MedChain Avitabile: Deliverable

Enrico Pezzano

October 2025

Abstract

Provide a concise summary of the project, its goals, and key outcomes. Highlight the most important implementation achievements and documentation artifacts.

Contents

1 Introduction

MedChain investigates how redactable blockchains can satisfy the GDPR Right to Erasure without abandoning the auditability healthcare regulators require. The project combines the chameleon hash redaction scheme by Ateniese et al. with the governance extensions proposed by Avitabile et al., delivering a permissioned ledger that allows controlled history updates while keeping cryptographic proofs verifiable. The scope covered by this deliverable spans the Python-based simulator, Solidity smart contracts, zero-knowledge tooling, and documentation needed to demonstrate the end-to-end workflow for privacy-preserving medical record management.

The current iteration emphasises a research-grade prototype that privileges correctness, traceability, and reproducibility over raw performance. Key accomplishments include: (i) a modular blockchain core that supports redactable blocks and role-based governance policies, (ii) an operational Groth16 proof pipeline built on circom/snarkjs and integrated with the medical redaction engine, (iii) encrypted IPFS storage with deterministic links between on-chain commitments and off-chain payloads, and (iv) a growing test suite and documentation set that anchor the implementation to compliance requirements. Remaining gaps—notably contract-orchestrated execution of redactions and production-grade consistency proofs—are tracked in the roadmap and inform the next development cycle.

2 Documentation Overview

The repository collects implementation notes, compliance guidance, and operating procedures alongside the code. Documentation lives close to the artefacts it describes to encourage short feedback loops: project-wide briefs reside under `docs/`, developer onboarding material is surfaced in the root `README.md`, and deliverable-specific sections are assembled through this LaTeX template. The `todo.md` backlog is treated as a living document that captures directives from supervisors and tracks progress on required enhancements.

2.1 Architecture Documentation

High-level design rationale and cryptographic integration notes are maintained in `docs/ZK_PROOF_IMPLEMENTAT` and `docs/CONSISTENCY_CIRCUIT_INTEGRATION_PLAN.md`. These documents describe how the redactable blockchain core, smart contracts, and proof systems fit together, and they are updated whenever major milestones land (e.g., the shift from simulated to real Groth16 proofs). The top-level `README.md` complements them with an overview of core features, command entry points, and the relationship between on-chain commitments and off-chain IPFS artefacts. Diagramming remains a TODO; the team keeps placeholders for future architecture figures as the contract orchestration layer stabilises.

2.2 Developer Documentation

Developer-facing resources emphasise reproducibility. The `README.md` provides bootstrap commands, environment variables, and demo invocations. Module-specific docstrings (for example in `medical/MedicalRedactionEngine.py` and `adapters/snark.py`) document extension points, while the adapters include inline comments that justify non-obvious design decisions such as IPFS retry strategies. Configuration helpers under `adapters/config.py` and the generated badges in `badges/` surface build and coverage status for new contributors. Outstanding documentation work includes formal coding standards and an explicit contribution guide.

2.3 User-Facing Documentation

User-oriented material focuses on demonstrators rather than polished manuals. The demo suite under `demo/` showcases standard redaction flows, IPFS integration, and zero-knowledge proof generation. The medical use-case notebooks and scripts document how synthetic datasets are generated and censored before being published via IPFS. A concise operator guide is planned once smart contract orchestration reaches feature parity with the simulator; in the meantime, the deliverable, demo walkthroughs, and inline CLI help serve as the main references for stakeholders evaluating the prototype.

3 Implementation Details

The codebase is organised to keep privacy-critical functionality isolated yet composable. Python orchestrates simulation, proof generation, and integration logic, while Solidity contracts and circom circuits implement the on-chain and zero-knowledge layers respectively. This separation allows rapid prototyping in the simulator without losing sight of deployment targets.

3.1 System Architecture

At the core, the `Models/` package extends the Ateniese redactable blockchain benchmark with explicit smart contract abstractions and role-aware governance policies. The `medical/MedicalRedactionEngin` module coordinates redaction requests, approval tracking, zero-knowledge proof generation, and proof-of-consistency checks produced by `ZK/ProofOfConsistency.py`. Integration layers under `adapters/` connect the simulator to external infrastructure: `adapters/snark.py` wraps the snarkjs CLI, `adapters/ipfs.py` manages encrypted storage and pinning, and `adapters/evm.py` (paired with `medical/backends.py`) exposes a Web3 client for the deployed Solidity contracts in `contracts/src/`. Circom circuits inside `circuits/` define the Groth16 redaction verifier, while generated artefacts are consumed both off-chain (Python) and on-chain (Solidity verifier contracts).

3.2 Data Flows

Medical records enter the system through the redaction engine, which serialises the payload, stores an encrypted copy in IPFS via the adapter layer, and anchors a commitment plus policy metadata to the blockchain model. Redaction requests trigger policy evaluation, multi-role approvals, and the creation of Groth16 proofs using `medical/circuit_mapper.py` to derive deterministic circuit inputs. Successful proofs and consistency checks are attached to the request, after which the chameleon hash trapdoor rewrites the affected block without breaking hash links. When operating against the Solidity deployment, the same flow persists, with the `MedicalDataManager.sol` contract emitting events that downstream services consume to update off-chain storage. Throughout the pipeline, personal data is encrypted-at-rest and provenance is maintained via hashes and event logs.

3.3 Technology Stack

- **Python 3.11**: primary language for the simulator, orchestration, and CLI demos.
- **Circom & snarkjs**: compile and evaluate Groth16 circuits, producing verifier calldata for Solidity.
- **Solidity + Hardhat**: implement smart contracts (`MedicalDataManager`, `RedactionVerifier`) and manage deployments, testing, and coverage.

- **Web3.py**: connect Python workflows to the deployed EVM contracts when `USE_REAL_EVM` is enabled.
- **IPFS (Kubo) + ipfshttpclient**: provide distributed, content-addressed storage with AES-GCM encryption of medical payloads prior to upload.
- **Cryptography & tooling**: AES-GCM key management, dotenv-based configuration, and pytest-driven verification.

3.4 Zero-Knowledge Proof Generation

The implementation delivers real Groth16 SNARK proofs integrated into redaction workflows, replacing prior simulations.

3.4.1 Circuit Input Mapping

The `MedicalDataCircuitMapper` class (in `medical/circuit_mapper.py`) bridges medical records and cryptographic circuit inputs. Medical data is serialized to canonical JSON format with deterministic field ordering:

$$\text{canonical}(R) = \text{JSON}(\{\text{“patient\_id”} : p_i d, \text{“diagnosis”} : d, \text{“treatment”} : t, \text{“physician”} : ph\}, \text{sorted\_keys}) \tag{1}$$

Data is converted to field elements compatible with BN254:

$$e_i = \Big(\text{SHA256}(data)[i \cdot n : (i+1) \cdot n] \bmod 2^{250}\Big) \tag{2}$$

Circuit inputs separate into public (verified on-chain) and private (prover secret) components:

Public inputs:

- Policy hashes (128-bit limbs): $(H_{\text{policy},0}, H_{\text{policy},1})$
- Merkle root: $(H_{\text{merkle},0}, H_{\text{merkle},1})$
- Original/redacted hashes: $(H_{\text{orig},0}, H_{\text{orig},1}, H_{\text{redact},0}, H_{\text{redact},1})$
- Authorization flag: $\text{policyAllowed} \in \{0, 1\}$

Private inputs:

- Data field elements: $(d_0^{\text{orig}}, d_1^{\text{orig}}, d_2^{\text{orig}}, d_3^{\text{orig}})$ and $(d_0^{\text{redact}}, \ldots)$
- Policy field elements: $(d_0^{\text{policy}}, d_1^{\text{policy}})$
- Merkle path elements and indices (optional for tree inclusion proofs)

3.4.2 Real Proof Generation

The `EnhancedHybridSNARKManager` (in `medical/my_snark_manager.py`) orchestrates real Groth16 generation via snarkjs CLI through the `adapters/snark.py` wrapper:

1. Extract medical record from redaction request
2. Prepare circuit inputs using `MedicalDataCircuitMapper`
3. Validate inputs conform to circuit specification

4. Call snarkjs to generate witness and proof
5. Verify proof off-chain before submission
6. Extract Groth16 components: (a, b, c) and public signals

Each proof is represented as:

$$\Pi_{\text{redaction}} = (\pi_a, \pi_b, \pi_c, \{\sigma_i\}_{i=0}^{8}) \tag{3}$$

where π_a, π_b, π_c are BN254 elliptic curve points and $\{\sigma_i\}$ are public signals including:

- Commitment to redacted data
- Nullifier for replay prevention
- Merkle root claim
- Policy and authorization flags

3.4.3 Consistency Proof Integration

The `ConsistencyProofGenerator` (in `ZK/ProofOfConsistency.py`) verifies that redaction operations maintain blockchain integrity across five check types: block integrity, hash chain consistency, Merkle tree validity, smart contract state transitions, and transaction ordering.

When a consistency proof is provided, it integrates into circuit public inputs via `prepare_circuit_inputs_w`

$$\text{pubInputs}_{\text{consistency}} = \text{pubInputs}_{\text{base}} \cup \{H_{\text{pre},0}, H_{\text{pre},1}, H_{\text{post},0}, H_{\text{post},1}, \text{consistencyValid}\} \tag{4}$$

Pre and post-state hashes are computed as:

$$H_{\text{state}} = \text{SHA256}(\text{canonical}(\mathcal{S})) \tag{5}$$

The circuit then verifies the redaction is cryptographically sound and consistency checks pass before generating a valid proof.

3.4.4 Implementation Status

All Phase 1 implementation files are marked with comment `### Bookmark1 for next meeting`:

- `medical/circuit_mapper.py`, `medical/my_snark_manager.py`, `medical/MedicalRedactionEngine.py`
- `ZK/SNARKs.py`, `ZK/ProofOfConsistency.py`, `adapters/snark.py`
- `tests/test_circuit_mapper.py`, `tests/test_snark_system.py`, `tests/test_consistency_system.py`, `tests/test_consistency_circuit_integration.py`

Test coverage includes 20+ unit tests for circuit mapping, 5+ SNARK system tests, 8+ consistency proof tests, and 5+ integration tests. All tests pass without blocking issues.

3.5 On-Chain Verification

Phase 2 extends Phase 1 with on-chain verification via smart contracts, enabling trustless, cryptographic validation of redaction operations on the blockchain.

3.5.1 Nullifier Registry

The `NullifierRegistry` contract maintains a mapping of used nullifiers to prevent replay attacks—the re-submission of an identical proof for unintended duplication:

$$\text{usedNullifiers} : \mathbb{B}_{32} \to \{0, 1\} \tag{6}$$

Each SNARK proof produces a unique nullifier via:

$$n = \text{hash}(\text{public\_signals} \| \text{timestamp} \| \text{prover\_address}) \tag{7}$$

When a redaction request is processed, the contract:

1. Extracts nullifier n from SNARK public signals
2. Queries registry: $\text{isNullifierUsed}(n)$
3. Reverts if true (already submitted)
4. Registers nullifier on success for audit trail

3.5.2 Groth16 Verifier Integration

The `RedactionVerifier_groth16` contract is auto-generated from snarkjs and implements:

$$\text{verifyProof}(\pi_a, \pi_b, \pi_c, \{\sigma_i\}) \to \{0, 1\} \tag{8}$$

The `MedicalDataManager` contract integrates this verifier via:

```
function requestDataRedactionWithProof(
    string memory patientId,
    string memory redactionType,
    string memory reason,
    uint[2] memory a,
    uint[2][2] memory b,
    uint[2] memory c,
    uint[9] memory publicSignals
) public onlyAuthorized returns (string memory requestId)
```

This function:

1. Calls verifier: $\text{valid} = \text{verifyProof}(a, b, c, \text{publicSignals})$
2. Extracts nullifier from signals
3. Checks nullifier registry for replay
4. Validates consistency proofs (if included)
5. Creates redaction request
6. Registers nullifier and emits audit event

Gas cost breakdown: Groth16 verification ($\sim 250k$ gas) + nullifier operations ($\sim 20k$ gas) + state updates ($\sim 50k$ gas) = $\sim 320k$ gas total (approximately \$20 at 100 gwei).

3.5.3 Python Backend

The `EVMBackend` class (in `medical/backends.py`) extends redaction workflows with on-chain verification:

```
def request_data_redaction_with_proof(
    patient_id: str,
    redaction_type: str,
    reason: str,
    medical_record_dict: Dict[str, Any]
) -> Optional[str]
```

This method:

1. Prepares circuit inputs via `MedicalDataCircuitMapper`
2. Generates SNARK proof via `SnarkClient`
3. Extracts proof components: (a, b, c), public signals
4. Calls `MedicalDataManager.requestDataRedactionWithProof()` on-chain
5. Returns request ID or None on failure

3.5.4 Circuit Extensions

The redaction circuit is extended with consistency proof inputs, adding to public inputs:

$$\{\text{preStateHash0}, \text{preStateHash1}, \text{postStateHash0}, \text{postStateHash1}, \text{consistencyCheckPassed}\} \tag{9}$$

The circuit verifies:

1. Original data hash matches circuit input
2. Redacted data hash is correct for operation type
3. Policy hash is authorized
4. If consistency enabled: pre/post-state hashes match provided proof

3.5.5 Testing and Deployment

Integration tests validate:

- SNARK proof generation with real artifacts
- On-chain submission and verification
- Nullifier replay prevention
- Consistency proof integration
- End-to-end workflow from request to verified redaction

Contract deployment steps:

1. Deploy `NullifierRegistry`

2. Deploy `RedactionVerifier_groth16`
3. Deploy updated `MedicalDataManager` with verifier and registry addresses
4. Configure environment variables with contract addresses
5. Run integration test suite

4 Results and Evaluation

The prototype has been exercised through automated tests, Hardhat simulations, and interactive demos. Validation emphasises deterministic proof generation, correctness of redaction policies, and the alignment between on-chain state, off-chain storage, and compliance expectations.

4.1 Validation Scenarios

- **Circuit and proof validation**: `pytest` targets such as `tests/test_circuit_mapper.py` ensure the medical circuit mapper produces valid public/private inputs for Groth16 proofs, while `tests/test_avitabile_redaction_demo.py` exercises the full redactable blockchain flow with approvals, trapdoor updates, and consistency checks.
- **Smart contract testing**: Hardhat tests under `contracts/test/` validate storage, approval thresholds, and verifier integration for `MedicalDataManager.sol`. Solidity coverage reports are exported to `contracts/coverage/` and surfaced through the repository badges.
- **Demo walkthroughs**: CLI demos in `demo/medchain_demo.py` and `demo/medical_redaction_demo.py` are used to rehearse GDPR Right-to-Erasure requests, highlighting the interaction between simulated consensus, SNARK proofs, and IPFS storage updates.

4.2 Metrics and KPIs

- **Build health**: GitHub Actions workflows (`tests.yml` and `contracts.yml`) report passing status at the time of writing, with coverage badges generated into `badges/python-coverage.svg` and `badges/solidity-coverage.svg`.
- **Proof integrity**: Real Groth16 proofs are generated via `SnarkClient.prove_redaction` and verified locally before redactions are executed; failures revert to prevent inconsistent ledger states.
- **Governance enforcement**: Policy thresholds configured in `MedicalDataContract` are respected in both simulator and contract tests, demonstrating that multi-role approvals gate every destructive operation.

4.3 Lessons Learned

Deploying real zero-knowledge tooling inside a research simulator requires disciplined artefact management: the team standardised on deterministic circuit inputs and explicit validation to avoid silent proof drift. Integrating IPFS taught the importance of encrypting payloads before upload and of treating pinning/unpinning as part of the redaction lifecycle. Finally, aligning simulated governance with on-chain contracts highlighted the need for shared data models and consistent event semantics so that auditors can trace the same operation across components.

5 Project Management

Project planning follows the directives captured in `todo.md`, which serves as a combined roadmap and status ledger. Weekly check-ins convert supervisory feedback into actionable tasks, while Git branches and GitHub Actions provide traceability for merged work.

5.1 Timeline

- **Phase 0 — Baseline Port**: Adapted the Ateniese redactable blockchain benchmark into the current modular simulator, laying the groundwork for injectable smart contracts and redaction policies. (Complete)
- **Phase 1 — Infrastructure Integration**: Added configurable adapters for IPFS, EVM access, and environment management; established encryption-at-rest and content-addressable linkage between on-chain and off-chain artefacts. (Complete)
- **Phase 2 — Zero-Knowledge Enablement**: Delivered the medical circuit mapper, real Groth16 proof pipeline, and proof-of-consistency tooling referenced in the latest documentation updates. (Complete)
- **Phase 3 — Contract Orchestration**: In progress. Outstanding tasks include wiring contract-based execution of redactions, finalising censored dataset publication, and broadening test coverage to negative cases.

5.2 Team Roles

Core development is led by Enrico Pezzano, who coordinates implementation across Python, Solidity, and circom. The Mobile IoT Security Lab reviewers provide design oversight, approve roadmap adjustments, and evaluate compliance alignment. Collaboration occurs via issue tracking in `todo.md`, code reviews on the GitHub repository, and synchronous milestone reviews when introducing new cryptographic components.

5.3 Risk Management

- **Proof Artefact Drift**: High impact. Mitigated by mandating checksum validation for `circuits/build/` artefacts and by failing fast when snarkjs binaries or zkeys are missing.
- **Regulatory Misalignment**: Medium impact. Addressed through explicit policy modelling in `MedicalDataContract` and by mapping requirements back to GDPR/HIPAA directives in backlog items.
- **External Dependency Availability**: Medium impact. Environment flags (`USE_REAL_IPFS`, `USE_REAL_EVM`) allow fallbacks to simulators, ensuring development remains unblocked when IPFS nodes or EVM endpoints are offline.
- **Security of Off-Chain Storage**: Medium impact. Countered by encrypting data before IPFS upload and by treating key rotation plus unpinning as first-class parts of the redaction workflow.

6 Future Work

Remaining tasks span protocol hardening, usability improvements, and compliance sign-off. The backlog in `todo.md` is the authoritative source; highlights are summarised here to guide the next development cycle.

6.1 Short-Term Priorities

- Complete end-to-end smart contract orchestration: invoke Groth16 verification and proof-of-consistency checks from `MedicalDataManager.sol` before allowing state changes.
- Extend the circom circuit and mapper to ingest consistency proof data, enforcing state-transition validity inside the proof system.
- Finalise the censored medical dataset pipeline by automating policy-based anonymisation, IPFS publication, and on-chain/off-chain linkage tests.
- Add negative-path testing for proof verification, policy breaches, and IPFS redaction edge cases to increase confidence ahead of demonstrations.
- Produce architecture diagrams and compliance mapping artefacts that visualise data flow and reference relevant GDPR/HIPAA clauses.

6.2 Long-Term Vision

- Deploy the solution on a managed permissioned blockchain and evaluate operational characteristics such as latency, throughput, and key management at scale.
- Investigate formal verification of smart contracts and circuits to strengthen assurance guarantees demanded by healthcare regulators.
- Introduce a role-aware operator dashboard that surfaces approvals, audit logs, and redaction history to non-technical stakeholders.
- Explore interoperability with existing health information systems (FHIR APIs, consent registries) to streamline data ingestion and audit trails.

A Appendix

This appendix captures configuration references and command snippets used during the current iteration.

A.1 Key Environment Variables

- `USE_REAL_IPFS`, `IPFS_API_ADDR`, `IPFS_GATEWAY_URL`: toggle and configure the real IPFS client in `adapters/ipfs.py`.
- `USE_REAL_EVM`, `WEB3_PROVIDER_URI`, `MEDICAL_CONTRACT_ADDRESS`: enable on-chain execution via `adapters/evm.py` and `medical/backends.py`.
- `CIRCUITS_DIR`: override the default location of circom artefacts consumed by `adapters/snark.py`.
- `TESTING_MODE`, `DRY_RUN`: adjust simulator behaviour for accelerated testing or preview runs.

A.2 Representative Commands

- **Run simulator**: `python Main.py` (set `TESTING_MODE=1` for fast mode).
- **Generate SNARK artefacts**: `cd circuits && ./scripts/compile.sh` followed by `PTAU=../tools/pot` `./scripts/setup.sh`.
- **Execute medical demo**: `python -m demo.medical_redaction_demo`.
- **Run Hardhat suite**: `cd contracts && npm test` (coverage emitted under `contracts/coverage/`).

References

[1] Vincenzo Botta, Vincenzo Iovino, and Ivan Visconti. Towards data redaction in bitcoin. *IEEE Transactions on Network and Service Management*, 19(4):3872–3883, 2022.

[2] Giuseppe Ateniese, Bernardo Magri, Daniele Venturi, and Ewerton R. Andrade. Redactable blockchain – or – rewriting history in bitcoin and friends. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 111–126, 2017.

[3] Gennaro Avitabile, Vincenzo Botta, Daniele Friolo, and Ivan Visconti. Data redaction in smart-contract-enabled permissioned blockchains. In *Proceedings of the 6th Distributed Ledger Technologies Workshop (DLT)*, Turin, Italy, 2024. CEUR-WS.org.