Salma Lemrazzeq
Information **S**ystems **S**ecurity **E**ngineer

Technical **W**hitepaper **& P**roject **R**eport

CERTIFLOW

A LOCAL HARDWARE-BACKED PKI SIGNING INFRASTRUCTURE

CertiFlow is a fully local digital signing infrastructure using a dedicated hardware signing module to protect private keys, enforce audit logging, and enable offline certificate validation. The system reproduces real-world PKI trust models while remaining transparent, affordable, and self-hosted.

2025 - Open **T**echnical **P**ortfolio **E**dition

ABSTRACT

This project presents the design and implementation of CertiFlow, a secure digital signing infrastructure built to strengthen document authenticity and institutional trust. The system introduces three coordinated components: the Signer Application, which allows users to create and manage their digital signatures; the CA Owner Application, which oversees certificate issuance, validation, and revocation; and the Verifier Application, which ensures independent verification of signed documents. Each component operates in a controlled environment, maintaining confidentiality and integrity through strong cryptographic mechanisms and clear trust boundaries. The platform also integrates a dedicated Hardware Signing Module to protect private keys and perform secure signing operations locally, reducing exposure to attacks and improving reliability. By combining automation, transparency, and hardware-based assurance, CertiFlow demonstrates a practical and resilient approach to institutional digital signatures. The work also explores deployment considerations, user workflows, and test results that confirm the robustness and stability of the system.

Keywords: Digital signature, Certification Authority, Hardware Signing Module, Verification, Trust, Security, Infrastructure.

RÉSUMÉ

Ce projet présente la conception et la mise en œuvre de CertiFlow, une infrastructure de signature numérique sécurisée visant à renforcer l'authenticité des documents et la confiance institutionnelle. Le système repose sur trois composants principaux : l'application du signataire, permettant la création et la gestion des signatures numériques ; l'application du responsable de l'autorité de certification, chargée de l'émission, de la validation et de la révocation des certificats ; et l'application de vérification, assurant le contrôle indépendant des documents signés. Chaque composant fonctionne dans un environnement maîtrisé, garantissant la confidentialité et l'intégrité grâce à des mécanismes cryptographiques robustes et à des frontières de confiance bien définies. La plateforme intègre également un module matériel de signature destiné à protéger les clés privées et à réaliser les opérations de signature de manière locale et sécurisée, limitant ainsi les risques d'exposition et augmentant la fiabilité du processus. En alliant automatisation, transparence et sécurité matérielle, CertiFlow propose une approche pratique et résiliente de la signature numérique institutionnelle. L'étude présente aussi les aspects de déploiement, les scénarios d'utilisation et les résultats de test qui confirment la solidité et la stabilité du système.

Mots-clés : Signature numérique, Autorité de certification, Module matériel de signature, Vérification, Confiance, Sécurité, Infrastructure.

Table of Contents

List of Figures

List of Tables

Acronyms and Abbreviations

A

- **ACME**: Automated Certificate Management Environment
- **AES**: Advanced Encryption Standard
- **API**: Application Programming Interface
- **APP**: Application
- **ASN.1**: Abstract Syntax Notation One

C

- **CA**: Certification Authority
- **CAO**: CA Owner Application
- **CDC**: Communications Device Class (USB)
- **CLI**: Command-Line Interface
- **COM**: Communications port (serial)
- **CPU**: Central Processing Unit
- **CRL**: Certificate Revocation List
- **CSR**: Certificate Signing Request
- **CSV**: Comma-Separated Values

D

- **DER**: Distinguished Encoding Rules
- **DFU**: Device Firmware Upgrade

E

- **ECC**: Elliptic Curve Cryptography
- **ECDSA**: Elliptic Curve Digital Signature Algorithm
- **EOL**: End of Line (protocol line ending)

F

- **FIPS**: Federal Information Processing Standards
- **FW**: Firmware

G

- **GB**: Gigabyte
- **GUI**: Graphical User Interface

H

- **HAL**: Hardware Abstraction Layer

- **HID**: Human Interface Device
- **HSM**: Hardware Security Module
- **HSMID**: Hardware Security Module Identifier
- **HTTP**: Hypertext Transfer Protocol
- **HTTP/JSON**: HTTP transport with JSON payloads

I

- **IETF**: Internet Engineering Task Force
- **INFO**: Device command returning status or metadata
- **IOC**: Initialisation and Configuration file (embedded project)
- **IP**: Internet Protocol
- **ISO**: International Organization for Standardization

J

- **JSON**: JavaScript Object Notation

K

- **KB**: Kilobyte
- **KEYGEN**: Device command to generate a key pair
- **KMS**: Key Management Service/Storage

L

- **LAN**: Local Area Network
- **LED**: Light-Emitting Diode

M

- **MB**: Megabyte
- **MCU**: Microcontroller Unit
- **MITM**: Man-in-the-Middle
- **MSI**: Multi-Speed Internal oscillator (clock source)

N

- **NIST**: National Institute of Standards and Technology

O

- **OCSP**: Online Certificate Status Protocol
- **OID**: Object Identifier
- **OTP**: One-Time Password
- **OS**: Operating System

P

- **PEM**: Privacy-Enhanced Mail format (Base64 wrapper)
- **PKA**: Public Key Accelerator (hardware block)
- **PKCS**: Public-Key Cryptography Standards (family)
- **PKCS#10**: Certificate request syntax used in CSRs
- **PKCS#11**: Cryptographic token interface standard
- **PKI**: Public Key Infrastructure
- **PIN**: Personal Identification Number
- **P-256**: NIST secp256r1 elliptic curve
- **PUTTY**: Terminal program used for serial testing

R

- **RDP**: Readout Protection (microcontroller flash access level)
- **RFC**: Request for Comments (IETF standard document)
- **REST**: Representational State Transfer
- **RNG**: Random Number Generator
- **RSA**: Rivest–Shamir–Adleman (public-key cryptosystem)
- **RTT**: Round-Trip Time
- **RX/TX**: Receive/Transmit (serial directions)

S

- **SAN**: Subject Alternative Name (X.509 extension)
- **SDK**: Software Development Kit
- **SHA-256**: Secure Hash Algorithm, 256-bit
- **SIG**: Signature operation/service or SIGN command
- **SPKI**: Subject Public Key Info (X.509 structure)
- **SQL**: Structured Query Language
- **SSH**: Secure Shell
- **SWD**: Serial Wire Debug

T

- **TFM**: Trusted Firmware-M
- **TLS**: Transport Layer Security
- **TPM**: Trusted Platform Module (referenced in comparisons)
- **TS**: Trust Store

- **TX/RX**: Transmit/Receive

U

- **UART**: Universal Asynchronous Receiver-Transmitter
- **UID**: Unique Identifier (factory MCU ID)
- **UI**: User Interface
- **UNLOCK**: Device command to open a session with PIN
- **USB**: Universal Serial Bus
- **USB FS/HS**: USB Full-Speed or High-Speed
- **UTC**: Coordinated Universal Time

V

- **VID/PID**: Vendor ID and Product ID (USB identifiers)
- **Verifier App**: Verifier Application (project component)

X

- **X.509**: ITU-T certificate and CRL standard
- **XML**: Extensible Markup Language

Y

- **YubiHSM**: Yubico Hardware Security Module (ecosystem example)

General Introduction

The digital world we live in today depends on trust, trust that the documents we read, sign, and share are real, complete, and unchanged. In institutions, administrations, and companies, official documents circulate every day, often in electronic form. The problem is that a simple PDF file can be altered in seconds, and without proper digital protection, it becomes impossible to prove who really signed it or when. This has turned digital signatures into an essential part of modern security. They make it possible to guarantee that a document has not been modified and that it truly comes from its author.

However, the reliability of a digital signature depends directly on how well the signing keys are protected. Most systems rely on software-based key storage, which is convenient but exposes private keys to the risk of theft, malware, or unauthorized access. In recent years, a new focus has emerged on hardware-based security, where keys are generated and stored inside a secure physical device known as a Hardware Security Module (HSM). These modules keep the most sensitive operations away from the host computer and prevent keys from being copied or extracted.

Unfortunately, professional HSMs are often complex and very expensive, which makes them difficult to use in smaller organisations, universities, or local environments where budgets and resources are limited. This project was born from the idea of building a local, affordable, and transparent alternative: a complete digital signing system where each user owns their own personal HSM, implemented on a small STM32 microcontroller, and managed through a local certification authority. The result is CertiFlow, a full document-signing ecosystem that works entirely offline and offers end-to-end trust without depending on external services or cloud platforms.

CertiFlow is made up of three desktop applications that communicate with each other through a local network and a hardware token. The User Application allows users to register, activate their HSM, and sign documents. The CA Owner Application manages user verification, certificate issuance, and system logs. Finally, the Verifier Application checks the authenticity of signed documents using the local trust chain. Together, these components reproduce the essential functions of a real-world PKI infrastructure, but in a form that is accessible, lightweight, and entirely self-hosted.

The practical part of this work also included the development of a custom firmware for the STM32U585 microcontroller, transforming it into a real hardware signing device. The firmware handles key generation, PIN-based access control, and secure signature operations, while ensuring that the private key never leaves the chip. Several prototypes were explored during development: one using simple USB storage for testing, another using a Python-based HSM emulator, and a final version running directly on the STM32 board.

This report presents the full path from concept to implementation**. The first chapter** gives the general background, explaining the principles behind digital signatures, public key infrastructure, and hardware security modules, while also reviewing similar existing solutions. **The second chapter** defines the project's context, objectives, and problem statement, and details the specifications and constraints that guided the design. **The third chapter** focuses on the conception and analysis of the system, describing its architecture, the interaction between components, and the reasoning behind key design choices. **The fourth chapter** presents the practical implementation: the technologies used, the developed applications, the embedded firmware, and the integration of the STM32-based HSM into the overall system. **The fifth chapter** evaluates the results, discusses testing outcomes, and reflects on the strengths and limitations of the proposed solution and proposes possible improvements and perspectives for future work, particularly the integration of more advanced security mechanisms such as TrustZone or secure communication channels. Finally, the conclusion summarizes the main achievements.

Chapter 1: Background and State of the Art

1. Introduction

The security of digital information has become a central concern for both individuals and institutions. Every day, files are exchanged, signed, and stored in electronic form, which makes their integrity and authenticity critical. Yet, these properties cannot be guaranteed without reliable cryptographic mechanisms. This chapter provides the general foundation for understanding how digital signatures protect documents and identities. It introduces the main concepts of modern cryptography, explains the role of public key infrastructure, and presents the use of hardware in protecting private keys. The goal is to give a clear background that supports the system analysis and implementation presented in later chapters.

2. Core Concepts of Digital Signatures

Digital signatures are one of the most effective ways to protect the authenticity and integrity of electronic documents. They work on the principle of linking a unique cryptographic identity to a person or system so that any change in the data can be detected. A digital signature acts as a digital equivalent of a handwritten signature, but with much stronger security guarantees. When someone signs a document, the process generates a small piece of data that depends entirely on the document's content and the signer's private key. If the document is later modified, the signature instantly becomes invalid.

Behind this process are two fundamental ideas in modern cryptography: **hashing** and **public-key cryptography**. Hashing transforms any message into a short, fixed-size code called a hash. Even the smallest change in the original message results in a completely different hash, which makes it ideal for detecting tampering. Algorithms like SHA-256 are commonly used because they produce consistent results that are extremely hard to reverse or predict. In a digital signature, the document's hash is the element that gets signed, not the document itself, which keeps the operation fast and secure.

Public-key cryptography adds identity and verification to the process. Each user has a key pair: a private key known only to the owner and a public key that others can use to check the signature. When a signature is created, the signer uses their private key to encrypt the hash. Anyone with the matching public key can then verify the signature by decrypting it and comparing the result to a freshly computed hash of the same document. If the two hashes match, the document is confirmed to be original and unmodified, and the signature is proven to come from the claimed signer.

This combination of hashing and asymmetric cryptography gives digital signatures three essential properties: integrity, since any modification breaks the signature; authenticity, because the signer's identity is bound to their key; and non-repudiation, meaning the signer cannot later deny their participation. These properties form the foundation of trust in electronic communication, online transactions, and document management systems.

Figure 1 below illustrates the main steps of digital signing and verification. On the left, the original data is first processed through a hash function to produce a unique fixed-size digest. This digest is then encrypted with the signer's private key to generate the signature, which can be attached to the document together with the signer's certificate. On the right, verification repeats the hashing process and compares the new digest with the one obtained by decrypting the signature using the signer's public key. If both values match, the document is confirmed to be authentic and unaltered.

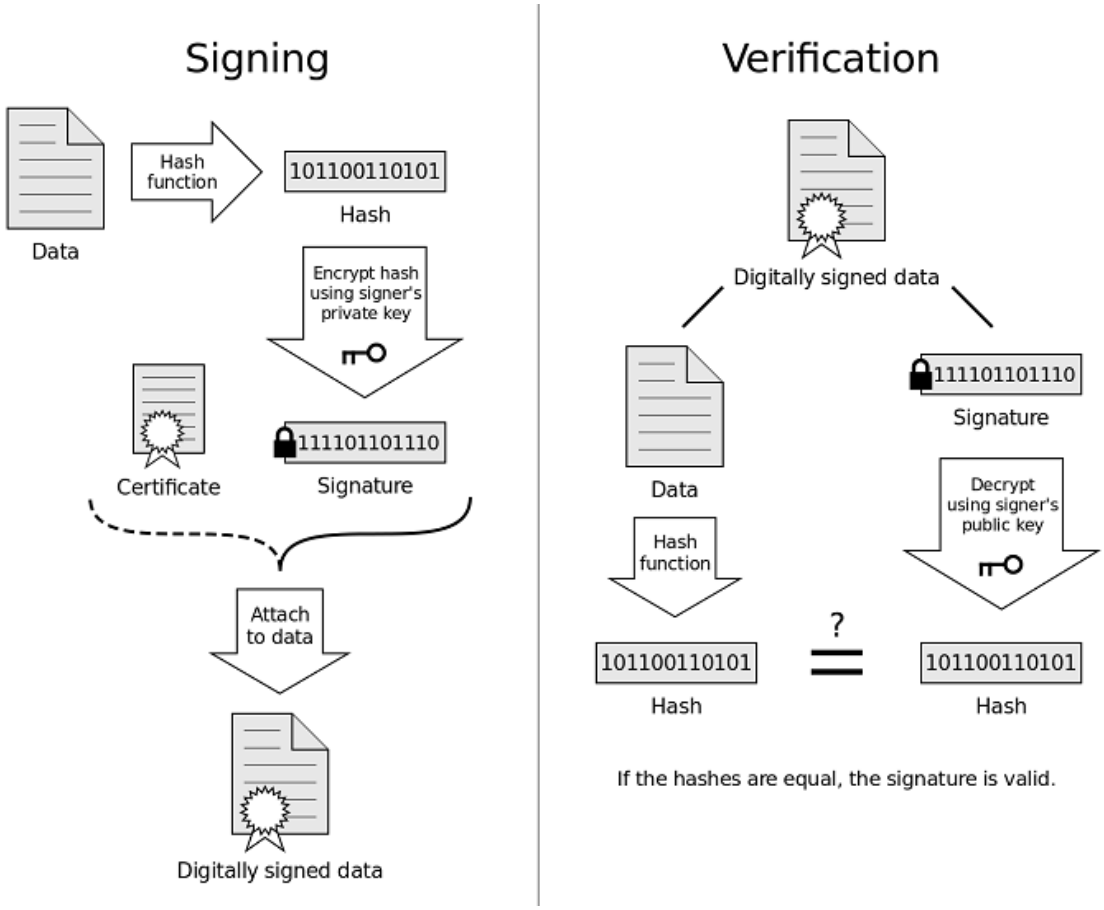


Figure 1: Digital signature: hash and key pair (1)

3. PKI and Trust

A digital signature alone proves that a document was signed with a specific private key, but it does not tell us who actually owns that key. To solve this problem, the security world relies on Public Key Infrastructure (PKI). PKI is a framework that links public keys to verified identities through digital certificates. These certificates are issued by trusted authorities called Certification Authorities (CAs), which act as the identity police of the system.

When a CA signs a certificate, it confirms that a certain public key truly belongs to a specific person, organisation, or device. Certificates are organized in a hierarchy of trust that usually starts from a root CA, followed by optional intermediate CAs, and finally reaches end-user certificates. Each level signs the one below it, forming a continuous chain of trust that allows any user to verify the authenticity of a certificate by tracing it back to a trusted root.

Trust also requires proper handling of revocation, which deals with certificates that must no longer be accepted, either because they were compromised, expired, or replaced. This is typically managed through a Certificate Revocation List (CRL) or an Online Certificate Status Protocol (OCSP) service. These mechanisms let systems check, in real time or on demand, whether a certificate is still valid or not.

Together, these elements create a trust model that extends beyond individual keys and users. PKI ensures that when a document is signed and verified, every participant can rely not only on the mathematics of cryptography but also on an organized system of verified identities.

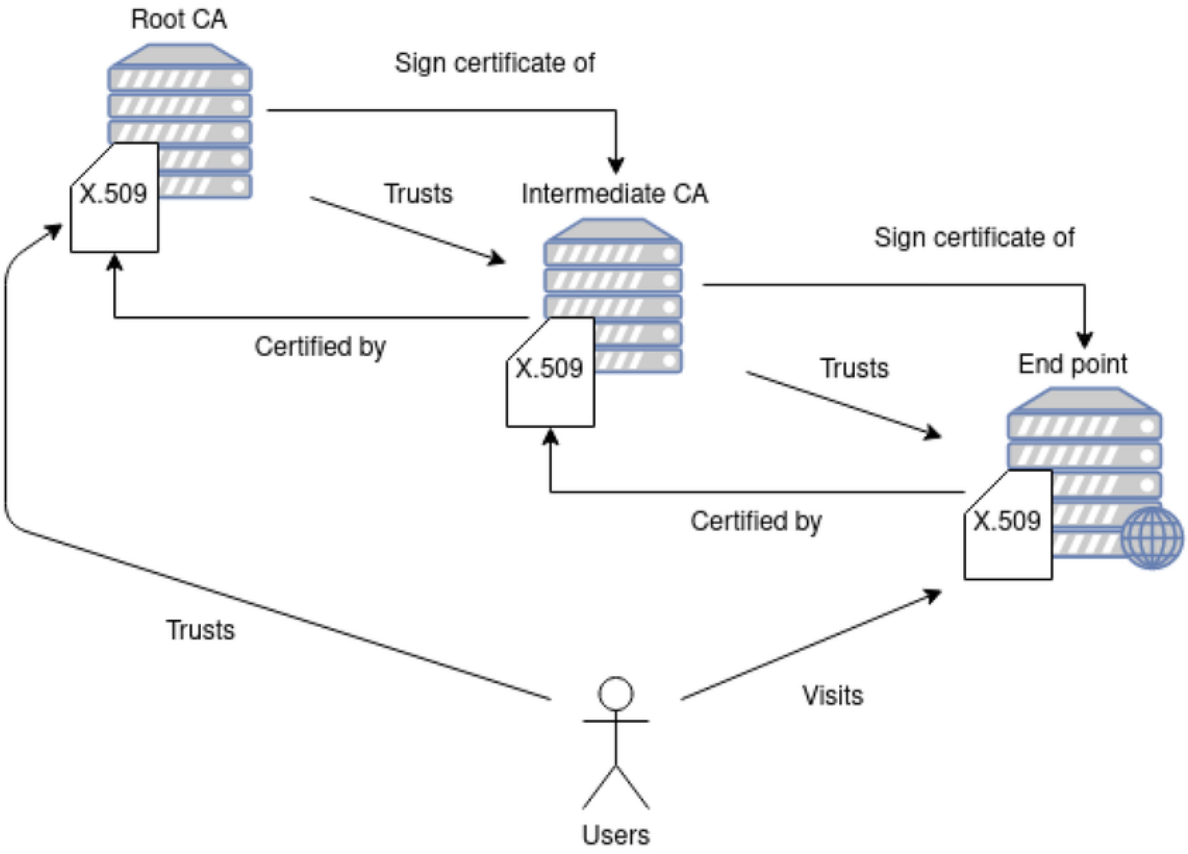


Figure 2: Certificate chain: root to end-entity (2)

Figure 2 presents the concept of a chain of trust in a public key infrastructure. The root CA sits at the top and signs the certificates of intermediate authorities, which in turn issue certificates to end users or devices. Each level confirms the authenticity of the one below it, allowing any certificate to be verified by tracing it back to the trusted root.

4. Hardware-Backed Key Protection

The security of a digital signature depends on how well the private key is protected. In many systems, keys are stored as files or software tokens on the computer that performs the signing. This approach is simple but fragile. If the device is infected with malware or an attacker gains access to the file system, private keys can be copied or stolen without leaving any visible trace. Once a private key is compromised, all documents signed with it lose their trust value.

To reduce these risks, modern security practices move sensitive key operations into dedicated hardware. A Hardware Security Module (HSM) is a physical device designed to generate, store, and use cryptographic keys in isolation from the host system. The main idea is that the private key never leaves the module. Signing and decryption are performed inside the hardware, and only the results are returned to the computer. This separation makes it much harder for attackers to extract or misuse the key, even if the main computer is compromised.

Hardware protection also improves reliability. Most HSMs include features such as secure memory, hardware random-number generators, tamper detection, and limited access through PINs or policies. They are used in servers, banking systems, certificate authorities, and personal tokens for secure authentication. However, these devices can be expensive and sometimes difficult to integrate, especially

in smaller organisations. For this reason, lighter alternatives have appeared, such as USB tokens and smartcards that use the same principle on a smaller scale.

In short, hardware-based protection transforms private keys from vulnerable files into physical secrets. It adds a tangible layer of defence to the cryptographic process, ensuring that trust in digital signatures extends beyond software boundaries.

Table 1: Comparison between software-based and hardware-based key protection

Aspect	Software Key Protection	Hardware Key Protection
Storage Location	Keys stored on the host computer (file system or software vault).	Keys stored inside a secure physical device (HSM, token, or smartcard).
Exposure Risk	High, malware or unauthorized access can copy or read private keys.	Very low, private keys never leave the hardware and cannot be extracted.
Operation Method	Signing and encryption done by the host system using the key file.	All cryptographic operations executed inside the hardware module.
Security Features	Depends on software protection, operating system, and user practices.	Includes secure memory, hardware RNG, PIN access, and tamper detection.
Cost and Complexity	Low cost, easy to deploy, but weaker protection.	Higher cost and setup effort, but strong long-term security.

Table 1 summarizes the main differences between software-based and hardware-based key protection. It shows how storing keys in dedicated hardware devices greatly reduces the risk of compromise by isolating cryptographic operations from the host system, while software storage remains easier to deploy but less secure in practice.

5. Local Ecosystem and Existing Solutions

Building a local, self-hosted signing environment typically combines three ingredients: a certificate authority (CA), hardware or software for key protection, and tooling to sign and verify documents. For the CA layer, widely used open-source options include **EJBCA** Community, a mature, full-featured PKI that handles issuance, revocation, and validation with scalable deployment options and a long maintenance history (3), and **CFSSL**, a lightweight toolkit that can act as a signing API or CLI for internal CAs in labs and private networks (4, 5). Both are actively documented and commonly used in production and test environments.

Modern teams also adopt **step-ca** when they want a small, automatable private CA that supports ACME, short-lived certificates, and easy installation on standard Linux hosts or single-board computers (6). **HashiCorp's Vault PKI** is another option when certificate issuance must be integrated with centralized secrets management and policy controls; it exposes both CLI and HTTP APIs for dynamic X.509 issuance (7). These tools are well-suited to offline or campus networks and can be secured further by storing the CA key in a hardware token.

For hardware-backed key protection, the landscape ranges from data-centre appliances to USB tokens. At the high end, **Thales Luna** devices exist as network, PCIe, or USB HSMs with tamper-resistance and management features aimed at enterprise Cas (8). At the accessible end, **YubiHSM** 2 provides a compact USB form factor for protecting CA keys and application keys, including a

Figure 3: Thales Luna device

FIPS-validated variant, which makes it attractive for smaller institutions (9). Open hardware–leaning options such as **Nitrokey HSM** 2 and **SmartCard-HSM** offer PKCS#11 interfaces and documented toolchains; they are frequently used for lab CAs or developer workflows (10, 11). When physical hardware is not available, **SoftHSM2** implements the PKCS#11 interface in software for development and CI; it can even be exposed over the network using pkcs11-proxy containers, with the explicit caveat that it is not for production (12).

Figure 4: Yubico YubiHSM 2 Device

In short, the current ecosystem already provides strong, mature tools for public-key management, document signing, and hardware-based key protection. However, most of these solutions are either costly, closed-source, or optimized for large infrastructures rather than local or academic environments. Open-source frameworks like EJBCA, step-ca, and SoftHSM2 fill important gaps but still depend on external servers or emulated security modules. Commercial tokens such as Thales Luna or YubiHSM 2 offer certified protection but lack flexibility and affordability for small deployments. These observations confirm that there is still room for practical, low-cost alternatives that combine open-source transparency with genuine hardware isolation. The system developed in this project aims to address that exact space by designing a local, USB-based HSM integrated into a self-contained certificate infrastructure, bringing secure signing capabilities to constrained or offline contexts

6. Conclusion

This chapter introduced the fundamental concepts that support secure digital communication and document signing. It explained how digital signatures combine hashing and asymmetric cryptography to guarantee authenticity and integrity, and how public key infrastructure extends this trust through verified identities and certificate hierarchies. The section on hardware protection highlighted the role of HSMs in securing private keys and compared software-based and hardware-based approaches. Finally, the overview of existing solutions showed that while many tools exist, few are adapted to local or low-cost environments. The next chapter builds on this foundation by defining the specific problem addressed in this project, its objectives, and the technical and functional requirements that guided the system design.

Chapter 2: Problem Statement and Specifications

1. Introduction

Before developing any system, it is important to first identify the problem it addresses, define clear objectives, and establish realistic boundaries for its scope. This chapter explains the motivation behind the project and the goals it was designed to achieve. It also describes the functional and technical requirements that guided its development, along with the main constraints and choices that influenced its design. These elements form the basis for the system's architecture and practical implementation discussed in the next chapters.

2. Problem Statement

In many institutions and small organisations, the process of signing official documents still relies heavily on paper, manual approval, or centralized online platforms that are often costly or inaccessible. Even when digital signatures are used, they tend to depend on third-party services or external hardware that lacks transparency, is expensive to scale, or doesn't offer full control over private key management. This becomes a real concern when dealing with sensitive documents, particularly in contexts where the authenticity and integrity of those files must be preserved, even when offline.

The core risk is the protection of private keys. When keys are stored in software on the host machine, malware or unauthorized access can copy them, which undermines every signature made with those keys. Moving key generation and signing into dedicated hardware reduces this exposure by keeping secrets inside the device and performing critical operations there. At the same time, binding a public key to a real identity requires certificates and a chain of trust anchored in a recognized authority, so signatures can be validated by others in a consistent way. In practice, many hardware options on the market are designed for large enterprises and come with higher costs and integration effort, which limits adoption in smaller settings (17, 18).

Problem statement: There is a lack of accessible, hardware-secured and locally operated systems for creating and verifying digital signatures in institutional or small-scale environments. Such a system must protect private keys from exposure, rely on standard PKI for identity and validation, operate reliably without external services, and remain feasible to deploy and maintain (13, 15). So how can a system be designed to offer the same level of trust and security as professional digital-signature platforms, while remaining fully local, affordable, and under the user's complete control?

3. Proposed Solution

To address the absence of accessible, hardware-secured signing systems for local environments, the proposed solution introduces a complete digital-signing framework that operates independently of external servers or online services. Its purpose is to provide the same level of trust and cryptographic assurance found in professional systems, but in a form that can be deployed and maintained within small organisations, universities, or administrative institutions.

The system relies on a combination of software and hardware components that together reproduce the core principles of a public key infrastructure (PKI) (13). Each user possesses a personal hardware device that generates and protects their private key, while a local certification authority (CA) manages identity verification and certificate issuance. This setup ensures that private keys never leave the user's device, and that each signature can be traced to a verified identity through a trusted certificate chain. By keeping all operations within a closed environment, the system maintains control over data and reduces exposure to external threats (15, 18).

The proposed approach also aims to balance security with practicality. It avoids dependence on expensive enterprise-grade hardware by adopting lightweight, dedicated modules capable of secure key storage and signing operations. The local CA component reinforces the trust model, ensuring that signatures remain verifiable within the institution's internal network, even without internet connectivity. Together, these elements create a self-contained infrastructure that is both secure and sustainable, offering a feasible alternative to traditional large-scale or cloud-based digital signature solutions (17, 19).

4. Project Objectives

The main objective of this project is to design and implement a secure local system that enables users to digitally sign documents while maintaining full control over their private keys. The system is designed to operate entirely offline and to serve as a practical alternative to commercial digital-signature platforms, particularly in environments where cost, autonomy, and data protection are priorities.

To achieve this goal, the project is built around two core components. The first is a local Certificate Authority (CA), hosted on a standalone computer or embedded system, which is responsible for issuing and managing user certificates, ensuring trust, and maintaining the integrity of the certification chain. The second is a hardware signing device based on a microcontroller, which functions as a lightweight Hardware Security Module (HSM). This device securely generates and stores cryptographic keys, performs signature operations internally, and prevents private key exposure to the host system (15, 17).

The specific objectives of the project include:

- → building a functional and secure local CA capable of issuing and managing X.509 certificates
- → Designing a desktop signing application that communicates with the hardware device and manages document signing and verification.
- → Developing embedded firmware for the microcontroller that handles key generation, secure storage, and signing operations.
- → Ensuring that private keys remain protected within the HSM device and are never exported or exposed.
- → Allowing users to sign and verify PDF documents using standard cryptographic validation tools.
- → Keeping the system self-contained, offline, and simple to deploy in real-world institutional settings.

These objectives combine technical and practical aims, emphasizing both system security and accessibility. The project seeks to deliver a tool that can be reused and adapted in academic, administrative, or organisational infrastructures that require local, hardware-based signing capabilities.

5. Scope and Limitations

This project focuses on creating a complete, self-contained digital-signature system that can function entirely offline, without relying on cloud infrastructure or third-party platforms. Its scope was defined to include only the essential components required to guarantee document authenticity, ensure secure key management, and establish a verifiable trust model, while keeping the overall design simple, efficient, and reproducible with standard hardware and open-source tools.

The system includes a local certification authority responsible for managing user identities and issuing digital certificates, an application that handles document signing and verification, and a secure mechanism for protecting private keys during signature operations. Together, these elements reproduce

the main principles of a public key infrastructure (PKI) (13), allowing users to generate, sign, and validate digital documents within a fully controlled environment. The design prioritizes independence, meaning all operations can be performed locally without external connectivity, ensuring both data confidentiality and long-term reliability (19).

The project is intended for controlled environments such as academic institutions, offices, or small organisations, where user identities can be verified through internal procedures or institutional accounts. It aims to provide a balance between accessibility, security, and practicality, showing that trust in digital communication does not necessarily depend on large-scale or commercial systems.

However, the system does have limitations. It was not designed for large infrastructures or high-volume certificate management. It excludes advanced enterprise features such as automated key recovery, networked databases, or real-time validation through OCSP (13). Certificate revocation is managed using local lists, and scalability remains limited by design. While the system follows the same principles as professional PKI solutions, it is primarily a proof of concept meant to demonstrate that strong digital-signature security can be achieved through local and autonomous infrastructures.

Despite these limits, the project establishes a solid foundation for secure document authentication in small-scale or offline contexts. It can be further expanded to include more advanced mechanisms, stronger hardware protection, and integration with broader institutional systems in future versions.

6. Planning and Methodology

The project followed a structured and progressive methodology to ensure that each stage contributed effectively to the final system. The work was divided into successive phases, each building upon the previous one. This approach helped maintain a clear workflow, from the initial research and analysis to the design, implementation, and validation of the system.

The adopted methodology was organized into five main stages. The preliminary study focused on understanding the principles of digital signatures, public key infrastructure, and hardware-assisted security. The specification and design phase identified system requirements, defined component interactions, and produced the conceptual architecture that guided the rest of the work. During the implementation phase, each subsystem was developed and tested individually before being integrated into the complete environment. The testing and validation phase ensured that the system met its functional objectives, verifying signature integrity, data authenticity, and operational reliability. Finally, the documentation and presentation phase consolidated all results into the final report and prepared the project for defence.

To organize and monitor progress, a project plan was established before the start of implementation. Two Gantt charts are presented below. The first represents the planned timeline defined at the beginning of the project, while the second shows the actual timeline, reflecting the real progression and adjustments made during development. The differences between the two charts illustrate how the initial schedule evolved to accommodate design refinements, debugging, and documentation work carried out in parallel with system testing.

Table 2 below presents an overview of the main phases and their estimated timelines, plus the actual timeline taken to finish them.

Table 2: Planned vs. Actual Project Phases

Phase	**Planned Duration**	**Actual Duration**	**Notes**
Preliminary Study	2 weeks	2 weeks	Literature review and analysis of existing PKI and HSM concepts.
Requirements & Design	2 weeks	4 weeks	Refinement of functional and security specifications.
Implementation	4 weeks	11 weeks	Integration and feature development took longer than expected.
Testing & Validation	2 weeks	10 weeks	Extra time for verification and user-level testing.
Documentation & Defense Preparation	2 weeks	13 weeks	Writing, revisions, and preparation for presentation.

Figure 5 shows the initial project schedule that was planned before development. The work was divided into five sequential phases, starting with the study and design of the system and ending with documentation and defence preparation. The schedule assumed steady progress with minimal overlap between phases.

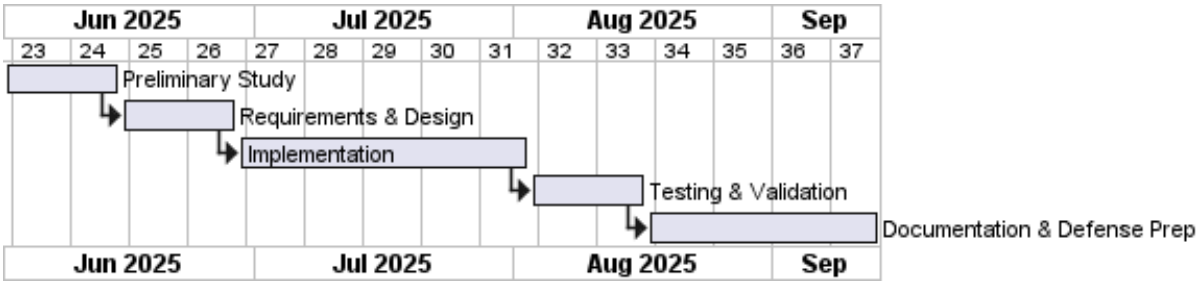


Figure 5: Gantt chart - Planned Project Schedule

Figure 6 presents the actual timeline observed during the project. Some phases required additional time due to integration challenges and testing iterations. Documentation and presentation work also began earlier than planned, overlapping with other phases. This reflects the project's adaptive and iterative progress, where design and verification evolved together.

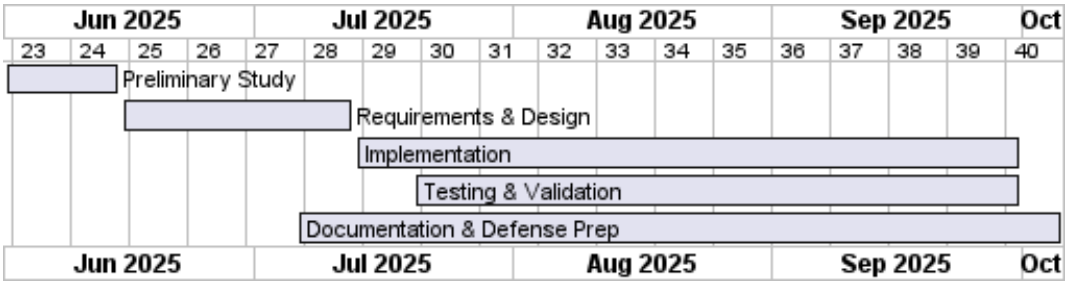


Figure 6: Gantt chart - Actual Project Timeline

7. Functional Specifications

The functional specifications define the expected behaviour of the system and describe the interactions between its main components. They translate the project's objectives into concrete functions that ensure document authenticity, user identity verification, and secure key management, all within a fully local environment. The specifications remain independent of any technical or implementation details and focus only on the logical operation of the system.

The proposed system involves three principal actors:

→ The **User**, who registers, obtains a certificate, and signs electronic documents.

→ The **Certification Authority (CA)**, which verifies identities, issues certificates, and manages revocation.

→ The **Verifier**, which checks document integrity, verifies certificate validity, and ensures that the signature originates from a trusted source.

These actors operate under the principles of public key infrastructure (PKI), ensuring that every digital signature can be linked to a validated identity and that the authenticity of documents can be verified offline and locally.

The functional specifications of the system are outlined as follows:

Table 3: Functional Requirements

ID	FUNCTION	DESCRIPTION
FR-1	User Registration	The system allows users to register with verified institutional identities before receiving a digital certificate.
FR-2	Certificate Issuance	The CA validates user information and generates a certificate that binds the user's identity to a public key.
FR-3	Secure Key Generation	The user's private key is generated and protected by a secure hardware module, ensuring it cannot be exported or duplicated.
FR-4	Document Signing	The system enables users to sign digital documents locally using their private keys.
FR-5	Signature Verification	The verifier application checks the signature's validity, confirms the document's integrity, and ensures the certificate is trusted.
FR-6	Certificate Revocation	The CA can revoke certificates when compromised or expired, updating the local revocation list.
FR-7	Offline Operation	All core functions work without internet access, maintaining trust through local verification.
FR-8	Logging and Traceability	Each major action (registration, signing, or verification) is logged to ensure traceability and accountability.
FR-9	User Interface	The system provides clear, accessible interfaces for each actor to perform their respective operations.
FR-10	Data Integrity	All stored and transmitted data must remain intact and verifiable at every stage of processing.

The system also follows several behavioural rules that govern its operation:

→ Every signature must be traceable to a valid and non-revoked certificate.

→ Private keys remain strictly protected within the secure hardware environment.

→ Only authorized and verified users can request or use certificates.

→ Each document verification must confirm both the integrity of the file and the validity of the associated certificate.

→ Revoked or expired certificates must immediately invalidate any dependent signatures.

This specification defines the expected functionality of the system from a user and organisational perspective. The next chapter presents the system's design and overall architecture, showing how these functions are structured and interact in practice.

8. Non-Functional Requirements

While functional requirements define what the system must do, non-functional requirements focus on how the system should behave. These qualities are just as important as core features, as they impact the system's usability, security, performance, and maintainability. Given the security-sensitive nature of digital signature systems, this project places a strong emphasis on protecting private keys, ensuring trust in the certificate chain, and maintaining a clean, lightweight design that works reliably in offline environments.

Table 4 summarizes the main non-functional requirements that define the expected quality and operational characteristics of the system. These requirements establish the standards for security, reliability, performance, and usability that the solution must meet to ensure consistent and trustworthy operation in real environments.

Table 4: Non-Functional Requirements

ID	CATEGORY	REQUIREMENT DESCRIPTION
NFR-1	Security	The system must protect all private keys and sensitive information from unauthorized access. All operations related to certificate management and document signing must preserve the confidentiality and integrity of data.
NFR-2	Reliability	The system must maintain consistent behaviour during operation, even when used offline or under limited resources. All transactions such as signing and verification must complete without data loss or corruption.
NFR-3	Performance	Signing and verification operations should execute efficiently for standard document sizes. The system must deliver smooth interaction and avoid long processing times during normal use.
NFR-4	Usability	Interfaces must be clear and intuitive for users with varying technical knowledge. Instructions, messages, and feedback should remain consistent and easy to understand throughout the system.

NFR-5	Maintainability	The architecture should be modular to allow updates or improvements without affecting the rest of the system. Future adjustments or feature extensions should require minimal structural changes.
NFR-6	Scalability	The system should be adaptable to different organisational sizes, from small institutions to larger administrative setups, without major reconfiguration.
NFR-7	Portability	The system should operate on standard computing environments and remain compatible with common digital-signature and certificate formats.
NFR-8	Availability	The system must remain accessible to users during normal working conditions and recover smoothly from potential interruptions or shutdowns.

These requirements ensure that the final solution is secure, dependable, and practical for real-world use. They also provide the foundation for the system architecture described in the next chapter, where these quality objectives are translated into design and structural decisions.

9. Constraints and Challenges

Every project faces certain constraints that shape its design and influence the final outcome. In this work, several factors (both technical and organisational) affected the scope, the planning, and the pace of progress. Understanding these constraints is important for interpreting the decisions made throughout the project and for identifying possible improvements in future versions.

Table 5: Main Constraints and Challenges

ID	CATEGORY	DESCRIPTION
C-1	Technical Constraints	The project had to rely on locally available hardware and open-source resources, which limited the range of components and tools that could be used. The design had to stay compatible with common systems and remain reproducible without specialized equipment.
C-2	Organisational Constraints	The project timeline was compressed, and several phases such as documentation, testing, and implementation had to progress in parallel. This required continuous adjustment of priorities and schedules to meet academic deadlines.
C-3	Security and Compliance Constraints	Balancing strong cryptographic protection with system simplicity was a recurring challenge. The design had to ensure key confidentiality and integrity while keeping the solution accessible to users with limited technical knowledge.
C-4	Human and Operational Constraints	Limited access to testers and institutional environments made it difficult to perform large-scale evaluations. The project relied mainly on controlled scenarios to validate functionality and reliability.

C-5	Resource Constraints	The work was conducted with modest hardware and time resources, which required focusing on essential functions rather than expanding into advanced features.
C-6	Environmental Constraints	The system needed to operate fully offline, which restricted the use of external validation or cloud-based components and required a fully self-contained trust model.

Table 5 summarizes the main constraints and challenges encountered during the project. These factors influenced design decisions, implementation priorities, and overall project management, shaping the final outcome into a realistic and adaptable solution.

Despite these limitations, the project achieved its main objectives by focusing on clarity, security, and practical usability. The constraints helped refine the scope of the work, leading to a functional and efficient prototype that remains adaptable for future improvements and broader institutional use.

10. Conclusion

This chapter defined the problem that motivated the project and established the framework guiding its development. It presented the objectives, scope, and expected behaviour of the system through detailed functional and non-functional requirements, while also identifying the main constraints that influenced its design. Together, these elements provide a clear foundation for understanding how the proposed solution was structured to meet real security and usability needs. The next chapter builds on this groundwork by introducing the system's overall architecture and conceptual design, explaining how the defined requirements are translated into a coherent and practical structure.

Chapter 3: System Conception and Analysis

1. Introduction

This chapter explains the system at a conceptual level: actors, components, responsibilities, and the labelled flows that connect them. It begins with a general overview of the system and its actors, followed by a detailed description of the architecture, data flow, and communication processes. Each part of the system is broken down and analysed based on its role, responsibilities, and interaction with other modules. The chapter also explains the key security principles that shaped the final architecture. Together, these elements provide a clear and organized foundation for the implementation work that follows.

2. Global System Overview

The system consists of three core applications and one secure hardware component operating inside a local environment. The User Application handles registration, requests a certificate from the CA Service, and performs document signing by delegating key operations to the Hardware Signing Module. The CA Service verifies identities, issues certificates, and maintains the local trust store, including certificate chains and revocation lists. The Verifier Application validates signed documents by checking integrity, certificate chains, and revocation status against the local trust data. These components interact through simple, well-defined exchanges: enrolment and certificate issuance, protected signing, and offline verification. The model below presents the actors, components, and main data flows without exposing any implementation details.

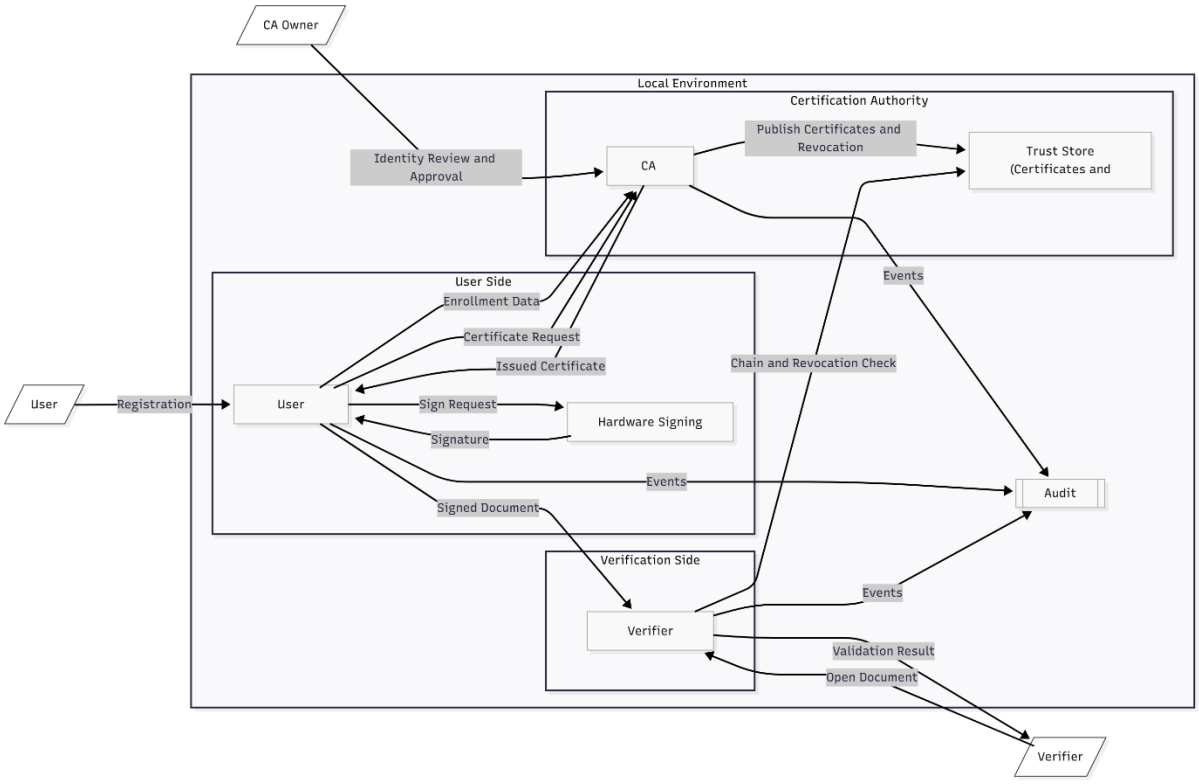


Figure 7: Global System Overview

Figure 7 shows the main actors and components in a local setup. The User enrols through the User Application, requests a certificate from the CA Service, and signs through the Hardware Signing Module. The CA Owner verifies identities, issues certificates, and publishes revocation data to the Trust Store. The Verifier uses the Verifier Application to validate signed documents offline against the Trust

Store. Only the essential flows are shown: enrolment, issuance, protected signing, verification, and revocation with basic auditing.

3. Role Definitions and System Actors

The security and functionality of this system depend heavily on how responsibilities are divided and how each actor interacts with the different components. This section defines the key actors involved, describes their roles, and illustrates their interactions through detailed use case diagrams. Each diagram captures a specific viewpoint (User, CA Owner, and System-wide), helping to clarify the actions each participant can perform and how those actions connect to the broader architecture.

A. System Actors and Responsibilities

In this part we will define the main actors of the system with their roles and responsibilities

Table 6: System Actors and Responsibilities

Actor	Description
User	A registered user who owns a personal USB HSM. The user is responsible for generating their own key pair, requesting a certificate from the local CA, signing PDF documents, and verifying digital signatures.
CA Owner	The trusted administrator who manages the Certificate Authority. This actor is responsible for initializing the CA, reviewing and approving certificate requests, issuing certificates, and maintaining the certificate revocation list (CRL).
Verifier	Any internal or external party that wants to verify the authenticity and integrity of a signed document. This can include another user, an institution, or any system that trusts the CA's root certificate.

Table 6 outlines the key actors involved in the system and their primary responsibilities. Each actor plays a specific role in ensuring secure document handling and trust management. The User is responsible for requesting certificates and signing documents through the system interface. The CA Owner oversees identity verification, certificate issuance, and revocation management to preserve trust across the network. The Verifier ensures the authenticity and integrity of signed documents by validating certificates and checking revocation data. Together, these roles define the operational structure and trust model that support the system's overall functioning.

B. Use Case Diagram - User Perspective

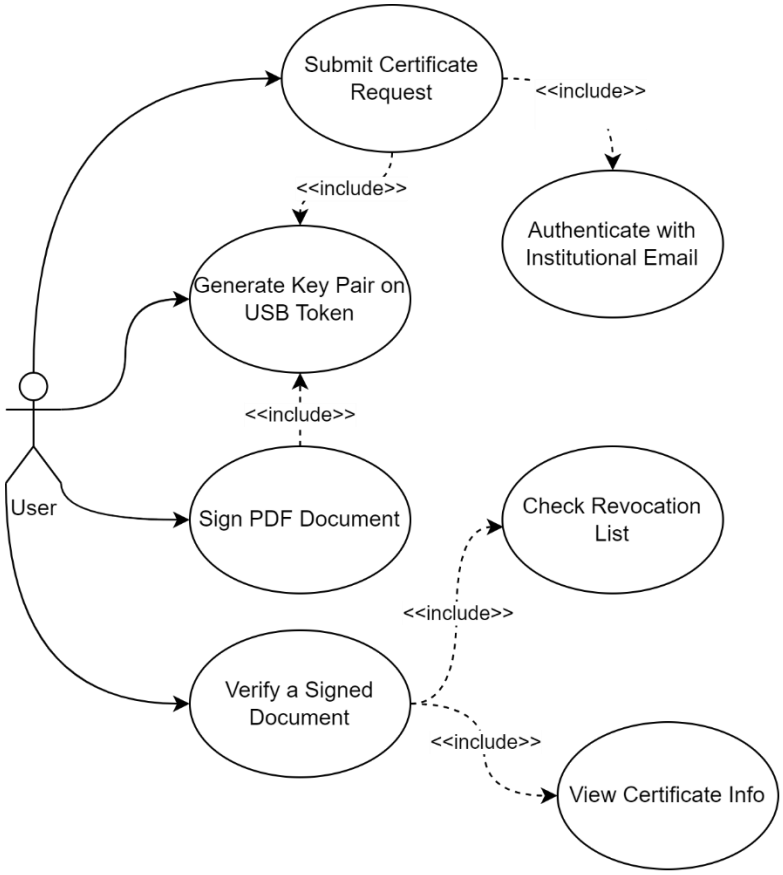


Figure 8: Use Case Diagram - User Perspective

Figure 8 presents the user's interaction with the system from a functional standpoint. It illustrates the main use cases available to the user, including registration, certificate request, document signing, and verification of signatures. Each operation is performed through the user interface, which communicates with the certification authority and the hardware signing module while keeping key operations protected. This diagram highlights how the user's actions are simplified and confined to clearly defined, secure processes within the system's workflow.

C. Use Case Diagram - CA Owner Perspective

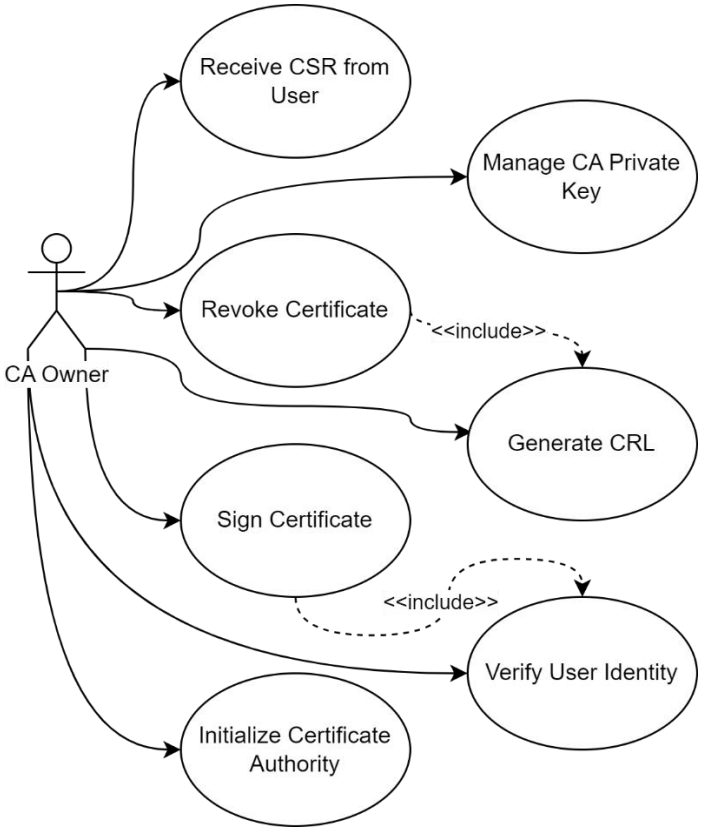


Figure 9: Use Case Diagram - CA Owner Perspective

Figure 9 illustrates the responsibilities of the certification authority (CA Owner) in maintaining trust within the system. It shows the main activities such as validating user identities, issuing and revoking certificates, managing the trust repository, and monitoring system operations. These processes ensure that every issued certificate is legitimate and that compromised or expired certificates can be revoked efficiently. The diagram emphasizes the CA Owner's central role in sustaining the integrity of the entire certification chain.

D. Global System Use Case Diagram - Full View

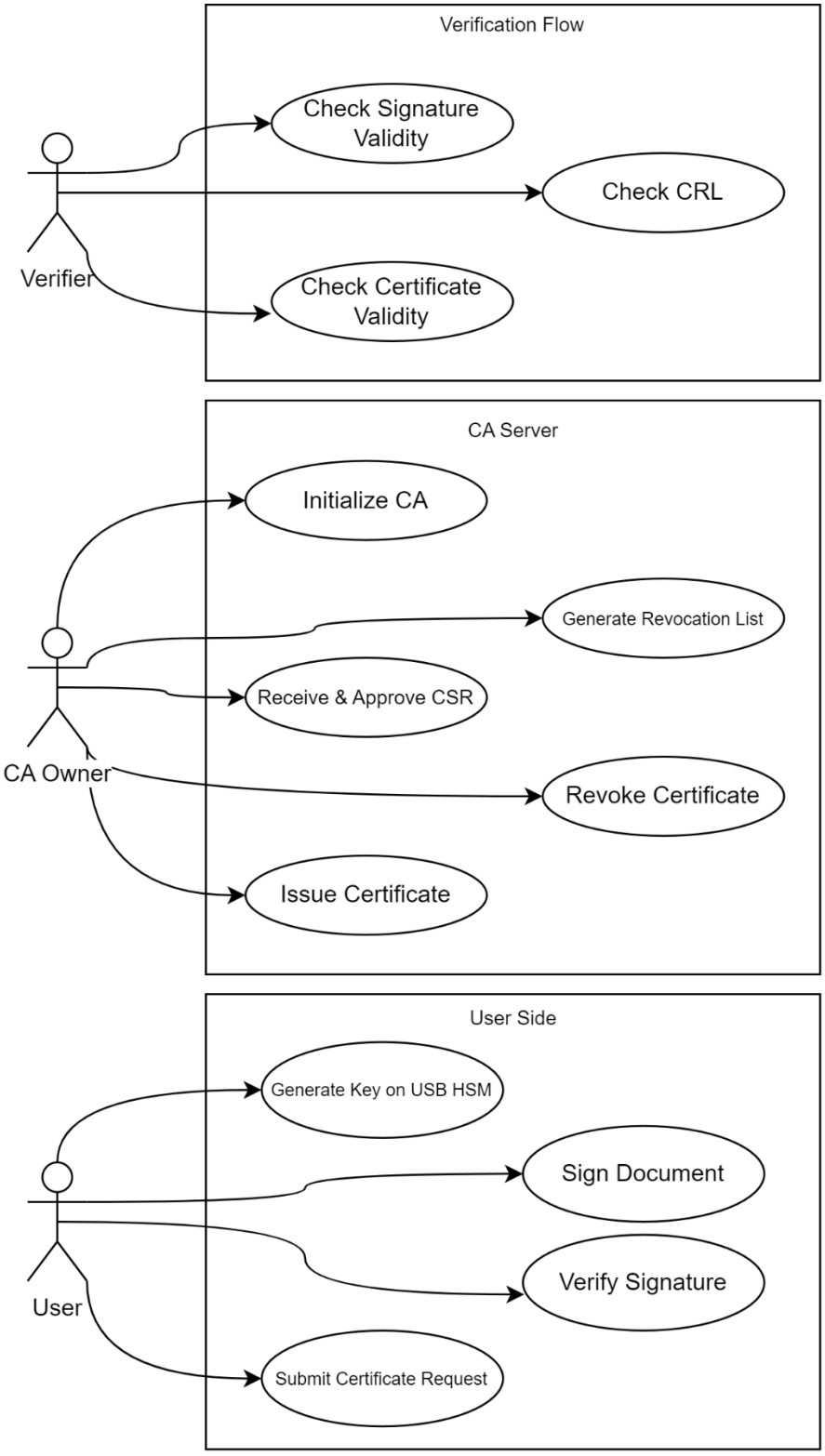


Figure 10: Global System Use Case Diagram

Figure 10 combines all actors and interactions into a unified overview of the system. It shows how the user, CA Owner, and verifier collaborate through defined exchanges (registration, certificate issuance, signing, and validation) to maintain document authenticity and trust. The diagram demonstrates the

complete operational flow and highlights the boundaries between user-side, authority-side, and verification processes, providing a holistic view of the system's functional ecosystem.

4. Functional Architecture

To ensure the system remains secure, modular, and maintainable, a layered architecture is used. Each layer has a clear responsibility and communicates only with adjacent layers. This separation keeps the design flexible and allows each part to evolve independently without affecting the others.

The system is composed of three core layers:

→ **Application Layer** handles user interaction, prepares documents for signing, and coordinates certificate requests and verification steps.
→ **Cryptographic Layer** performs sensitive operations such as key generation, secure key storage, and digital signatures inside a protected environment.
→ **Trust and Identity Layer** provides identity validation, certificate issuance and revocation, and maintains the local trust store for verification.

A. Layered Functional Architecture of the System

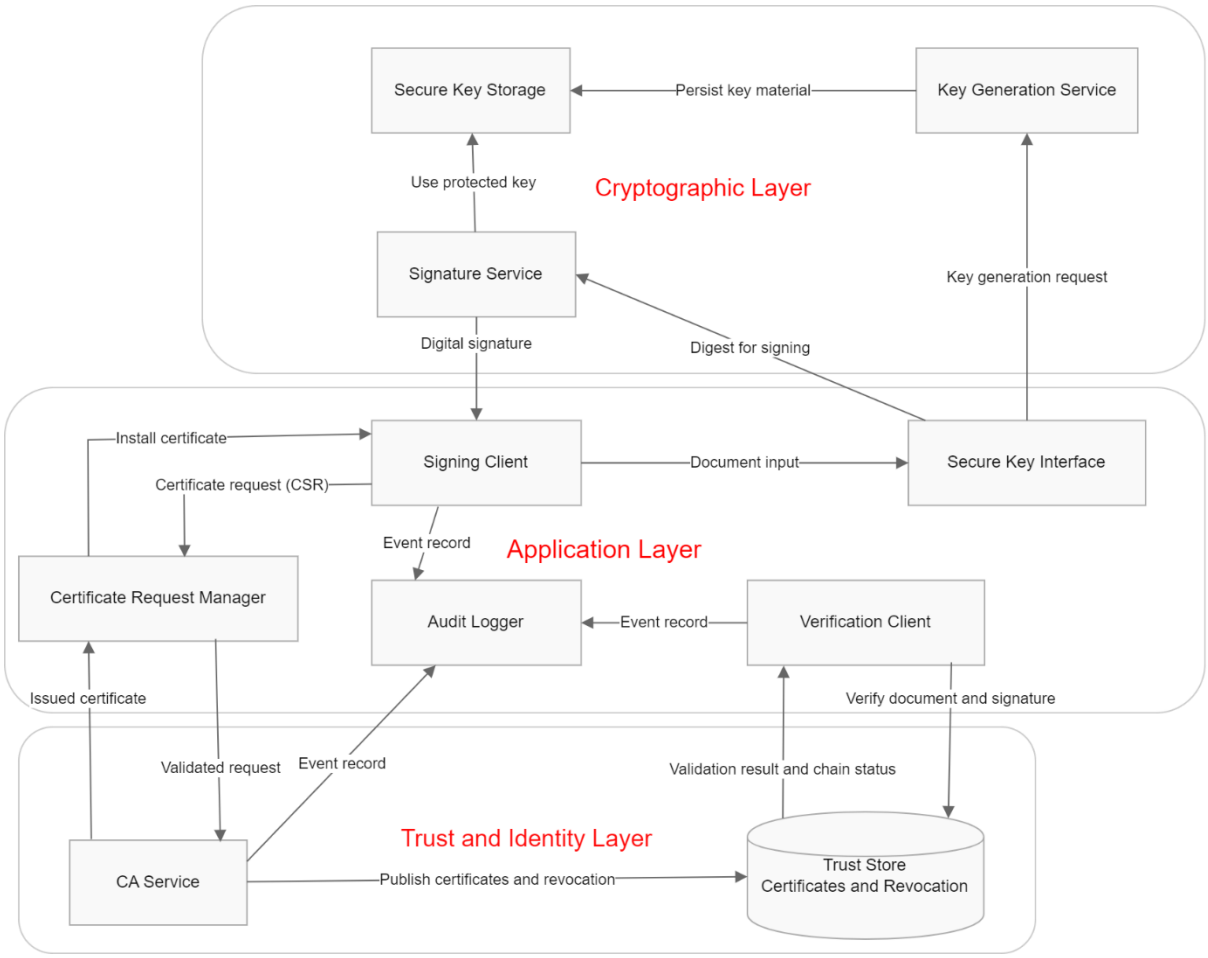


Figure 11: Functional Layered Architecture of the System

Figure 11 shows the three-layer structure and the main responsibilities of each layer. The Application Layer manages user actions and orchestrates certificate requests and verification. The Cryptographic Layer isolates key generation, key storage, and signature operations inside a protected environment.

The Trust and Identity Layer manages the certification authority and the trust store used for offline validation. Interactions are limited to adjacent layers to preserve isolation and clarity.

B. Functional Responsibilities per Component

Table 7 presents the logical distribution of responsibilities among the system's main components, organized by layer. Each layer focuses on a specific aspect of the system's operation, ensuring modularity and clear separation of concerns. The Application Layer manages user interaction and high-level coordination, the Cryptographic Layer secures all key-related operations, and the Trust & Identity Layer governs certificate management and trust validation. The mapping below clarifies how these parts collectively support the secure and offline document-signing workflow.

Table 7: Functional Responsibilities per Component

LAYER	COMPONENT	RESPONSIBILITY
APPLICATION LAYER	Signing Client	Prepares document input, coordinates signing, embeds the resulting signature in the document.
	Secure Key Interface	Sends signing requests to the protected environment and receives signature outputs.
	Certificate Request Manager	Builds and submits certificate requests to the CA and updates local certificate state.
	Verification Client	Validates signed documents against the trust store and reports results clearly to the user.
	Audit Logger	Records key user and system events for traceability.
CRYPTOGRAPHIC LAYER	Key Generation Service	Generates user key pairs inside the protected environment.
	Signature Service	Performs digital signatures over prepared digests without exposing private keys.
	Secure Key Storage	Keeps private keys in a protected store and prevents export.
TRUST & IDENTITY LAYER	CA Service	Verifies identities, issues certificates, and manages revocation.
	Trust Store Manager	Publishes certificates and revocation data for local, offline verification.

C. Key Design Benefits

The system's functional architecture was designed to achieve simplicity, clarity, and long-term reliability. Each layer operates independently but cooperates through well-defined interfaces, ensuring that every component can evolve without compromising the system's integrity. The main benefits of this design can be summarized as follows:

- → **Security through Isolation**: Sensitive operations such as key generation and digital signing are handled exclusively within a protected environment, preventing any exposure of private keys or critical data.
- → **Local Trust and Autonomy**: The certification authority and trust store operate locally, allowing institutions to maintain full control over identity management, certificate issuance, and revocation without external dependence.
- → **Offline Functionality**: All core operations (from signing to verification) can be performed without internet access, which enhances reliability in constrained or high-security environments.
- → **Modularity and Maintainability**: The clear separation of layers simplifies updates, debugging, and future improvements, as changes in one component do not affect the others.
- → **Transparency and Traceability**: Every critical action is recorded in an audit log, ensuring accountability and enabling security reviews when needed.
- → **Scalability for Institutional Use**: Although lightweight by design, the system can be extended to accommodate additional users or integrated with broader institutional infrastructures without altering its core structure.

5. Component Breakdown and Responsibilities

This section presents an overview of the system's main components, describing the responsibilities of each within the global architecture. The system is composed of three primary applications: a Signer Application, a Certification Authority Interface, and a Verification Application. Each of these is further divided into logical modules that operate collaboratively to ensure the integrity, security, and traceability of operations.

The goal of this modular breakdown is to provide a clear understanding of how responsibilities are distributed and how components interact, while keeping the design maintainable, secure, and adaptable to institutional use cases.

Below is a conceptual diagram showing the core components grouped by application and responsibility. It highlights the key modules of each system and the logical flows between them.

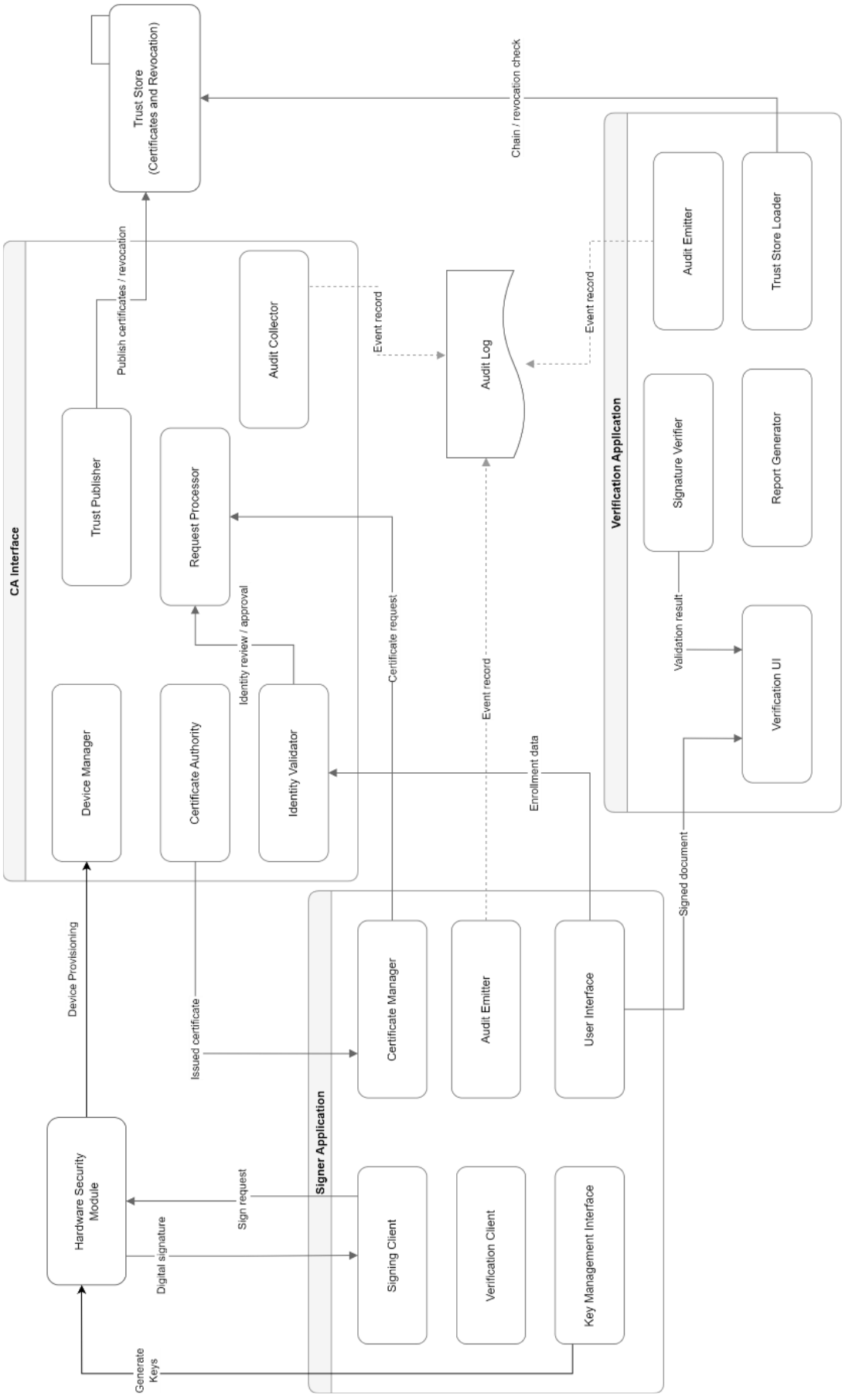


Figure 12: Logical Breakdown of System Components

Figure 12 organizes the system into three applications (Signer Application, CA Interface, and Verification Application) plus the Hardware Signing Module and Trust Store. It shows each component's role and the labelled core flows: enrolment, certificate issuance, protected signing, offline verification, revocation publication, and audit logging.

Table 8: Component-Level Responsibilities

APPLICATION	COMPONENT	RESPONSIBILITY
SIGNER APPLICATION	User Interface	Guides users through signing, verification, and certificate management.
	Key Management Interface	Handles secure interaction with key material during signing and setup.
	Certificate Manager	Initiates certificate requests and tracks issuance status.
	Signing Client	Coordinates document preparation and signature embedding.
	Audit Emitter	Records key user actions for traceability.
CA INTERFACE	Identity Validator	Manages identity confirmation and enrolment approval.
	Request Processor	Validates requests and enforces policy checks.
	Certificate Authority	Issues, renews, and revokes certificates.
	Trust Publisher	Publishes valid certificates and revocation data to the trust store.
	Device Manager	Oversees binding of secure devices to user identities.
	Audit Collector	Aggregates operational events for oversight.
VERIFICATION APPLICATION	Verification UI	Presents results and guidance to the verifier.
	Signature Verifier	Validates document integrity and signature authenticity.
	Trust Store Loader	Synchronizes local trust data and revocation lists.
	Report Generator	Produces a clear validation report for records.
	Audit Emitter	Records verification events for traceability.

Table 8 groups the system's components by application and summarizes each unit's role and responsibilities.

6. Data Flow and Communication

In a system built to ensure the authenticity and integrity of signed documents, it is critical to understand how data flows between each of the system's components. This part explains how

information moves between the core components of the system inside a local trust domain. It introduces the main exchanges that enable enrolment and certificate issuance, protected signing, and offline verification, while keeping sensitive material confined to the Hardware Signing Module. The emphasis is on clear boundaries and minimal exposure: private keys never leave the protected hardware, certificate control remains local, and data exchanges use a local link under institutional control.

We begin with a level-1 data-flow diagram that shows the components and labelled flows: digest for signature, signature output, certificate request, certificate, revocation publication, verification query, and audit record. The following sequence diagrams then detail three core operations: document signing, certificate request and issuance, and signature verification.

A. Level 1 Data Flow Diagram - Digital Signing System

This level-1 diagram presents a high-level view of how information moves inside the digital signing system. It focuses on the meaning of each exchange between the Signer Application, the Hardware Signing Module, the CA Owner Application, the Trust Store, and the Verifier Application. The flows highlight certificate request and issuance, protected signing where private keys remain inside the hardware, publication of trust data, and offline verification under local institutional control.

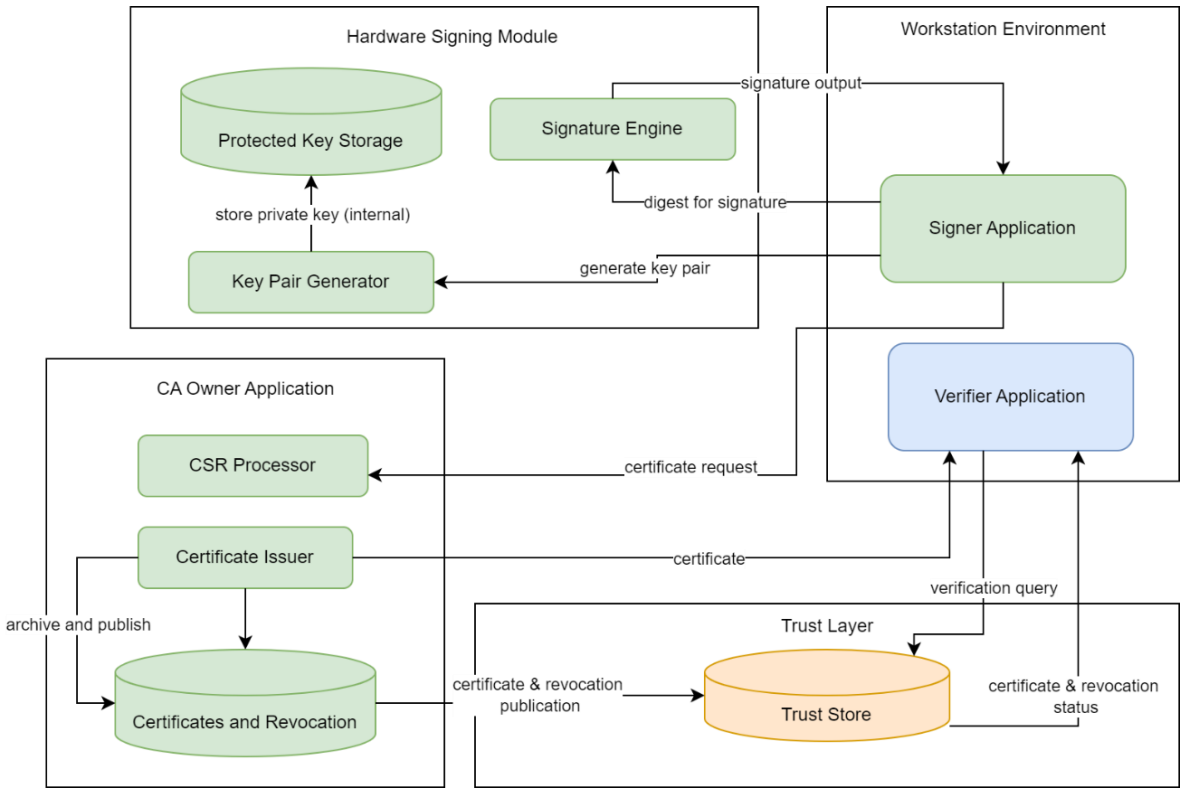


Figure 13: Level 1 Data Flow Diagram – Digital Signing System

Figure 13 outlines the core components and their data exchanges within a local trust domain. The Signer Application prepares a document digest and requests a signature from the Hardware Signing Module, which performs the operation internally and returns only the signature output. For enrolment, the Signer Application submits a certificate request to the CA Owner Application, which approves issuance and returns the certificate while also publishing trust data to the Trust Store. The Verifier Application later validates signed documents by consulting the Trust Store, retrieving certificate information and revocation status without relying on external services. Flows are labelled to indicate intent (*digest for signature*, *signature output*, *certificate request*, *certificate*, *revocation publication*, *and verification*

queries) while the trust boundaries keep sensitive material isolated inside the Hardware Signing Module and the authority functions within the institutional domain.

B. Sequence Diagram 1 - Document Signing with Hardware Signing Module

This sequence describes how a document is signed within the local trust domain. The Signer Application prepares a digest of the content, requests a protected signature from the Hardware Signing Module, then assembles the final signed artefact. Sensitive material never leaves the hardware; only the signature output is returned to the application. The flow is intentionally minimal and offline, with all exchanges occurring over a local link under institutional control.

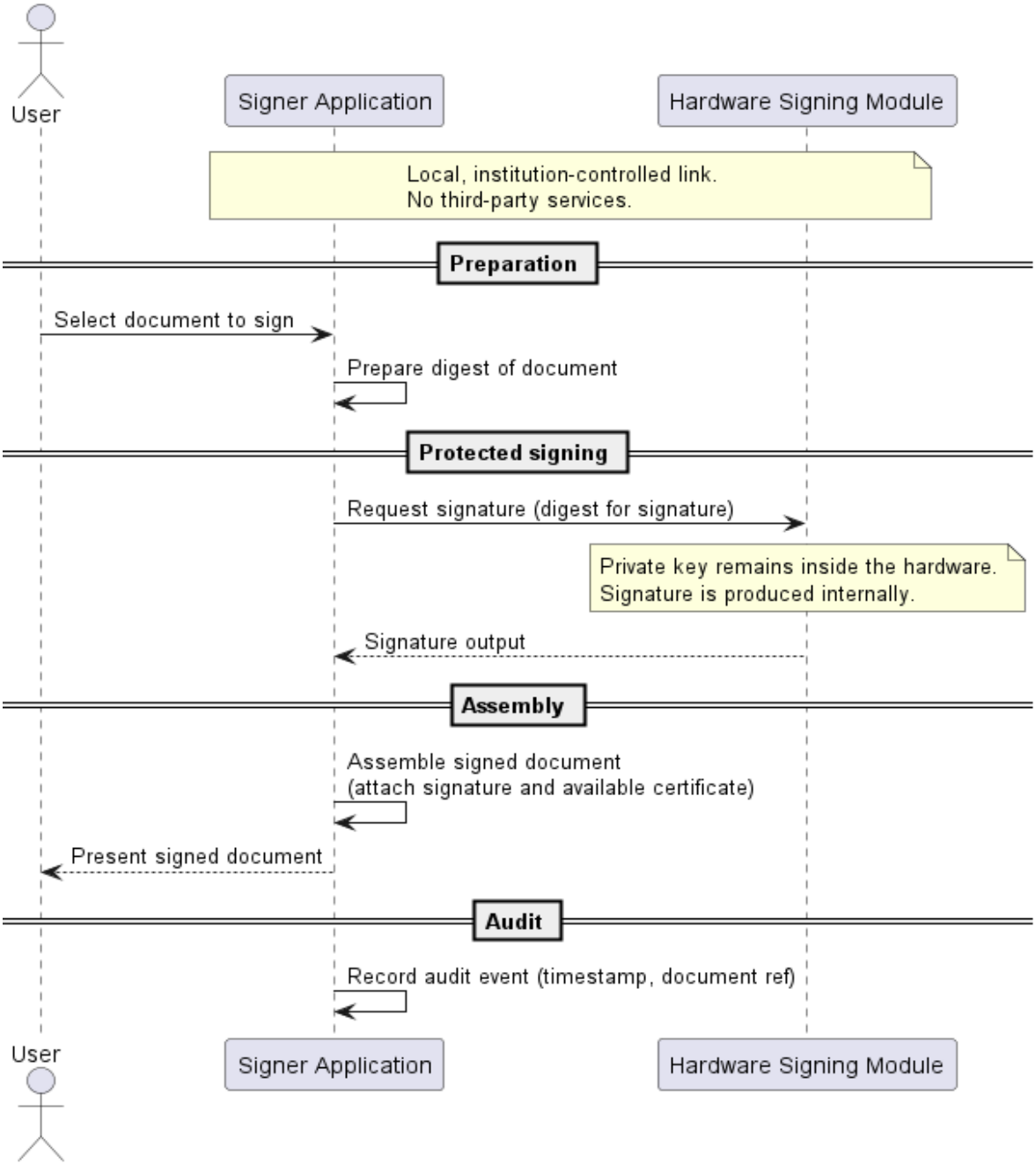


Figure 14: Sequence Diagram - Document Signing with Hardware Signing Module

Figure 14 shows the end-to-end signing operation within a local environment. The Signer Application first prepares a digest of the document, then requests a protected signature from the Hardware Signing Module. The module performs the cryptographic operation internally and returns only the signature output, ensuring private keys remain isolated. The Signer Application assembles the final signed document by embedding the returned signature and any available certificate information required for later verification. The process is offline, uses a local link under institutional control, and records a minimal audit event to support traceability without exposing sensitive material.

C. Sequence Diagram 2 - Certificate Request and Issuance

This sequence explains how an identity within the local environment obtains a certificate. The Signer Application coordinates with the Hardware Signing Module to ensure a key pair exists, constructs a certificate request, and submits it to the CA Owner Application for review and issuance. The CA validates the request according to local policy, issues the certificate upon approval, and publishes trust data to the Trust Store. The process remains offline and institution-controlled, and private keys never leave the Hardware Signing Module.

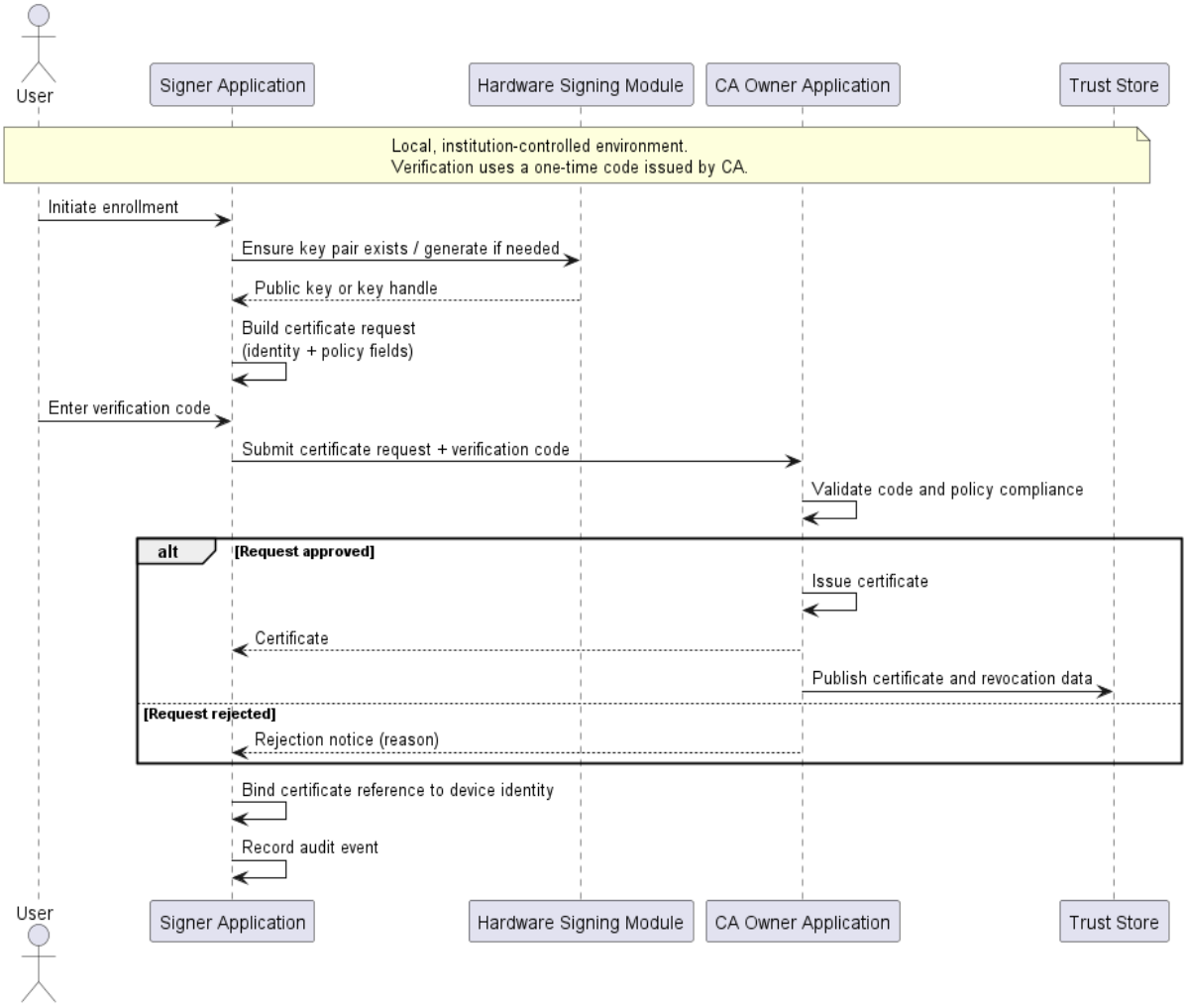


Figure 15: Sequence Diagram - Certificate Request and Issuance

Figure 15 shows how a certificate is obtained within the local trust domain. The Signer Application first ensures a key pair is present inside the Hardware Signing Module and retrieves the corresponding public key or handle. It then constructs a certificate request containing identity attributes and policy-relevant fields, and submits it to the CA Owner Application. If approved, the CA issues the certificate,

returns it to the Signer Application, and publishes the certificate and revocation information to the Trust Store for later verification. The Signer Application binds the issued certificate to the device identity and records a minimal audit trail. Throughout the process, private keys remain inside the Hardware Signing Module and the entire flow operates offline under institutional control.

D. Sequence Diagram 3 - Signature and Certificate Verification

This sequence shows how a signed document is validated in an offline, institution-controlled environment. The Verifier Application extracts the embedded signature and certificate information, consults the Trust Store for the relevant certificate data and revocation status, and then validates the certificate chain and the signature over the document digest.

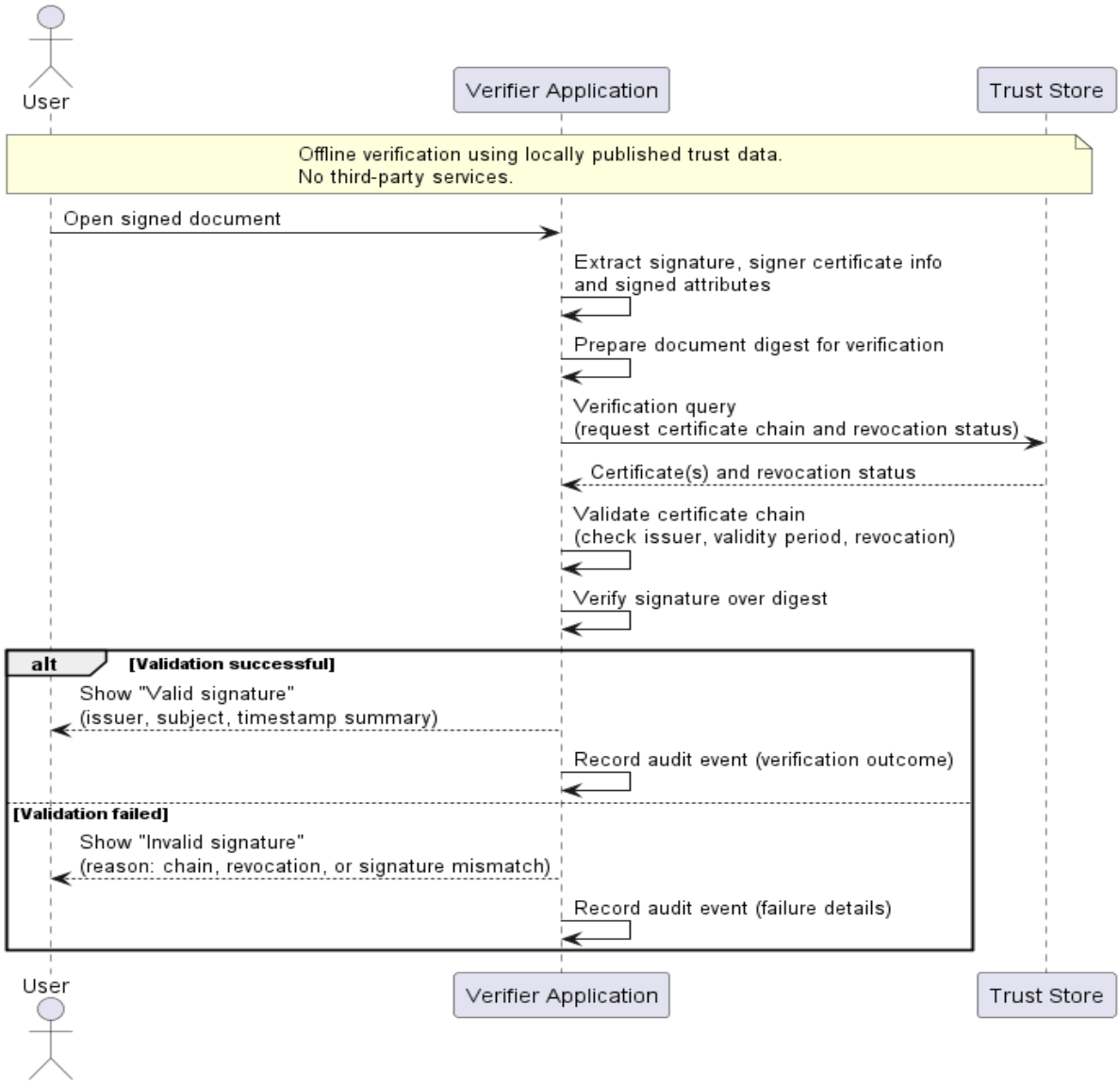


Figure 16: Sequence Diagram - Signature and Certificate Verification

Figure 16 details the offline verification workflow. The Verifier Application opens the signed document, extracts the signature and the signer's certificate references, and prepares the document digest for checking. It queries the Trust Store to obtain the necessary certificate chain and the current revocation status. With this local information, the Verifier validates the chain against the trust anchors and validity constraints, then verifies the signature over the prepared digest. If all checks pass, the result

is reported as a valid signature with a concise summary; otherwise the user is shown a clear failure reason. An audit record of the verification outcome is stored locally. All steps operate within the institutional trust domain without external dependencies.

E. Sequence Diagram 4 - Command Exchange (Signer App & Hardware Signing Module)

This part captures the minimal, offline command–response flow between the Signer Application and the Hardware Signing Module. It shows only what's essential: start a session, identify the module, ensure a key pair exists, request a protected signature over a prepared digest, optionally fetch the public key, and close the session. Private keys never leave the module.

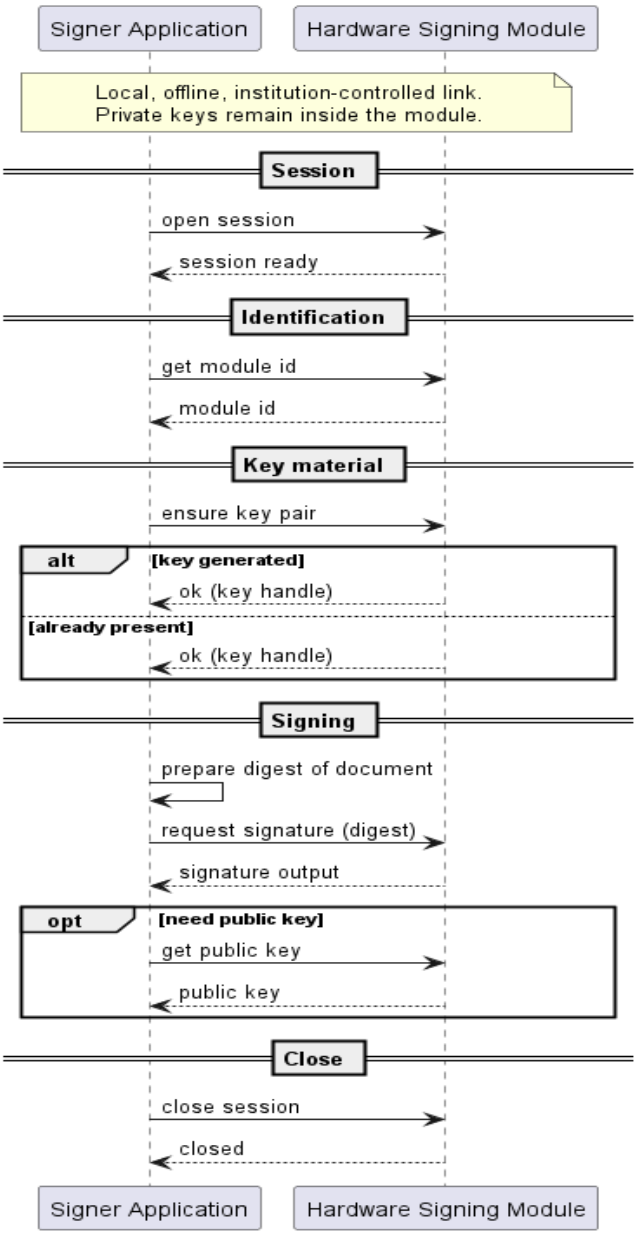


Figure 17: Sequence Diagram - Command Exchange (Signer App & Hardware Signing Module)

Figure 17 focuses on the essential local interaction between the Signer Application and the Hardware Signing Module. The application opens a session, retrieves a stable module identifier, and ensures that a key pair exists inside the module. For signing, the application prepares a digest of the document and

sends a request; the module performs the operation internally and returns only the signature output. When needed, the application can fetch the corresponding public key for enrolment or verification contexts. The session is then closed. The flow is offline, under institutional control, and keeps private keys confined to the Hardware Signing Module.

F. Key Data Flows

This part summarizes the essential exchanges that enable protected signing, local issuance, and offline verification. Each row states the flow, the endpoints, what is actually exchanged, why it is exchanged, and the single most important security note for that flow.

Table 9: Key Data Flows Across Components

FLOW	SOURCE → DESTINATION	EXCHANGED DATA	PURPOSE	SECURITY NOTE
DIGEST FOR SIGNATURE	Signer Application → HSM	Document digest; signing parameters	Request a protected signature over prepared content	Integrity must be preserved; private key remains inside hardware.
SIGNATURE OUTPUT	HSM → Signer Application	Signature blob; status	Return the protected signature result	Only the signature leaves the module; no private key exposure.
CERTIFICATE REQUEST	Signer Application → CA Owner Application	Identity attributes; public key or handle; policy fields	Obtain a certificate for local identity	Request must be authentic and policy-compliant.
CERTIFICATE	CA Owner Application → Signer Application	End-entity certificate; issuance metadata	Provide the issued certificate for future validation	Chainable to local trust anchors; returned only after approval.
REVOCATION PUBLICATION	CA Owner Application → Trust Store	Revocation entries; updated trust data	Make status information available for offline checks	Must be tamper-evident and timely within the local domain.
VERIFICATION QUERY	Verifier Application → Trust Store	Certificate chain references; status query	Retrieve materials needed to validate a signature	Responses must be authentic and reflect current local status.
AUDIT RECORD	Signer/Verifier Application → Local audit log	Timestamp; operation; reference; outcome	Maintain minimal traceability	Avoid sensitive payloads; integrity of records is required.

Table 9 captures the minimal flows required for secure operation in an offline, institution-controlled environment. The Signer Application sends only a digest to the Hardware Signing Module and receives a signature output, preserving private key isolation. Enrolment proceeds through a certificate request to the CA Owner Application, which returns the certificate and publishes revocation information to the Trust Store. During validation, the Verifier Application queries the Trust Store for certificate and status data. Each flow highlights a single security priority to keep the model precise and technology-neutral.

G. Security Note

The system operates entirely within an institution-controlled environment and enforces strict key isolation. Private keys are generated and stored inside the Hardware Signing Module and never leave it, applications exchange only non-sensitive inputs (digests, identity fields) and receive signature outputs. Certificate issuance and revocation are handled locally by the CA Owner Application, which publishes trust data to a local Trust Store to enable offline verification. Communication paths are minimal and purpose-bound, reducing exposure and simplifying validation. Integrity of all exchanged data and audit records is a priority, confidentiality is applied where identity or issuance details are present. Trust boundaries are explicit: hardware domain for key material, authority domain for issuance and status, and workstation domain for application logic.

7. Certificate Lifecycle and Trust Chain

This section explains how trust is established and maintained within the system. A certificate binds a public key to an identified signer so that verifiers can confirm who produced a given signature. The lifecycle covers how certificates are requested, issued, used in signing, and later validated or revoked when necessary. All operations occur inside an institution-controlled environment: keys are generated and kept within the Hardware Signing Module, issuance and revocation are governed by the CA Owner Application, and verifiers rely on a locally published Trust Store for offline checks. The trust chain begins at the locally managed root, extends through issued certificates, and is evaluated by the Verifier Application without depending on external services.

A. Trust Chain and Certificate Lifecycle Diagram

This part summarizes how a certificate is created, trusted, used, and retired within the local domain. Keys are generated inside the Hardware Signing Module, the CA Owner Application issues certificates, the Trust Store publishes trust data, and verifiers rely on that local publication for offline checks.

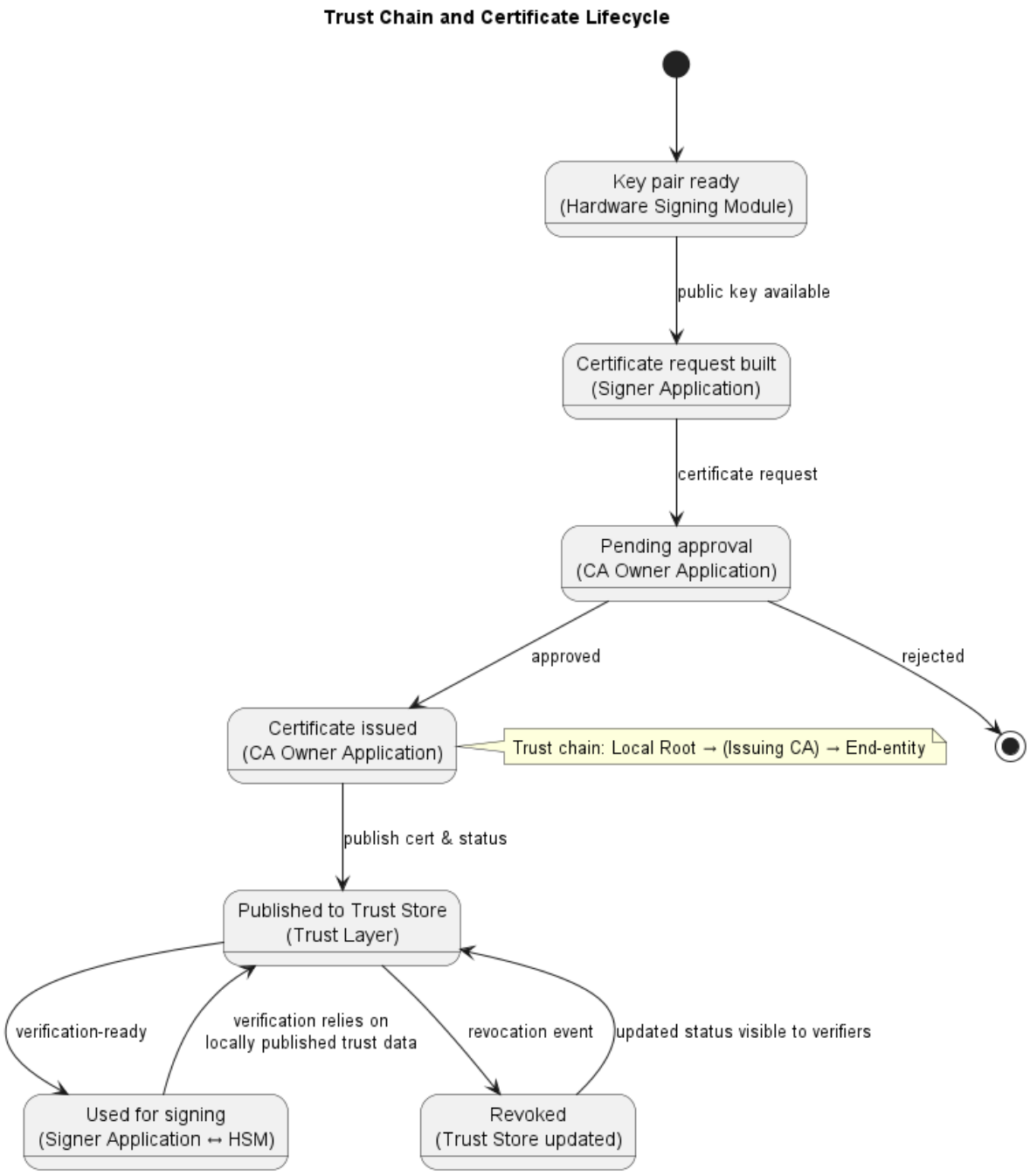


Figure 18: Certificate Lifecycle and Trust Chain

Figure 18 shows the local lifecycle: keys are prepared inside the Hardware Signing Module; the Signer Application builds a request that the CA Owner Application validates and, if approved, issues as a certificate. The certificate and status are published to the Trust Store, enabling offline verification. During use, signatures reference this published trust data. If a revocation occurs, status is updated in the Trust Store and verifiers observe the change without external dependencies.

B. Certificate Lifecycle Stages

This section lists the certificate lifecycle stages used in the system model. Each stage names the actor(s) involved and the essential goal, while respecting the rule that private keys remain inside the Hardware Signing Module.

1. **Key generation**: Hardware Signing Module generates and protects the private key; a public key or key handle becomes available to the Signer Application.
2. **Enrolment / CSR construction**: Signer Application creates a certificate request containing identity attributes and the public key or handle.
3. **Submission**: Signer Application submits the certificate request to the CA Owner Application together with the verification code issued for that enrolment.
4. **Validation**: CA Owner Application validates the verification code and checks the request against local policy and identity records; approved requests proceed to issuance.
5. **Issuance**: CA Owner Application issues the end-entity certificate and returns it to the Signer Application.
6. **Publication**: CA Owner Application publishes the certificate and revocation metadata to the Trust Store for offline verifier access.
7. **Binding and use**: Signer Application binds the certificate reference to the device identity and uses the Hardware Signing Module to produce signatures.
8. **Verification**: Verifier Application consults the Trust Store for certificate chain and revocation status, then validates signatures locally.
9. **Renewal / Revocation**: Certificates are renewed or revoked by the CA Owner Application; revocation updates are published to the Trust Store and consumed by verifiers.
10. **Audit & retention**: Signer and Verifier Applications record minimal audit entries (operation, timestamp, outcome) for traceability; sensitive payloads are excluded.

C. Trust Anchors and Chain Validation

Trust in this system starts from anchors managed by the institution. The root (and any intermediate issuers, if used) is controlled by the CA Owner Application and published to the local Trust Store. Verifier Applications rely only on this locally published set; there is no dependence on external directories or online status services. End-entity certificates bind a public key to an identified signer, while the private key remains inside the Hardware Signing Module. During verification, the Verifier Application reconstructs a path from the signer's certificate to a local trust anchor using only data available in the Trust Store.

Chain validation checks are straightforward and offline. The Verifier confirms that each certificate in the path is correctly signed by the next authority, observes validity periods, and enforces policy constraints exposed in the certificates. It then consults the Trust Store for the current local revocation state and applies it to the path. If the chain is intact, within time, not revoked, and consistent with the intended use of the certificate, the signature over the document digest is verified with the signer's public key. All decisions are made within the local trust domain, using only data under institutional control.

8. Security Principles and Design Choices

This system keeps private keys inside dedicated hardware, governs certificates locally, and lets verifiers make offline decisions using a published trust set. Boundaries are explicit, flows are minimal, and anything outside institutional control sits in a clearly labelled non-secure zone.

A. Threat Model

The model assumes an institution-controlled environment with four main domains: a **Hardware Signing Module** that holds private keys and performs signing, a **CA Owner Application** domain that controls issuance and revocation, **Workstation apps** for signing and verification, and a **Trust Publication layer** that exposes anchors and revocation for offline use. Actors are bound to

domains: Signer, CA Owner, Verifier. Enrolment is gated by a one-time verification code and signatures are produced only over digests; private keys never leave the hardware.

Table 10: Core threats and design controls

Threat	**Asset**	**Boundary**	**Likely vector**	**Control in design**	**Residual risk / note**
Workstation compromise	Private signing key	Workstation ↔ Hardware module	Malware attempts key extraction	Key isolation in hardware; digest-only requests	Physical/firm-ware hardening of device still matters.
Tampering with inputs/outputs	Digest or signature	Workstation ↔ Hardware module	Intercept/alter digest or returned signature	Minimal labelled flows; signature verified downstream	Enforce strict input checks and audit.
Unauthorized enrolment	Issuance	Workstation ↔ CA domain	Register device/user without approval	One-time verification code + policy checks before issuance	Add lifetime/ attempt limits for the code.
Stale or forged status	Revocation /anchors	CA domain ↔ Trust publication	Old or spoofed status seen by verifiers	Local publication used by verifiers; snapshot required before verify	Prefer signed status artefacts to strengthen integrity.
Post-signature alteration	Signed document	Non-secure zone	Content edited in transit/storage	Offline verification against local trust set	Always treat signed files as untrusted until verified.
Operator misuse	CA issuance	Inside CA domain	Accidental or out-of-policy issuance	Manual approval + logging/audit trail	Policy constraints should be explicit and reviewed.

Table 10 explains how the core threats map to assets, boundaries, and the specific controls enforced by this design. It shows that attempts to extract private keys from the workstation are neutralized by hardware key isolation and digest-only calls; tampering with inputs or returned signatures is constrained by minimal, labelled flows and downstream verification; unauthorized enrolment is blocked by the verification-code gate and CA policy checks; stale or forged status is addressed through local publication of anchors and revocation in the Trust Store; post-signature alteration of documents is handled by offline validation within the Verifier Domain; and operator-side issuance risks are bounded by approval workflows and auditable records. Each row clarifies the asset at risk, the boundary a threat would need to cross, and the control that stops or detects it, making the relationship between the coming diagram's domains and the system's security guarantees explicit and easy to audit.

B. Trust Boundary Diagram

The trust boundary diagram shows security domains, attached actors, trust levels, and what crosses each boundary. Colours express trust level; the Non-Secure Zone holds artefacts outside your control (e.g., a signed file on random media) that are still verifiable against local trust.

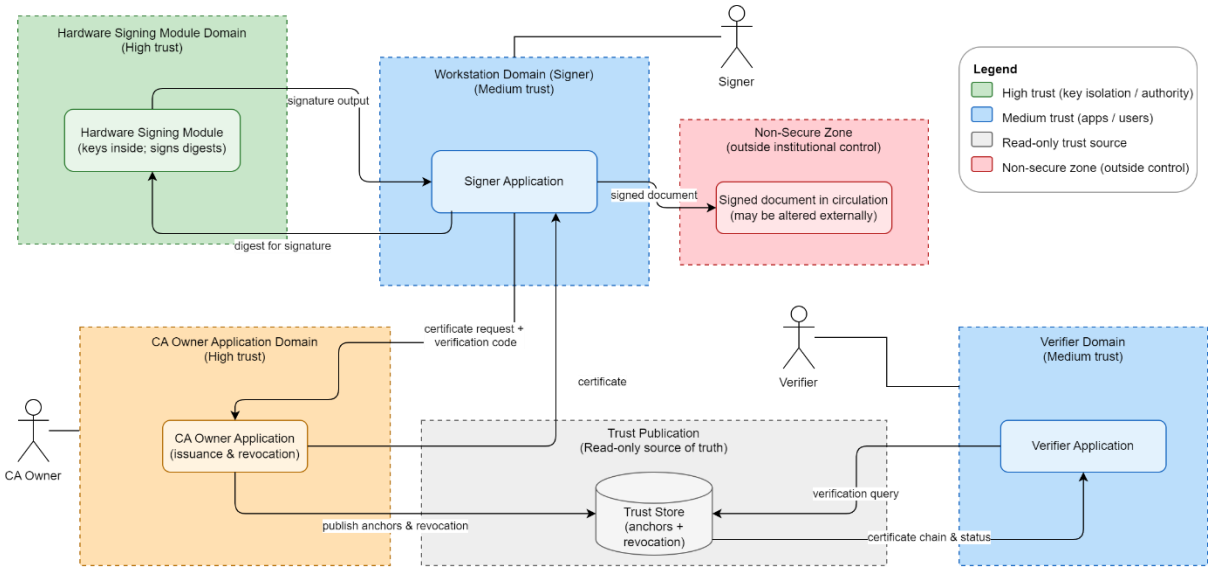


Figure 19: Trust Boundary Diagram

Figure 19 presents the system's trust boundaries as color-coded security domains with attached actors and the minimal, labelled exchanges permitted across them. The Hardware Signing Module Domain (high trust, green) isolates private keys and performs signature operations internally; only digests enter and only signature outputs leave. The CA Owner Application Domain (high trust, amber) governs identity and issuance, validating certificate requests that arrive with a verification code and returning issued certificates while publishing anchors and revocation data to the read-only Trust Publication layer. The Workstation Domain (Signer) (medium trust, blue) hosts the Signer Application and its human actor; it prepares digests, requests protected signatures from the hardware, and initiates enrolment with the CA. The Verifier Domain (medium trust, blue) contains the Verifier Application and actor; it relies solely on locally published trust data to validate artefacts. The Trust Publication layer (grey) exposes anchors and revocation as a read-only source of truth for offline verification. Finally, the Non-Secure Zone (red) represents any location outside institutional control where signed documents may reside or be altered; such artefacts are treated as untrusted until the Verifier validates them against the local trust set. The diagram emphasizes least-privilege exchanges, strict key isolation, and a verification model that depends only on data under institutional control.

C. Security Design Principles

To support long-term trust and robust document integrity, the following principles were used as the foundation for the system's security design:

→ **Private key isolation:** Private key isolation. Key generation and signing happen inside the Hardware Signing Module; applications send digests and receive signatures.

→ **Local certificate control:** The CA Owner Application validates requests (including the verification-code gate), issues certificates, records revocations, and publishes trust data for verifiers.

→ **Offline verification:** Verifiers rely on locally published anchors and status, not external services. Signed content can live anywhere; its trust comes from local validation, not storage location.
→ **Minimal exposure:** Only what's needed crosses boundaries, and each flow is intentionally labelled to keep reviews simple and attack surfaces small.
→ **Traceability without leakage:** Apps log operations and outcomes, not secrets; logs support audits while keeping sensitive material out of band.

This security model provides a strong and self-contained foundation for offline document signing and verification, suitable for sensitive environments that require both autonomy and cryptographic assurance.

9. Conclusion

Chapter 3 defines the system at a technology-neutral level, it introduces the actors and components (Signer Application, Hardware Signing Module, CA Owner Application, Verifier Application, and the Trust Store), explains their responsibilities, and describes how information moves between them through labelled, minimal exchanges. It presents the trust boundaries and data flows that keep private keys isolated in hardware and all decisions within an institution-controlled environment, and the security assumptions and design principles that follow from this model. The result is a coherent conception in which the CA governs identity and lifecycle, the Hardware Signing Module performs protected signing, the Trust Store publishes the local source of truth, and the Verifier validates artefacts from outside the secure domain without depending on external services.

Chapter 4: Implementation

1. Introduction

This chapter presents the practical realisation of the system named CertiFlow. It begins with a system overview that links the delivered build to the design, then sets out the shared foundations across the platform: data formats, interfaces, and security controls. The technology choices and their rationale are summarised before the implemented parts are described in turn: the HSM path from encrypted KeyStore to embedded firmware, the CA Owner application and API, the User application, and the Verifier application. The chapter closes with a concise account of limitations, constraints, and engineering challenges to frame the evaluation that follows.

2. System overview (from design to build)

The implemented system delivers a local, device-bound signing workflow. It consists of four concrete parts: the STM32U5-based hardware security module (HSM) that generates and protects keys; the User application that drives registration and signing; the CA Owner application with a local API that handles activation, approvals and certificate issuance; and the Verifier application that validates signatures using a published trust snapshot. Each part has a narrow role and well-defined interfaces, so the whole path remains auditable and simple to operate.

This section links the design to the running build in two ways. First, the deployment view shows where each component lives and how they communicate: USB CDC for the device link, HTTP/JSON for local services, and files for trust distribution. Second, the execution flow traces a typical journey from device provisioning and approval to document signing and verification. These two figures act as the top-level map for the rest of this chapter.

Table 11: Main CertiFlow components and their core functions

COMPONENT	ROLE IN THE SYSTEM	MAIN INTERACTION
HSM FIRMWARE	Generates, stores, and uses private keys securely on the STM32U5 board	Communicates with the User Application over USB CDC
USER APPLICATION	Handles user registration and document signing	Exchanges commands with the HSM and requests services from the CA API
CA OWNER APPLICATION & API	Issues and manages certificates, activation codes, and trust artefacts	Receives CSRs from users and publishes the root certificate and CRL
VERIFIER APPLICATION	Validates signed documents using the published trust data	Operates offline with local copies of root and revocation lists

Table 11 summarises the essential elements of the implemented CertiFlow platform before presenting their detailed layouts. It outlines each component's main responsibility and the way it interacts with the rest of the system, providing a concise bridge between the textual overview and the diagrams that follow.

A. Deployment view

The deployment view presents the physical and logical arrangement of the implemented CertiFlow components and their communication links. It clarifies where each element of the system operates (on the embedded hardware or within the desktop applications) and how data travels between them. This overview ensures that the following technical sections can be interpreted with a clear

understanding of the boundaries and interactions among the HSM, the applications, and the trust infrastructure.

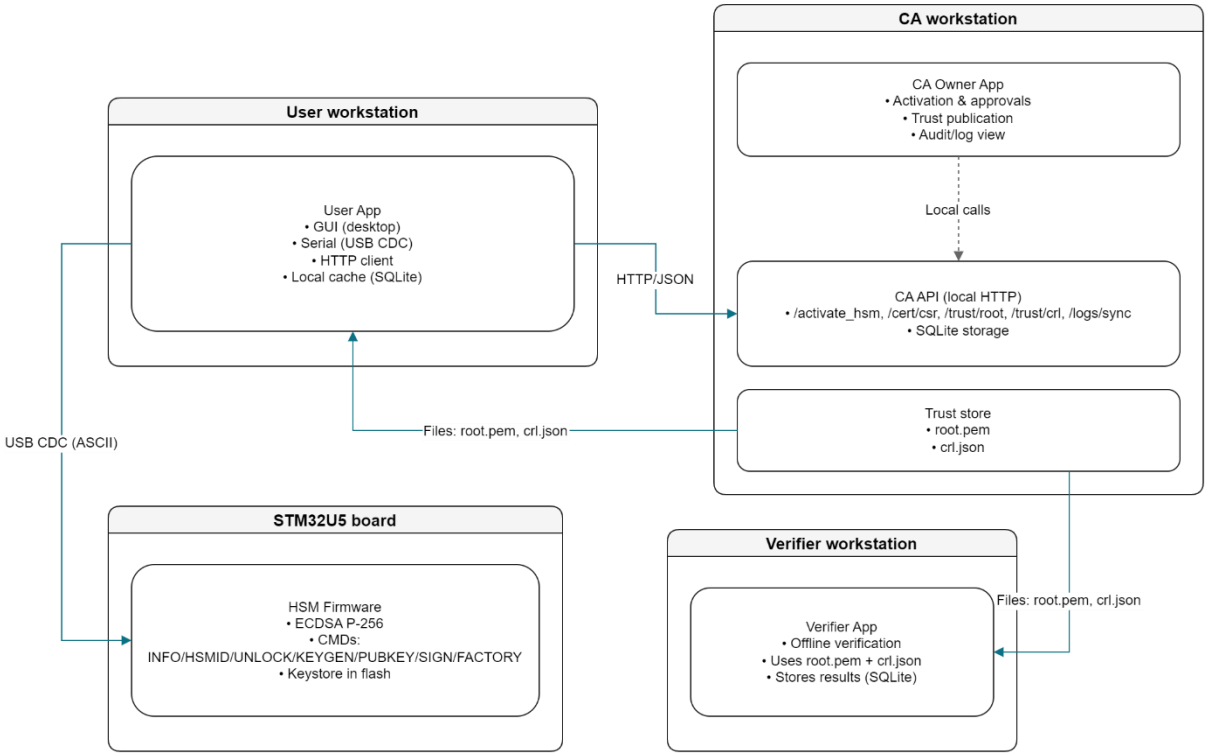


Figure 20: Deployment architecture of the implemented CertiFlow system

Figure 20 illustrates the physical and logical placement of the four main CertiFlow components and their communication links. At the hardware layer, the STM32U5-based HSM connects to the User Application via a USB CDC channel that carries ASCII command–response exchanges. On the CA workstation, the CA Owner desktop interface and the local API cooperate through an internal loopback connection and share a single SQLite database for user and certificate records. The Verifier Application operates independently, relying only on the exported trust artefacts (root certificate and CRL) provided by the CA. The arrows show the principal data paths:

→ **USB CDC** between the User App and HSM for secure command execution.
→ **HTTP/JSON** between User/CA components for registration, activation, and certificate management.
→ **File distribution** for trust artefacts consumed by the Verifier App. This architecture supports offline operation while maintaining clear trust boundaries between device, issuer, and verifier.

B. Execution flow

The execution flow describes the chronological sequence of operations that connect the system's components during normal use. It tracks the movement of information from device provisioning to certificate issuance, document signing, and final verification. This view highlights the cooperation between hardware and software, showing how trust is established, maintained, and validated throughout the process.

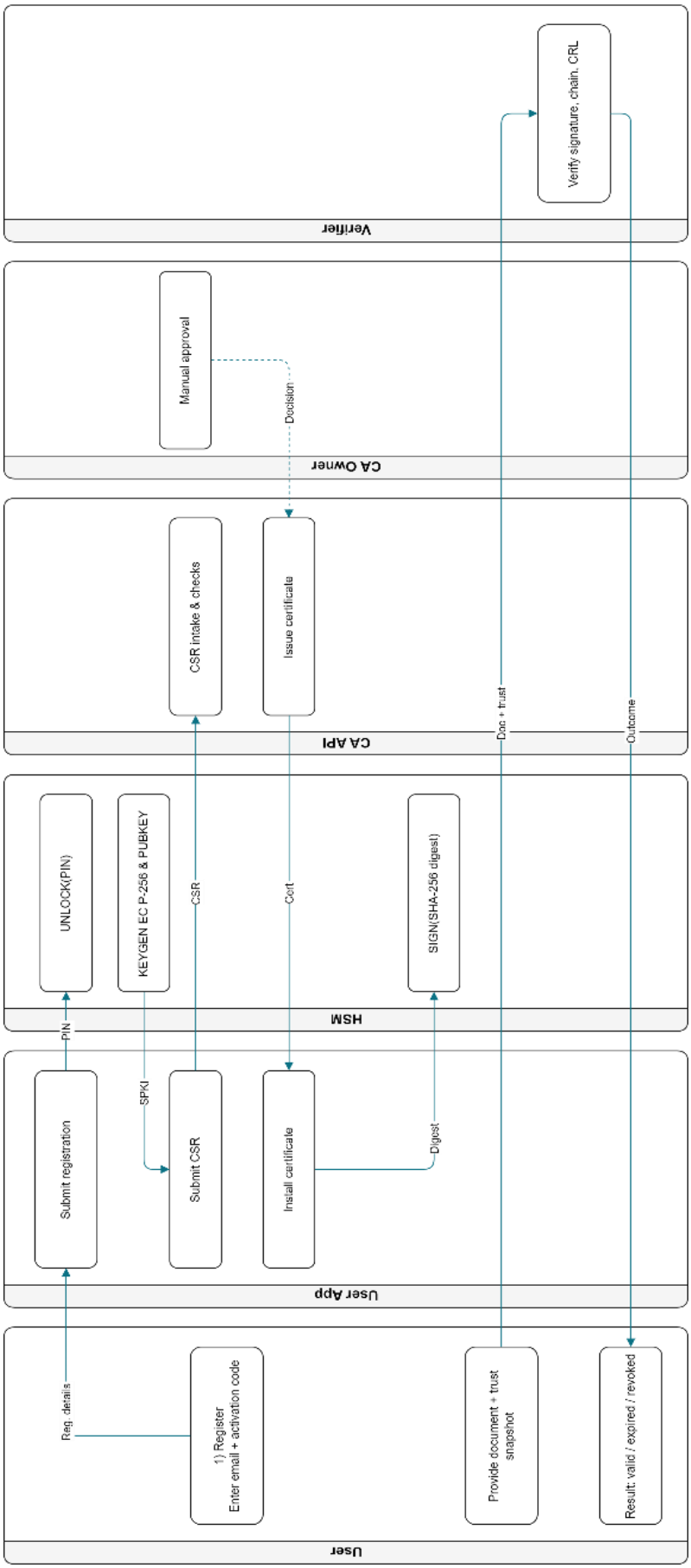


Figure 21: End-to-end operational flow of CertiFlow

Figure 21 summarises the chronological interactions among the system's actors during provisioning, certification, signing, and verification. It begins with the device binding phase, where the CA Owner issues an activation code and associates the HSM's unique ID with a user account. The registration phase follows, where the User App validates the activation code, unlocks the device with a PIN, and generates a key pair directly on the HSM. In the certificate issuance phase, the public key is wrapped in a CSR and sent to the CA API, the CA Owner reviews the request and, once approved, a signed X.509 certificate is issued back to the user. The signing phase then enables document signing through the HSM, which performs hashing and ECDSA operations internally. Finally, the verification phase is executed by the Verifier Application, which checks signature validity, certificate trust chain, and revocation status using the locally cached trust snapshot. This flow enforces separation of duties (generation and signing on the HSM, validation through trusted artefacts) and ensures the integrity of documents even when disconnected from the CA network.

3. Technology choices and rationale

The implementation of CertiFlow relied on a series of technical choices that balanced practicality, security, and maintainability. This section explains and justifies those decisions, starting with the selected hardware platforms and continuing through the development software, libraries, and supporting tools. Each subsection highlights why the component or technology was adopted, what advantages it offers to the system, and how it supports the overall objectives of confidentiality, integrity, and operational simplicity.

A. Hardware platforms

i. STM32U585CIU6 (WeAct Studio Black Pill)

The core of the CertiFlow hardware is the STM32U585CIU6 microcontroller board manufactured by WeAct Studio. This device belongs to the STM32U5 series from STMicroelectronics, which integrates an Arm Cortex-M33 CPU running up to 160 MHz, 2 MB of Flash, and 786 kB of SRAM (20). It provides a built-in USB Full Speed peripheral, hardware PKA and AES engines, a true RNG, and support for Arm TrustZone technology (21). These features make it particularly suitable for implementing an embedded Hardware Security Module because cryptographic operations can be off-loaded to dedicated hardware and keys can remain in protected memory regions (22).

The MCU's security capabilities (read-out protection levels, memory-protection unit (MPU), flash isolation, and secure boot provisions) were decisive factors in its selection (23). Even though the prototype disables TrustZone to simplify firmware integration, the same chip supports trusted execution partitions for future versions (24). The U5 series is also known for its ultra-low-power operation, which allows deterministic performance while maintaining energy efficiency (25). Compared with alternative development boards, it offered the best ratio between cost, security functionality, and toolchain maturity.

The board's USB CDC capability eliminates the need for proprietary drivers, which simplifies the connection with the host applications. Its integrated debug interface (ST-LINK V3) enables in-circuit flashing and real-time monitoring, which improves reliability during testing. The choice therefore aligns perfectly with CertiFlow's design constraints: a self-contained, auditable, and easily reproducible security device.

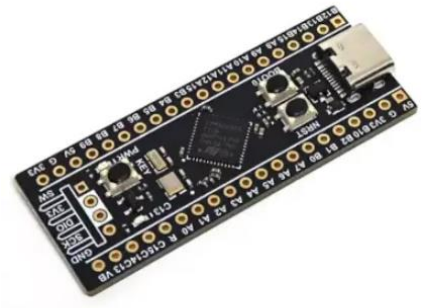


Figure 22: STM32U585CIU6 WeAct Studio Black Pill Board

ii. Raspberry Pi for CA Owner and API hosting

For the CA Owner application and the associated Flask-based API, a Raspberry Pi device is proposed as an optional dedicated host. The Python-based implementation of the CA stack runs smoothly on ARM Linux distributions, and all required libraries are fully supported on modern Raspberry Pi models (26). The Raspberry Pi 5 in particular offers a quad-core Cortex-A76 CPU at 2.4 GHz, 8 GB RAM options, and native Gigabit Ethernet (27). This performance comfortably exceeds the needs of a small Certificate Authority service, which mainly performs light database operations and cryptographic signing.

Figure 23: Raspberry Pi 5/4GB

The platform's compact form factor and low energy consumption make it an ideal candidate for a physically protected "CA appliance." The device can be enclosed in a tamper-evident case, placed in a secure location, and powered continuously. Logical hardening is also feasible through restricted SSH access, firewall rules, and filesystem read-only modes. In addition, the Flask framework's modest resource requirements have been demonstrated on Raspberry Pi platforms in numerous small-scale deployments (28). Using such a board therefore offers a realistic and inexpensive path toward operational deployment while maintaining the security properties of the CA Owner role.

B. Firmware development environment

The firmware was implemented using STM32CubeIDE, an integrated environment provided by STMicroelectronics that bundles Eclipse and GCC toolchains with configuration and code-generation utilities (29). It supports device initialization through the .ioc file, integrated debugger, and direct flashing via ST-LINK. This approach ensures consistency across builds and speeds up low-level setup.

The project relies on several STM32Cube middleware packages. Azure RTOS ThreadX handles concurrency, while USBX provides a reliable CDC-ACM USB device stack. These components were selected for their stability, low footprint, and official support within STM32CubeU5 (30). Cryptographic operations leverage the on-chip hardware accelerators via the HAL CRYP and PKA drivers; random material is supplied by the HAL RNG interface. This combination of libraries provides efficient key management and deterministic execution while maintaining code maintainability.

Table 12: STM32 firmware libraries used in CertiFlow

LIBRARY / MODULE	FUNCTION IN FIRMWARE	REASON FOR SELECTION
HAL_RCC	Clock and power configuration	Standard, stable interface for system clock control
HAL_FLASH	Flash erase / program operations	Required to manage KeyStore storage securely
HAL_CRYP & HAL_PKA	Hardware crypto acceleration	Utilises AES and ECC engines for faster operations

HAL_RNG	Random-number generation	Provides entropy for PBKDF salts and nonces
USBX CDC-ACM	USB device communication	Easiest way to expose the HSM over virtual COM
THREADX	RTOS kernel	Lightweight task scheduler with deterministic timing

Table 12 describes the core STM32 firmware libraries employed in the HSM module. Each was part of the official STM32CubeU5 package and chosen for proven reliability, direct hardware access, and minimal maintenance overhead.

Firmware was flashed and debugged using , which integrates with CubeIDE and allows memory read/write verification and option-byte configuration (31). PuTTY was used as a serial console to test command exchanges and verify responses from the HSM device (32).

C. Software stack and application development

The three desktop applications (User, CA Owner, and Verifier) were written in Python, which was chosen for its cross-platform compatibility, mature ecosystem, and rapid prototyping capabilities. Python's rich standard library and third-party packages simplified the handling of HTTP requests, JSON data, and cryptographic material. Each application relies on lightweight frameworks and databases to keep the overall footprint small.

The CA API employs the Flask micro-framework, which offers a clear route mapping model and is sufficient for the low-traffic, local-hosted nature of the service (33). SQLite provides persistent storage without the complexity of a server-based database, ensuring that all data remain within a single protected file. PySide6 powers the GUI for both the CA Owner and User applications, delivering responsive interfaces while remaining purely Python-based. PySerial handles USB CDC communication with the HSM device, chosen because it abstracts port management and supports Windows and Linux out of the box. The requests library simplifies REST interactions between clients and the API.

Table 13: Main Python libraries and their roles in CertiFlow

LIBRARY	ROLE IN THE PROJECT	REASON FOR SELECTION
FLASK	Web-API framework for CA server	Minimal footprint, easy to secure
PYSIDE6	Desktop UI framework	Native look, fast UI development
PYSERIAL	Serial communication handler	Reliable access to USB CDC devices
REQUESTS	HTTP client for API calls	Simple and well-maintained
SQLITE	Local database management	Zero-configuration storage
CRYPTOGRAPHY	ECDSA / hashing operations	FIPS-aligned implementation
LOGGING / DATETIME	Local audit and timestamps	Built-in and portable

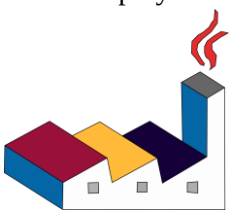
Table 13 summarises the Python libraries that support each part of the CertiFlow applications. The selection prioritised portability, maintainability, and alignment with standard cryptographic and PKI requirements.

Source-code management relied on Git with remote hosting on **GitHub**, chosen for its distributed branching model, pull-request workflow, and built-in code review that suits incremental development across firmware and multiple desktop applications. The platform's issue tracking, release tagging and Actions-based automation gave a single place to manage changes, build artefacts and test runs without additional infrastructure. Using protected branches and mandatory reviews improved traceability around sensitive changes to cryptographic code and protocol handlers, which is essential in a security-oriented project (34).

Visual Studio Code served as the primary editing environment because it provides first-class Python support, integrated debugging, and seamless Git operations in one window, which reduced context switching during implementation and testing. The editor's extension ecosystem covers Python language servers, linting and formatting, serial-port helpers and Markdown previews, so the same tool could be used to modify firmware-adjacent scripts, desktop UI code and documentation with consistent diagnostics. The integrated terminal and problem matcher simplified reproducible build steps and let the project standardise instructions for new environments without changing tools (35).

For diagrams created from scratch and for layout-heavy figures, **draw.io** was used because it produces editable vector diagrams that export cleanly to PDF and SVG while remaining easy to version as XML alongside the source code. Layering and reusable shape libraries kept the deployment and execution views visually consistent across revisions, which reduced the cost of updating figures when interfaces evolved. The ability to export at fixed resolutions avoided last-minute scaling artefacts when placing figures into the report template (36).

Where diagrams benefited from text-as-code, **PlantUML** was used so sequence, class and deployment diagrams could live as plain text in the repository and be regenerated in the build pipeline, preventing drift between diagrams and implementation. Text-based diagrams also made reviews simpler, since proposed edits appear as normal diffs and can be discussed inline with pull requests. This approach kept the "System overview" figures reproducible and ensured that any renaming of components or endpoints could be propagated mechanically (37).



4. Cross-cutting foundations

This section describes the elements that are shared by all components of CertiFlow: the data artefacts they exchange, the contracts that govern those exchanges, and the security controls that apply system-wide. The aim is to keep the later component sections short by defining formats, interfaces, and policies once, here, in a compact and consistent form.

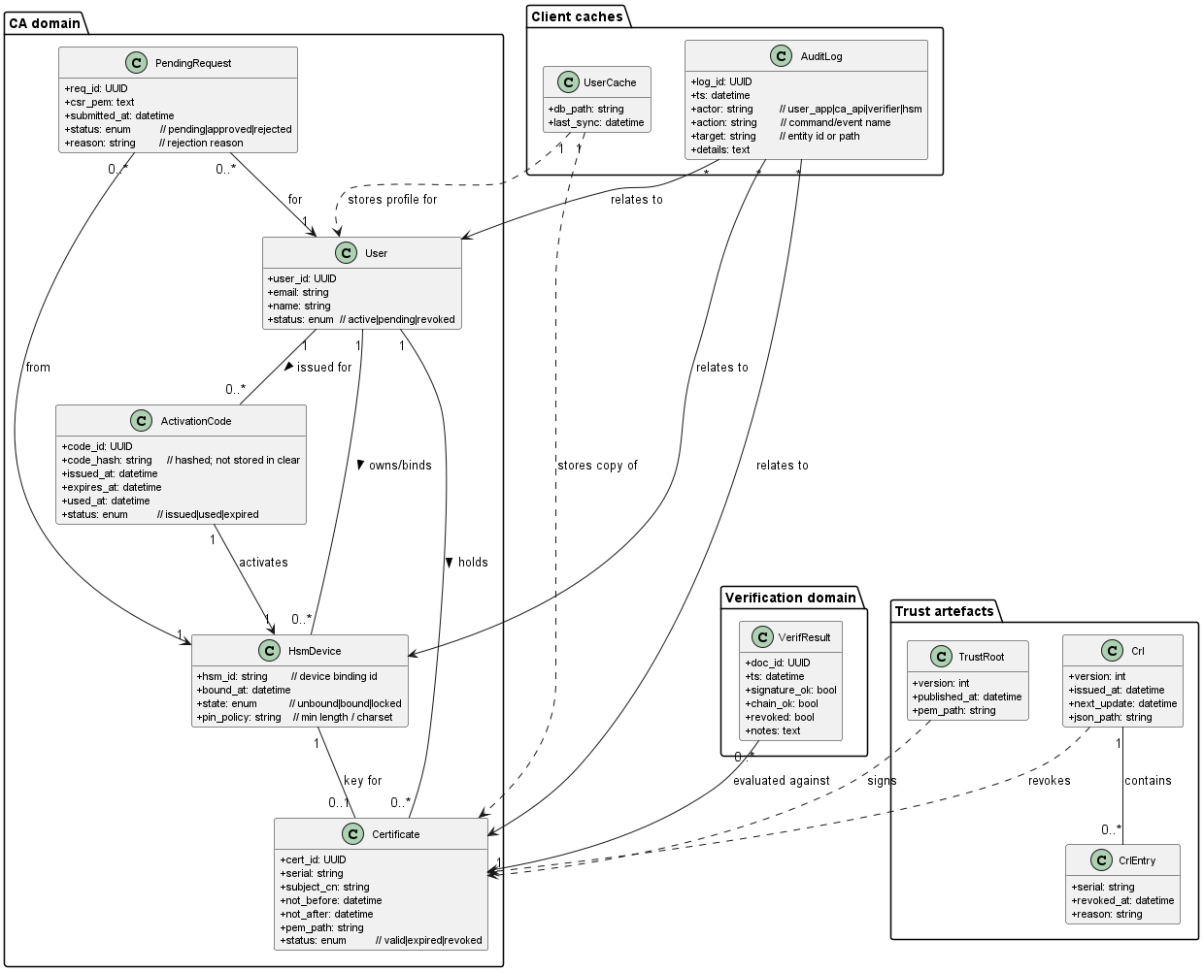


Figure 24: Core classes and persistent entities in CertiFlow

Figure 24 captures the shared data model used across CertiFlow: the CA domain manages users, bound HSM devices, activation codes, CSR-backed pending requests, and issued certificates; trust artefacts consist of a published root and a CRL with per-certificate revocation entries; client caches store local profiles and audit events; and the verifier records per-document results that reference the evaluated certificate. Relationships show ownership and flow: users bind devices and receive activation codes, devices originate CSRs that become pending requests, approved requests yield certificates, trust artefacts sign or revoke those certificates, and audit and verification records link back to the entities they concern.

A. Data and formats

This section defines the data artefacts exchanged across CertiFlow and the formats in which they are stored. It identifies where each artefact resides, which component produces it, and which component consumes it

Table 14: Data artefacts, storage and ownership

ARTEFACT	FORMAT	STORED AT	PRODUCED BY	CONSUMED BY
ROOT CERTIFICATE	PEM (X.509)	Trust store bundle	CA Owner	User App, Verifier
END-ENTITY CERTIFICATE	PEM/DER (X.509)	User cache, CA DB	CA Owner	Verifier
CRL	JSON (version, issued, revoked)	Trust store bundle	CA Owner	Verifier, User App
PUBLIC KEY	DER SPKI	User cache	HSM (via User App)	CA API (CSR), Verifier
SIGNATURE	DER ECDSA (base64 in metadata)	With signed doc	HSM	Verifier
AUDIT EVENT	JSON/row	Local SQLite (per app)	All apps	CA (sync), auditor

Table 14 summarises the artefacts that persist across components, their on-disk formats, and which subsystem creates or reads them.

B. Interfaces and contracts

This section specifies the communication contracts that bind components together. It sets the framing rules, message shapes and minimum behavioural guarantees for device–host exchanges and service endpoints, so that implementations remain interoperable and failures are predictable.

i. USB CDC line protocol

This part describes the core text-based command channel exposed by the HSM over a virtual serial port. It defines line termination, command syntax, expected replies and error signalling, as well as the preconditions for stateful operations such as unlocking and signing, so host applications can interact deterministically.

Table 15: USB command surface

COMMAND	ARGUMENTS	PRECONDITIONS	SUCCESS	FAILURE
INFO	—	—	OK info …	ERR …
HSMID	—	—	OK <uid>	ERR …
UNLOCK	<pin>	Not locked	OK	ERR LOCKED
KEYGEN	EC P256	Unlocked	OK	ERR ARG
PUBKEY	—	Key exists	OK <spki-b64>	ERR …
SIGN	SHA256 <64-hex>	Unlocked, key exists	OK <sig-b64>	ERR …

Table 15 summarises the HSM's command interface in a compact form, listing each operation with its arguments, preconditions, and the expected success or failure replies. It captures the framing model used on the wire and the state transitions that matter to clients.

ii. REST API

This part outlines the HTTP interface used by the applications to reach the CA services. It presents the endpoint catalogue and the minimal JSON request and response bodies required for provisioning, issuance, trust publication and log synchronisation, keeping the contract simple and implementation-neutral.

Table 16: REST endpoints

METHOD	PATH	PURPOSE	MINIMAL REQUEST	TYPICAL RESPONSE
POST	/api/activate_hsm	Bind device to user	{email, hsm_id, code}	{ok, message}
POST	/api/cert/csr	Submit CSR	{csr_pem}	{ok, cert_pem}
GET	/api/trust/root	Fetch root	—	PEM
GET	/api/trust/crl	Fetch CRL	—	JSON
POST	/api/logs/sync	Push audit entries	[{…}]	{ok, count}

Table 16 presents the minimal contract for the CA-facing HTTP API, showing the method, path, purpose, and the smallest viable request and typical response for each route. It is intended as a quick reference for client implementers.

C. Security controls implemented

This section states the project-wide security controls that are enforced across components. It consolidates the effective parameters for authentication, key protection and session handling on the device, along with issuer-side checks and the trust distribution model, to provide a single view of the system's baseline posture.

Table 17: Enforced security parameters

CONTROL	SETTING
PIN POLICY	Length and charset as implemented
PBKDF	PBKDF2-HMAC-SHA-256, 20 000 iterations, 16-byte salt
SESSION	Idle timeout: 120 seconds
LOCKOUT	After 5 failures, penalty 30 seconds
ZEROISATION	Sensitive buffers wiped after PBKDF, keygen, sign
TRUST	Root PEM + CRL JSON; offline verification supported

Table 17 consolidates the system-wide security parameters that the implementation actually enforces, including PIN policy, key-derivation settings, lockout and session timings, zeroization points, and the trust artefacts used during verification.

5. Component implementations

This section presents the implemented components of CertiFlow in a clear, evidence-led manner. Each part focuses on what was built, how it operates, and how it interacts with the rest of the system, using concise text supported by screenshots. The aim is to document behaviour rather than repeat design intent, so the emphasis stays on concrete responsibilities, interfaces, and observable results.

A. HSM implementations: from encrypted KeyStore to embedded device

This part documents the evolution of CertiFlow's signing module from a file-based KeyStore on removable storage, through a Python-based HSM emulator, to the final embedded firmware running on the STM32U5 board. The aim is to show how each stage incrementally improved key isolation, operational reliability, and auditability, while keeping the command contract stable for the desktop applications.

i. Encrypted KeyStore on USB storage

The initial approach stored an encrypted KeyStore file on a removable USB drive and protected it with a user password. Keys were created on the host and sealed using a password-derived key; signing operations were performed in software after decrypting the private material in process memory. This solution was quick to implement and allowed early end-to-end testing, including the CA Owner's login workflow which still relies on a similar encrypted store for administrative credentials. Its main limitations were exposure of private keys in host memory during use, sensitivity to filesystem corruption or accidental deletion, and the absence of hardware-enforced lockout or zeroization.

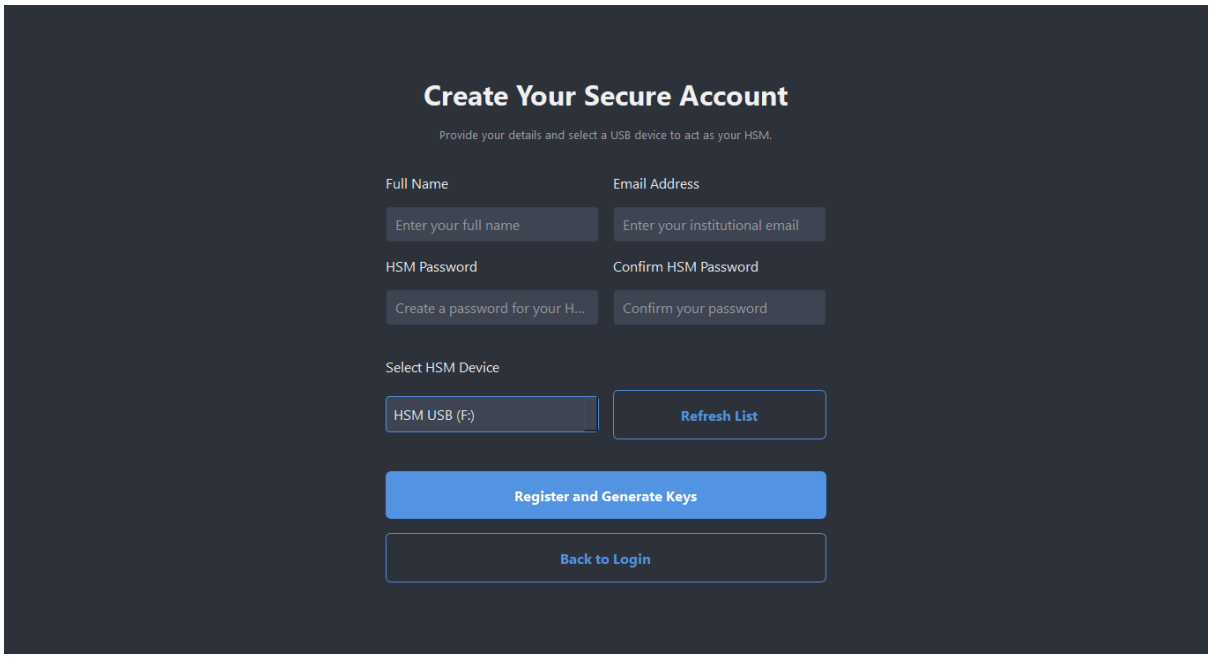


Figure 25: Screenshot - User Registration Page using Encrypted KeyStore on USB storage

Figure 25 represents the first registration page for the signer app, that was based on storing the keypair in a USB storage unit, encrypting it using the password that the user would enter in the same page. All encryption and signing functions were in the handlers of the app, so the "HSM" only had one function, and that is storing the keypair.

ii. Python HSM emulator (device-compatible protocol)

The second stage replaced host-side key handling with a local "virtual device" implemented in Python that mimicked the final HSM's line protocol. The emulator generated keys once, kept them in a protected process space, and serviced commands such as INFO, HSMID, UNLOCK, KEYGEN, PUBKEY, and SIGN using the same request/response framing as the hardware device. To exercise it like real hardware, a virtual serial-port pair was created with com0com so that the emulator listened on the first port while PuTTY attached to the second, allowing end-to-end tests without a physical board. This preserved the desktop applications' behaviour while eliminating most file-handling risks and enabling deterministic tests. It also allowed edge cases and error codes to be stabilised before touching firmware.

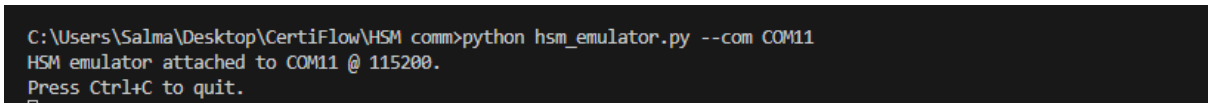


Figure 26: Screenshot - HSM python emulator

Figure 26 shows how the emulator was run, using the port reserved by com0com, and to test the emulator, we had to connect to the other port reserved by com0com to PuTTY or a testing script, and run the commands to see if they get a proper answer

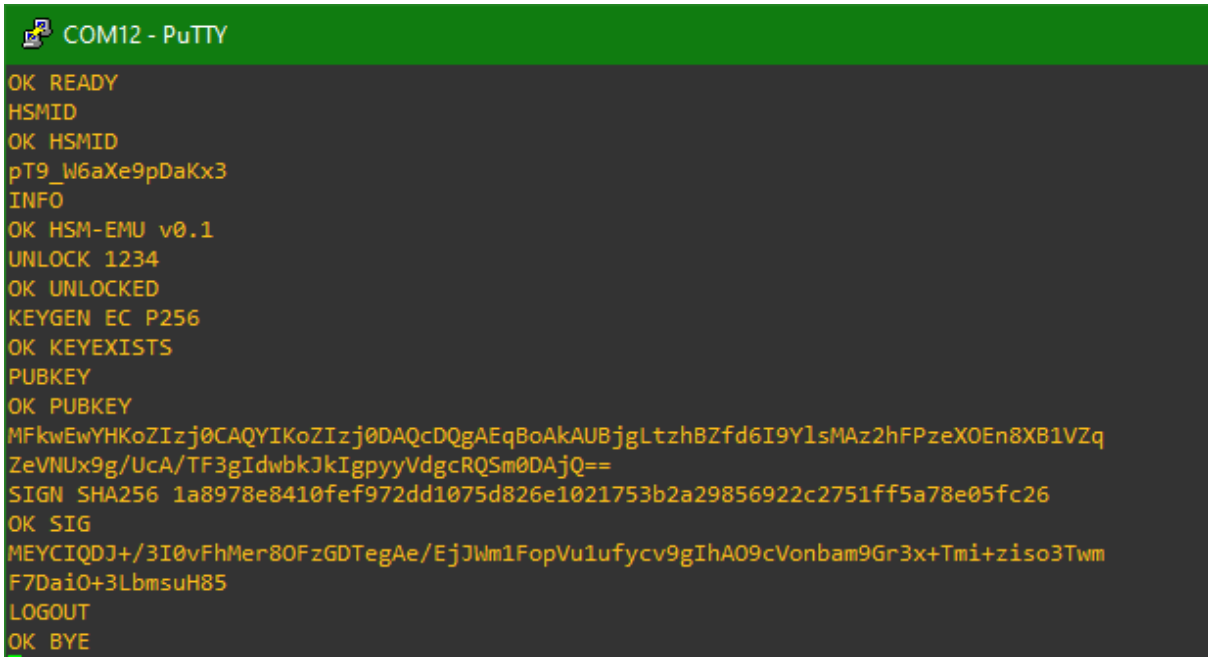


Figure 27: Screenshot - HSM python emulator test using PuTTY

In **Figure 27** we can notice replies from the emulator, the emulator starts by a greeting reply OK READY, that assured the communication between the emulator and PuTTY, then it replies to each command with the right answer, this emulator worked as an early prototype for the firmware that was about to be built

iii. STM32U5 firmware HSM (final device)

The final stage moved the same command surface to an embedded target that generates and stores keys inside the MCU's non-volatile memory and executes ECDSA operations via the device runtime. The firmware enforces a PIN-gated session, a failure-count lockout with penalty, and explicit zeroization of sensitive buffers after PBKDF, key generation, and signing. It persists the KeyStore in flash with a versioned header and integrity check, and exposes a single USB CDC interface so the desktop applications can connect without custom drivers.

First step that has to be taken to make sure the board is connected, is checking the Device Manager for any new COM port

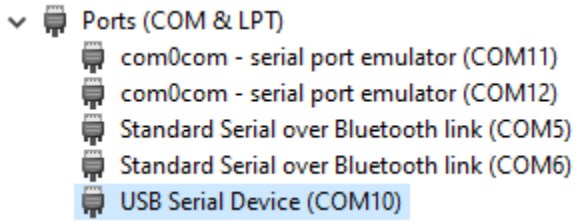


Figure 28: Screenshot - STM32 board enumerating on Device Manager

Figure 28 is a screenshot of the device manager window after connecting the STM32U5 board, and it shows what COM port the board is reserving, which appears to be COM10

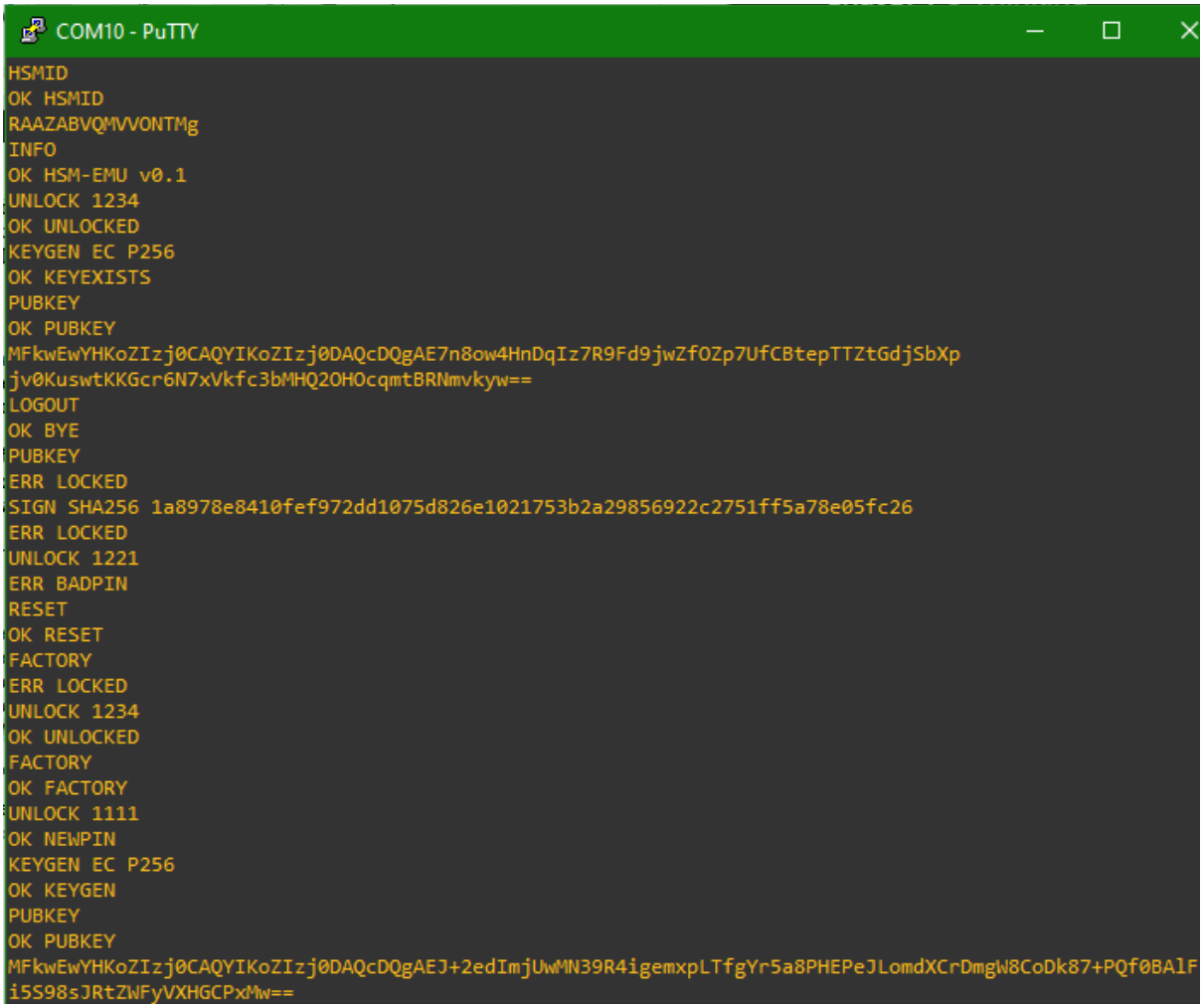


Figure 29: Screenshot - STM32U5 firmware HSM using PuTTY

Figure 29 shows the communication between PuTTY and the HSM Board, the test was successful and all commands were handled correctly, a key pair already existed on the board at the beginning, but we were able to FACTORY reset the board, after unlocking it with the right PIN. Errors and exception were handled correctly, returning ERR + the error code that show's exactly why the board returned an exception.

iv. Summary comparison of HSM implementation stages

Table 18 compares the three stages against key custody, isolation level, common risks, and operational strengths. It provides a single view of why the final embedded device was adopted while acknowledging the practical value of earlier stages for prototyping and current CA login use.

Table 18: Summary comparison of HSM implementation stages

STAGE	KEY CUSTODY	ISOLATION	TYPICAL RISKS	OPERATIONAL STRENGTHS
ENCRYPTED USB KEYSTORE	Host process memory during use; encrypted at rest on USB	None beyond OS permissions	Password reuse, host malware, file loss/corruption	Fast to prototype, simple recovery and backup
PYTHON HSM EMULATOR	Host process memory; sealed by emulator process	Process-level only	Same host malware risk; emulator crash loses session	Protocol-accurate tests, stable error handling, no USB driver issues
STM32U5 FIRMWARE HSM	On-device flash; never leaves MCU	Hardware boundary; PIN, lockout, zeroization	Physical attacks; device loss without backups	True key isolation, deterministic behaviour, minimal host trust

B. CA Owner application and API

This part covers the administrative desktop interface and its local API that together provide provisioning, approval, issuance, and trust publication. It explains the main screens, the data stored in the local database, and the HTTP endpoints that the other applications consume. The evidence set will include UI captures for activation and approvals, example API requests and responses, and short traces of audit events.

i. Setup page

On first run the application shows a multi-step wizard instead of the login screen. The wizard guides the root administrator through the welcome step, credential entry with validation, HSM path selection, a confirmation step, and a progress page that initialises the root CA before signalling completion.

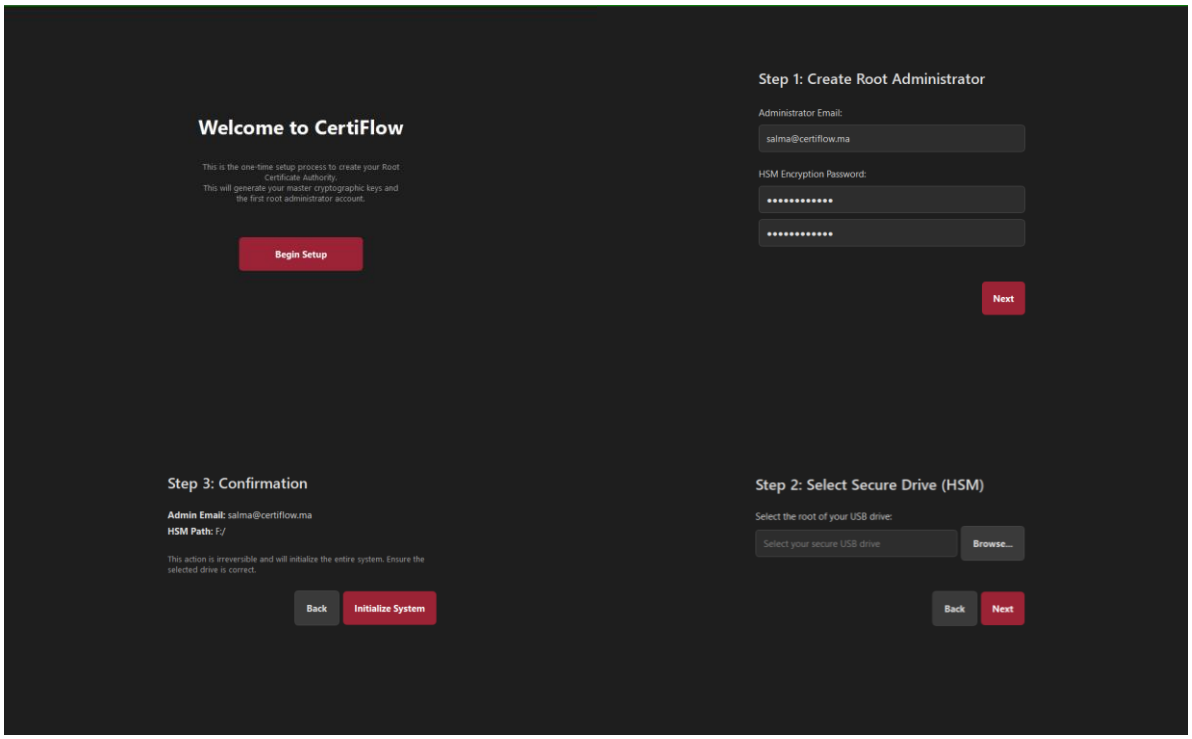


Figure 30: Screenshot - CAO Setup Page

ii. Login page

The main window loads this view first so administrators land on a focused sign-in screen. It centres email and password inputs, displays validation errors, and dispatches an authentication action when the user clicks Authenticate or presses Enter. On success it hands the collected credentials to the next part of the flow.

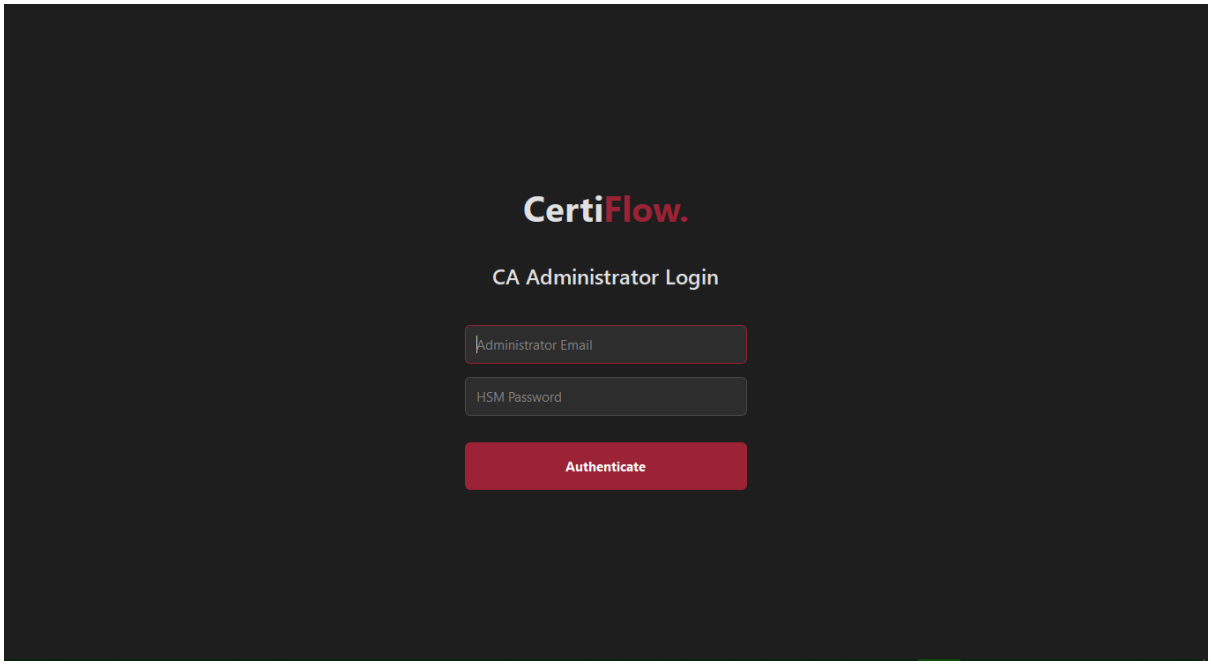


Figure 31: Screenshot - CAO Login Page

iii. HSM Wait page

After a login attempt, the interface switches to this view while the application searches for the removable-drive HSM. A background worker scans mounted drives, attempts admin login against any discovered store, surfaces precise failure messages, and allows the user to cancel back to the login screen.

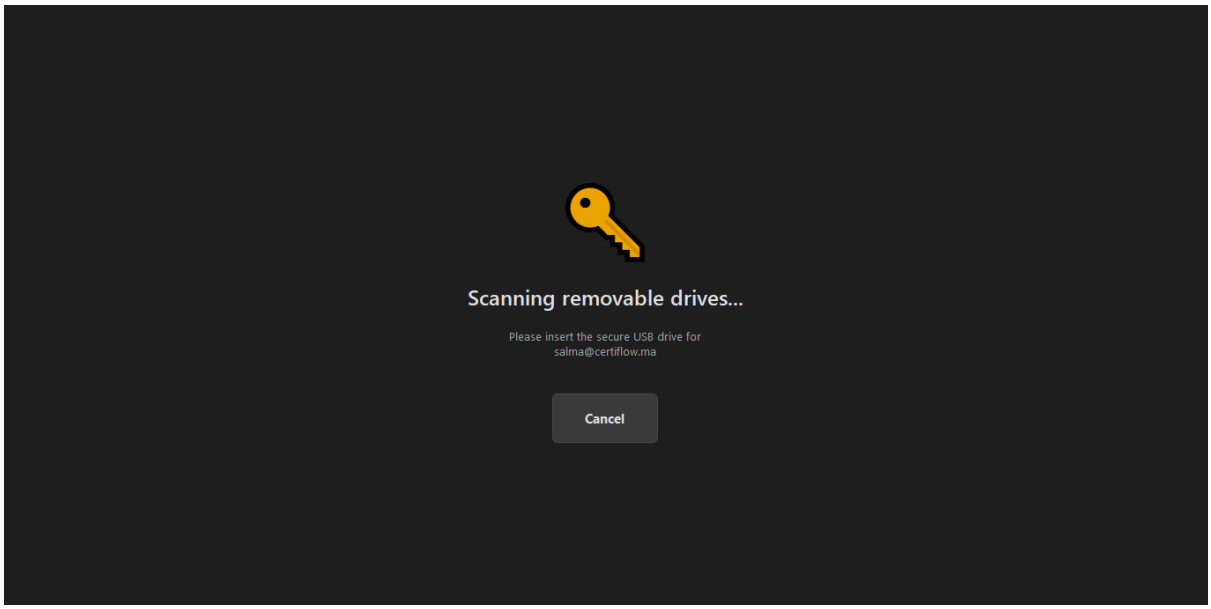


Figure 32: Screenshot - CAO HSM Wait page

iv. Dashboard page

Once authenticated, the dashboard presents high-level status: statistic cards, a list of pending certificate requests with their verification state, and quick approval or rejection actions. Actions are confirmed via dialogs, routed to the CA API, and the lists refresh automatically.

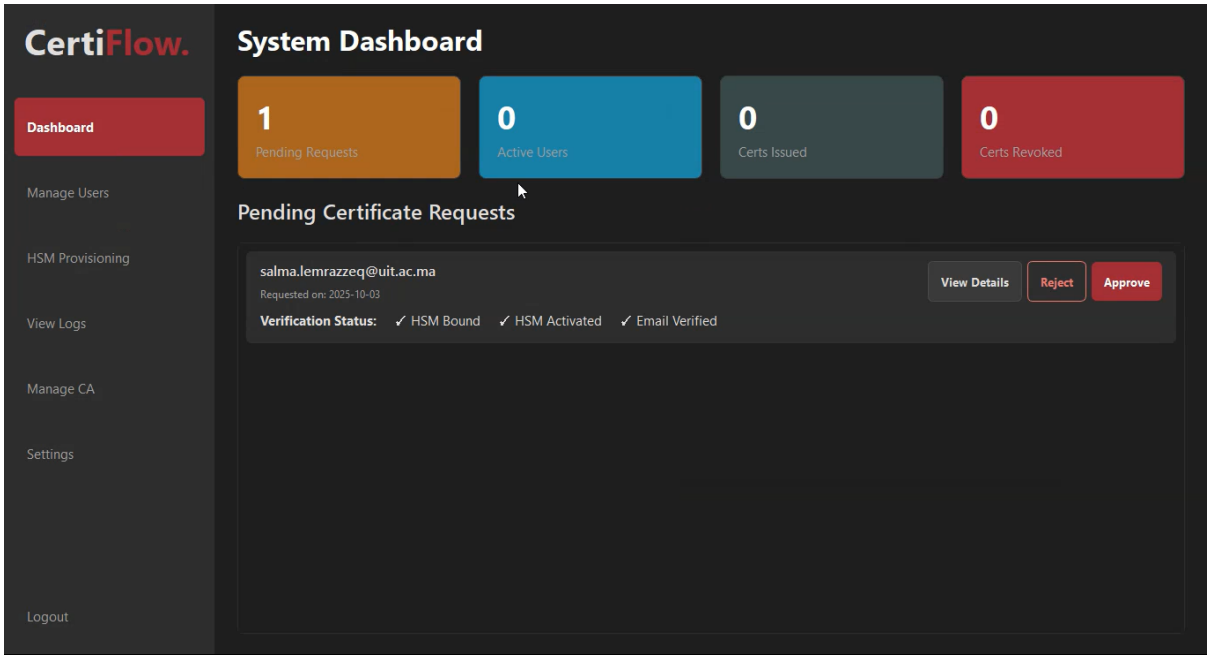


Figure 33: Screenshot - CAO Dashboard Page

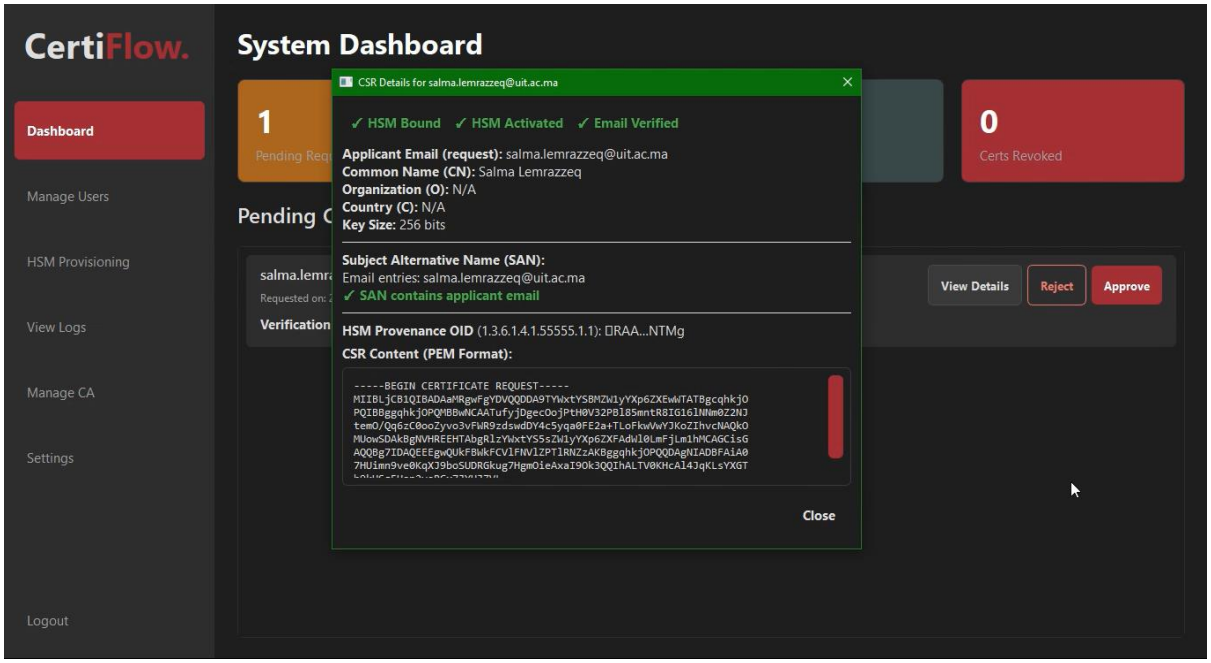


Figure 34: Screenshot - Pending request details Dialog

v. Manage Users page

This management view lists users with filtering and status widgets, refreshes periodically, and shows a detail panel for the selected user. From here administrators can view bound devices, examine issued certificates, and trigger certificate revocation with a reason prompt.

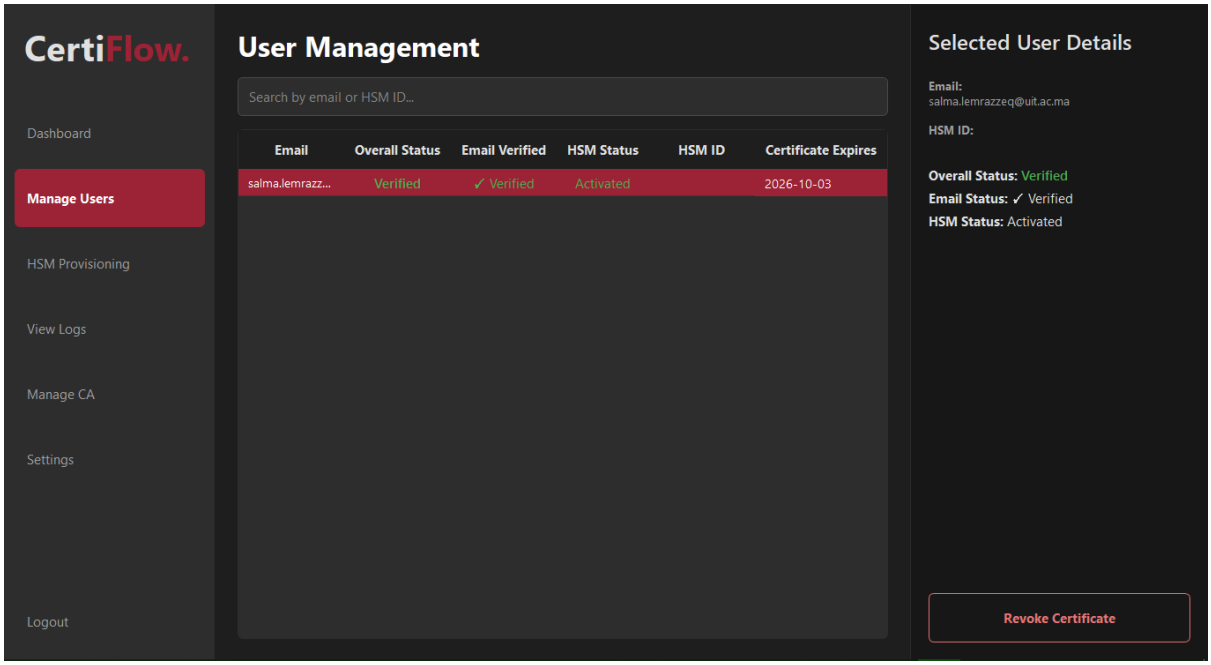


Figure 35: Screenshot - CAO Manage Users page

vi. HSM Management page

The provisioning console scans for newly connected devices and separates them into "detected" and "bound" tables. It supports binding a device to a user, regenerating or viewing activation codes, reassigning ownership, and revoking bindings through guided dialogs.

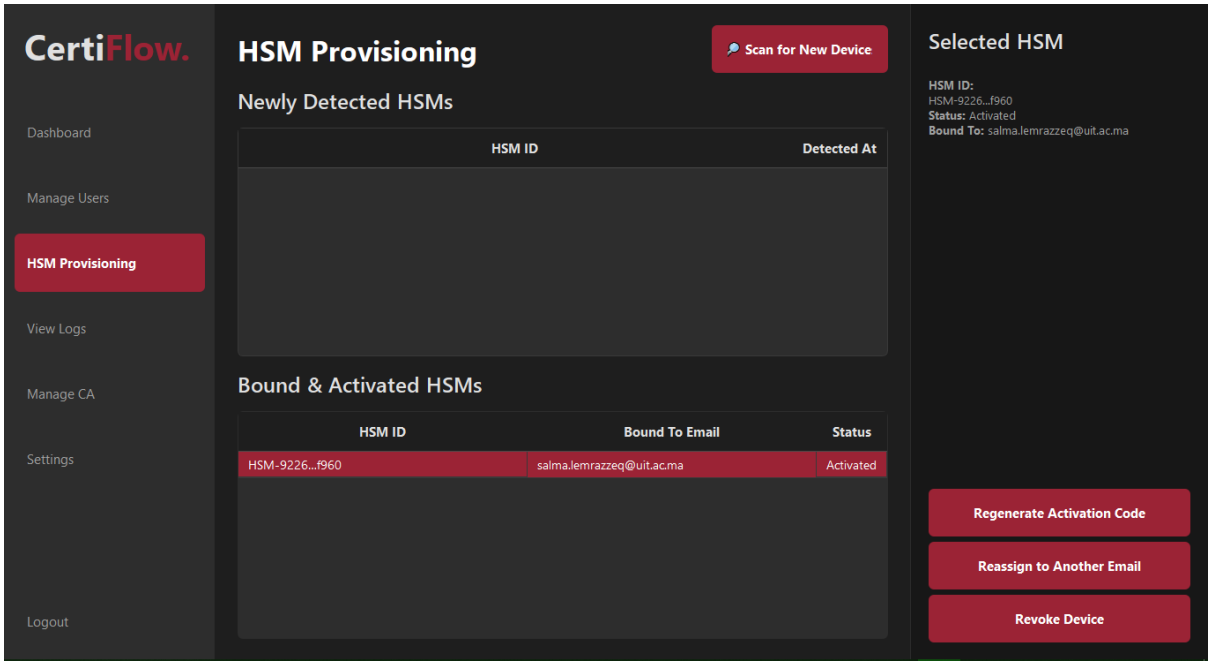


Figure 36: Screenshot - CAO HSM Management page

vii. View Logs page

The audit log viewer reloads entries on a schedule, offers free-text filtering, and displays a details panel for the selected record with categorised metadata. It provides a quick way to trace administrative actions and API events.

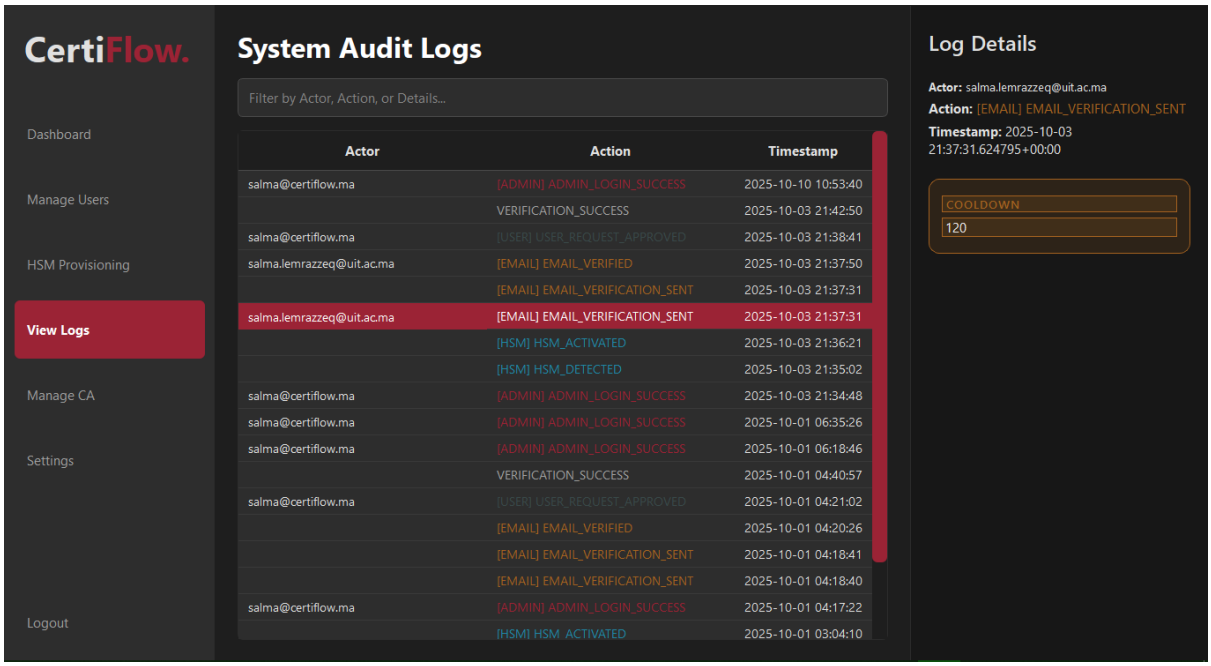


Figure 37: Screenshot - CAO View Logs page

viii. Manage CA page

The administrator roster lists CA operators with masked HSM identifiers and enforces permissions so only root administrators can add or remove accounts. All changes pass through confirmation flows to prevent accidental modifications.

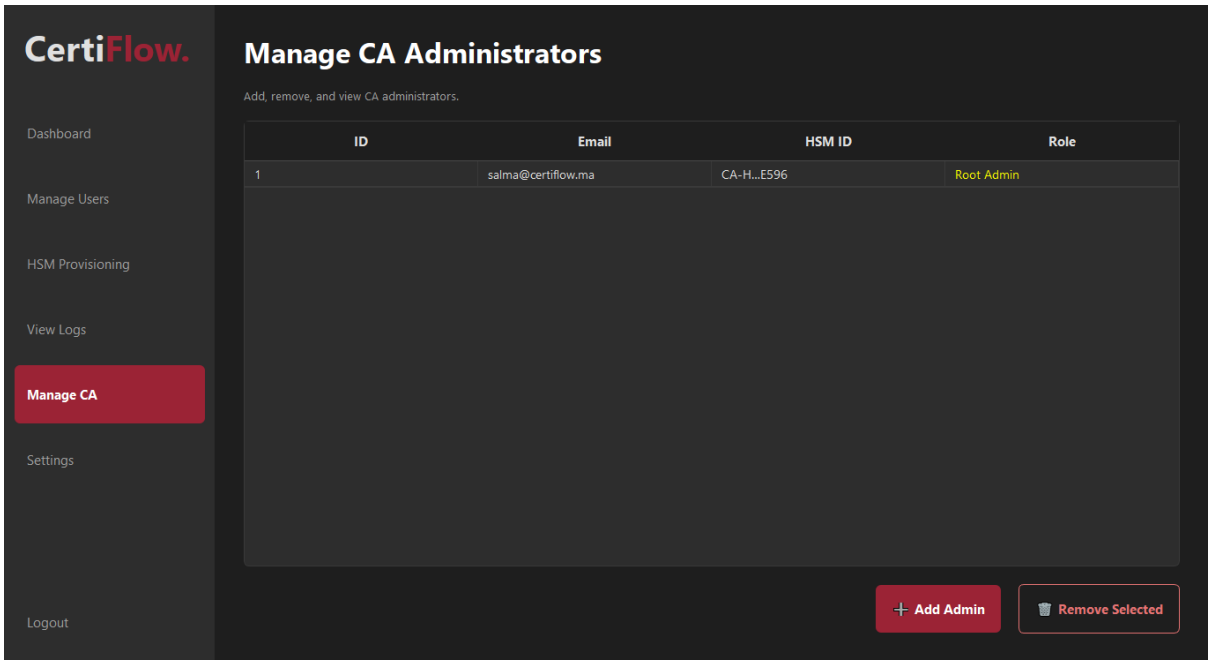


Figure 38: Screenshot - CAO Manage CA page

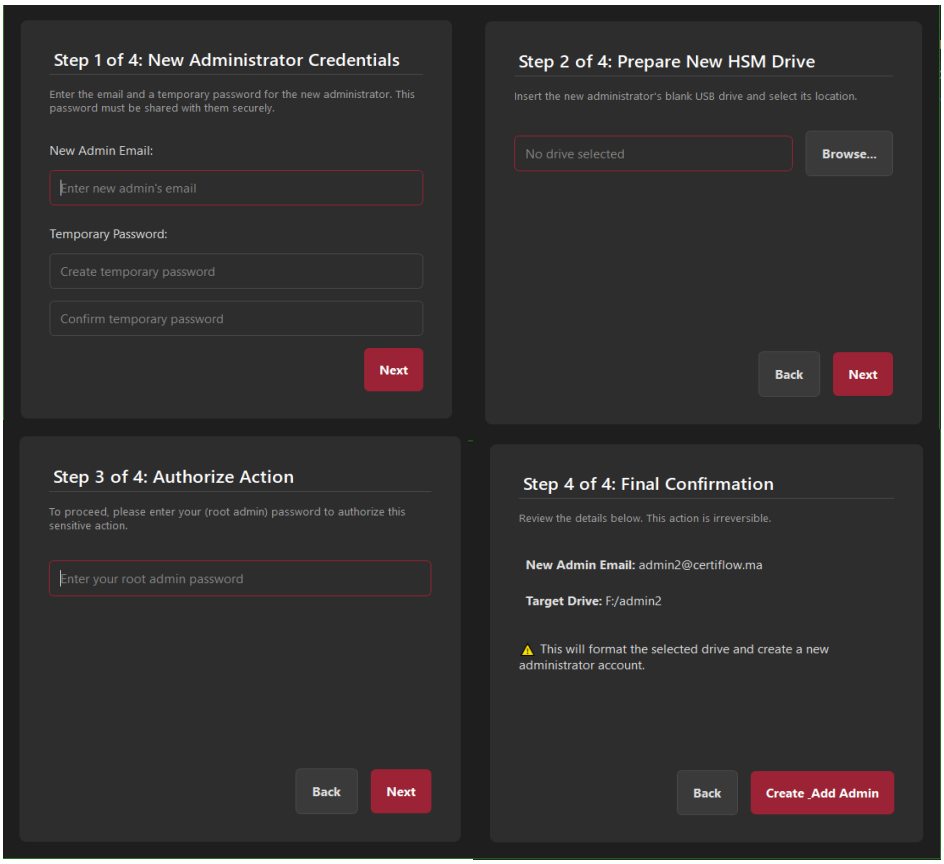


Figure 39: Screenshot - CAO add new Admin Dialogs

ix. Settings page

The system settings view highlights missing or misconfigured email settings, and provides actions for password change, backup, restore, and viewing certificates. Each operation validates session prerequisites before execution and guides the user through any required confirmation.

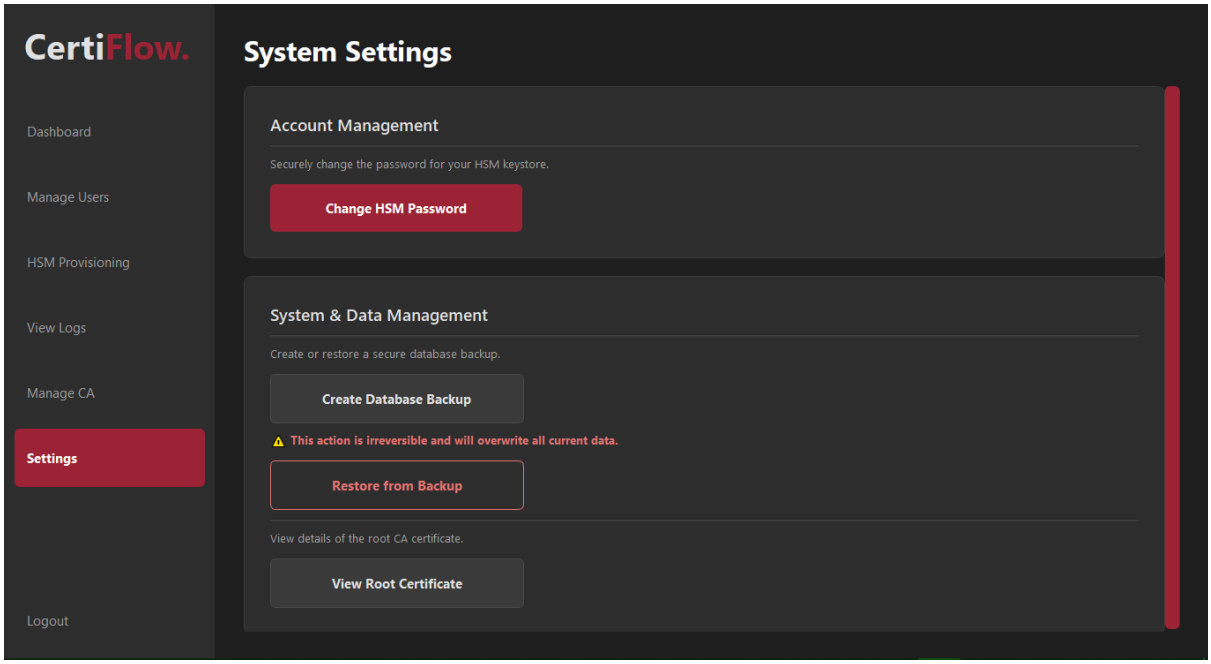


Figure 40: Screenshot - CAO Settings Page

C. User (signer) application

This part documents the desktop application used by end users for enrolment and document signing. It summarises the registration and email-verification flow, the way the app drives the HSM for cryptographic operations, and the artefacts it produces and stores locally. Screenshots will show the key pages, while brief logs will demonstrate the path from PIN entry to signature generation and audit logging.

i. Login page

This page presents a centred card with the CertiFlow logo, institutional email field, PIN field, and Login/Register actions. Pressing Login validates inputs, calls the authentication routine, and routes to HSM scanning, pending approval, or rejection as appropriate. Register opens the enrolment workflow.

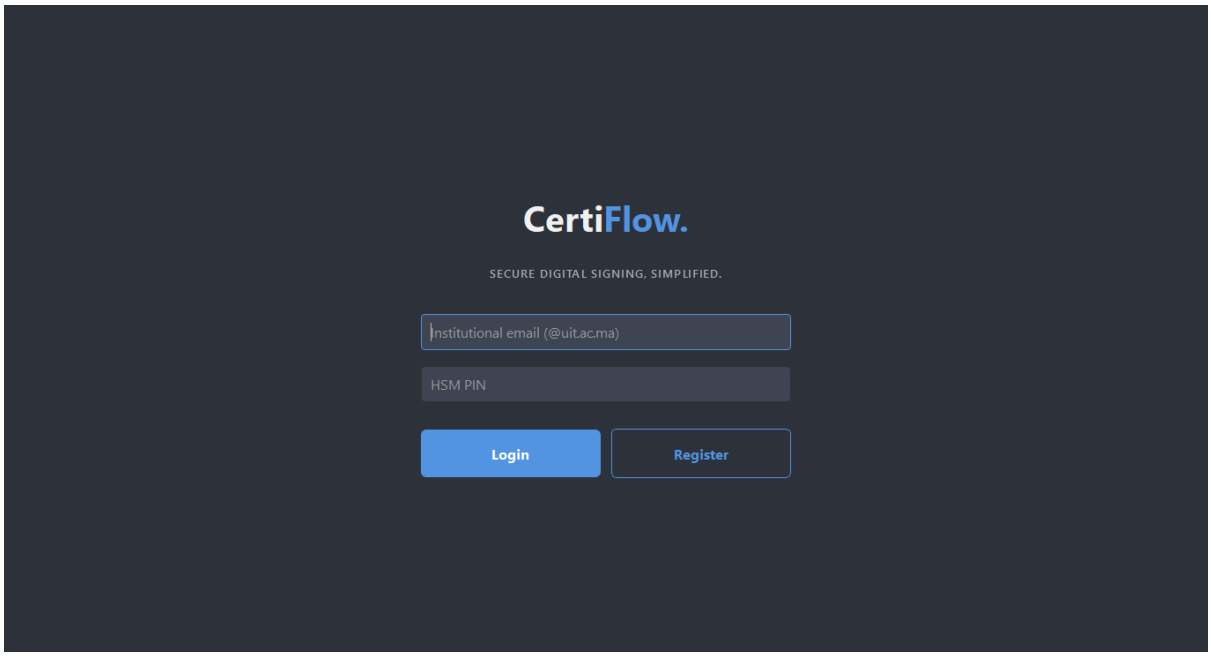


Figure 41: Screenshot - User Login Page

ii. Registration page

A three-step process: select a detected device, submit the activation code for server verification, then provide email, name, and a chosen PIN. A worker performs registration, shows progress, and emits completion or error so the flow can advance to approval tracking.

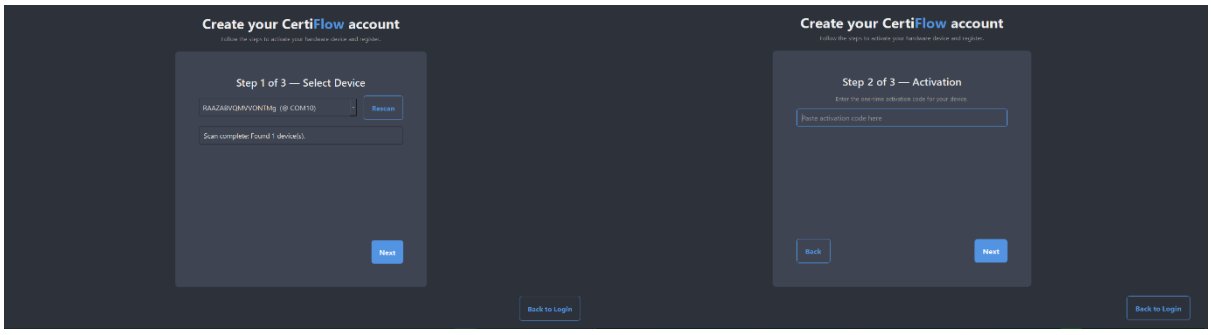


Figure 42: Screenshot - User Registration Page Step 1 & 2

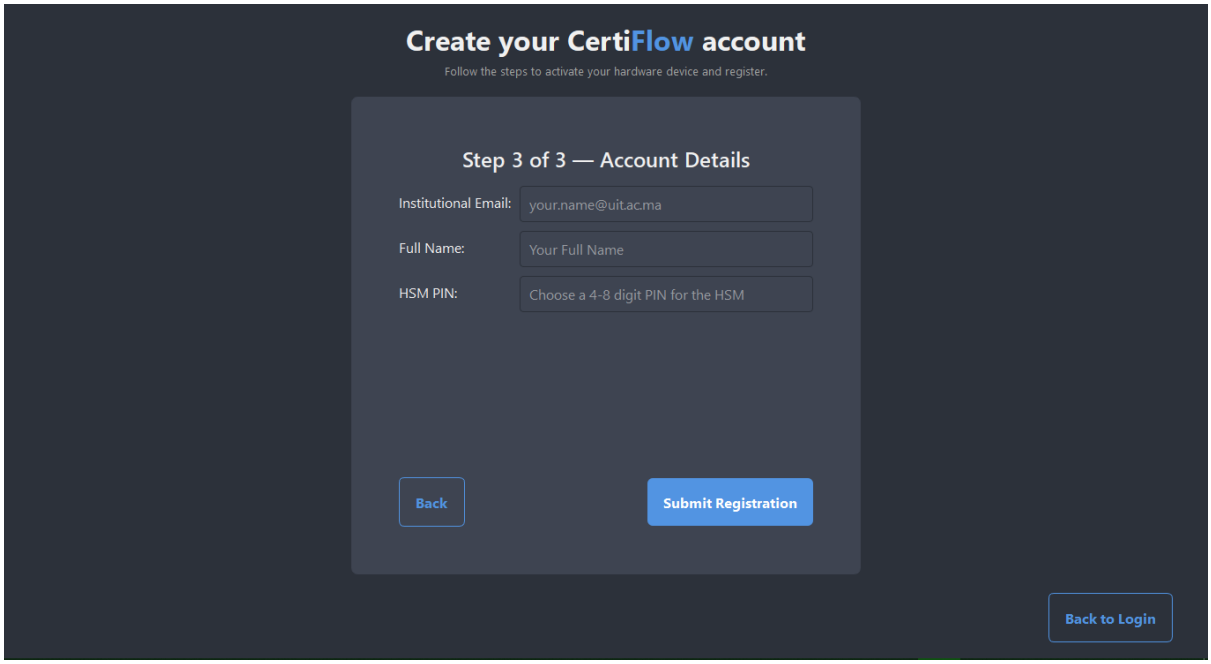


Figure 43: Screenshot - User Registration Page Step 3

iii. Pending Approval page

While the CA reviews the request, the page displays branded messaging with a spinner and periodically polls the user status endpoint. Verified users are returned to the login page to start a normal session; denials redirect to the rejection view; connectivity issues are shown with brief guidance and a Back to Login option.

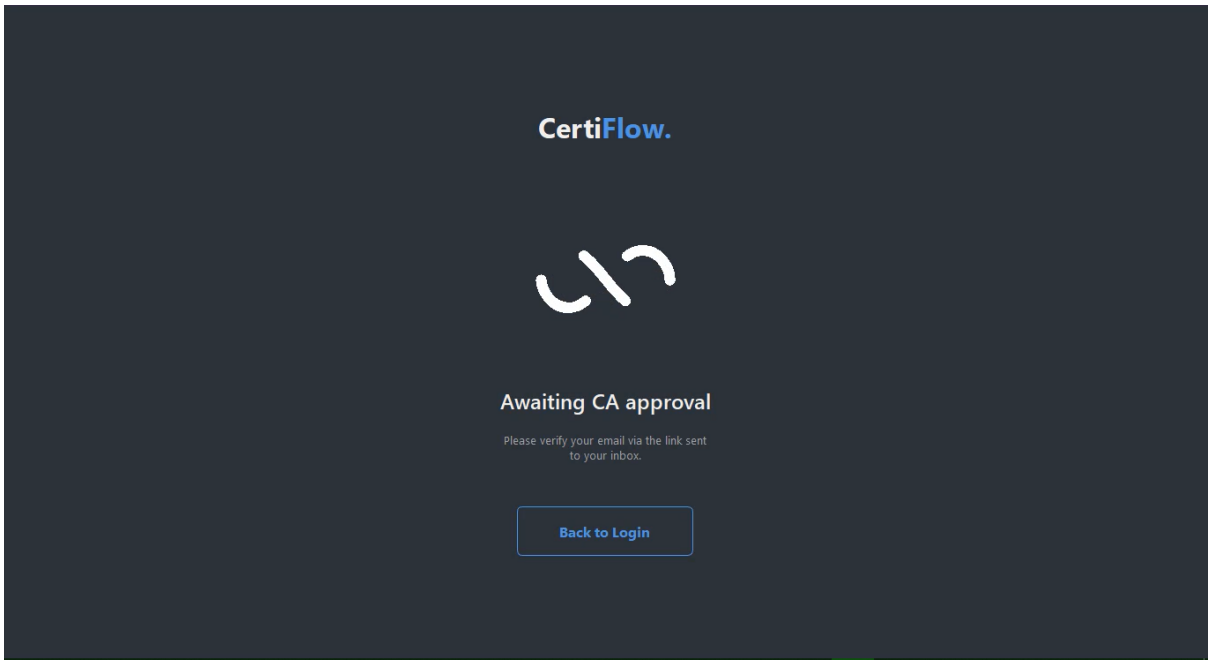


Figure 44: Screenshot - User Pending Approval page

iv. Home page

Authenticated users land on a dashboard with a greeting and status cards summarising certificate validity, security gates, recent activity, and trust synchronisation state. Interactions on the cards trigger navigation or trust refresh, and cached data populate expiry warnings, gate badges, recent documents, and CRL metadata.

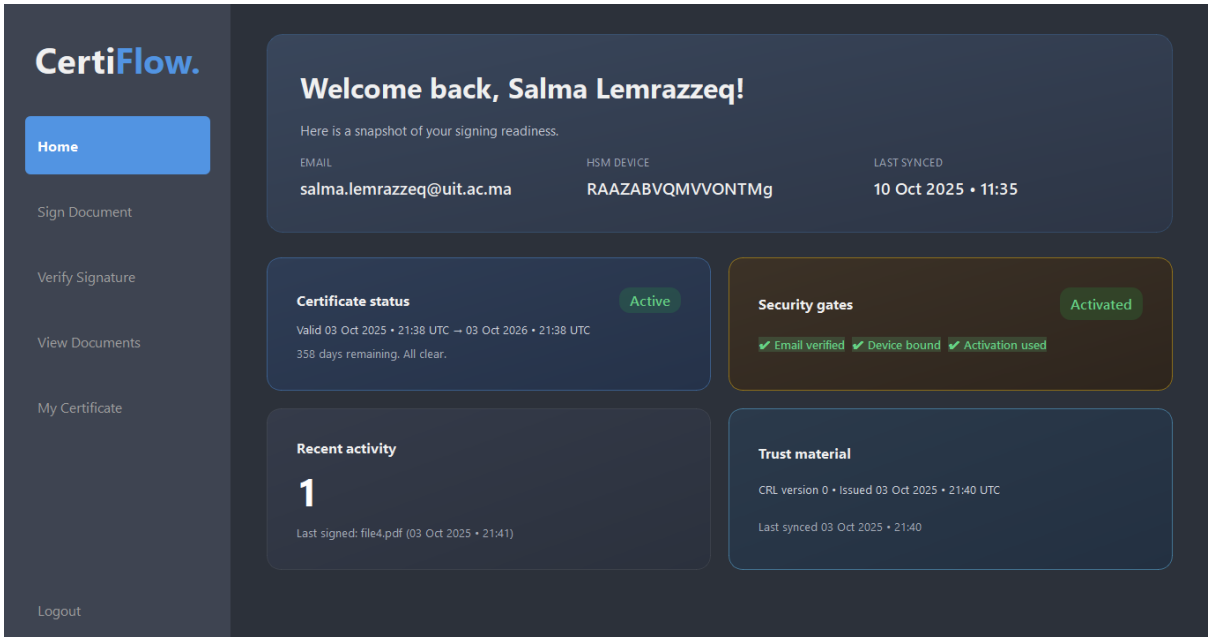


Figure 45: Screenshot - User Home Page

v. Sign Document page

This page allows the user to choose a PDF, enter the HSM PIN, and start signing. A worker runs the signing routine, shows progress, reports success or failure, records an audit entry, and resets the form while keeping the interface responsive.

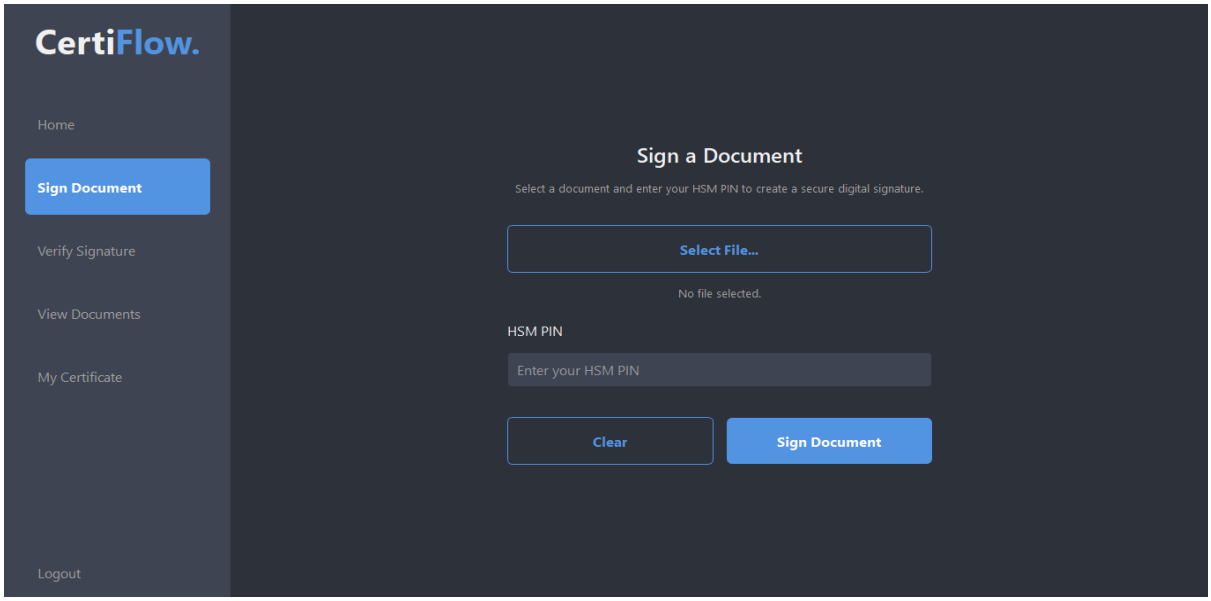


Figure 46: Screenshot - User Sign Document page

vi. Verify Signature page

This page accepts a signed PDF and the signer's email, then runs verification. Controls are disabled during the check, and a styled result dialog presents outcomes or input errors before the page returns to an idle state for another attempt.

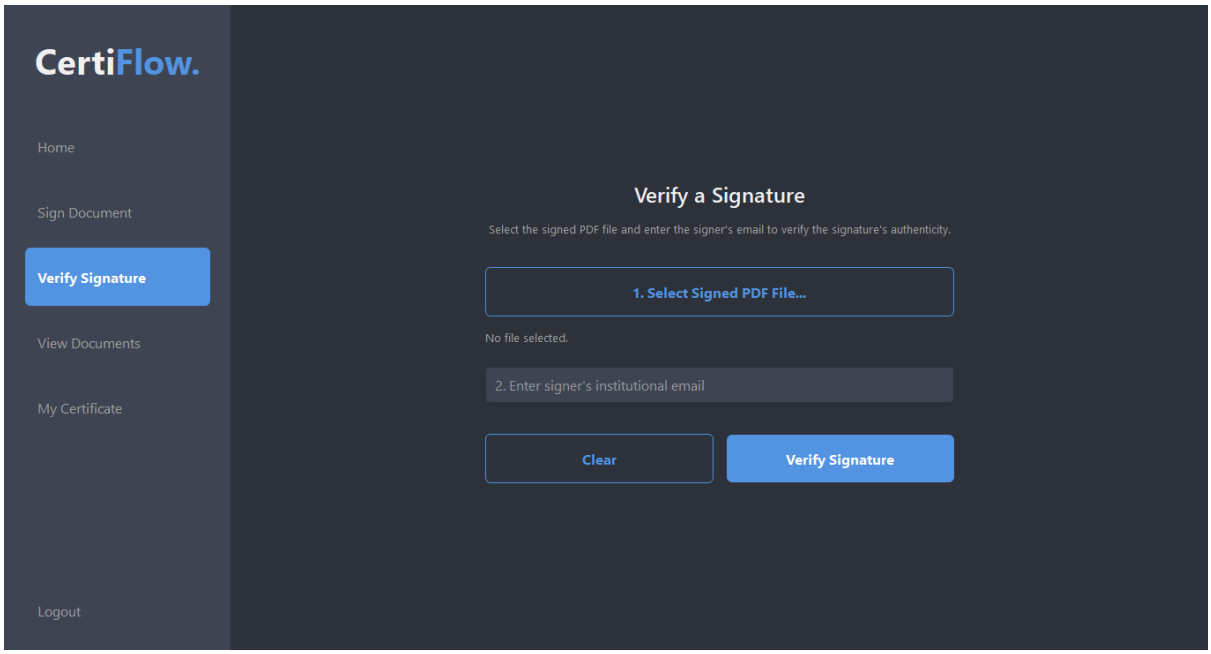


Figure 47: Screenshot - User Verify Signature page

vii. User Info page

The User Info Page displays certificate subject, issuer, validity window, serial number, and local HSM status flags from the cache. Renewal hints update button state and messages; a Renew action emits a signal to open the renewal flow.

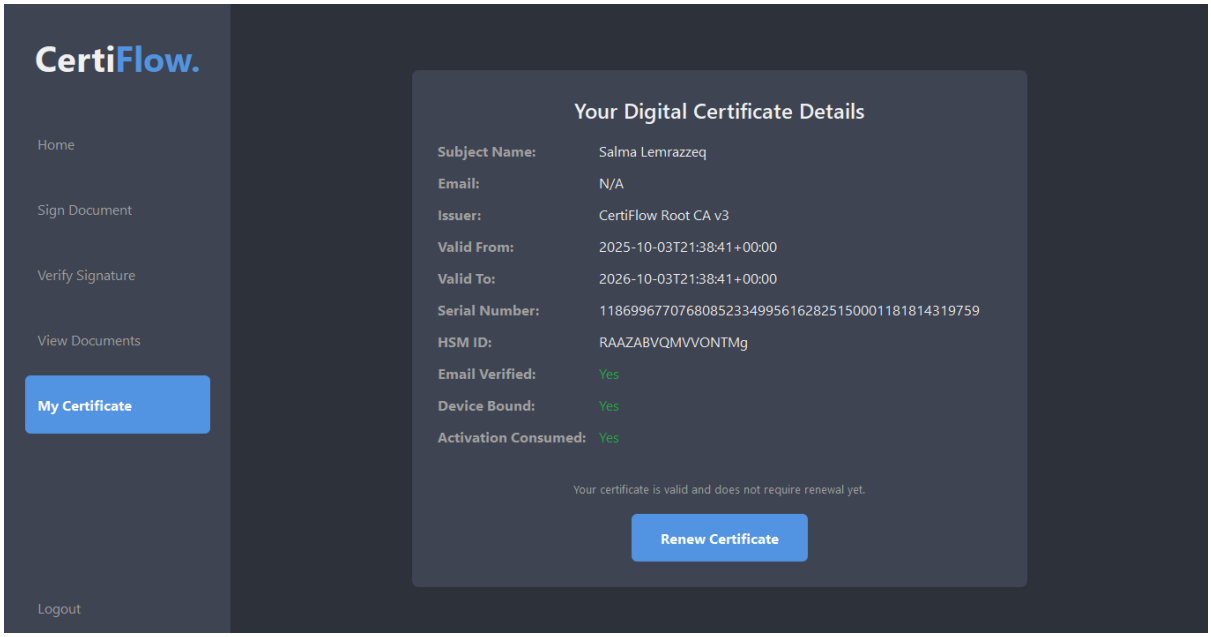


Figure 48: Screenshot - User Info page

viii. Renew Certificate page

This page requests the HSM PIN and launches a renewal worker that submits the request to the CA. The page locks the action during processing, shows success or failure, emits follow-up signals, and resets on completion.

ix. View Documents page

The view documents page shows a scrollable history of signed documents with filenames, timestamps, and signer emails from the local database. Users can open containing folders, resolve missing files via a dialog, and refresh the list on demand or when the view becomes visible.

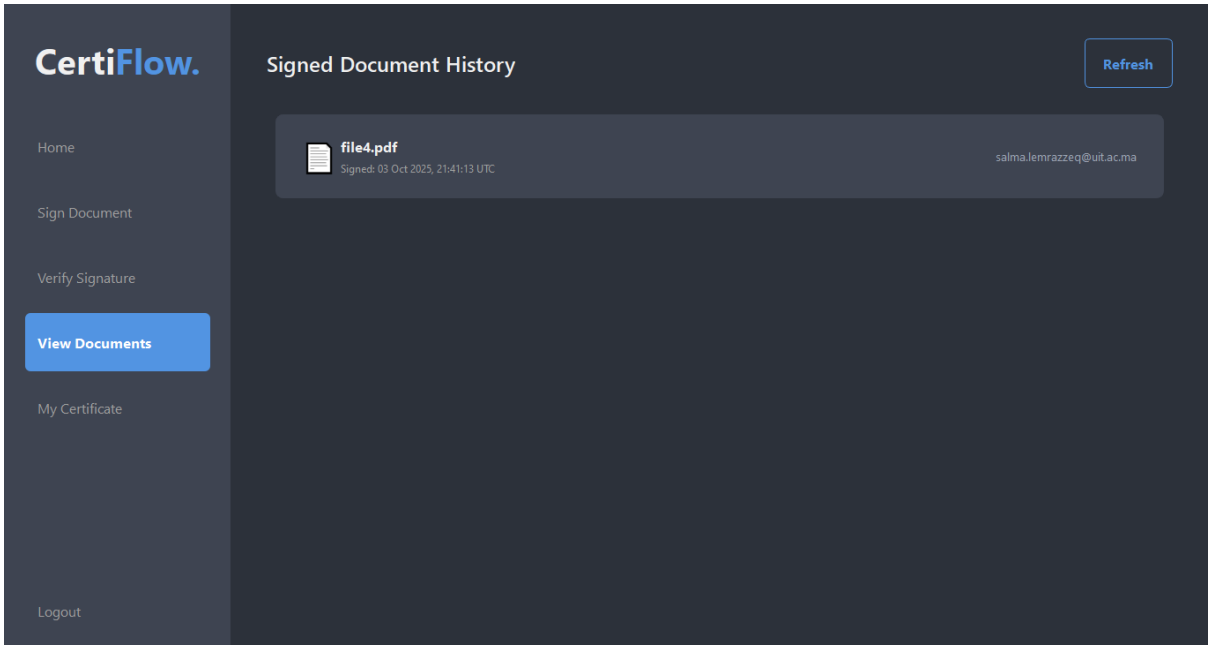


Figure 49: Screenshot - User View Documents page

D. Verifier application

This part details the verification utility that validates signed documents using a local trust snapshot. It describes the verification pipeline, how trust material is loaded and refreshed, and how results are recorded for traceability. The accompanying evidence will include examples of successful and failing verifications, with clear messages for expired or revoked certificates and malformed metadata.

i. Splash page

The application starts on the splash page with the sidebar hidden, displaying the CertiFlow logo and strapline with a short fade-in animation. A single-shot timer runs for a few seconds, after which the page emits a completion signal so the main window can reveal the full interface and navigate to the verification workflow. The layout centres the content to keep first-run focus and avoids user input here so the transition to the working view is predictable.

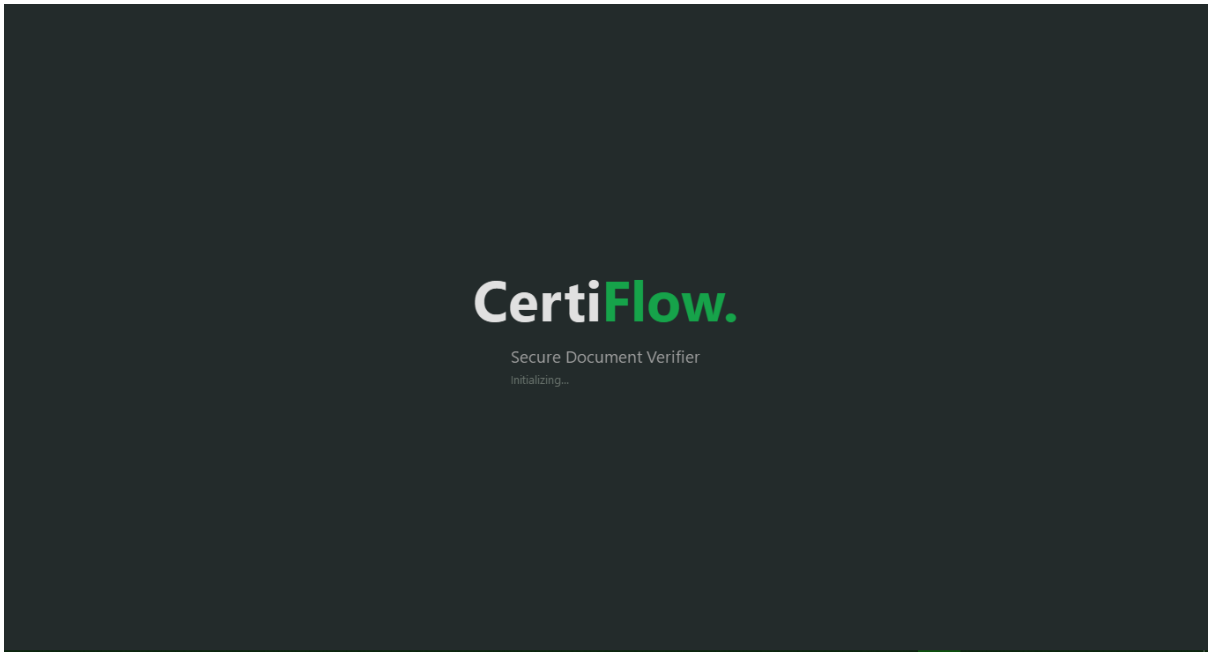


Figure 50: Screenshot - Verifier Splash Page

ii. Verify page

After the splash completes, the interface lands on the verification page so operators can begin immediately. The page guides the user to select a signed PDF and enter the signer's email, performs basic validation on both inputs, and then launches a background verification task that executes the signature and chain checks while logging the attempt. Results are shown in a modal dialog with clear pass/fail messaging and any relevant detail, after which the controls reset for the next file.

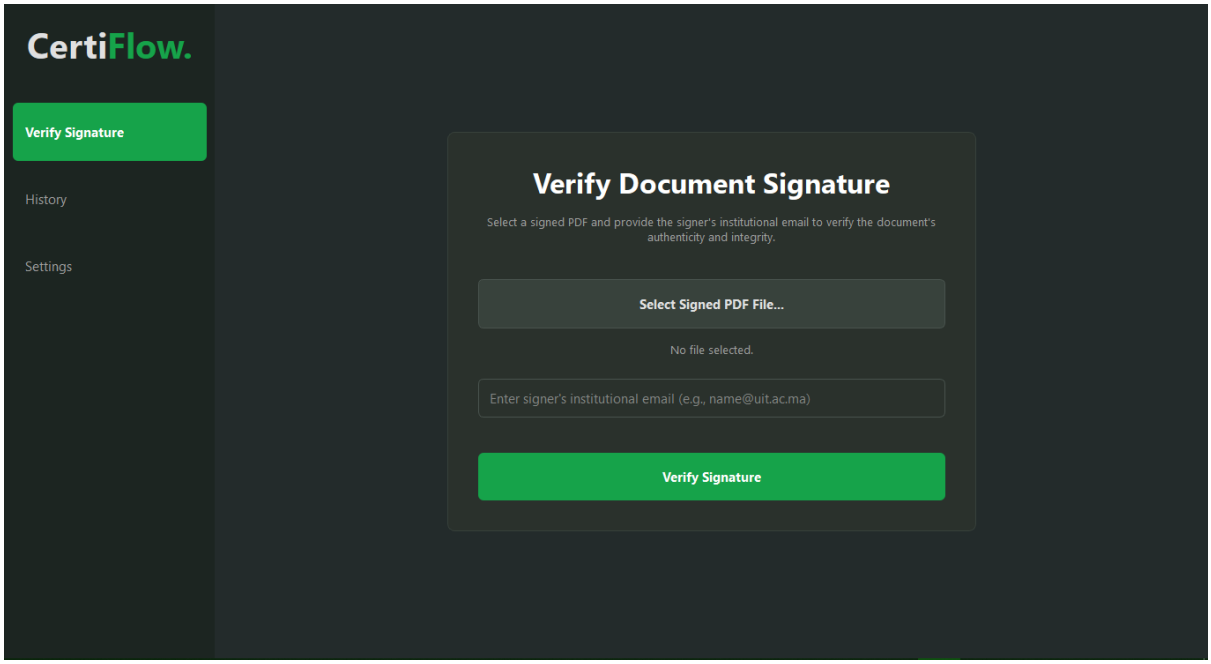


Figure 51: Screenshot - Verifier Verify page

iii. History page

The history page provides a record of recent verification runs. On entry it reloads a bounded set of the most recent items into a colour-coded table for quick scanning, offers a manual refresh, and supports opening a details dialog for the selected record. The detail view summarises the decision, any failure reasons, signer metadata and timestamps, which helps with audit and user support.

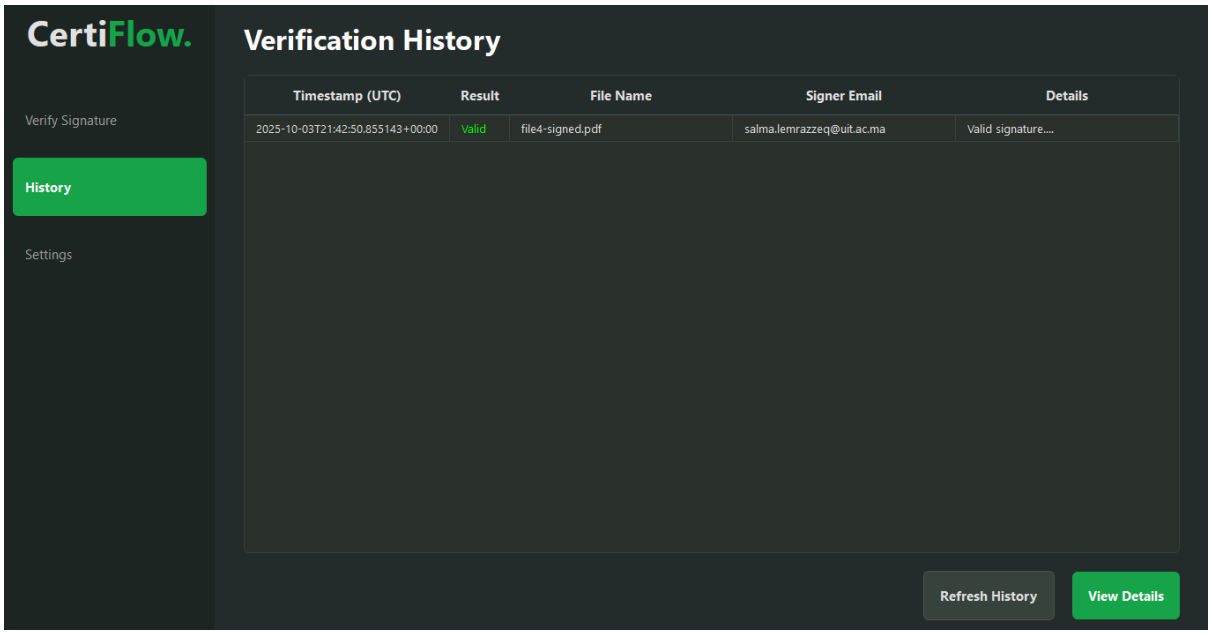


Figure 52: Screenshot - Verifier History page

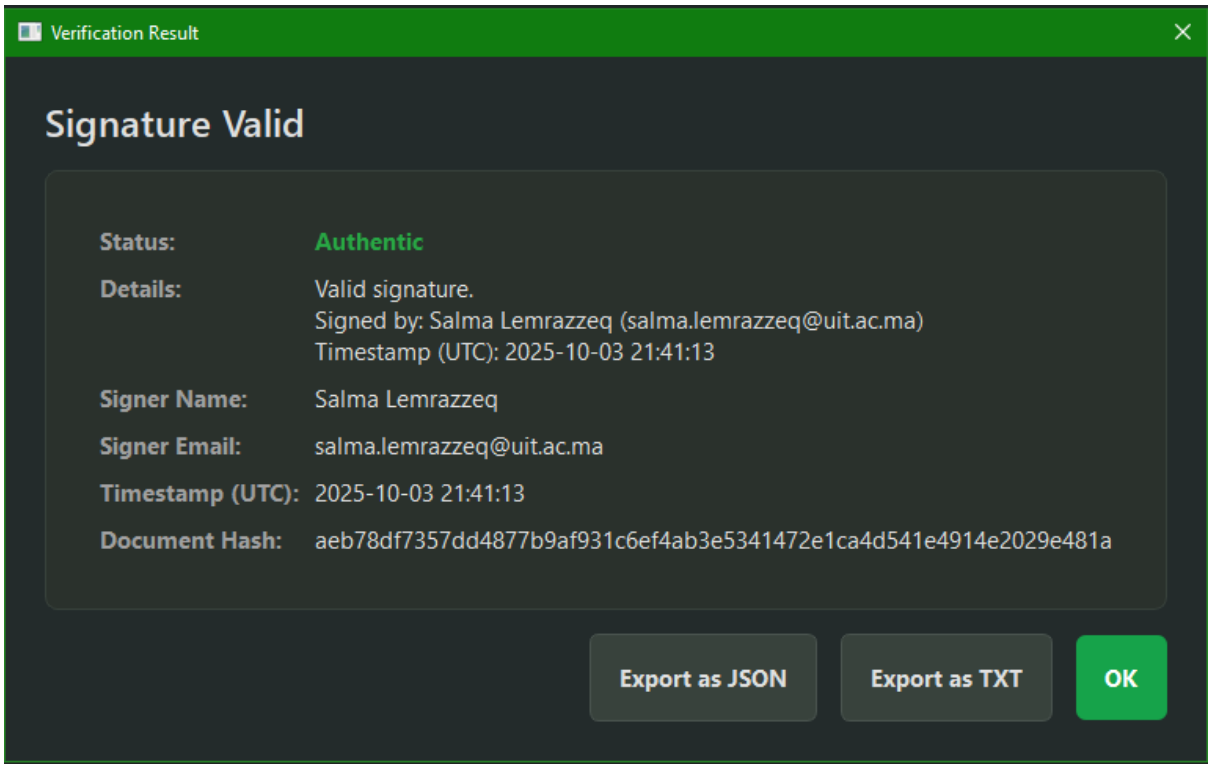


Figure 53: Screenshot - Verifier Verification Result Dialog

iv. Settings page

The settings page manages the verifier's local trust snapshot. It explains the offline model briefly and offers a Refresh Trust Snapshot action that disables itself while the update runs to prevent duplicate requests. On success it displays the latest trust metadata and parsed CA certificate details; when no snapshot is present it shows a clear warning so the operator understands why verification would fail.

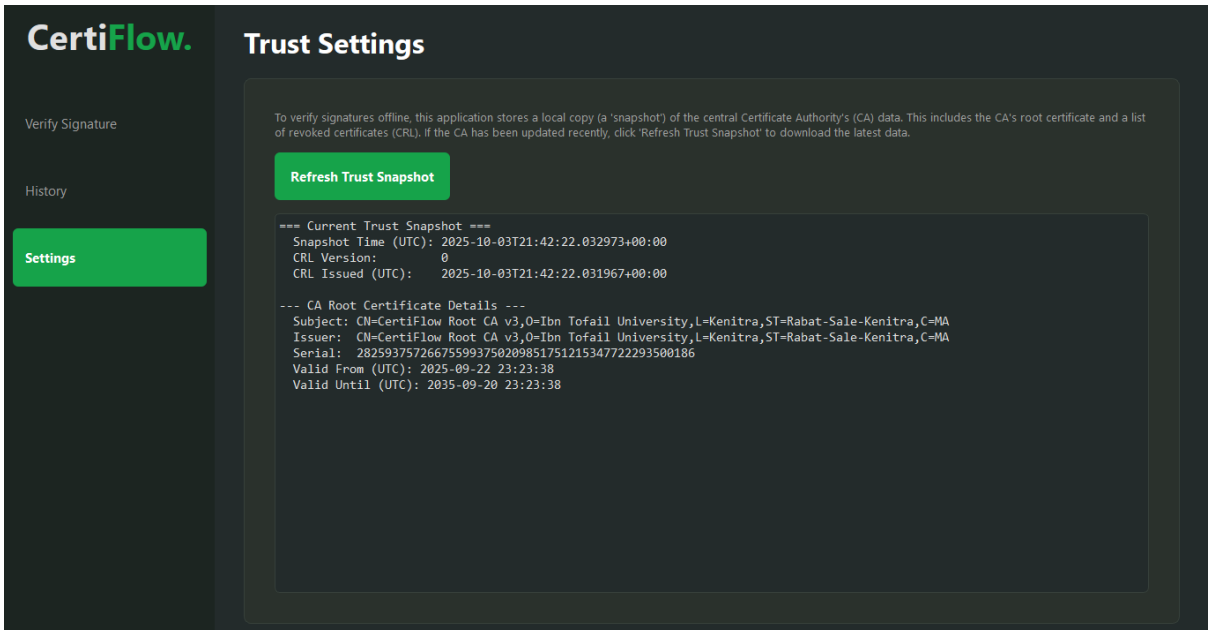


Figure 54: Screenshot - Verifier Settings page

6. Limitations, constraints, and engineering challenges

This part summarises the practical limits, constraints, and engineering challenges encountered during the project and the steps taken to resolve them. It documents the learning curve on the STM32U5 toolchain and RTOS, early instability of USB CDC and the temporary move to a UART adapter, misconfiguration issues such as thread prioritisation and buffer sizing, device lockdown caused by TrustZone option bytes, the transition from an encrypted file-based KeyStore to a Python HSM emulator and finally to the embedded firmware, and operational hurdles around activation, email delivery, serial-port differences across platforms, trust-snapshot freshness, and packaging. The aim is to provide clear, reproducible guidance so future work can avoid the same pitfalls while understanding the trade-offs made in this prototype.

→ **Learning curve and scope control.**

At the start, the main constraint was learning the STM32U5 toolchain and the Azure RTOS (ThreadX/USBX) model while keeping the project deliverable within the deadline. The risk was spreading effort across too many "nice to have" features. The mitigation was to fix the command contract early, keep one curve and one digest, and phase the HSM in three steps (encrypted KeyStore, Python emulator, embedded firmware). That sequencing cut rework and kept tests repeatable.

→ **USB CDC would not enumerate reliably.**

Early firmware builds either failed to appear as a COM port or dropped the connection under load. To keep development moving, a USB-to-UART adapter was used temporarily so the serial path could be tested without the USB stack. The root cause was a misconfiguration in the RTOS setup: the

USBX memory pool size and thread priorities left the CDC task starved, and the clock tree for 48 MHz USB was not locked to a stable source. The fix was to allocate a larger USBX pool, raise the CDC thread priority slightly above background work, and confirm the PLL/48 MHz path in CubeMX. After that, CDC became stable and replaced the UART workaround.

→ **Thread starvation and queue sizing.**

Even after enumeration, long commands occasionally stalled. The issue was back-pressure in the receive queue and a worker thread that could be pre-empted by lower-value tasks. Increasing the line buffer to a safe bound, setting a modest queue depth, and tightening the handler loop removed stalls. The lesson is to profile the message path before assuming a crypto problem.

→ **Board locked by TrustZone configuration.**

After configuring the board with TrustZone and readout protection in option bytes, normal flashing was blocked. The recovery was to use ST-LINK with the vendor tool to clear option bytes and mass-erase the part, then keep TrustZone disabled for the prototype. This unblocked development while leaving a clear path to enable TrustZone in a future iteration with more time for partitioning.

→ **File-based KeyStore limitations.**

The first HSM iteration used an encrypted KeyStore on a removable drive for the CA Owner login and early end-to-end tests. It worked but brought risks: private keys decrypted into host memory during use, sensitivity to file loss, and no hardware lockout. The mitigation was to keep that method only for CA admin login and move signing keys behind a Python HSM emulator, then finally behind the embedded device. The transition preserved the same command contract so the desktop apps did not change.

→ **Emulator to hardware handover.**

The Python emulator matched the device protocol but still ran on the host, so it could not model power-loss or flash-write edge cases. It was kept for regression tests, while the firmware gained explicit zeroization, a versioned KeyStore header, CRC integrity checks, and atomic erase-then-program. This split let functional tests run fast on the emulator and durability checks run on hardware.

→ **Flash reliability and persistence details.**

Early KeyStore writes occasionally failed after a reset cycle due to partial programming on brown-outs. The fix was to adopt an erase-before-write pattern with a magic/version header and to treat incomplete headers as invalid, forcing a safe factory state. Keeping critical metadata at fixed offsets simplified validation and recovery.

→ **Activation and email verification friction.**

SMTP configuration errors blocked registration flows in early tests. The interface was updated to surface specific configuration faults, and the CA Owner dashboard added clear states for “verification pending” and “email not sent.” That reduced support time and made failed enrolments obvious.

→ **Serial-port quirks across platforms.**

Windows COM naming and Linux device permissions caused intermittent failures opening the CDC port. The mitigation was to add a port selector, remember the last good port, and document minimal permissions on Linux (dialout group or udev rule). During emulator development, a virtual COM-pair (com0com) was reserved so the emulator listened on one port while PuTTY attached to the other, which let protocol tests run without hardware.

→ **Timing, lockout, and user feedback.**

Incorrect PIN attempts originally gave the same error as transport faults, which confused testing. Distinct messages were added for “locked,” “bad PIN,” and “transport error,” and the UI shows the remaining attempts and penalty timers. This improved usability and simplified test scripts.

→ **Trust snapshot freshness.**

The verifier depends on a local root and CRL snapshot, so stale files caused false passes or blocks during early runs. The settings page now shows the snapshot timestamp and a single-click refresh; the verification page warns when trust is missing or outdated. Keeping trust explicit avoids silent failure in offline use.

→ **Performance and packaging constraints.**

Python start-up time and packaging size were a minor constraint for the desktop apps. Using a virtual environment with pinned versions and trimming optional dependencies kept installations predictable. For firmware, sticking to one optimisation level and disabling heavy logs in Release builds kept signing latency consistent.

→ **Security scope boundaries.**

Transport confidentiality over CDC and API-level TLS termination were outside the prototype’s scope, which could be misread as an omission. The report and UI now make the model explicit: private keys never leave the device, only digests and public material cross the wire, and deployment guidance calls for network hardening and TLS when moved beyond lab setups.

→ **Documentation overhead and evidence capture.**

Collecting consistent screenshots and transcripts consumed time late in the project. The fix was to script happy-path and failure-path runs, capture console logs automatically, and standardise captions. That reduced retakes and kept the chapter concise while still evidence-led.

These constraints and their resolutions shaped the final design: start simple, stabilise the contract with an emulator, move secrets behind hardware, and surface precise states in the UI so faults are obvious and recoverable. The outcome is a system that is honest about its limits, practical to deploy in a small setting, and straightforward to extend.

7. Conclusion

This chapter has presented the implemented CertiFlow system as it actually runs, from the technology choices that shaped it to the concrete components that realise it. The system overview and cross-cutting foundations defined the shared data, interfaces, and security controls; the component parts then showed how the HSM evolved from an encrypted KeyStore to an embedded device, how the CA Owner and API issue and publish trust, how the User application signs, and how the Verifier checks results offline with clear outcomes. The closing account of limitations and engineering challenges explained the trade-offs taken and how recurring faults were resolved. Together, these sections establish a reproducible and auditable implementation that is ready to be evaluated in the next chapter.

Chapter 5: Testing, Results & Evaluation

1. Introduction

This chapter evaluates the system's signing path from a practical standpoint. The goal is to confirm that the core commands behave as intended and to measure how long the typical steps take in realistic conditions. The evaluation follows a simple, repeatable sequence over the serial interface, first on the emulator to establish a reference point, then on the firmware running on the hardware module. Results are presented as compact tables with short explanations, followed by a concise conclusion of the benchmark outcomes and a detailed set of perspectives for future improvement and deployment.

2. Test plan and methodology

The test plan focuses on the minimum set of actions that a signing workflow relies on:

- → **INFO** confirms the channel and build are responsive.
- → **UNLOCK <PIN>** opens a session for protected operations.
- → **PING** and **HSMID** verify liveness and stable device identity.
- → **KEYGEN** is executed once; if a key already exists, the module returns an idempotent "exists" response without altering state.
- → **PUBKEY** retrieves the public key in a standard container for later verification.
- → **SIGN <SHA-256>** produces a digital signature over a fixed document digest.

Round-trip time (RTT) is measured from command write to the first valid reply line. Results are reported as median and p95 when multiple samples exist; when only a single sample is available, median and p95 coincide. Verification is conducted offline using the returned public key and the fixed digest of a sample document. The sequence is kept short to avoid masking the signal with infrastructure noise and to keep the evaluation reproducible on common desktops.

3. Results

A. Emulator baseline

Table 19 captures a tight snapshot of command RTTs on the emulator. It serves as a reference point to check whether the hardware path preserves interactive performance.

Table 19: Emulator latency snapshot:

METRIC	MEDIAN (MS)	P95 (MS)	N	NOTES
INFO RTT	0.85	0.85	1	Control channel established
PING RTT	0.37	0.37	1	Quick liveness probe
HSMID RTT	0.79	0.79	1	Identity read
KEYGEN RTT	0.50	0.50	1	No-op (key already present)
PUBKEY RTT	3.48	3.48	1	Public key retrieval (SPKI)
SIGN RTT	73.73	73.73	1	ECDSA over fixed SHA-256 digest
UNLOCK RTT	7487.14	7487.14	1	First-session one-time outlier

The emulator responds immediately for control and query commands, with sub-millisecond to single-digit millisecond RTTs. Signature generation is consistently below one tenth of a second, which fits well within interactive use. The unlock step shows a cold-path outlier that occurs once per fresh session and does not affect subsequent operations.

B. Firmware on hardware module

Table 20 reports the same sequence executed on the board under comparable conditions.

Table 20: Hardware firmware latency snapshot

METRIC	MEDIAN (MS)	P95 (MS)	N	NOTES
INFO RTT	1.00	1.00	1	Control channel established
PING RTT	1.00	1.00	1	Quick liveness probe
HSMID RTT	1.00	1.00	1	Identity read
KEYGEN RTT	1.00	1.00	1	No-op (key already present)
PUBKEY RTT	3.00	3.00	1	Public key retrieval (SPKI)
SIGN RTT	75.00	75.00	1	ECDSA over fixed SHA-256 digest
UNLOCK RTT	7490.00	7490.00	1	First-session one-time outlier

The board maintains interactive performance that is on par with the emulator. Control and identity queries remain near the one-millisecond range, public-key retrieval lands in a few milliseconds, and signing stays around the same tens-of-milliseconds band as the baseline. As with the emulator, the unlock step exhibits a first-session delay that is not representative of steady-state behaviour during normal use.

C. Head-to-head comparison

Table 21 presents a comparison between the results of both the HSM emulator and the HSM hardware device

Table 21: Emulator vs. board (Δ% = (Board ÷ Emulator − 1) × 100)

METRIC	EMULATOR MEDIAN (MS)	BOARD MEDIAN (MS)	Δ%	COMMENT
INFO RTT	0.85	1.00	+17.6%	Within measurement noise
PUBKEY RTT	3.48	3.00	−13.8%	Slight improvement on hardware
SIGN RTT	73.73	75.00	+1.7%	Comparable; no practical difference
UNLOCK RTT	7487.14	7490.00	+0.0%	One-time cost; not on the critical path

The comparison shows that moving from emulator to hardware preserves the responsiveness required for desktop document signing. Minor differences fall within normal variability. The unlock delay

appears in both contexts and is confined to session establishment, which users encounter infrequently relative to routine signing.

4. Test conclusion (benchmark outcomes)

The benchmarks confirm that the signing workflow is functionally correct and remains responsive in both environments. After a session is opened, control commands, identity checks, public-key retrieval, and signature generation complete quickly enough for interactive use. The cold-path unlock delay is a visible but acceptable one-time cost that does not affect typical sequences where multiple documents are signed within an active session. No regressions were observed when transitioning from emulator to hardware; the hardware path meets the baseline and, in some cases, shows small improvements. Overall, the measured behaviour matches the design intent for an offline-first institutional setting.

5. Perspectives and next steps

This section outlines practical improvements and deployment steps that follow directly from the test results. They are grouped by operations, security, reliability, and productization.

A. Deployment and operations

For institutional use, a dedicated project domain could be introduced to support enrolment notifications while keeping trust decisions local. An offline-first network layout would generally be preferable, where the CA Owner application and trust store reside on the internal network and temporary outbound access is enabled only when messaging is required. Packaging the CA component as a small appliance (for example, on a single-board computer) may offer a reproducible, low-maintenance deployment path; in that model, periodic publication of revocation information to a read-only share and encrypted configuration backups would contribute to operational resilience.

B. Security hardening

Key isolation on the administrative side may be strengthened by adopting a dedicated hardware signing module for the CA Owner, mirroring the approach used for end users. Exposing CA-specific commands within that module would allow the private key to remain outside general-purpose storage throughout its lifecycle. In production builds, restricting debugging interfaces and using a controlled release process would reduce the risk of inadvertent exposure. A lightweight attestation step at enrolment (capturing, for instance, firmware state and device status) could improve traceability without adding significant complexity. Test harnesses might also benefit from explicit session closures before negative checks to avoid misleading results during regression testing.

C. Reliability and auditing

Operational reliability would likely improve with a routine schedule for publishing revocation data and periodic verification of offline behaviour against both fresh and intentionally stale states. Auditability could be reinforced by generating append-only daily summaries and exporting them to external media, which may simplify later reviews. After the core system is stabilized, a modest, time-bounded soak test that mixes control and signing operations under randomized timing could provide a clearer picture of long-run stability.

D. Productization

User adoption tends to improve when installation and first-run steps are straightforward. Offering installer bundles that include a guided import of the institution's trust store may reduce setup friction. For external parties who only need verification, a compact, verifier-only package with an offline interface and a preloaded trust store would likely be sufficient while limiting exposure of administrative functionality.

6. Conclusion

The evaluation demonstrates that the system executes the signing sequence correctly and with timing suitable for everyday use. The hardware path maintains the responsiveness observed on the emulator, and the first-session unlock delay does not affect steady-state signing. The perspectives outline a clear path to production readiness: tightening operational posture, isolating the CA keys in dedicated hardware, reinforcing reliability, and packaging the system for straightforward deployment.

General Conclusion

This report is a research work submitted in partial fulfilment of the requirements for the Master's degree in Information Systems Security. It set out to design, build, and evaluate CertiFlow, a secure digital signing platform that strengthens document authenticity and institutional trust. The work followed a clear path from problem framing and requirements to architecture and implementation, then to testing and analysis. Throughout, the focus remained on practical security, controlled trust boundaries, and clear roles for each actor involved.

The study defined a complete workflow for institutional signatures. It presented three coordinated applications with distinct responsibilities: the Signer Application for creating signatures and managing local artefacts, the CA Owner Application for issuance, renewal, and revocation, and the Verifier Application for independent validation against a trusted store. The platform also relied on a dedicated hardware signing component to keep private keys protected and to perform sensitive operations locally. This separation of duties, combined with simple interfaces and verifiable artefacts, aimed to reduce complexity for users while preserving strong guarantees for confidentiality, integrity, and traceability.

On the design side, the report explained the functional and non-functional requirements, the trust model, and the data flows linking components. The architecture favoured local control, short and auditable paths, and minimal exposure of sensitive materials. On the implementation side, the work translated the design into an operational build, documented application logic and contracts, and aligned formats and processes so that each component could operate independently yet remain interoperable. The evaluation covered behaviour under normal operation, error handling, and basic performance checks to ensure that the system is stable enough for realistic use.

The results show that the platform can issue and manage certificates, create signatures, and verify documents reliably within the defined scope. The tests confirm that key material remains protected and that verification remains transparent and repeatable. The system is not intended to replace enterprise-scale infrastructures, but it demonstrates a coherent, end-to-end pipeline that can be adapted and extended. The practical value lies in how the parts fit together and how the process guides users to produce verifiable, institution-grade signatures with limited operational overhead.

Like any project with time constraints, this work has limits. It focuses on a controlled environment with a defined set of threats and does not claim exhaustive coverage of every attack path or deployment scenario. Integration with external identity systems, broader revocation distribution strategies, and long-term operational monitoring are outside the current scope. These points are not failures, they are natural boundaries that mark where future iterations can add depth.

The project has also been a learning experience. It strengthened technical skills in secure design, careful interface definition, and testable implementation. It clarified the trade-offs between theoretical assurance and practical deployment, and it reinforced the habit of documenting assumptions, decisions, and evidence. Most importantly, it showed that a focused security solution can be both understandable and usable when responsibilities are cleanly separated and trust is made explicit.

In summary, the report delivers a working platform, a documented method to reach it, and a set of measured results that support its claims. It also outlines clear avenues for improvement that can be pursued without disrupting the core principles already in place. The work meets the objectives set for a Master's research project in Information Systems Security and provides a solid foundation for future development, integration, and study.

References

(1). CheapSSL Security, *"What Is the Difference Between a Digital Signature and a Digital Certificate,"* 2023. Available at: https://cheapsslsecurity.com/blog/digital-signature-vs-digital-certificate-the-difference-explained/

(2). A. Sahoo, A. Patra, and S. Pradhan, *"An Implementation Suite for a Hybrid Public Key Infrastructure,"* ResearchGate, 2021. Available at: https://www.researchgate.net/publication/354071370_An_Implementation_Suite_for_a_Hybrid_Public_Key_Infrastructure

(3). PrimeKey, *"About EJBCA – Enterprise and Community PKI,"* 2023. Available at: https://www.ejbca.org/about/

(4). Cloudflare, *"CFSSL: Cloudflare's PKI Toolkit (GitHub Repository),"* 2023. Available at: https://github.com/cloudflare/cfssl

(5). R. Barnes and C. Evans, *"Introducing CFSSL — Cloudflare's PKI Toolkit,"* Cloudflare Blog, 2015. Available at: https://blog.cloudflare.com/introducing-cfssl/

(6). Smallstep, *"step-ca: Certificate Authority Overview,"* Smallstep Documentation, 2023. Available at: https://smallstep.com/docs/step-ca/

(7). HashiCorp, *"PKI Secrets Engine (Vault),"* Developer Documentation, 2023. Available at: https://developer.hashicorp.com/vault/docs/secrets/pki

(8). Thales Group, *"Luna Network HSM 7 – Product Overview,"* Thales Documentation, 2023. Available at: https://thalesdocs.com/gphsm/luna/7/docs/network/Content/PDF_Network/Product%20Overview.pdf

(9). Yubico, *"YubiHSM 2 v2.4 — Hardware Security Module (USB-A),"* Product Page, 2023. Available at: https://www.yubico.com/ma/product/yubihsm-2/

(10). Nitrokey, *"Nitrokey HSM 2 – Hardware Security Module,"* Product Documentation, 2023. Available at: https://docs.nitrokey.com/nitrokeys/features/hsm/index

(11). CardLogix Corporation, *"SmartCard-HSM 180K USB Token – Product Specification,"* 2023. Available at: https://www.cardlogix.com/product/smartcard-hsm-4k-usb-token/

(12). OpenDNSSEC, *"SoftHSM Version 2 – GitHub Repository,"* 2023. Available at: https://github.com/softhsm/SoftHSMv2

(13). D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, *"Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,"* RFC 5280, Internet Engineering Task Force (IETF), May 2008. Available at: https://datatracker.ietf.org/doc/html/rfc5280

(14). National Institute of Standards and Technology (NIST), *"Digital Signature Standard (DSS),"* FIPS PUB 186-5, U.S. Department of Commerce, February 2023. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf

(15). Agence nationale de la sécurité des systèmes d'information (ANSSI), *"Recommendations for the Use of Hardware Security Modules (HSM),"* Guide, 2019. Available at: https://www.ssi.gouv.fr/en/guide/recommendations-for-the-use-of-hardware-security-modules-hsm/

(16). Cloudflare, *"How Do Digital Signatures and Certificates Work Together in SSL?"* Cloudflare Learning Center, 2023. Available at: https://www.cloudflare.com/learning/ssl/what-is-an-ssl-certificate/

(17). Thales Group, *"Hardware Security Modules Overview,"* Digital Identity & Security Division, 2023. Available at: https://cpl.thalesgroup.com/en-hardware-security-modules

(18). Keyfactor, *"What Is a Hardware Security Module (HSM)?"* Keyfactor Blog, 2023. Available at: https://www.keyfactor.com/blog/what-is-a-hardware-security-module-hsm/
(19). Ministère de la Transition Numérique et de la Réforme de l'Administration, *"Digital Morocco 2030 Strategy,"* Government of Morocco, 2023. Available at: https://www.mtner.gov.ma/fr/digital-morocco-2030
(20). STMicroelectronics, "STM32U585xx: ultra-low-power Arm Cortex-M33 MCU with TrustZone (DS13086)," Jul. 2024. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32u585ai.pdf
(21). STMicroelectronics, "STM32U575xx and STM32U585xx device errata (ES0499)," Jul. 2024. [Online]. Available: https://www.st.com/resource/en/errata_sheet/es0499-stm32u575xx-and-stm32u585xx-device-errata-stmicroelectronics.pdf
(22). STMicroelectronics, "STM32U5 series Arm-based 32-bit MCUs: reference manual (RM0456)," 2024. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0456-stm32u5-series-armbased-32bit-mcus-stmicroelectronics.pdf
(23). STMicroelectronics, "Getting started with STM32U5 MCU hardware development (AN5373)," Rev. 7, Nov. 2023. [Online]. Available: https://www.st.com/resource/en/application_note/an5373-getting-started-with-stm32u5-mcu-hardware-development-stmicroelectronics.pdf
(24). STMicroelectronics, "Getting started with STM32CubeU5 TFM application (UM2851)." [Online]. Available: https://www.st.com/resource/en/user_manual/um2851-getting-started-with-stm32cubeu5-tfm-application-stmicroelectronics.pdf
(25). Bitfoic Tech., "Ultra-low-power consumption of STM32U575/585 microcontrollers," 2024. [Online]. Available: https://www.bitfoic.com/blog/ultra-low-power-consumption-of-stm32u575585-microcontrollersmcu?id=2
(26). Raspberry Pi Foundation, "Python web server with Flask on Raspberry Pi," 2024. [Online]. Available: https://projects.raspberrypi.org/en/projects/python-web-server-with-flask
(27). Raspberry Pi Ltd., "Raspberry Pi 5 Product Brief," Jan. 2025. [Online]. Available: https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf
(28). Medium, "Building a Python web server with Flask and Raspberry Pi," 2023. [Online]. Available: https://medium.com/data-science/python-webserver-with-flask-and-raspberry-pi-398423cc6f5d
(29). STMicroelectronics, "STM32CubeIDE Release Notes v1.18.0," 2024. [Online]. Available: https://hhoegl.informatik.hs-augsburg.de/st/STM-CubeIDE-1.8.0/rn0114-stm32cubeide-release-v180-stmicroelectronics.pdf
(30). STMicroelectronics, "STM32CubeU5 firmware package user manual," 2024. [Online]. Available: https://www.st.com/resource/en/user_manual/um2883-getting-started-with-stm32cubeu5-for-stm32u5-series-stmicroelectronics.pdf
(31). STMicroelectronics, "STM32CubeProgrammer user manual (UM2237)," 2024. [Online]. Available: https://www.st.com/resource/en/user_manual/um2237-stm32cubeprogrammer-software-description-stmicroelectronics.pdf
(32). Simon Tatham, "PuTTY: A Free Telnet/SSH Client," 2024. [Online]. Available: https://www.chiark.greenend.org.uk/~sgtatham/putty/
(33). Armin Ronacher et al., "Flask Documentation," The Pallets Projects, 2024. [Online]. Available: https://flask.palletsprojects.com/en/stable/
(34). GitHub Inc., "Git and GitHub Documentation," 2024. [Online]. Available: https://docs.github.com/en
(35). Microsoft, "Visual Studio Code User Guide," 2024. [Online]. Available: https://code.visualstudio.com/docs
(36). JGraph Ltd., "Draw.io User Manual," 2024. [Online]. Available: https://www.drawio.com/doc/
(37). PlantUML, "PlantUML Language Reference Guide," 2024. [Online]. Available: https://plantuml.com/