CertRoot - Design Specifications

1 Revision History

Revision	Date	Created By	Description
0.1	October 1st, 2025	Nguyen, Thien Phuc	Initial version

2 Project Overview

2.1 Problem

In engineering and manufacturing, design reviews often end without an immutable record of the final files and feedback. This creates a trust gap, since firms cannot independently prove that an approved design hasn't been altered. The result is exposure to IP risks, disputes, and supply chain liability.

2.2 Solution

CertRoot integrates with CoLab to create a blockchain-based audit log for every closed review. Final design files and comment threads are hashed and sealed on-chain, providing a verifiable record of integrity. This ensures designs can be independently validated at any time, giving firms stronger IP protection, reduced liability, and greater client confidence.

2.3 Development team

- Nguyen, Thien Phuc (Gerard)
- Saha, Priyanka
- Gajjar, Ronit Hirenbhai
- Manyam, Anil
- Alam, Sadia

2.4 Client

- Mr. Freddie Pike, Staff Developer and Technical Onboarding Manager at CoLab Software

2.5 Project Timeline

- Start date: September 30th, 2025
- End date: November 30th, 2025

3 How CertRoot Works

Think of a **hash** like a **fingerprint for a file**.

- If you change even one pixel in a drawing or one word in a comment, the fingerprint comes out completely different.

CertRoot takes the **fingerprint (hash) of your design files and reviews threads** and writes it to a **blockchain**.

- Blockchain is just a special type of database that nobody can secretly change. Once something is written, it stays there forever.

Later, if someone uploads the same file again, CertRoot re-creates the fingerprint and checks it against what's stored on the blockchain.

- If they match → file is authentic, untouched since approval.
- If they don't match → file has been changed.

So in plain terms:

- **Cryptography = making a file's unique fingerprint.**
- **Blockchain = keeping that fingerprint in a tamper-proof ledger.**

Together, they give the 'digital notary' effect.

4 Project Deliverables

The final product is a modular suite of tools designed for seamless integration into CoLab's Python backend and React frontend environments.

4.1 The Core Trust Layer (Solidity Smart Contract)

EVM Smart Contract: The fully developed, tested, and audited Solidity smart contract deployed to the agreed-upon EVM-compatible network.

- **Functions:** Contains only the essential public functions:
  - recordHash (for certification).
  - checkHashExists (for verification).
- **Immutability:** The final compiled ABI (Application Binary Interface) and Contract Address will be delivered for CoLab's integration reference.

4.2 The Python SDK (Backend Engine)

Python Package (certroot-sdk): A fully documented, pip-installable Python package.

- **Secure Certification:**
  - The core function, sdk.certify_file(file_path, metadata), handles local hashing, secure wallet key management, gas estimation, and transaction signing via web3.py.
- **Integrity Check:**
  - The sdk.check_integrity(hash) function performs the simple query for internal system checks.
- **Code Quality:**
  - All Python code will be unit-tested and conform to Python best practices, including robust exception handling for network and transaction failures.
- **Documentation:**
  - Comprehensive README and integration guide with clear examples for CoLab engineers.

4.3 The React Component Library (Frontend Integration)

NPM Package (@certroot/react-ui): A modular npm package containing key components.

- **Certification Component:**
  - A drop-in <CertifyButton>component that triggers the necessary API calls to the Python backend to initiate the certification process.
- **Verification Component:**
  - A component like <IntegrityBadge>that displays the verification status (e.g., "Certified" / "Tampered") directly in the CoLab UI.
- **Documentation:**
  - Clear usage instructions, props definitions, and examples for quick adoption.

4.4 The Public Verification Portal (External Assurance)

This crucial tool ensures third-party trust and compliance.

- **Standalone Web Application:**
  - A single-page application (SPA), hosted separately, designed for external use (auditors, clients).
- **Functionality:**
  - Allows a user to upload a file to perform a local hash calculation and then makes a direct, free read-only view call to the EVM contract.
- **Trust:**
  - Provides independent, trustless proof of integrity without requiring the auditor to interact with CoLab's production systems.

5 Functional Requirements

5.1 Python SDK

Requirement	Description
Local File Hashing	• Must generate SHA-256 hash values for files and comment threads locally on server. • supporting large CAD/image/PDF files via efficient I/O streaming.
Secure EVM Certification	• sdk.certify_file(), must securely manage the certification wallet's private key.
Record Data Structure	Input: • The file hash(es). • Comment hash. • CoLab metadata (User ID, Project ID). • Timestamp. Output: • A single, compact data structure for the Solidity contract.
Internal Integrity Check	• sdk.check_integrity(hash) method to quickly query the EVM contract for a hash's existence.
CoLab Workflow Integration	• Provide clear hooks to automatically trigger certification upon a 'ReviewClosed' event in CoLab's Python backend.

5.2 EVM Smart Contract

Requirement	Description
Data Immutability	• Store hash records permanently on the EVM-compatible chain using the recordHash() function.
Verification Retrieval	• Expose a read-only view function (checkHashExists()) that allows for quick, gas-free public verification checks.
ABI and Address	• The final Contract Address and the Application Binary Interface (ABI) JSON file must be delivered for all integration efforts.

5.3 React Component Library & Portal

Requirement	Description
Certification Trigger Component	• Deliver a reusable React component (e.g., <CertifyButton>) that triggers the certification flow via CoLab's backend API.
Public Verification Portal	• A standalone React application where an external user can upload a file, calculate the hash locally in the browser, and make a direct EVM view call for verification.
Display Results	Verification results: • Must be clear and display block details. • Show Timestamp. • Show metadata. • URL link to the EVM block explorer.

6 Non-Functional Requirements

Requirement	Description
Security (Wallet)	• Absolute priority must be placed on the secure management and isolation of the Python SDK's private key used for signing certification transactions.
Performance (Hashing)	• Python SDK hashing implementation must be optimized to handle multi-gigabyte files efficiently to avoid backend latency.
Usability (React)	• The React components and the Verification Portal must adhere to CoLab's UI standards. • Provide clear status feedback (e.g., "Transaction Pending," "Certified") to minimize user confusion about blockchain processes.
Scalability (EVM)	• The smart contract must be gas-optimized to ensure transaction costs remain low even as the usage volume increases.
Documentation	• Provide comprehensive documentation for the Python SDK (installation, usage) and the React components (props, examples).

7 Use Cases

Use Case	Primary Actor/Component	Description (EVM & SDK Context)
1. Close Review & Seal Design		
	CoLab Backend (Python SDK)	• The Python SDK automatically triggers upon the ReviewClosed event. • It securely hashes all final artifacts, bundles the data • It uses the certification wallet to sign and submit the transaction (paying gas) to the EVM contract.
2. Verify File Integrity		
	External Auditor (Public React Portal)	• The auditor uploads the file to the React Portal. • The browser calculates the hash and makes a direct, gas-free view call. • EVM contract is deployed for instant, trustless verification.
3. API-based Integration		
	CoLab Backend Engineering Team	• A user imports the certroot-sdk Python package. • `sdk.certify_file()` can be integrated into core review closure logic, abstracting the complex EVM process.
4. View Blockchain Record		
	CoLab User (React Component)	• A user clicks an "Integrity Proof" link. • The React Component retrieves the transaction details from a public EVM Block Explorer. • The system will redirect and display the timestamp and certification metadata.
5. Optional File Archival		
	Python SDK & External Storage	• The Python SDK manages the secure upload of large files to an external system (e.g., S3). • It then logs the file's hash and the resulting storage URL/pointer in the EVM contract metadata.
6. Audit Trail & Reporting		
	Compliance Officer (CoLab Backend Logic)	• Python SDK can be used to query a range of recorded hashes from the EVM contract. • Provide a compiled report that shows every sealed design for a given project ID for regulatory review.

8 Development Schedule

8.1 Phase 1: Foundation & Backend Focus

Date Range	**Focus (4 Devs)**	**Solo Focus (1 Dev)**	**Client Meeting**
Oct 1 – Oct 4	Project setup: EVM/Solidity environment, Python SDK boilerplate, initial CI/CD foundation.	Verification Portal UI: Build the front-end shell for the public verification tool (React).	None
Oct 7 – Oct 11	Solidity MVP: Develop the immutable smart contract (recordHash). Develop the core Python hashing function.	Verification Portal UX: Design and implement the verification result display logic.	Oct 13: Kick-off/Review
Oct 14 – Oct 18	Python SDK Write Logic: Implement web3.py for secure transaction signing (wallet management, gas estimation).	Demo Prep: Integrate the client-side hashing function for the public portal.	None
Oct 21 – Oct 25	Integration & Test: Full E2E testing (Python to Solidity testnet). Finalize Python SDK CI/CD.	Demo Prep: Final E2E testing of the verification read function on the portal.	Oct 25: Phase 1 Demo

8.2 Phase 2: Client Integration & Polish

Date Range	**Focus (4 Devs)**	**Maintenance (1 Dev)**	**Client Meeting**
Nov 4 – Nov 8	React Component 1: Develop the core ¡CertifyButton¿ and set up the NPM package structure and CI/CD.	Maintenance & Docs: Write comprehensive Python SDK documentation and fix any Phase 1 issues.	None
Nov 11 – Nov 15	React Component 2 & Portal: Develop the ¡IntegrityBadge¿ and finalize the Public Verification Portal UI/UX.	Refinement: Final Python SDK changes based on client review feedback.	Nov 17: Component Review
Nov 18 – Nov 22	QA & Packaging: Full integration testing (React to Python SDK). Final NPM package and Python wheel preparation.	Release Prep: Finalize all documentation, release notes, and prepare for handover.	None
Nov 25 – Nov 29	Final Submission: Prepare the final presentation and deliver all source code, libraries, and documentation.	Handover Prep: Prepare a technical walkthrough of the Python SDK security features.	Dec 1: Final Handover