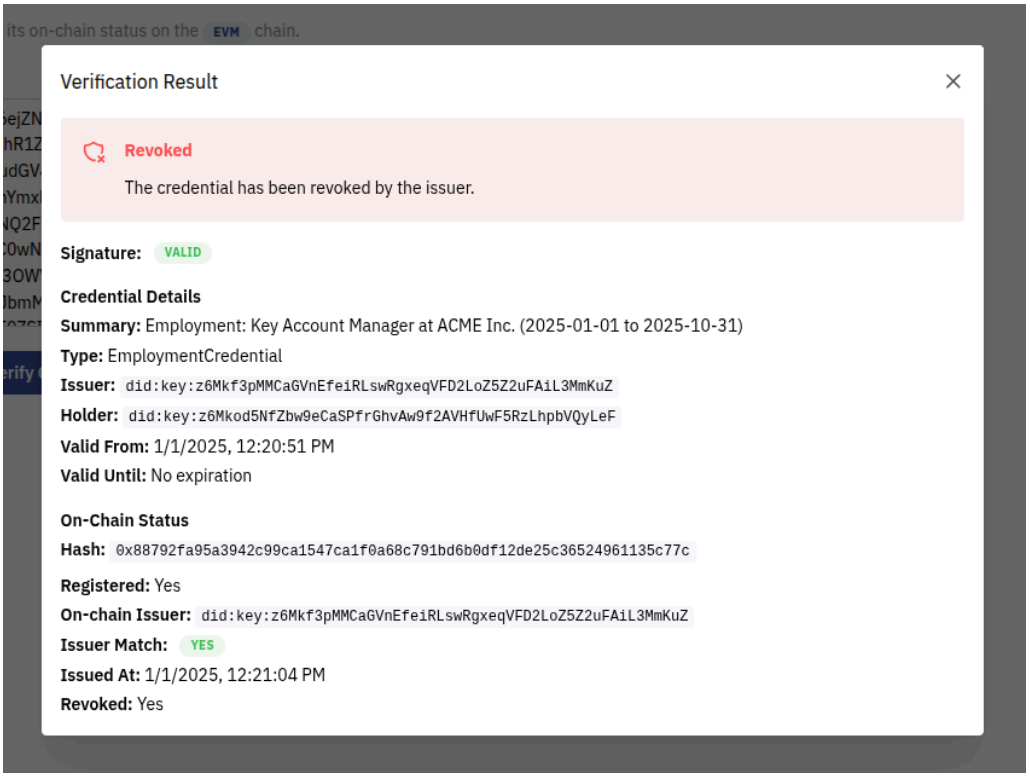


Step 5: Revoke the credential.

Return to the Credentials tab and click “Revoke” on the registered credential. After confirmation, the status badge updates to “Revoked.”

Step 6: Verify again.

Switch back to the Verify tab and verify the same JWT token again. This time, the result shows REVOKED: the signature is still valid (the JWT itself has not changed), but the on-chain status indicates the issuer has withdrawn their attestation.



This walkthrough demonstrates the core value proposition of the protocol: credentials are created and signed off-chain (fast, free), anchored on-chain (tamper-proof), and verifiable by anyone with the JWT and blockchain access (independent, instant). Revocation is transparent – a verifier always sees the current status.

8. Results

We ran all three credential scenarios on both chains. Each scenario was executed via the automated scenario scripts on local test networks (Ethereum via the Ethereum Local Testing Starter Pack, IOTA via the IOTA Local Testing Starter Pack). The same credential types, claims, and lifecycle operations were performed on both chains, providing a direct comparison.

EVM Scenario Results

All three scenarios completed successfully on EVM.

Scenario	Operation	Gas Used	Status
Employment	Register	140,242	VALID
Employment	Revoke	31,976	REVOKED
Certification	Register (with expiration)	140,290	VALID
Endorsement 1	Register	140,242	VALID
Endorsement 2	Register	140,242	VALID
Endorsement 3	Register	140,242	VALID

Registration cost is highly consistent at approximately 140,242 gas units (slightly higher at 140,290 when an expiration timestamp is included, due to the non-zero value requiring marginally more storage). Revocation is significantly cheaper at 31,976 gas – it only flips a boolean and sets a timestamp in an existing storage slot. Verification (view call) costs zero gas.

IOTA Scenario Results

All three scenarios completed successfully on IOTA.

Scenario	Operation	Cost (NANOS)	Status
Employment	Register	4,974,800	VALID
Employment	Revoke	1,000,000	REVOKED
Certification	Register (with expiration)	3,994,400	VALID
Endorsement 1	Register	3,994,400	VALID
Endorsement 2	Register	3,994,400	VALID
Endorsement 3	Register	3,994,400	VALID

The first registration (employment credential) costs 4,974,800 NANOS – higher than subsequent registrations (3,994,400 NANOS each) because the first entry in a Table incurs additional storage allocation overhead. Subsequent registrations are consistent at 3,994,400 NANOS. Unlike EVM, the expiration timestamp does not affect the cost because IOTA's storage model charges per-object rather than per-byte. Revocation costs 1,000,000 NANOS (the minimum computation cost), with no net storage change because it modifies an existing record. Verification costs nothing – it reads on-chain state directly via `iotax_getDynamicFieldObject` without submitting a transaction.

An important caveat: these costs are from local test networks. On IOTA mainnet, basic transactions are feeless (subsidized by the network's economic model). The NANOS costs shown here reflect the local test network's fee structure, which mirrors the gas accounting but not the mainnet economics. For a production deployment on IOTA, the cost of credential anchoring would effectively be zero, making it particularly attractive for high-volume issuers.

Cross-Chain Comparison

Metric	EVM (Ethereum)	IOTA Rebased
Register cost	~140,242 gas	3,994,400 – 4,974,800 NANOS
Revoke cost	~31,976 gas	~1,000,000 NANOS
Verify cost	0 (view call)	0 (direct read)
Cost unit	Gas (priced in ETH)	NANOS (1 IOTA = 10^9 NANOS)
Mainnet cost model	Gas price varies with demand	Feeless for basic transactions
Timestamp precision	Seconds	Milliseconds
Contract language	Solidity	Move
Contract size	170 lines	223 lines
Backend code size (Python)	220 lines	640 lines
Read mechanism	View function call	Dynamic field lookup
Signing scheme	secp256k1 (ECDSA)	Ed25519 + intent-signing
Python SDK	web3.py	None (raw JSON-RPC)

The absolute cost numbers between EVM and IOTA are not directly comparable – they use different units with different real-world valuations. What is comparable is the relative cost structure: on both chains, registration is the most expensive operation (new storage allocation), revocation is cheaper (modifying existing storage), and verification is free. This consistent pattern validates that the protocol's cost characteristics are not chain-specific.

Observations

The protocol is genuinely chain-agnostic. The same credential, signed the same way with the same hash, is verifiable on either chain. The chain-specific code (backend modules and smart contracts) implements the same interface, and the credential operations layer is entirely chain-independent. Adding a third blockchain would require only a new backend module and contract – no changes to the credential format, signing, or verification logic.

Developer experience is the primary differentiator, not capability. Both chains can implement the protocol correctly. The difference is in how much work it takes. EVM's mature Python tooling (web3.py) made the backend implementation straightforward. IOTA's lack of a Python SDK required three times more code to achieve the same result. This gap is a maturity issue, not a fundamental limitation – TypeScript developers using IOTA's official SDK would likely find the experience comparable to web3.py.

IOTA's object model enables a natural read pattern. The direct dynamic field read (`iotax_getDynamicFieldObject`) is arguably more elegant than EVM's view-function approach. It reads the on-chain data structure directly rather than executing contract code, and it returns the full record in a single RPC call. This pattern was discovered as a workaround for devInspect limitations on v1.1.0, but it turned out to be the better approach regardless.

Move's type safety provides stronger compile-time guarantees. The Move compiler enforces resource ownership, prevents double-spending of objects, and catches type errors that Solidity would only surface at runtime. The test framework's `expected_failure` annotations allow precise specification of which error codes are expected. However, Move's strict type system also means more explicit code – the IOTA contract is 31% larger than the Solidity contract despite implementing the same logic.

The credential format is the truly portable layer. The most significant finding is that the JWT credential itself – not the smart contract, not the backend code – is the portability layer. A credential issued for the Ethereum implementation is bit-for-bit identical to one issued for IOTA, because the issuance and signing happen entirely off-chain. The on-chain component is an anchoring convenience, not a dependency.

9. Future Directions

Selective Disclosure

The most important planned extension is selective disclosure: the ability to prove specific claims within a credential without revealing the full document. A job applicant could prove they worked at a specific company without revealing their salary or exact dates. A certification holder could prove they hold a valid certification without revealing its level.

The current credential architecture is designed to support this. Fields within `credentialSubject` are flat (not nested), enabling per-field disclosure. Implementation approaches include SD-JWT (Selective Disclosure for JWTs, an IETF draft), BBS+ signatures (which enable zero-knowledge proofs over signed attributes), and hash-based field commitments (where each field is individually hashed and the Merkle root is anchored on-chain).

Selective disclosure is scoped as a separate follow-up paper in the Consensix Labs research series. It builds on the credential foundation established here without requiring changes to the on-chain contracts or the verification protocol.

Production Identity

Moving beyond `did:key` for production use requires DID methods that support key rotation, organizational identity, and revocation of the identifiers themselves. Candidates include `did:web` (using existing web infrastructure), `did:iota` (using IOTA's Decentralized Identity framework), and `did:ion` (using Bitcoin's blockchain for DID anchoring). Integration with existing identity providers (OAuth, SAML) could enable issuer authentication without requiring issuers to manage their own DIDs.

Credential Storage and Wallets

The proof of concept stores credentials as JWT files on disk. A production system would need a proper credential wallet – a secure, user-controlled store that manages credentials, handles backup and recovery, supports cross-device synchronization, and provides a consistent presentation interface. This is relevant to Consensix Labs' C6 Test Wallet project, which could integrate credential management alongside its existing multi-chain wallet capabilities.

Scaling and Performance

For high-volume issuers, the per-credential anchoring model can be replaced with batch anchoring. The issuer collects a batch of credential hashes, constructs a Merkle tree, and anchors only the root hash on-chain. Verification then involves checking a credential's hash against the Merkle proof and confirming the root hash on-chain. This reduces on-chain costs from one transaction per credential to one transaction per batch, with no change to the verification guarantee.

More efficient revocation mechanisms (bitmap-based status lists as defined in the W3C Bitstring Status List specification, or cryptographic accumulators) would scale better than per-credential revocation records for registries with millions of credentials.

Additional Blockchain Networks

The protocol is chain-agnostic by design. Future implementations could target Cosmos SDK / CosmWasm (relevant to the planned Consensix chain), Bitcoin (via OP_RETURN or Taproot commitments for hash anchoring), or Substrate / Polkadot. Each additional chain would further validate the protocol's portability claim and provide data for the developer experience comparison.

Appendix A: Project File Reference

Repository Structure

```
decentralized-professional-credentials/
├── compose.yml                              # Docker Compose (CLI service + UI profile)
├── Dockerfile                               # Python container (shared by CLI and API)
├── .env.example                             # Environment configuration template
├── run_scenarios_evm.sh                     # End-to-end EVM scenario script
├── run_scenarios_iota.sh                    # End-to-end IOTA scenario script
├── app/
│   ├── cli.py                               # CLI entrypoint (Click)
│   ├── api.py                               # REST API entrypoint (FastAPI)
│   ├── credential.py                        # W3C VC v2.0 creation and JWT signing
│   ├── did.py                               # did:key generation and resolution
│   ├── chain_evm.py                         # EVM backend (web3.py + py-solc-x)
│   ├── chain_iota.py                        # IOTA backend (pure JSON-RPC)
│   └── requirements.txt                     # Python dependencies
├── web/
│   ├── Dockerfile                           # Multi-stage build (Node 22 + nginx)
│   ├── nginx.conf                           # nginx config (serves app + proxies /api/)
│   ├── src/
│   │   ├── App.jsx                          # Main app shell with navigation and chain selector
│   │   ├── views/
│   │   │   ├── IssueView.jsx                # Issue credential form
│   │   │   ├── CredentialsView.jsx          # Credential wallet with status badges
│   │   │   └── VerifyView.jsx               # Credential verification
│   │   └── ...
│   └── package.json                         # React + Mantine 8 dependencies
├── contracts/
│   ├── evm/
│   │   └── CredentialRegistry.sol           # Solidity contract (0.8.28)
│   └── iota/
│       ├── Move.toml                        # Move package manifest (v1.1.0)
│       ├── sources/
│       │   └── credential_registry.move     # Move contract
│       └── tests/
│           └── credential_registry_tests.move # Move unit tests (6 tests)
├── samples/
│   ├── employment_claims.json               # Sample employment credential claims
│   ├── certification_claims.json            # Sample certification credential claims
│   └── endorsement_claims.json              # Sample peer endorsement claims (3 endorsers)
├── credentials/                             # Generated credential JWTs (gitignored)
├── keys/                                    # Generated keypairs (gitignored)
└── reports/                                 # Generated reports (gitignored)
```

Environment Variables

Variable	Required	Description
`EVM_RPC_URL`	For EVM	Ethereum JSON-RPC endpoint
`EVM_DEPLOYER_PRIVATE_KEY`	For EVM	Deployer account private key (hex)
`EVM_ISSUER_PRIVATE_KEY`	For EVM	Issuer account private key for register/revoke
`EVM_CONTRACT_ADDRESS`	For EVM	Deployed contract address (set after deploy)
`IOTA_RPC_URL`	For IOTA	IOTA full node JSON-RPC endpoint
`IOTA_DEPLOYER_PRIVATE_KEY`	For IOTA	Deployer account private key (iotaprivkey1…)
`IOTA_ISSUER_PRIVATE_KEY`	For IOTA	Issuer account private key for register/revoke
`IOTA_CONTRACT_ADDRESS`	For IOTA	Deployed contract address (PACKAGE_ID:REGISTRY_ID)
`IOTA_STARTER_PACK_DIR`	For IOTA	Path to the IOTA Local Testing Starter Pack

Appendix B: Full Scenario Output

B.1: EVM Scenarios

```
═══════════════════════════════════════════════════════════
  Setup: Generating Keys
═══════════════════════════════════════════════════════════

→ Cleaning up previous run artifacts...
→ Generating issuer key (employer)...
Generated keypair: employer
  DID:  did:key:z6Mkif2g9x66Sry7YEM7uyG1i2GpD9SQLSK5TJcMRnAKSf5b
  Saved to: /app/keys/employer.json
→ Generating issuer key (certification body)...
Generated keypair: certbody
  DID:  did:key:z6MkvcmYDW4tAufms4XunUEhzQL9aZGq75esGTdhcS4NtftX
  Saved to: /app/keys/certbody.json
→ Generating endorser keys (3 peers)...
Generated keypair: endorser-bob
  DID:  did:key:z6MkpJVPEiX6DroR8HShMbHWSLk9xfH68hfuhCor4cmKfqEu
  Saved to: /app/keys/endorser-bob.json
Generated keypair: endorser-carol
  DID:  did:key:z6MknQujbBWJxiRSRiudWLy9NzVkWQtSnH2mSB55ZnFMNeog
  Saved to: /app/keys/endorser-carol.json
Generated keypair: endorser-dave
  DID:  did:key:z6MkfR82B2UmWSF1vcYokcLwtYUZ7tNVqZTPwH26gWM1m25F
  Saved to: /app/keys/endorser-dave.json
→ Generating holder key (Alice)...
Generated keypair: alice
  DID:  did:key:z6MkesEmD6CYnNLAdLEiWC1JE2AckndT95L7StJmpzVngvyf
  Saved to: /app/keys/alice.json
  Alice's DID: did:key:z6MkesEmD6CYnNLAdLEiWC1JE2AckndT95L7StJmpzVngvyf

═══════════════════════════════════════════════════════════
  Setup: Deploying Contract
═══════════════════════════════════════════════════════════

→ Deploying CredentialRegistry on evm...
Deploying CredentialRegistry on evm...
Contract deployed at: 0xf2EAC808a7669eCdf9Dd250918CAB3ff9244064D

Add to your .env file:
  EVM_CONTRACT_ADDRESS=0xf2EAC808a7669eCdf9Dd250918CAB3ff9244064D
→ Contract address: 0xf2EAC808a7669eCdf9Dd250918CAB3ff9244064D

═══════════════════════════════════════════════════════════
  Scenario 1: Employment Credential
═══════════════════════════════════════════════════════════

→ Issuing employment credential...
Issuer: did:key:z6Mkif2g9x66Sry7YEM7uyG1i2GpD9SQLSK5TJcMRnAKSf5b
Credential: Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
Hash: 0x91431f5f29eec9f353550e11d6b9d1359307b37a5dae053ca7b7781f12af12c0
Saved to: credentials/employment_91431f5f.jwt
→ Registering on-chain...
Credential: Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
Hash: 0x91431f5f29eec9f353550e11d6b9d1359307b37a5dae053ca7b7781f12af12c0
```

```
Issuer DID: did:key:z6Mkif2g9x66Sry7YEM7uyG1i2GpD9SQLSK5TJcMRnAKSf5b
Expiration: none
Registering on evm...
Registered. Gas used: 140242
→ Verifying (should be VALID)...
Checking signature...
  Signature: VALID
  Credential: Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
  Issuer: did:key:z6Mkif2g9x66Sry7YEM7uyG1i2GpD9SQLSK5TJcMRnAKSf5b
  Holder: did:key:z6MkesEmD6CYnNLAdLEiWC1JE2AckndT95L7StJmpzVngvyf

Checking on-chain status (evm)...
  Hash: 0x91431f5f29eec9f353550e11d6b9d1359307b37a5dae053ca7b7781f12af12c0
  Registered: yes (at block timestamp 1775306055)
  Issuer (on-chain): did:key:z6Mkif2g9x66Sry7YEM7uyG1i2GpD9SQLSK5TJcMRnAKSf5b

  Result: VALID
→ Revoking credential (employer terminates attestation)...
Revoking credential 0x91431f5f29eec9f353550e11d6b9d1359307b37a5dae053ca7b7781f12af12c0 on evm...
Revoked. Gas used: 31976
→ Verifying again (should be REVOKED)...
Checking signature...
  Signature: VALID
  Credential: Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
  Issuer: did:key:z6Mkif2g9x66Sry7YEM7uyG1i2GpD9SQLSK5TJcMRnAKSf5b
  Holder: did:key:z6MkesEmD6CYnNLAdLEiWC1JE2AckndT95L7StJmpzVngvyf

Checking on-chain status (evm)...
  Hash: 0x91431f5f29eec9f353550e11d6b9d1359307b37a5dae053ca7b7781f12af12c0
  Registered: yes (at block timestamp 1775306055)
  Issuer (on-chain): did:key:z6Mkif2g9x66Sry7YEM7uyG1i2GpD9SQLSK5TJcMRnAKSf5b
  Revoked: yes (at block timestamp 1775306061)

  Result: REVOKED

═══════════════════════════════════════════════════════════
  Scenario 2: Certification Credential
═══════════════════════════════════════════════════════════

→ Issuing certification credential (expires 2027-01-15)...
Issuer: did:key:z6MkvcmYDW4tAufms4XunUEhzQL9aZGq75esGTdhcS4NtftX
Credential: Certification: AWS Solutions Architect - Professional from Amazon Web Services
Hash: 0xecaecbcaeb454e5d63aa801e71ba2367f41594a32d6eca04fc33247444682c60
Saved to: credentials/certification_ecaecbca.jwt
→ Registering on-chain...
Credential: Certification: AWS Solutions Architect - Professional from Amazon Web Services
Hash: 0xecaecbcaeb454e5d63aa801e71ba2367f41594a32d6eca04fc33247444682c60
Issuer DID: did:key:z6MkvcmYDW4tAufms4XunUEhzQL9aZGq75esGTdhcS4NtftX
Expiration: 1799971200
Registering on evm...
Registered. Gas used: 140290
→ Verifying (should be VALID -- not yet expired)...
Checking signature...
  Signature: VALID
  Credential: Certification: AWS Solutions Architect - Professional from Amazon Web Services
  Issuer: did:key:z6MkvcmYDW4tAufms4XunUEhzQL9aZGq75esGTdhcS4NtftX
  Holder: did:key:z6MkesEmD6CYnNLAdLEiWC1JE2AckndT95L7StJmpzVngvyf

Checking on-chain status (evm)...
  Hash: 0xecaecbcaeb454e5d63aa801e71ba2367f41594a32d6eca04fc33247444682c60
```

```
  Registered: yes (at block timestamp 1775306070)
  Issuer (on-chain): did:key:z6MkvcmYDW4tAufms4XunUEhzQL9aZGq75esGTdhcS4NtftX
  Expiration: 1799971200

  Result: VALID

  Note: Expiration is checked against the blockchain's block timestamp.
  To demonstrate expiration, the contract would need to be tested with
  a credential whose validUntil is in the past, or by advancing the
  Hardhat node's time. See the research paper for discussion.

============================================================
  Scenario 3: Peer Endorsements
============================================================

→ Issuing endorsement from endorser-bob...
Issuer: did:key:z6MkpJVPEiX6DroR8HShMbHWSLk9xfH68hfuhCor4cmKfqEu
Credential: Endorsement: Bob Martinez endorses [distributed systems, technical leadership]
Hash: 0xfd0b43b30f8bb20f685f565dce6fe9ce43f01d4bc09e81ca2e13488d67526608
Saved to: credentials/endorsement_fd0b43b3.jwt
→ Issuing endorsement from endorser-carol...
Issuer: did:key:z6MknQujbBWJxiRSRiudWLy9NzVkWQtSnH2mSB55ZnFMNeog
Credential: Endorsement: Carol Chen endorses [system design, cross-team collaboration]
Hash: 0xd60dfd8a69579f6d476827382c29d0e2a4ad434587a1f11b989f1d8a2c160dcb
Saved to: credentials/endorsement_d60dfd8a.jwt
→ Issuing endorsement from endorser-dave...
Issuer: did:key:z6MkfR82B2UmWSF1vcYokcLwtYUZ7tNVqZTPwH26gWM1m25F
Credential: Endorsement: Dave Wilson endorses [API design, documentation]
Hash: 0x10f7a6ba8443a6fa5ae031970b614e2267c46b01d0986d239f2c0ed09a6c52f0
Saved to: credentials/endorsement_10f7a6ba.jwt
→ Registering all endorsements on-chain...
Credential: Endorsement: Dave Wilson endorses [API design, documentation]
Hash: 0x10f7a6ba8443a6fa5ae031970b614e2267c46b01d0986d239f2c0ed09a6c52f0
Issuer DID: did:key:z6MkfR82B2UmWSF1vcYokcLwtYUZ7tNVqZTPwH26gWM1m25F
Expiration: none
Registering on evm...
Registered. Gas used: 140242
Credential: Endorsement: Carol Chen endorses [system design, cross-team collaboration]
Hash: 0xd60dfd8a69579f6d476827382c29d0e2a4ad434587a1f11b989f1d8a2c160dcb
Issuer DID: did:key:z6MknQujbBWJxiRSRiudWLy9NzVkWQtSnH2mSB55ZnFMNeog
Expiration: none
Registering on evm...
Registered. Gas used: 140242
Credential: Endorsement: Bob Martinez endorses [distributed systems, technical leadership]
Hash: 0xfd0b43b30f8bb20f685f565dce6fe9ce43f01d4bc09e81ca2e13488d67526608
Issuer DID: did:key:z6MkpJVPEiX6DroR8HShMbHWSLk9xfH68hfuhCor4cmKfqEu
Expiration: none
Registering on evm...
Registered. Gas used: 140242
→ Verifying all endorsements...

Checking signature...
  Signature: VALID
  Credential: Endorsement: Dave Wilson endorses [API design, documentation]
  Issuer: did:key:z6MkfR82B2UmWSF1vcYokcLwtYUZ7tNVqZTPwH26gWM1m25F
  Holder: did:key:z6MkesEmD6CYnNLAdLEiWC1JE2AckndT95L7StJmpzVngvyf

Checking on-chain status (evm)...
  Hash: 0x10f7a6ba8443a6fa5ae031970b614e2267c46b01d0986d239f2c0ed09a6c52f0
  Registered: yes (at block timestamp 1775306079)
```

```
  Issuer (on-chain): did:key:z6MkfR82B2UmWSF1vcYokcLwtYUZ7tNVqZTPwH26gWM1m25F

  Result: VALID

Checking signature...
  Signature: VALID
  Credential: Endorsement: Carol Chen endorses [system design, cross-team collaboration]
  Issuer: did:key:z6MknQujbBWJxiRSRiudWLy9NzVkWQtSnH2mSB55ZnFMNeog
  Holder: did:key:z6MkesEmD6CYnNLAdLEiWC1JE2AckndT95L7StJmpzVngvyf

Checking on-chain status (evm)...
  Hash: 0xd60dfd8a69579f6d476827382c29d0e2a4ad434587a1f11b989f1d8a2c160dcb
  Registered: yes (at block timestamp 1775306082)
  Issuer (on-chain): did:key:z6MknQujbBWJxiRSRiudWLy9NzVkWQtSnH2mSB55ZnFMNeog

  Result: VALID

Checking signature...
  Signature: VALID
  Credential: Endorsement: Bob Martinez endorses [distributed systems, technical leadership]
  Issuer: did:key:z6MkpJVPEiX6DroR8HShMbHWSLk9xfH68hfuhCor4cmKfqEu
  Holder: did:key:z6MkesEmD6CYnNLAdLEiWC1JE2AckndT95L7StJmpzVngvyf

Checking on-chain status (evm)...
  Hash: 0xfd0b43b30f8bb20f685f565dce6fe9ce43f01d4bc09e81ca2e13488d67526608
  Registered: yes (at block timestamp 1775306085)
  Issuer (on-chain): did:key:z6MkpJVPEiX6DroR8HShMbHWSLk9xfH68hfuhCor4cmKfqEu

  Result: VALID

═══════════════════════════════════════════════════════════
  Summary
═══════════════════════════════════════════════════════════

All scenarios completed on evm.

Credentials issued: 5
Keys generated: 6

Credential files:
  certification_ecaecbca.jwt
    Certification: AWS Solutions Architect - Professional from Amazon Web Services
    Hash: 0xecaecbcaeb454e5d63aa801e71ba2367f41594a32d6eca04fc33247444682c60

  employment_91431f5f.jwt
    Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
    Hash: 0x91431f5f29eec9f353550e11d6b9d1359307b37a5dae053ca7b7781f12af12c0

  endorsement_10f7a6ba.jwt
    Endorsement: Dave Wilson endorses [API design, documentation]
    Hash: 0x10f7a6ba8443a6fa5ae031970b614e2267c46b01d0986d239f2c0ed09a6c52f0

  endorsement_d60dfd8a.jwt
    Endorsement: Carol Chen endorses [system design, cross-team collaboration]
    Hash: 0xd60dfd8a69579f6d476827382c29d0e2a4ad434587a1f11b989f1d8a2c160dcb

  endorsement_fd0b43b3.jwt
    Endorsement: Bob Martinez endorses [distributed systems, technical leadership]
    Hash: 0xfd0b43b30f8bb20f685f565dce6fe9ce43f01d4bc09e81ca2e13488d67526608
```

B.2: IOTA Scenarios

```
═══════════════════════════════════════════════════════════
  Step 0: Compiling Move Package
═══════════════════════════════════════════════════════════

→ Building credential_registry Move package via starter pack iota-tools...
FETCHING GIT DEPENDENCY https://github.com/iotaledger/iota.git
INCLUDING DEPENDENCY Iota
INCLUDING DEPENDENCY MoveStdlib
BUILDING credential_registry
  Build output:
total 16
drwxr-xr-x 3 root root 4096 Apr  4 14:36 .
drwxr-xr-x 5 root root 4096 Apr  4 14:36 ..
-rw-r--r-- 1 root root 1637 Apr  4 14:36 credential_registry.mv
drwxr-xr-x 4 root root 4096 Apr  4 14:36 dependencies

═══════════════════════════════════════════════════════════
  Setup: Generating Keys
═══════════════════════════════════════════════════════════

→ Cleaning up previous run artifacts...
→ Generating issuer key (employer)...
Generated keypair: employer
  DID:  did:key:z6MkehBgptTgBhVtoLZ6AwXnHQNvLmtVsbt9kNHXpgetFHR4
  Saved to: /app/keys/employer.json
→ Generating issuer key (certification body)...
Generated keypair: certbody
  DID:  did:key:z6MkhwjomwvQPV7n58rS6S8Db7igeAv2igzMAm9hgk5sBezL
  Saved to: /app/keys/certbody.json
→ Generating endorser keys (3 peers)...
Generated keypair: endorser-bob
  DID:  did:key:z6MkpkkEdYhPVxLgzgzdtn4DT253tGaLuroeQw3bFSVZjXLE
  Saved to: /app/keys/endorser-bob.json
Generated keypair: endorser-carol
  DID:  did:key:z6Mkq2KdJ8hArAPzJNDN9JhScevGTRo5JnC8hom9HodTYGQT
  Saved to: /app/keys/endorser-carol.json
Generated keypair: endorser-dave
  DID:  did:key:z6Mkm2PsG8Xx5dkAdYT95gDV5jbqEjjpKin6RupByuPrq1kk
  Saved to: /app/keys/endorser-dave.json
→ Generating holder key (Alice)...
Generated keypair: alice
  DID:  did:key:z6Mkjp7Ac8e79nwQBuv7Y3Zb9ji45BEzdgCFsS8kk9vkWyvz
  Saved to: /app/keys/alice.json
  Alice's DID: did:key:z6Mkjp7Ac8e79nwQBuv7Y3Zb9ji45BEzdgCFsS8kk9vkWyvz

═══════════════════════════════════════════════════════════
  Setup: Deploying Contract
═══════════════════════════════════════════════════════════

→ Deploying CredentialRegistry on iota...
Deploying CredentialRegistry on iota...
Contract deployed at: 0x8adbab65dd003ea330c914737eddcd96be1bebf3160ce035da5a051d1af10a77:0xfd95e216740f7ca0

Add to your .env file:
  IOTA_CONTRACT_ADDRESS=0x8adbab65dd003ea330c914737eddcd96be1bebf3160ce035da5a051d1af10a77:0xfd95e216740f7c
→ Contract address: 0x8adbab65dd003ea330c914737eddcd96be1bebf3160ce035da5a051d1af10a77:0xfd95e216740f7ca094
```

```
============================================================
  Scenario 1: Employment Credential
============================================================

→ Issuing employment credential...
Issuer: did:key:z6MkehBgptTgBhVtoLZ6AwXnHQNvLmtVsbt9kNHXpgetFHR4
Credential: Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
Hash: 0x39c5c3597aa078552084402fefab8548da329e3776490d48e46b721b1dedb2d5
Saved to: credentials/employment_39c5c359.jwt
→ Registering on-chain...
Credential: Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
Hash: 0x39c5c3597aa078552084402fefab8548da329e3776490d48e46b721b1dedb2d5
Issuer DID: did:key:z6MkehBgptTgBhVtoLZ6AwXnHQNvLmtVsbt9kNHXpgetFHR4
Expiration: none
Registering on iota...
Registered. Cost (NANOS): 4974800
→ Verifying (should be VALID)...
Checking signature...
  Signature: VALID
  Credential: Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
  Issuer: did:key:z6MkehBgptTgBhVtoLZ6AwXnHQNvLmtVsbt9kNHXpgetFHR4
  Holder: did:key:z6Mkjp7Ac8e79nwQBuv7Y3Zb9ji45BEzdgCFsS8kk9vkWyvz

Checking on-chain status (iota)...
  Hash: 0x39c5c3597aa078552084402fefab8548da329e3776490d48e46b721b1dedb2d5
  Registered: yes (at block timestamp 1775306187)
  Issuer (on-chain): did:key:z6MkehBgptTgBhVtoLZ6AwXnHQNvLmtVsbt9kNHXpgetFHR4

  Result: VALID
→ Revoking credential (employer terminates attestation)...
Revoking credential 0x39c5c3597aa078552084402fefab8548da329e3776490d48e46b721b1dedb2d5 on iota...
Revoked. Cost (NANOS): 1000000
→ Verifying again (should be REVOKED)...
Checking signature...
  Signature: VALID
  Credential: Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
  Issuer: did:key:z6MkehBgptTgBhVtoLZ6AwXnHQNvLmtVsbt9kNHXpgetFHR4
  Holder: did:key:z6Mkjp7Ac8e79nwQBuv7Y3Zb9ji45BEzdgCFsS8kk9vkWyvz

Checking on-chain status (iota)...
  Hash: 0x39c5c3597aa078552084402fefab8548da329e3776490d48e46b721b1dedb2d5
  Registered: yes (at block timestamp 1775306187)
  Issuer (on-chain): did:key:z6MkehBgptTgBhVtoLZ6AwXnHQNvLmtVsbt9kNHXpgetFHR4
  Revoked: yes (at block timestamp 1775306190)

  Result: REVOKED

============================================================
  Scenario 2: Certification Credential
============================================================

→ Issuing certification credential (expires 2027-01-15)...
Issuer: did:key:z6MkhwjomwvQPV7n58rS6S8Db7igeAv2igzMAm9hgk5sBezL
Credential: Certification: AWS Solutions Architect - Professional from Amazon Web Services
Hash: 0x6b410c015d05277365c2b78d66fdd3e80a282fa97e9900e16843d6f88f36f873
Saved to: credentials/certification_6b410c01.jwt
→ Registering on-chain...
Credential: Certification: AWS Solutions Architect - Professional from Amazon Web Services
Hash: 0x6b410c015d05277365c2b78d66fdd3e80a282fa97e9900e16843d6f88f36f873
Issuer DID: did:key:z6MkhwjomwvQPV7n58rS6S8Db7igeAv2igzMAm9hgk5sBezL
```

```
Expiration: 1799971200
Registering on iota...
Registered. Cost (NANOS): 3994400
→ Verifying (should be VALID -- not yet expired)...
Checking signature...
  Signature: VALID
  Credential: Certification: AWS Solutions Architect - Professional from Amazon Web Services
  Issuer: did:key:z6MkhwjomwvQPV7n58rS6S8Db7igeAv2igzMAm9hgk5sBezL
  Holder: did:key:z6Mkjp7Ac8e79nwQBuv7Y3Zb9ji45BEzdgCFsS8kk9vkWyvz

Checking on-chain status (iota)...
  Hash: 0x6b410c015d05277365c2b78d66fdd3e80a282fa97e9900e16843d6f88f36f873
  Registered: yes (at block timestamp 1775306194)
  Issuer (on-chain): did:key:z6MkhwjomwvQPV7n58rS6S8Db7igeAv2igzMAm9hgk5sBezL
  Expiration: 1799971200

  Result: VALID

  Note: Expiration is checked against the IOTA Clock timestamp.
  Unlike EVM, IOTA uses millisecond-precision timestamps from the
  network Clock object. See the research paper for discussion.

═══════════════════════════════════════════════════════════
  Scenario 3: Peer Endorsements
═══════════════════════════════════════════════════════════

→ Issuing endorsement from endorser-bob...
Issuer: did:key:z6MkpkkEdYhPVxLgzgzdtn4DT253tGaLuroeQw3bFSVZjXLE
Credential: Endorsement: Bob Martinez endorses [distributed systems, technical leadership]
Hash: 0x798312104038f2b06b33801af12efc93157f9d4902b1b5f0e9d506366636b5bb
Saved to: credentials/endorsement_79831210.jwt
→ Issuing endorsement from endorser-carol...
Issuer: did:key:z6Mkq2KdJ8hArAPzJNDN9JhScevGTRo5JnC8hom9HodTYGQT
Credential: Endorsement: Carol Chen endorses [system design, cross-team collaboration]
Hash: 0x61bcc86ad29ea66ff2b74637a87bf47f3e685064f7671d3eaf21fe30639372af
Saved to: credentials/endorsement_61bcc86a.jwt
→ Issuing endorsement from endorser-dave...
Issuer: did:key:z6Mkm2PsG8Xx5dkAdYT95gDV5jbqEjjpKin6RupByuPrq1kk
Credential: Endorsement: Dave Wilson endorses [API design, documentation]
Hash: 0x23d1a202b529b9371b9a6333c97c1324fd10ab73a53a27c2b2126423b5b6f852
Saved to: credentials/endorsement_23d1a202.jwt
→ Registering all endorsements on-chain...
Credential: Endorsement: Dave Wilson endorses [API design, documentation]
Hash: 0x23d1a202b529b9371b9a6333c97c1324fd10ab73a53a27c2b2126423b5b6f852
Issuer DID: did:key:z6Mkm2PsG8Xx5dkAdYT95gDV5jbqEjjpKin6RupByuPrq1kk
Expiration: none
Registering on iota...
Registered. Cost (NANOS): 3994400
Credential: Endorsement: Carol Chen endorses [system design, cross-team collaboration]
Hash: 0x61bcc86ad29ea66ff2b74637a87bf47f3e685064f7671d3eaf21fe30639372af
Issuer DID: did:key:z6Mkq2KdJ8hArAPzJNDN9JhScevGTRo5JnC8hom9HodTYGQT
Expiration: none
Registering on iota...
Registered. Cost (NANOS): 3994400
Credential: Endorsement: Bob Martinez endorses [distributed systems, technical leadership]
Hash: 0x798312104038f2b06b33801af12efc93157f9d4902b1b5f0e9d506366636b5bb
Issuer DID: did:key:z6MkpkkEdYhPVxLgzgzdtn4DT253tGaLuroeQw3bFSVZjXLE
Expiration: none
Registering on iota...
Registered. Cost (NANOS): 3994400
```

```
→ Verifying all endorsements...

Checking signature...
  Signature: VALID
  Credential: Endorsement: Dave Wilson endorses [API design, documentation]
  Issuer: did:key:z6Mkm2PsG8Xx5dkAdYT95gDV5jbqEjjpKin6RupByuPrq1kk
  Holder: did:key:z6Mkjp7Ac8e79nwQBuv7Y3Zb9ji45BEzdgCFsS8kk9vkWyvz

Checking on-chain status (iota)...
  Hash: 0x23d1a202b529b9371b9a6333c97c1324fd10ab73a53a27c2b2126423b5b6f852
  Registered: yes (at block timestamp 1775306201)
  Issuer (on-chain): did:key:z6Mkm2PsG8Xx5dkAdYT95gDV5jbqEjjpKin6RupByuPrq1kk

  Result: VALID

Checking signature...
  Signature: VALID
  Credential: Endorsement: Carol Chen endorses [system design, cross-team collaboration]
  Issuer: did:key:z6Mkq2KdJ8hArAPzJNDN9JhScevGTRo5JnC8hom9HodTYGQT
  Holder: did:key:z6Mkjp7Ac8e79nwQBuv7Y3Zb9ji45BEzdgCFsS8kk9vkWyvz

Checking on-chain status (iota)...
  Hash: 0x61bcc86ad29ea66ff2b74637a87bf47f3e685064f7671d3eaf21fe30639372af
  Registered: yes (at block timestamp 1775306203)
  Issuer (on-chain): did:key:z6Mkq2KdJ8hArAPzJNDN9JhScevGTRo5JnC8hom9HodTYGQT

  Result: VALID

Checking signature...
  Signature: VALID
  Credential: Endorsement: Bob Martinez endorses [distributed systems, technical leadership]
  Issuer: did:key:z6MkpkkEdYhPVxLgzgzdtn4DT253tGaLuroeQw3bFSVZjXLE
  Holder: did:key:z6Mkjp7Ac8e79nwQBuv7Y3Zb9ji45BEzdgCFsS8kk9vkWyvz

Checking on-chain status (iota)...
  Hash: 0x798312104038f2b06b33801af12efc93157f9d4902b1b5f0e9d506366636b5bb
  Registered: yes (at block timestamp 1775306205)
  Issuer (on-chain): did:key:z6MkpkkEdYhPVxLgzgzdtn4DT253tGaLuroeQw3bFSVZjXLE

  Result: VALID

═══════════════════════════════════════════════════════════
  Summary
═══════════════════════════════════════════════════════════

All scenarios completed on iota.

Credentials issued: 5
Keys generated: 6

Credential files:
  certification_6b410c01.jwt
    Certification: AWS Solutions Architect - Professional from Amazon Web Services
    Hash: 0x6b410c015d05277365c2b78d66fdd3e80a282fa97e9900e16843d6f88f36f873

  employment_39c5c359.jwt
    Employment: Senior Engineer at Acme Corp (2022-03-01 to 2024-06-30)
    Hash: 0x39c5c3597aa078552084402fefab8548da329e3776490d48e46b721b1dedb2d5

  endorsement_23d1a202.jwt
```

```
    Endorsement: Dave Wilson endorses [API design, documentation]
    Hash: 0x23d1a202b529b9371b9a6333c97c1324fd10ab73a53a27c2b2126423b5b6f852

  endorsement_61bcc86a.jwt
    Endorsement: Carol Chen endorses [system design, cross-team collaboration]
    Hash: 0x61bcc86ad29ea66ff2b74637a87bf47f3e685064f7671d3eaf21fe30639372af

  endorsement_79831210.jwt
    Endorsement: Bob Martinez endorses [distributed systems, technical leadership]
    Hash: 0x798312104038f2b06b33801af12efc93157f9d4902b1b5f0e9d506366636b5bb
```

This paper accompanies the Decentralized Professional Credentials proof of concept, developed by Consensix Labs. The code is provided for research and educational purposes. It has not been audited for production use.