Politecnico di Torino

Politecnico di Torino

Master of science in Computer Engineering

A.Y. 2024/2025

Graduation Session October 2025

EduWallet

A Blockchain-Enabled Digital Wallet for Managing University Course Credits

Supervisor:
Valentina Gatteschi

Candidate:
Diego Da Giau

Abstract

While the world is rapidly transitioning from traditional Web 2.0 systems, dominated by large companies monetizing centralized data, toward Web3 solutions built on blockchain's decentralized nature, the academic sector remains largely reliant on centralized databases and static documents, whether digital or on paper. This outdated approach hampers the sharing of academic records between institutions, which often requires the production, validation, and verification of documents, as well as agreement on common formats.

This work proposes a solution to modernize academic records management. By leveraging blockchain and Ethereum smart contracts, we lay the foundation for a system based on tamper-proof technologies. Through the adoption of account abstraction, we are able to develop a user-friendly solution that enables gasless interactions for users. EduWallet relies on a Software Development Kit that allows universities to interact with on-chain functionalities, and a browser extension that enables students to manage their data. A key feature of the proposed solution is full data ownership, which is entirely granted to students, who control access to their academic records. Additionally, a decentralized storage system is used to store students' certificates, minimizing the need for on-chain data storage.

EduWallet aims to provide a comprehensive environment within academic institutions, allowing universities to streamline bureaucratic processes and align with the latest Web3 standards. The proposed solution also serves as a practical example of how to effectively integrate on-chain and off-chain components into a cohesive system.

Sammendrag

Samtidig som verden raskt beveger seg bort fra tradisjonelle Web 2.0-systemer, dominert av store selskaper som tjener på sentraliserte data, og over til Web3-løsninger basert på *blockchain*-teknologiens desentraliserte natur, er akademia fortsatt i stor grad avhengig av sentraliserte databaser og statiske dokumenter, enten digitale eller fysiske. Denne utdaterte tilnærmingen vanskeliggjør deling av akademiske opplysninger mellom institusjoner, og medfører ofte behov for produksjon, validering og verifisering av dokumenter, samt enighet om felles formater.

Dette arbeidet foreslår en løsning for å modernisere håndteringen av akademiske opptegnelser. Ved å ta i bruk *blockchain* og smartkontrakter på Ethereum legges det til rette for et system basert på manipulasjonssikre teknologier. Gjennom bruk av kontoabstraksjon utvikles en brukervennlig løsning som muliggjør gassfrie interaksjoner for sluttbrukeren. EduWallet benytter et *Software Development Kit* som lar universiteter samhandle med blokkjedefunksjonalitet, samt en nettleserutvidelse som gjør det mulig for studenter å administrere sine egne data. En sentral egenskap ved løsningen er at studentene har fullt eierskap til sine data, og selv kontrollerer tilgangen til sine akademiske opplysninger. I tillegg benyttes et desentralisert lagringssystem for oppbevaring av studentenes sertifikater, noe som reduserer behovet for datalagring direkte på blokkjeden.

EduWallet har som mål å tilby et helhetlig digitalt miljø for akademiske institusjoner, slik at universiteter kan effektivisere byråkratiske prosesser og tilpasse seg moderne Web3-standarder. Den foreslåtte løsningen fungerer også som et praktisk eksempel på hvordan man kan integrere *on-chain*- og *off-chain*-komponenter i et sammenhengende system.

Contents

Figures

Tables

Code Listings

Acronyms

API Application Programming Interface . xvi, 16–18, 42, 43, 48, 49, 62, 101, *Glossary:* Application Programming Interface

AWS Amazon Web Services. xvi, 48, 62, 101

BTC Bitcoin. 4, 5, 23

CID Content Identifier. 6, 26, 48, 49, 62, 63

CLI Command Line Interface. xi, 19, 36, 50–53, 59, 65, 75

dApp decentralized application. 5, 6

DCC Digital Credential Consortium. 7–9

ECTS European Credit Transfer and Accumulation System. 26, 38, 44, 52, 53

EOA Externally Owned Account. 5, 23–25, 31–33, 37, 41–47, 52, 56–59

ETH Ether. xix, 4, 5, 23, 31, 58, 66, 71

EVM Ethereum Virtual Machine. 5, 23

EW EduWallet. 1, 2, 9, 11, 12, 15–19, 21, 23, 24, 27, 28, 30, 31, 33, 35, 36, 40, 42, 49, 50, 52, 55, 60, 62, 65, 66, 69, 71, 72

FR functional requirement. 15–17, 26, 29, 31, 32, 35, 36, 53, 60, 65, 69, 71

ID identifier. 32, 37, 43–45

IPFS InterPlanetary File System. xv, xvi, 6, 48, 49, 62, 101

IT Information Technology. 36, 43

LMS Learning Management System. 8, 16–19, 36, 50, 52

MIT Massachusetts Institute of Technology. 7, 8

NFR non-functional requirement. 15, 17, 18, 23, 33, 36, 42, 44, 47, 48, 69, 71

PDF Portable Document Format. 63

QR Quick Response. 8, 9, 72

SCA Smart Contract Account. 21, 23–25, 29, 32, 37, 41–47, 52, 53, 56–58

SDK Software Development Kit. xv, 19, 21, 27, 29, 30, 32, 36, 42–48, 50, 52, 61, 62, 65, 75

UI User Interface. 37, 42, 65, 72

URL Uniform Resource Locator. xv, 36, 40, 48, 62

UX User Experience. 8, 9, 11, 50, 72

WWW World-Wide Web. 3

Glossary

Application Programming Interface A set of rules and protocols that allow different software applications to communicate with each other, enabling them to exchange information or functionalities. xvii, 16

double-spending The double-spending problem refers to the malicious act of spending the same digital currency multiple times. In digital systems without adequate safeguards, this vulnerability could allow users to duplicate funds and undermine the integrity of the monetary system. 4

hash In computer science, a hash function is an algorithm that takes an input of arbitrary length and produces a fixed-size output, commonly referred to as a hash or digest. xv, 4, 6, 49, 60, 61

Keccak-256 A cryptographic hash function that belongs to the Keccak family. It produces a fixed 256-byte hash. 60, 61

mempool Short for memory pool. In general computing, it refers to a temporary memory area used to manage pending operations. In the context of blockchain and account abstraction, a mempool is a temporary storage area to hold UserOerations that have been broadcast to the network but are not yet included in a block. 24

nonce A nonce is a number that can be used only once in a cryptographic communication. In Ether, the nonce represents the number of transactions sent by a specific address and is used to ensure the uniqueness and validity of each transaction. 56

peer-to-peer A peer-to-peer network is a distributed system architecture in which all participating nodes, called peers, have equal authority and responsibility. 4, 6

salt A salt is a random sequence of data used in cryptography. It is added to a password before hashing, with the purpose of making the resulting hash unique, even if two users have the same password. 37

Chapter 1

Introduction

We are currently living in a globalized world, where distance is increasingly irrelevant. People move freely across borders, and thanks to digital technologies, everyone is interconnected and can access services regardless of location. Schools and universities are deeply affected by this globalization, and their internal operations are rapidly evolving. Today, many students receive education from a variety of sources, often through online courses offered by institutions in different countries. A further driver of this change is the rise of academic mobility programs, such as the Erasmus+[1] program, an initiative by the European Union designed to promote student and faculty exchanges and foster intercultural experiences.

However, these opportunities introduce challenges in managing and sharing students' academic records. Typically, each university maintains its own format and system for storing academic data, which creates friction when this information needs to be exchanged. When a student transfers to another institution or participates in an exchange program, they must present verified academic documentation, including diplomas and course transcripts. Currently, students are required to request certified digital or paper documents from their home institution, which must be signed and validated by the university. Then, they submit these to the receiving institution, which must verify the authenticity of both data and signatures. This multi-step process is burdensome for both students and universities, involving significant administrative overhead and document handling.

Given our experience with the Erasmus+ exchange program, and the tedious process of requesting documentation, waiting for it, and resolving discrepancies between universities regarding content and validation, we decided to develop a solution to provide both students and universities with a unified system for storing and sharing academic records. EduWallet (EW) leverages blockchain technology, utilizing its tamper-proof and verifiable nature to allow universities to securely issue academic records and certificates, while enabling students to access them through ownership of an academic wallet in which the records are stored. User interaction is a central focus of this work, as the solution is intended for students of all backgrounds, without requiring specific computer science skills. Therefore,

[1] `https://erasmus-plus.ec.europa.eu`

the system is designed to abstract the complexity of its blockchain-based core.

The remainder of this document is structured as follows: the next chapter, Chapter 2, provides the necessary background information on blockchain and the technologies used within EW. Chapter 3 presents the related work. Based on the context established in the previous chapters, Chapter 4 explains the problem addressed in this thesis and introduces the use case that guided the system design. Chapter 5 outlines the requirements derived from these use cases. The design of the proposed solution is presented in Chapter 6, which gives an overview of the components that build EW, and is further detailed in Chapter 7 and Chapter 8, which describe the on-chain and off-chain elements, respectively. The tools and code used for the development of the system are discussed in Chapter 9. The results are presented and analysed in Chapter 10, focusing on system costs and compliance with the specified requirements. Chapter 11 offers a discussion of the solution and its contributions, along with suggestions for future work. Finally, Chapter 12 concludes the thesis by summarizing the work.

Chapter 2

Background Material

The aim of this chapter is to provide the reader with the fundamentals concepts necessary to understand this work. All the background knowledge presented here is related to emerging decentralized technologies, which are leveraged to modernize the university system in line with Web3 paradigm.

The chapter begins by tracing the evolution of the internet and websites, from the earliest technologies to today's decentralized web. It then delves into blockchain technologies, with a particular focus on one of the most prominent platforms, Ethereum. The final section explores decentralized storage solutions, which serve as a bridge between fully on-chain systems and traditional web-based services.

2.1 From Web 1.0 to Web3

The World-Wide Web (WWW) [1] was launched in 1991 with the publication of the first website[1] by the English computer scientist Tim Berners-Lee. The goal behind the WWW was to create a digital archive of collective human knowledge, accessible to anyone, everywhere. The first iteration of the web is known as Web 1.0, that was composed primarily of static web pages [2], where users' main activities were limited to reading content created by technically skilled individuals or interacting via emails and chat rooms [3]. A major limitation of this model was the stateless nature of protocols like HTTP, which prevented websites from storing or recall user data. As a result, web pages were static and offered limited interactivity, making them unattractive for commercial purpose since monetization was difficult.

Web 2.0 emerged to address these issues by introducing dynamic websites to the WWW. In the early 1990s, Lou Montulli, a developer at Netscape, introduced browser cookies [4], small data files stored by web pages to retain user information. This innovation enabled websites to become dynamic, offering personalized experiences based on user behaviour and preferences. It also paved the way for

[1] `https://info.cern.ch/`

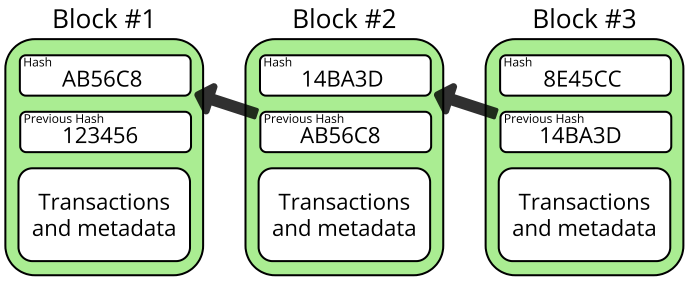


Figure 2.1: Blockchain structure showing the chain of blocks.

monetization, as companies could now track and analyse their data. Web 2.0 gave rise to modern tech giants such as Google and Microsoft, whose business models heavily rely on collecting and leveraging user data. However, this shift led to a new issue: user data became the property of centralized platforms, which could exploit or sell it without users' consent.

Web3 aims to reverse this trend by decentralising data ownership and returning control to the users [5][6]. In the Web3 paradigm, individuals can own their data and choose how it is shared or monetized. The core technology enabling this transformation is the blockchain, a distributed ledger that facilitates interactions with digital services without relying on centralized structures.

2.2 Blockchain

A blockchain [7] is a linked structure composed of data packages called blocks (hence the name blockchain). Each block contains multiple operations, known as *transactions*, along with metadata such as the cryptographic hash of the previous block. This structure links each block to its predecessor, forming the chain depicted in Figure 2.1, ensuring the integrity of the entire ledger. Since each block includes the hash of the previous one, tampering with a block would require altering all the subsequent blocks in the chain, making such modifications computationally infeasible.

Blockchains are typically maintained by a peer-to-peer network [8], where each participant, referred to as *node*, stores a copy of the blockchain. This decentralized replication ensures data availability, fault tolerance, and security. Nodes follow a consensus protocol, which defines the rules by which transactions are validated and blocks are added to the chain.

The first blockchain was introduced in 2008 by Satoshi Nakamoto, the pseudonymous creator (or group of creators) of Bitcoin (BTC) [9][10]. It continues to function as the public ledger for all BTC transactions, effectively solving the double-spending problem without the need for a centralized authority. Another widely known blockchain is Ethereum, which operates alongside its native cryptocurrency, Ether (ETH).

2.3 Ethereum

Ethereum[2] was launched in 2015 based on an idea of Vitalik Buterin [11][12]. Buterin's vision was to enable the deployment of decentralized application (dApp), programs that run autonomously on a blockchain without relying on external, centralized infrastructure. To realize this vision, he and his collaborators created the Ethereum Virtual Machine (EVM), a decentralized virtual machine that executes transactions. In contrast to BTC's limited instruction set, the EVM supports a broad range of operations, including loops and conditional statements. This flexibility makes it possible to develop *smart contracts*, which are programs containing both code and data that execute on the Ethereum blockchain.

When users perform on-chain operations, such as sending cryptocurrency or invoking a smart contract function, they consume computational resources. To compensate nodes for providing these resources and to prevent abuse (for example, infinite loops), Ethereum measures resource usage in units called *gas*. Each individual operation requires a specific amount of gas, and the total *gas fee* for a transaction is calculated by multiplying the gas used by the current *gas price*. Because gas is paid in ETH, the cost of executing an operation varies with its market.

Users hold ETH and pay transaction fees via Externally Owned Account (EOA) [13], which can initiate transactions and serve as the interface between users and the blockchain. An EOA is derived from a private key, which is also required to cryptographically sign transactions. The other account type in Ethereum is the *contract account* [13], which corresponds to a deployed smart contract. Contract accounts can hold cryptocurrency but cannot initiate transactions on their own; they can only send transactions in response to receiving one. Both contract accounts and EOAs are publicly referenced by their address, which serves as the identifier used to send funds or invoke contract functions.

Due to its versatility and the wide range of operations it supports, Ethereum has become the foundation for numerous Layer 2 solutions [14] over time. Layer 2 solutions are protocols built on-top of the main blockchain (Layer 1) to provide additional features, such as an increased transaction throughput or reduced fees. The core idea behind these solutions is to process transactions off-chain and periodically submitting a summary about them on the Layer 1 chain. Some notable examples of Ethereum Layer 2 solutions are:

- *Polygon*[3], which offers faster and cheaper transactions compared to the Ethereum main network.
- *Arbitrum*[4], that enhances scalability while preserving compatibility with Ethereum smart contracts [15].
- *ZKsync*[5], a solution that ensures high security and fast finality through the use of validity proofs.

[2] `https://ethereum.org`
[3] `https://polygon.technology`
[4] `https://arbitrum.io`
[5] `https://www.zksync.io`

- *Optimism*[6], which prioritises simplicity and close integration with the Ethereum ecosystem.
- *Starknet*[7], that introduces its own high-performance language, Cairo[8].

2.4 Decentralized Storage

Since blockchain operations consume gas, and gas costs are significantly higher than those in traditional Web 2.0 solutions, especially for storage [16], dApps often rely on decentralized storage solutions to handle large amounts of data and high-volume files [6][17]. These solutions distribute data across the nodes in the network, in contrast to traditional storage approaches that depend on centralized data centres.

One widely adopted decentralized storage system is IPFS [18], a protocol built on a peer-to-peer network architecture, similar to blockchains. When users upload a file to IPFS, such file is assigned a unique identifier called a Content Identifier (CID), which corresponds to the hash of the file's content [19]. Files can be replicated across multiple nodes to enhance availability and security. To retrieve a file, users must use its CID to locate and access the nodes storing it. This content-addressed design ensures file integrity and verifiability, as any change to the file would produce a new CID.

[6] `https://www.optimism.io`
[7] `https://www.starknet.io`
[8] `https://www.cairo-lang.org/`

Chapter 3

Related Work

In recent years, many academic institutions have begun exploring blockchain-based solutions to streamline certificate issuance and verification, leveraging the technology's tamper-resistant properties. This chapter presents the related work, discussing each project's approach and its contributions in relation to the problem addressed in this thesis.

3.1 Blockcerts

In 2018, the Massachusetts Institute of Technology (MIT) introduced Blockcerts platform [20] as an open-source standard for issuing cryptographically verifiable academic credentials. Originally developed to allow MIT students to receive digital versions of their academic certificates, Blockcerts has since evolved into a community-led project[1]. The platform provides:

- A set of libraries (e.g., Python and JavaScript) for creating, issuing, and verifying credentials.
- Mobile applications that allow students to receive, store, and share verifiable certificates.
- Schemas and specifications for building systems that adhere to the Blockcerts standard.

3.2 Digital Credential Consortium

In 2019, MIT joined eleven other institutions worldwide to form the Digital Credential Consortium (DCC)[2], including the University of Milano-Bicocca (Italy) and the University of Toronto (Canada). The goal of the DCC is to establish interoperable standards for academic credentials and to facilitate the secure exchange of students' certificates across institutions. Building upon the foundation laid by

[1] https://www.blockcerts.org

[2] https://digitalcredentials.mit.edu

MIT's Blockcerts, the consortium evolved the project to support a broader and more integrated ecosystem. It provides students with mobile applications to access, verify, and share their academic credentials, and offers institutions an administrative dashboard for issuing and managing certificates.

3.3 PublicEduChain

PublicEduChain [21] is a project developed by Gazi University (Turkey) that enables students to manage their educational data on Ethereum. In this model:

- Each student deploys a personal smart contract via third-party wallets such as MetaMask or Trust Wallet.
- Students register the address of their smart contract within the university's Learning Management System (LMS), which must be modified to support this integration.
- Authorized LMS users (faculty, staff, or other institutions) can write courses information into the student's smart contract.

3.4 Cerberus

Tariq et al. introduced Cerberus [22], a blockchain-based system for certificate verification focused on document authenticity. In Cerberus:

- Universities apply to join the network; once approved, they can upload certificates directly to the blockchain.
- Upon document upload, a unique Quick Response (QR) code is generated and linked to the on-chain record.
- Students and other stakeholders can verify any certificate's authenticity by scanning the QR code.

3.5 Conclusions

The existing solutions, summarized in Table 3.1, exhibit several shortcomings relative to our objectives. First, most platforms, including Blockcerts, the DCC, and Cerberus, focus exclusively on degree certificates without accounting for the full spectrum of academic records that universities issue. While diplomas attest to degree completion, individual course transcripts contain valuable information that students often use for academic exchanges or employment opportunities.

PublicEduChain offers a more comprehensive "academic wallet" model but suffers from poor User Experience (UX). Students are responsible for developing and deploying their own smart contracts, which limits adoption to those with technical expertise and places a significant training burden on institutions.

Cerberus, despite addressing certificate verification, does not shift data ownership to students. Instead, it retains the traditional paradigm in which universities

Table 3.1: Comparison of related solutions

Solution	Focus	Data Ownership	UX
Blockcerts	Certificates	Students	Libraries and mobile applications
DCC	Certificates	Students	Mobile applications and admin dashboard
PublicEduChain	Full academic records	Students	Students must write and deploy smart contracts
Cerberus	Certificates	Universities	Verification via QR code

issue and hold certificates; students merely verify and share them via QR codes. In contrast, Web3 principles advocate for a model in which students possess and fully control their academic records, granting or revoking access as needed.

By offering a user-friendly UX for both students and institutions, supporting the full spectrum of academic records, and ensuring that students retain complete ownership of their data, EW presents a comprehensive solution that aligns with the vision and values of decentralized technologies.

Chapter 4

Problem Statement

This chapter builds upon the related works presented in Chapter 3, which represent the current state of the art for our problem, and the background concepts introduced in Chapter 2, which describe decentralized technologies with a focus on the blockchain. It delves into the specific problem addressed by this work, and introduces the use case that guided the design of EW.

As introduced in Chapter 1, our problem is the management of secure and verifiable academic information. Since we are rapidly moving into Web3 technologies, students and institutions require a decentralized system of records-keeping that enables them to share this information without the use of traditional document and validation methods. Based on the related works discussed earlier, such a system must cover all aspects of a student's academic career and provide stakeholders with a streamlined environment that prioritizes UX and interaction. More in detail, EW aims to enable university to issue certificates and record courses evaluation in a tamper-proof and secure system, leveraging on blockchain characteristics to provide these features.

On the students side, instead, we want to follow Web3 rule about data ownership and design a system in which students are the owner of their academic data; this means that they can manage the access to their academic wallet and decide the entity that can view and modify their information.

4.1 Use Case

By analysing this problem, we developed a use case, illustrated in Figure 4.1, to simulate the usage of EW. The scenario involves four main actors:

- The **home university**, which issues certificates and records evaluations during the student's regular academic career.
- The **host university**, which requires access to the student's academic data when the exchange project begins.
- The **student**, who participates in an exchange project, such as the Erasmus+ program. At the beginning of the program, they must send their academic

history to the host university. At the end of the program, the home university must certify the results obtained at the host institution.

- The **administrator** of the EW system, who manages the registration of students and universities in the platform.

The first step is the registration of universities in the EW system. Universities request access from the system administrator, who verifies their information and, upon approval, registers them into the platform.

Once the universities are present in the system, the student begins their academic career by enrolling in the home university. As with traditional enrolment, the university verifies the student's identity and, once validated, creates an account for them in EW. With the account created and the appropriate permissions, the home university can begin registering the student's academic evaluation.

After some years of regular study, the student decides to participate in an exchange program at the host university. To process the application, the host university needs access to the student's academic records. It sends a request via EW, which the student receives and approves by granting the necessary permission.

Once granted access, the host university retrieves the student's history directly from EW and proceeds with the application. Upon acceptance, the students signs the learning agreement and begins the program. The host university then requests permission to add new academic records to the student's wallet. The student again approves this request. After completing the exams, the host university record the evaluations in the student's account. At the end of the exchange program, the home university accesses the list of completed courses from EW to update the student's academic career.

Finally, the student revokes the host university access permissions, as they no longer need access to their academic wallet. The student and both universities have successfully completed the streamlined management of exchange program documentation using EW.

The scenario highlights the need for a system that balances security with usability, while ensuring that students retain control over their academic data. The key features required in such a solution include secure credential management, decentralized data ownership, streamlined inter-institutional communication, and tamper-proof record keeping. The following chapter translates the features outlined in the use case and problem analysis into specific requirements for the EW system.

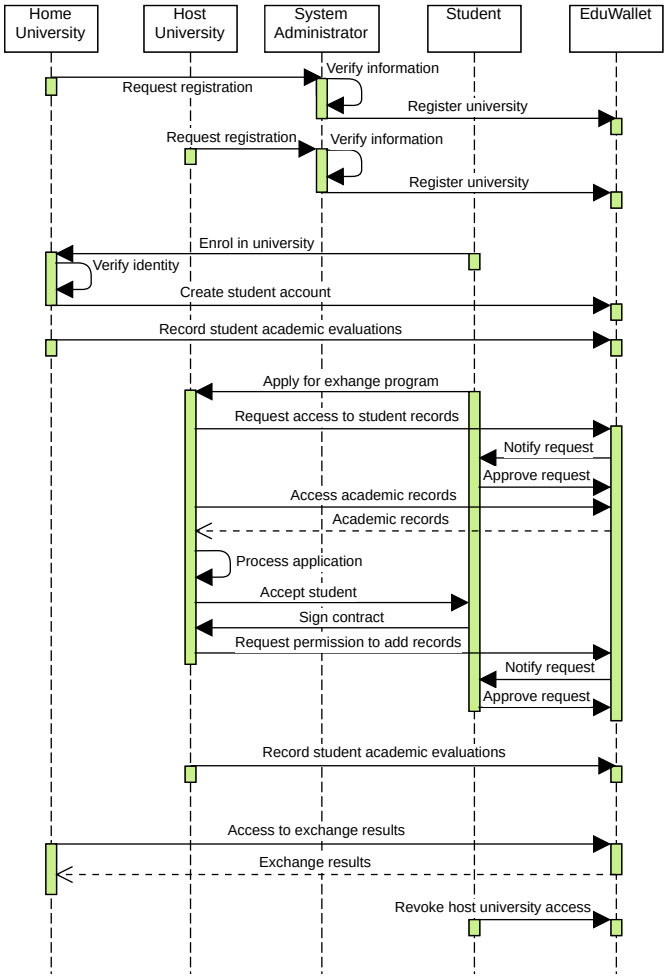


Figure 4.1: Sequence diagram illustrating the chronological interactions between student, universities, EduWallet, and the system administrator during the academic exchange process.

Chapter 5

Requirements

In this chapter, our aim is to describe both the functional requirements (FRs) and non-functional requirements (NFRs) that guided us in the development of the comprehensive academic wallets system. These requirements are summarized in Table 5.1 and Table 5.2, respectively. They were derived from an in-depth analysis of the use case presented in Chapter 4, as well as the specific needs and demands of the various stakeholders engaged with the system. This stakeholder group includes the administrator of the EW system, students, and universities.

5.1 Functional Requirements

The primary FR for the system administrator is the ability to register universities following their subscription request. This process must be preceded by a thorough verification of the provided data, in order to ensure that only trusted institutions are granted access to the system. The validation step is fundamental, as universities are the entities entitled to register new students in the academic record. Consequently, if malicious actors are authorized to access the system, its security and the veracity of the stored data could be compromised.

Regarding universities, they must be able to initiate a subscription by submitting their institutional information, including name and country. Upon approval, they require secure authentication mechanisms to access the system and exercise their privileges. Authentication is a critical precondition for nearly all operations, including the viewing and modification of students' data and academic records. These operations may involve retrieving the list of courses attended by a student or issuing a new certification. Furthermore, authenticated universities are responsible for creating EW account for newly enrolled students who do not yet posses one, as well as requesting permission from students to access their existing wallets. Delegating the creation of student accounts to universities accelerates the data verification process, as the system relies on the institution's trustworthiness to validate student information.

Academic records are strictly personal data and must be handled with extreme caution. For this reason, without proper authentication, access to student, whether

Table 5.1: Functional Requirements

System Administration	
FR1	Allow the system administrator to verify and approve universities requesting access to the EW system.
University	
FR2	Enable universities to register and subscribe to the system.
FR3	Provide secure authentication mechanisms for universities to access the platform.
FR4	Allow universities to create new smart contract wallets for students upon enrolment.
FR5	Enable universities to read from and issue academic records to students' smart wallets.
FR6	Implement authorization controls to ensure that only permitted universities can access or modify specific academic records.
FR7	Provide a mechanism for universities to request and obtain permission from students before accessing or modifying their academic records.
FR8	Provide Application Programming Interface s (APIs) that allow universities to integrate the EW system with their existing LMS.
Student	
FR9	Students must own and manage their academic smart wallets independently.
FR10	Enable students to securely authenticate and access their smart wallets.
FR11	Provide students with a web-based interface to view and manage their academic records.
FR12	Allow students to grant and revoke access permissions to their academic records for specific institutions.

for viewing or modification, must be strictly restricted. Finally, since blockchain technologies can be difficult to interact with, due to their relatively recent development, the system should expose comprehensive API endpoints. These can support seamless integration with university LMS, enabling them to interact with and utilize the full rage of features offered by EW.

Students' FRs are closely aligned with the core principles of Web3 ownership: students must retain full control over their academic wallets. Consequentially, the EW system must enable students to securely authenticate and access their academic records via a web interface, which also provides the ability to view and administer their data. A web application eliminates the need to install dedicated software on user devices and ensures a smoother multi-device experience. Once authenticated, students should be able to grant universities explicit permissions to access their records, with fine-grained control over viewing and modification rights. This distinction is essential to manage different scenarios appropriately. For example, in the context of an exchange program, the host university should only be allowed to access a student's academic records, and not modify them.

5.2 Non-Functional Requirements

In addition to the presented core features, the system must fulfil several NFRs that ensure robustness, maintainability and alignment with the principles of decentralization.

To preserve its independence, EW must avoid reliance on third-party cryptocurrency wallets, such as MetaMask or Coinbase, as well as on proprietary technologies. These constraints are essential to prevent external dependencies that could compromise the system's availability or security due to changes in external policies or services.

A key consideration for blockchain-based systems is the cost associated with on-chain storage operations [16]. To address this, the system must minimize blockchain storage usage by storing only essential data directly on-chain, while leveraging alternative technologies to manage and store files.

Academic records, by nature, must be verifiable and resistant to unauthorized modifications. The system must ensure that these records are tamper-proof and can be verified by third parties at any time, in alignment with the integrity requirements of academic data.

Simplicity and ease of use are also critical. Since our target users are not necessary experts in blockchain or, more generally, in computer technologies, the system must offer an intuitive means of accessing and managing academic data. As such, both universities and students must be supported with a user-friendly interface that facilitates seamless interaction with the EW platform.

Finally, the adoption of blockchain technologies is driven by the desire to embrace the defining characteristic of Web3 applications: decentralization. EW should be designed to operate as a decentralized system, thereby avoiding the

Table 5.2: Non-Functional Requirements

NFR1	The system shall operate without dependency on third-party wallet providers such as MetaMask.
NFR2	The system shall minimize reliance on third-party technologies to enhance security and maintain control.
NFR3	On-chain storage costs shall be minimized by storing only essential data, excluding large files.
NFR4	Academic records shall be tamper-proof and verifiable by authorized third parties.
NFR5	The system shall provide an intuitive and user-friendly interface for both students and university administrators.
NFR6	The system architecture shall be designed to maximize decentralization wherever feasible.

limitations and risks associated with centralized architectures, such as data security vulnerabilities, scalability constraints, and privacy concerns.

5.3 Constraints and Assumptions

The system operates under the assumption that the user authentication phase is not the primary focus of the project. Therefore, it is sufficient to implement a basic mechanism to identify users and grant them access to their respective privileges. A key requirement is that this method be easily replaceable or upgradable, allowing for future integration of more sophisticated authentication solutions.

As the system is intended to serve as a Web3 extension for traditional LMS platforms, a critical constraint is that all on-chain operations must remain within acceptable gas limits. This ensures that blockchain transactions linger affordable and practical for real-world use. It is also assumed that universities and their technical staff possess a fundamental understanding of blockchain technologies, including key concepts such as wallets and transactions. This baseline knowledge is essential for effectively utilize the API provided by EW.

Chapter 6

System Architecture

This chapter introduces the architecture of EW and provides a high-level overview of its components, each designed to satisfy the requirements outlined in Chapter 5. The original project is available through the GitHub repository linked in Appendix A.1.

EW employs a hybrid architecture that combines on-chain and off-chain elements, thereby delivering a user-friendly, secure and technologically advanced academic record system. The integrated components, illustrated in Figure 6.1, are:

- **Smart Contracts**: A suite of on-chain contracts that implement the core system logic, including the creation of students and university accounts creation and the management and retrieval of academic records.
- **Browser Extension**: A graphical interface through which students can manage their academic wallets and interact with their records.
- **SDK**: A TypeScript library intended for integration into university LMS, facilitating seamless interaction with EW's blockchain components.
- **Decentralized Storage System**: An off-chain solution for storing and retrieving certification files, which reduces on-chain storage costs by handling large documents externally.

In addition to these core components, we developed a simple yet complete **CLI**, which serves as a testing and demonstration tool and enables users to perform all operations typically available to universities, thereby simplifying the interaction with our SDK. Because the focus of our work is on the interaction of universities and students with the academic registry, the system administrator's core functionalities[1] have been inserted directly in the CLI. This design decision streamlines our use case and reduces unnecessary complexity.

The next chapters provide a detailed breakdown of the on-chain (Chapter 7) and off-chain (Chapter 8) designs.

[1] The approval and subscription of universities

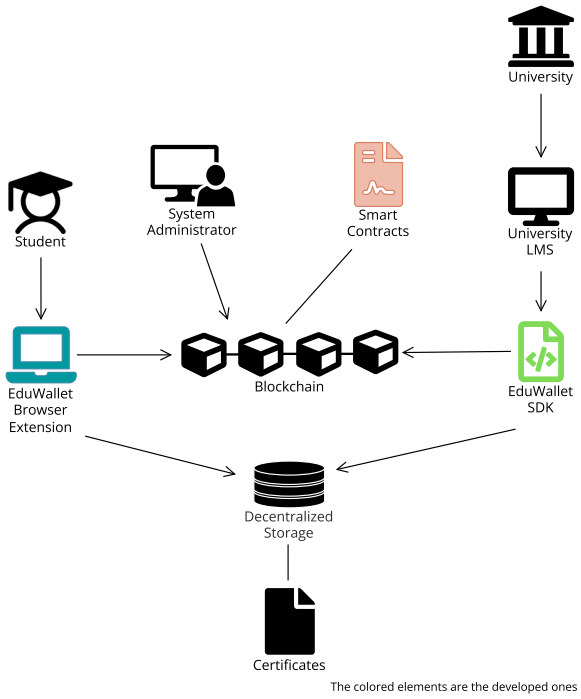


Figure 6.1: Base architecture of the EduWallet system

Chapter 7

On-Chain Design

This chapter details the on-chain design of the EW system, which is implemented entirely via smart contracts. As introduced previously, these contracts embody the core logic of the platform, exposing the functionality required by both the SDK and the browser extension. Their overall structure is depicted in the class diagram in Figure 7.1.

The EW system comprises seven custom smart contracts, represented as white classes in the diagram. (*Result* is not a smart contract, see Section 7.3 for more details).

1. **SmartAccount**: Defines the structure of a Smart Contract Account (SCA) following the account abstraction specification.
2. **Student**: Represents an individual student in the system.
3. **University**: Represents a university entity.
4. **StudentDeployer**: Responsible for deploying Student contracts.
5. **UniversityDeployer**: Deploys University contracts.
6. **StudentsRegister**: Manages and stores information about students and universities.
7. **Paymaster**: Sponsors blockchain transactions made by students and universities.

It addition, the system relies on five external contracts, shown in green in Figure 7.1, which are derived from established libraries such as *OpenZeppelin*:

1. **EntryPoint**: A contract that receives transactions from smart accounts and executes user operations.
2. **AccessControl**[1]: Provides a comprehensive role-based access control mechanism.
3. **Ownable**[2]: Implements a simple ownership model, allowing the designated owner to perform privileged operations via restricted functions.
4. **BaseAccount**: Abstract contract defining the core behaviours of a SCA under the account abstraction specification.

[1] `https://docs.openzeppelin.com/contracts/5.x/api/access`

[2] `https://docs.openzeppelin.com/contracts/5.x/api/access#Ownable`

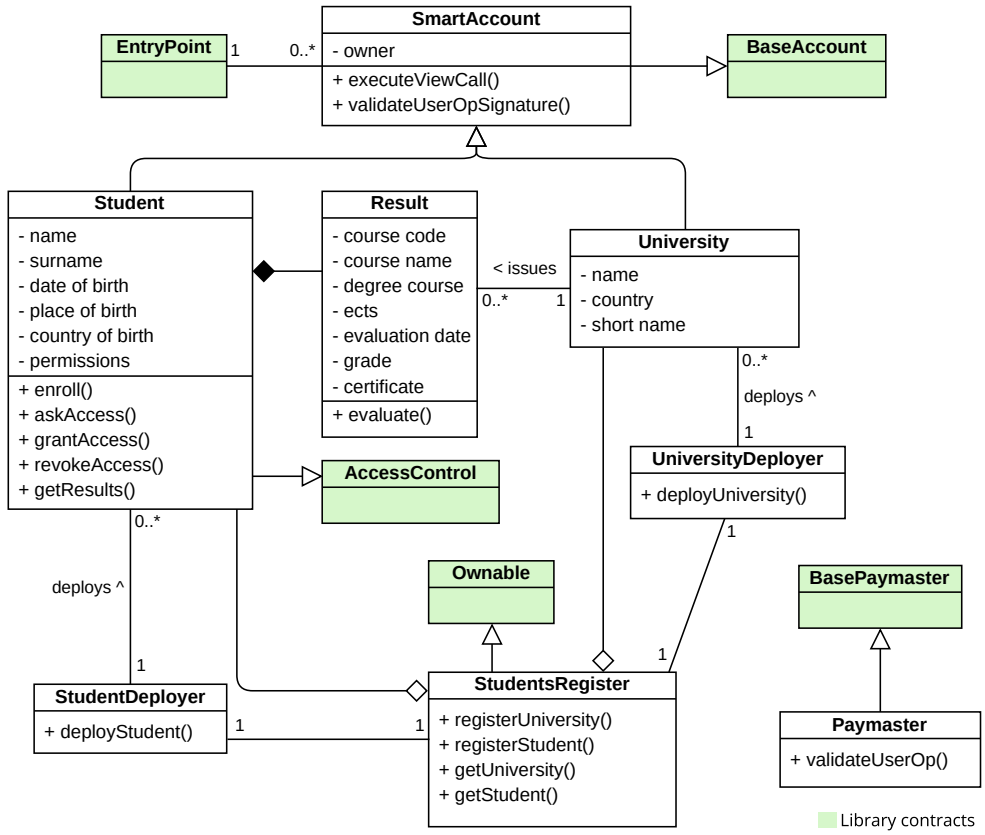


Figure 7.1: Class diagram representing smart contracts architecture

5. **BasePaymaster**: Abstract contract defining the structure of a paymaster that sponsors user transactions.

Before detailing the design of each custom smart contract and its role in the academic records workflow, we first justify our choice on on-chain technologies and platforms.

7.1 Blockchain Technologies

The development of a blockchain-based system begins with the selections of a suitable blockchain platform. We selected Ethereum, a public blockchain known for its extensive developer community and rich ecosystem of features particularly suitable for an academic record systems [21][20]. Ethereum supports smart contract development in multiple languages and serves as the foundation for various layer 2 solutions that enhance performance and scalability. This flexibility allows the system to be initially developed for the Ethereum main network and later migrated to a Layer 2 chain to take advantage of specific features, with minimal development overhead.

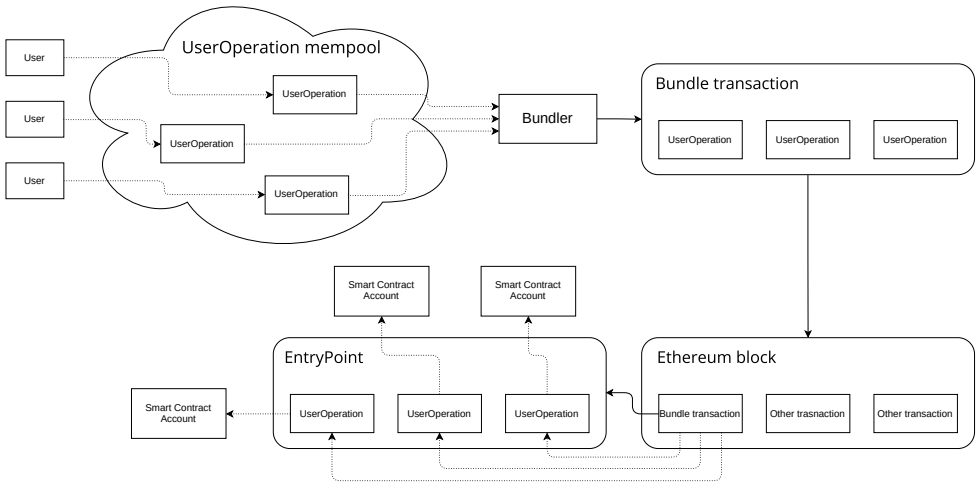


Figure 7.2: UserOperation life cycle within account abstraction protocol.

For the implementation, we selected Solidity[3] as the programming language. Solidity is an object-oriented language, designed specifically for writing smart contracts on Ethereum and the EVM, influenced by C++, JavaScript and Python[4]. It is the most widely adopted language in the ETH ecosystem, supported by and active and large developer community. Given our prior experience with Solidity, this choice was natural and well-suited to our objectives.

7.1.1 Account Abstraction

One of the most significant obstacles to the widespread adoption of blockchain technologies, and Web3 applications in general, is the complexity involved in interacting with the blockchain. To perform on-chain operations, users are typically required to create and manage cryptocurrencies wallets (EOA), fund them with tokens (e.g., BTC or ETH), and only then are they able to access and utilize decentralized platforms. During the design of the system, we aimed to eliminate this burden for users, in order to streamline the interactions among EW, universities and students, thereby addressing NFR 5 in Table 5.2. These considerations motivated the adoption of account abstraction within the system.

ERC-4337, introduced in 2021 by Vitalik Buterin and others [23][24], defines account abstraction on Ethereum. It enables users to leverage SCA with custom logic as their primary on-chain accounts, replacing traditional EOA. User actions are encapsulated in pseudo-transactions called *UserOperations*, whose life cycle is illustrated in Figure 7.2. Each UserOperation may bundle multiple on-chain actions, such as contract calls or token transfers, and includes all necessary execution data. When creating a UserOperation, the user specifies the target SCA and signs the operation to authorize both its execution and associated gas consump-

[3] `https://soliditylang.org/`

[4] `https://github.com/ethereum/solidity/blob/develop/docs/index.rst`

tion. Signed UserOperations are submitted to a mempool, where specialized nodes called bundlers collect and aggregate them into a single on-chain transaction. This transaction is sent to an entry point contract on Ethereum, which unpacks each UserOperation and dispatches it to the designated SCA. The SCA then executes the required actions and handles fee payments, effectively becoming the transaction sender.

A key feature of ERC-4337 is the ability to designate a paymaster, such as our system's Paymaster contract, to sponsor gas fees on the user's behalf. If the paymaster approves, it covers the transaction costs, allowing users to interact with EW without managing wallets or tokens directly.

Some features introduced by this standard in Ethereum are:

- **Custom Verification Logic**: SCAs have the capacity to implement custom-made authentication and validation mechanisms, going beyond the traditional public-key signature model.
- **Sponsored Transactions**: Developers can delegate gas payment to paymaster, transforming the user fee experience.
- **Bundled Operations**: Users can combine multiple transactions into a single UserOperation, reducing the aggregate cost compared to separate transactions.

However, one of the trade-offs of using account abstraction is the higher transaction cost. Due to the added complexity and number of steps involved, executing a UserOperation is typically about four times more expensive than a traditional transaction initiated by an EOA [24].

In our implementation, since the system runs in a local test environment where no public bundlers are available, UserOperations are sent directly to the entry point contract, which then forwards them to the appropriate SCA for execution.

7.2 SmartAccount

Depicted in Figure 7.3, *SmartAccount* contract serves as an abstract base for all SCAs in the account-abstraction layer of the EW system. Both University and Student contracts inherit from SmartAccount, which in turn extends BaseAccount, an abstract contract supplied by the ERC-4337 standard. BaseAccount implements core ERC-4337 logic, including support for both single and batched UserOperations and a mechanism to refund the entry point contract for gas expenditures incurred during transaction execution. SmartAccount boosts these capabilities with two key features:

- **Validate UserOperation signature**: Upon receiving a signed UserOperation, the contract verifies that the signature originates from the owner of the SCA. The owner is specified when the contract which implements SmartAccount is deployed by providing the address to its constructor. Once validated, the account can perform the specified operations by leveraging the execution routines provided by BaseAccount.

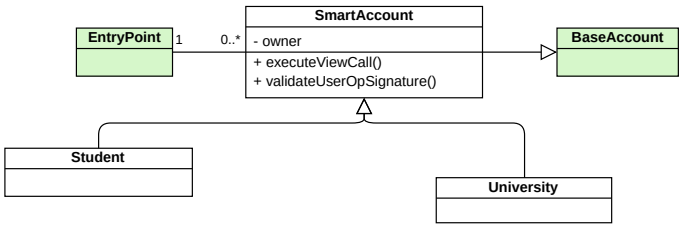


Figure 7.3: Class diagram showing a focused view of the SmartAccount contract and its associated interactions. This view is a subset of the broader system architecture.

- **Execute read-only function**: To allow gasless access to read-only (*view*) functions that are permissioned in our system, the contract offers a dedicated method, restricted to only the owner of the account, for executing such calls via the SCA. This avoids the need for standard UserOperations execution, and their associated gas costs, when users only require data retrieval. These view calls are performed by providing SmartAccount the address of the target contract and the encoded function data (*calldata*) to send to it (i.e., the function signature and parameters in encoded form).

Moreover, because SCAs support customizable signature validation logic and decouple externally EOAs from on-chain execution, SmartAccount can be readily adapted to integrate alternative authentication schemes.

7.3 Student

The *Student* contract encapsulates the majority of the system's logic. Its primary responsibilities include storing the student's personal information, such as name, surname, date of birth, place of birth and country of birth, as well as managing their academic records. These records are stored using the structure presented in Code listing 7.1.

Code listing 7.1: Result structure within the Student smart contract

```
/**
 * @dev Represents an academic result
 * @param code Course code
 * @param name Course name
 * @param university Smart wallet address of the university that created the record
 * @param degreeCourse Name of the degree program
 * @param ects ECTS credits for the course
 * @param grade Final grade
 * @param date Date when the grade was assigned
 * @param certificateHash CID of the IPFS file representing the certificate
 */
struct Result {
    string code;
    string name;
    address university;
    string degreeCourse;
```

```
    uint16 ects;
    string grade;
    uint date;
    string certificateHash;
}
```

Due to Solidity's limited support for floating-point numbers, and because the European Credit Transfer and Accumulation System (ECTS) credits may not always be whole number, the ECTS value is stored as the original number multiplied by 100. The type *uint16* is used for this purpose, allowing values up to 655.35[5], which is more than sufficient for academic credit systems. A smaller unsigned integer type, such as *uint8*, supports only values from 0 to 2.55 in this context, which is clearly inadequate. The field *certificateHash* stores the CID of the certificate, acting as a reference to its location in the decentralized storage system, as seen in Section 2.4.

Additional functionalities, all addressing FR 5 in Table 5.1, include enabling universities to:

- Retrieve a student's personal and academic information.
- Enrol the student in a new course.
- Record an evaluation for course the student has already attended.

All such interactions are governed by a strict access control mechanism, implemented to fulfil the relevant functional requirements outlined in Table 5.1, namely FR 6, FR 7, and FR 12. This mechanism is based on the *AccessControl* library, specifically the *AccessControlEnumerable* extension, which enables the definition of roles and their association with specific addresses. Within the Student contract, four distinct roles are defined:

1. *reader*
2. *writer*
3. *reader requester*
4. *writer requester*

These roles cover all possible scenarios. When an institution requests read or write access to a student's academic records, its smart account address is assigned the corresponding role. Since AccessControlEnumerable supports enumeration of role bearers, the student can query and view which institutions have pending access requests. When an institution attempts to access or modify a student's academic records, the Student contract verifies whether the caller's address has been granted the appropriate role, thereby enforcing access restrictions. This mechanism requires the Student contract to support a set of permission management functions, including the ability for students to grant, revoke, and inspect permissions (FR 12 in Table 5.1), as well as for universities to request and verify their access rights (FR 7 in Table 5.1). The underlying AccessControl library restricts the permission management functions to holders of the *default admin* role, which is defined within the library itself. The Student contract assign this role to the stu-

[5]The maximum number representable with 16 bits is 65535

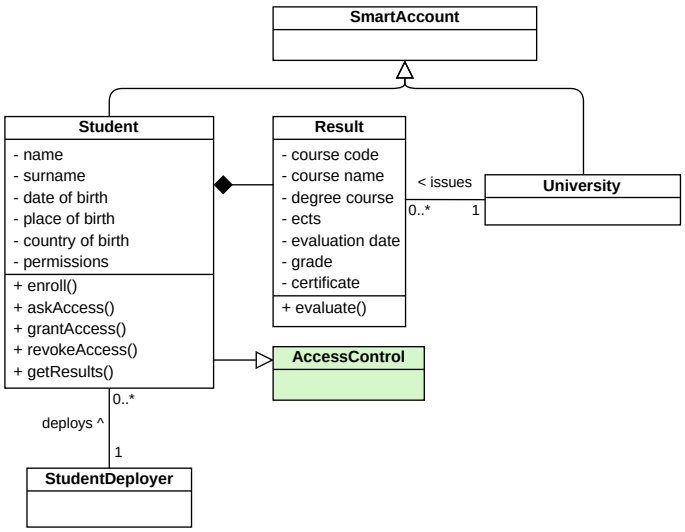


Figure 7.4: Class diagram showing a focused view of the Student contract and its associated interactions. This view is a subset of the broader system architecture.

dent, thereby enabling them to manage and control access to their academic data autonomously.

7.3.1 Deployment and Interaction Flow

As illustrated in Figure 8.8 and Figure 7.4, Student contracts are deployed by the StudentDeployer, which is invoked by the StudentsRegister contract when a university registers a new student in the EW system. Delegating students registration to universities decentralizes the validation process: since universities already verify applications during their standard admissions procedures, they supply authenticated data directly to the system, thereby avoiding centralization and reducing administrative overhead. Relying solely on a central administrator to register both universities and potentially tens of thousands of students would be impractical.

Upon registration, the initiating university is automatically granted write permissions for the student's academic record, reflecting its role as the enrolling institution responsible for issuing evaluations. All subsequent interactions with the Student contract are carried out either by the browser extension or the SDK. The browser extension is responsible for retrieving personal and academic data and managing permissions on the student's behalf. The SDK, on the other hand, acts on behalf of universities to access and modify the student's academic wallet. To facilitate these interactions, the Student contract defines several structured data types, which are used for functions inputs and outputs. These structures improve code readability and usability by grouping related data into cohesive types. Instead of requiring users to pass multiple separate parameters in a specific order, an approach that increases the risk of errors, developers can simply import the

relevant structure and populate its fields. The data types defined in the Student contract are:

- **EnrollmentInfo**: Contains the information required to enrol a student in a course; used as input for the enrolment function.
- **EvaluationInfo**: Contains the data needed to record an evaluation; used as input for the evaluation function.
- **Result**: Previously presented, this structure stores the details of a course attended by the student and is used to return the student's academic records.
- **StudentBasicInfo**: Represents the student's personal information.
- **StudentInfo**: A composite structure that includes StudentBasicInfo and a list of Result structures, representing the student's complete academic profile.

7.3.2 Vulnerabilities and Scalability

This design is suitable not only for the testing environment but also for deployment in a real-world platform. In Solidity, the usage of aggregated data types, such as arrays or mappings, entails a gas cost that increases with the size of the data structure, particularly when looping or enumerate its content. However, in the context of a student's academic career, the number of universities requiring access to their academic wallet is typically limited, likely fewer than ten. As a result, the size of the permissions data structures remain small and the associated gas cost does not increase to a level that would pose significant issue.

A more significant concern arises with the storage of student's academic results, which are currently stored together in a single array. Recording a new evaluation for an existing course requires iterating through the array to locate the appropriate entry. If a student's results set grows to several tens of entries, this operation can incur high costs, undermining platform usability. To mitigate this issue, future enchantments can preserve the existing results array, allowing all records to be retrieved in a single transaction, while also maintaining a secondary mapping from the composite key (university address, course code) to the index of the corresponding result in the array. Although this mapping introduces a slight increase in the storage consumption, it enables constant-time lookups and updates for specific courses. This significantly lowers the gas cost for evaluations.

Moreover, to minimize the overall cost of the EW system and its operational expenses, all functions that do not modify the on-chain state, i.e., those that perform only data retrieval, are implemented as read-only operations. In Solidity, these are marked with the *view* or *pure* modifier and execute without gas cost. Given that read operations are generally more frequent than writes, this design choice further curtails the system's cumulative gas expenditures.

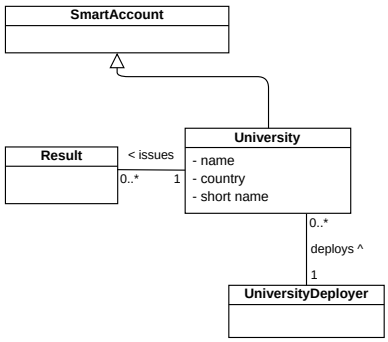


Figure 7.5: Class diagram showing a focused view of the University contract and its associated interactions. This view is a subset of the broader system architecture.

7.4 University

Since the primary focus of this work is on the interaction between students and their academic records, as well as the ownership of such data, the institutional accounts (SCA) of universities, implemented through the *University* contract, are designed to be simpler than those of students, as illustrated in Figure 7.5. Similar to the Student contract, the University contract extends the SmartAccount contract, enabling it to function as a SCA compatible with the account abstraction specification. As a result, universities can use their account, via the SDK, to perform blockchain transactions, which are then sponsored by the Paymaster (see Section 7.7).

In addition, because universities are identified solely by their contract address in interactions with other smart contracts, the University contract also stores descriptive metadata, including the institution's name, country, and a short identifier. Apart from the UniversityDeployer, which is responsible for deploying the contract, the only components that interact directly with the University contract are the SDK and the browser extension. When these components need to access university-related information, they do so using the institution's contract address. For instance, when a student retrieves their academic records, each record references the issuing university by it address. The browser extension must then query the blockchain to access the corresponding University contract and extract the relevant metadata. All such metadata functions are implemented as read-only calls, enabling gass-free data retrieval.

7.5 StudentDeployer and UniversityDeployer

The deployment of Student and University contracts, corresponding to FRs 2 and 5 in Table 5.1 respectively, is handled by the *StudentDeployer* and *UniversityDeployer* contracts, as depicted in Figure 7.6. When a system administrator registers a new

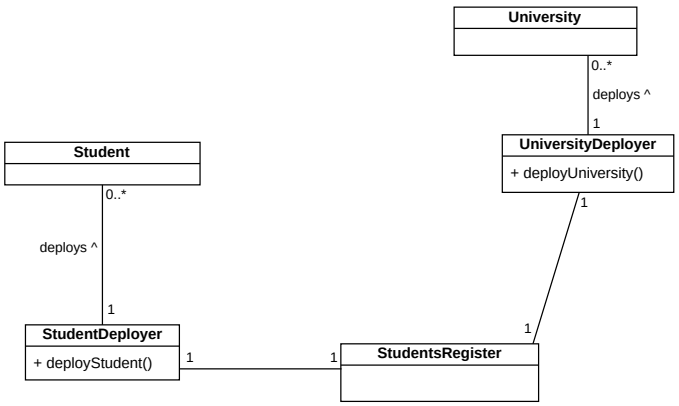


Figure 7.6: Class diagram showing a focused view of the StudentDeployer and UniversityDeployer contracts and their associated interactions. This view is a subset of the broader system architecture.

university, or a university registers a new student through the SDK, they interact with the StudentsRegister smart contract. This contract, in turn, invokes one of the deployer contracts to create the corresponding smart contract instance. This architecture implements the factory pattern, a design principle commonly used in object-oriented programming to abstract the creation of objects. In the context of the EW system, the deployer contracts abstract and encapsulate the instantiation of new Student and University contracts.

The adoption of the factory pattern offers several advantages over embedding the deployment logic directly within the StudentsRegister contract:

- It separates the contract creation logic from the registration logic, improving modularity and maintainability.
- It reduces the complexity of the StudentsRegister contract by externalizing the deployment process.
- It minimizes the contract size of StudentsRegister. In Solidity, deploying a contract via the *new* keyword requires embedding the bytecode of the deployed contract, which increases the size of the calling contract. Since Solidity enforces a maximum contract size of 24576 bytes, including large deployment code directly could exceed this limit. Using external deployer contracts bypass this issue.

The decision to centralize deployments through the StudentsRegister contract was also motivated by gas efficiency. On blockchain platforms, reducing the number of transactions typically leads to lower gas costs. By combining the deployment of a contract and the registration of its address into a single transaction, the system reduces the overall gas consumption required for onboarding new entities.

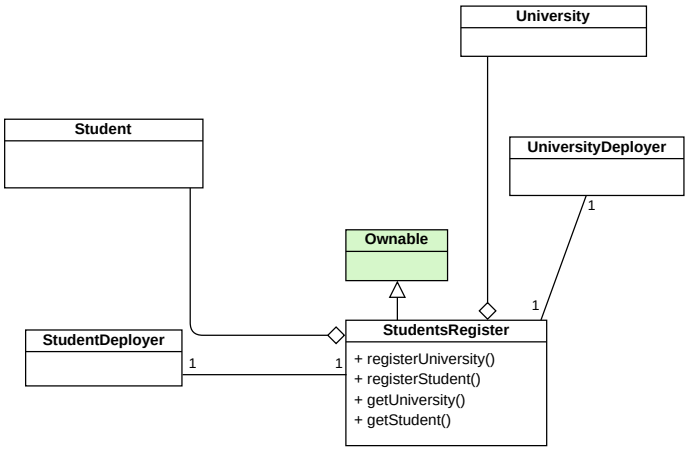


Figure 7.7: Class diagram showing a focused view of the StudentsRegister contract and its associated interactions. This view is a subset of the broader system architecture.

7.6 StudentsRegister

This section presents the *StudentsRegister* smart contract, which functions as the EW ledger. It stores the addresses of all student and university academic wallets and links them to their respective ETH wallets. This contract is deployed by the system administrator, who thereby becomes its owner and gains access to the functionalities restricted through the *Ownable* library contract. The Ownable contract, provided by the OpenZeppelin library, establishes an ownership model where certain functions are accessible only to the contract's owner. It also enables ownership transfer to another ETH wallet or smart contract, if needed. During deployment, the StudentsRegister contract requires the addresses of the already deployed StudentDeployer and UniversityDeployer contracts, which it must invoke to create new smart accounts. Additionally it must be configured with the address of the EntryPoint contract responsible for enabling account abstraction.

7.6.1 Core Functionalities

As illustrated in Figure 7.7, the StudentsRegister contract provides five core functionalities:

1. **Register a new university**: This function allows the system administrator (i.e., the contract owner) to register a new university within the EW system by submitting its institutional information, name, country, a short identifier, and its EOA address. The address is stored within the StudentsRegister contract and also utilized in the corresponding University contract to define account ownership. This functionality fulfils FR 1 in Table 5.1. To prevent duplicate entries, the function checks whether the university's EOA address is already present in the registry. Upon successful validation, the Students-

Register contract invokes the UniversityDeployer contract to instantiate the university's smart account and stores the resulting address alongside the provided wallet address. Access to this function is restricted to the contract owner to ensure that only verified and authorized institutions can be registered.

2. **Retrieve a university smart account address**: This function allows a university to obtain the address of its SCA by referencing its EOA address. This functionality supports FR 3 in Table 5.1 and is integrated into the SDK, which uses the university's EOA for authentication and retrieves the corresponding smart account address to enable operations under the account abstraction model. As a read-only operation, it incurs in no gas cost.
3. **Register a new student**: This function, restricted to validated universities, allows them to register a new student by providing the student's EOA address along with relevant personal details. To prevent duplicate registrations, the function verifies whether the student's EOA is already present in the system. This operation fulfils requirement FR 4 in Table 5.1. The wallet address is recorded in the StudentsRegister contract and referenced in the associated Student contract to establish ownership. Universities invoke this functionality through the SDK, as with all other smart contract interactions.
4. **Retrieve a student smart account address**: This function enables a student to retrieve the address of their SCA using their EOA address. It is primarily used during the login process handled by the browser extension, addressing requirement FR 10 in Table 5.1. When a student logs in using their identifier (ID) and password, the extension derives the corresponding EOA and queries the StudentsRegister contract to retrieve the associated smart account address. If a valid address is returned, the authentication process is considered successful. As this function does not alter any state on the blockchain, it is implemented as a view function to leverage gas-free execution.

7.6.2 Scalability and Gas Considerations

In contrast to the Student contract, already addresses the issue of unbounded gas costs typically associated with dynamically growing data structures. It stores student and university addresses in mappings, which provide direct, constant-time access by key. Since the system does not require iteration over all entries during standard transaction execution, the gas cost of insertion operations remains stable and independent of the total number of registered entities. This architectural choice ensures that the registry can scale efficiently without incurring increasing gas fees.

7.7 Paymaster

One of the system's key feature is that blockchain usage is nearly transparent for users. Students do not directly interact with wallets or perform transactions

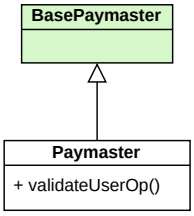


Figure 7.8: Class diagram showing a focused view of the Paymaster contract. This view is a subset of the broader system architecture.

themselves, and universities only need the private key associated with their EOA to manage their institutional smart wallet. This level of abstraction is made possible by the *Paymaster*, a smart contract deployed on the blockchain that sponsors all transactions made by users. This design directly addresses NFR 5 (see Table 5.2), which emphasizes minimizing the complexity of interactions with the system for end users.

Without the Paymaster, the system would require a mechanism to fund user wallets, presenting three primary options:

1. Each user funds their own wallet.
2. Universities fund the wallets of both students and themselves.
3. The EW system centrally manages and funds all wallets.

Each approach has significant drawbacks. The first and second require users to manage EOA and purchase tokens, which increases complexity and cost, especially burdensome for students. The third alternative still introduces administrative overhead and security concerns related to managing a large number of wallets.

Our solution utilizes a paymaster that, as shown in Figure 7.8, implements the BasePaymaster abstract contract, developed as part of ERC-4337. For simplicity, our current implementation sponsors all transactions it receives, without validating their origin or gas cost. The only enforced constraint, inherited from BasePaymaster, is that they must be routed through a known EntryPoint contract.

This configuration is suitable for testing environments such as local or test networks, where there is no risk of losing real tokens. In a real-world deployment, a more robust implementation would be necessary, specifically, one that integrates with the StudentsRegister contract to verify that the transaction sender is a verified student or university.

Chapter 8

Off-Chain Design

In this chapter, we present the design of the EW off-chain components, developed to offer a user-friendly interface for stakeholders and support the integration of system functionalities. Together with the smart contracts described in Chapter 7, these components form the complete system architecture, as illustrated in Figure 8.1.

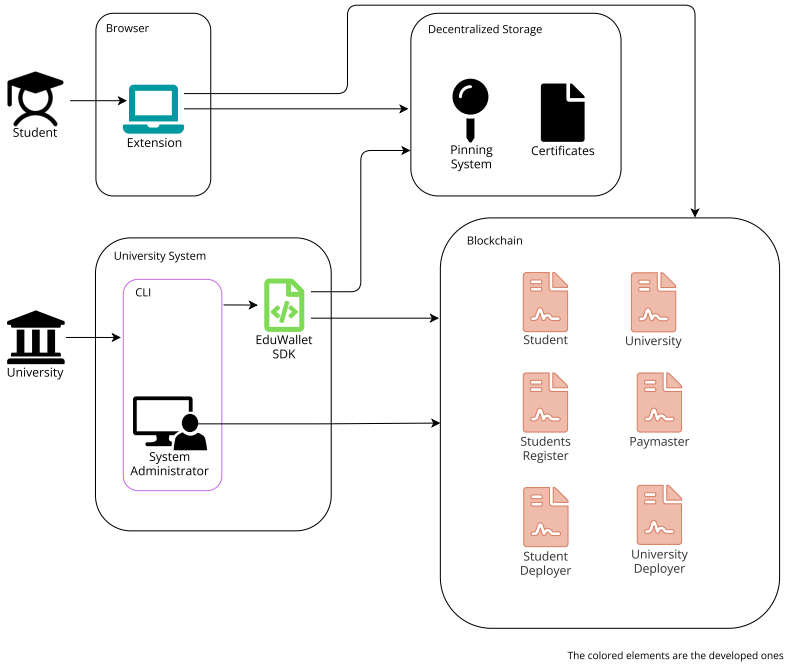


Figure 8.1: Complete architecture of the EduWallet system

Given that students using EW are not expected to have expertise in blockchain technologies, the platform must provide a simple yet effective interface. This interface should enable them to interact with their academic wallets, retrieve academic results, and manage access permissions. These requirements address FR 11 in Table 5.1. Similarly, smart contracts interactions may pose challenges even for

universities and their Information Technology (IT) departments. Therefore, the system must also offer a simplified and accessible interface for institutional use, as specified inFR 8 in Table 5.1. To satisfy these usability needs, two primary components were developed: a **browser extension** for student interactions with their academic wallets, and a **SDK** designed to facilitate the integration of EW within university systems.

Moreover, NFR 3 in Table 5.2 emphasizes the need to minimize on-chain storage consumption and associated costs by limiting blockchain use to essential data only. To support this goal, the system incorporates a **decentralized storage** solution, which is responsible for storing and providing access to large data files, such as academic certificates and other supporting documentation.

Finally, to enable comprehensive testing of the environment, a **CLI** was developed. This tool simulates a real university LMS and allows for testing the SDK functionalities, as well as its interaction with smart contracts and the decentralized storage layer.

8.1 Browser Extension

This section presents the browser extension, which serves as the primary interface for students to interact with their academic wallets. Its main objective is to abstract the complexity of on-chain operations by offering a user-friendly interface that aligns with the design and usability standards of traditional web applications, addressing FR 11 in Table 5.1.

8.1.1 Technological Choices

To achieve the highest level of decentralization and autonomy for users, we chose to implement a browser extension rather than a traditional web application. Conventional web applications, typically accessed via a URL, rely on a server to host both the user interface (front-end) and the business logic (back-end), but this architecture introduces central points of control and potential failure. Such reliance would conflict with NFR 6 in Table 5.2, which emphasizes the importance of minimizing centralization within the system's design. In contrast, a browser extension operates more like a lightweight desktop application, but within the browser environment. This eliminates the need for an external server to host the interface and, since the core logic of the EW system resides in smart contracts deployed on the blockchain, no additional server-side back-end is required. This architecture ensures that both the user interface and the underlying logic remain fully decentralized, depending only on the local extension and on-chain infrastructure. Beyond decentralization, browser extensions offer a compact, wallet-like user experience. Rather than a full screen web page, they present a small, easily accessible window via an icon in the browser toolbar. This interaction model is common among cryptocurrency wallets, such as MetaMask, offering users an intuitive means of managing on-chain assets. Adopting the same paradigm for academic

records helps students access and control their data seamlessly, reinforcing the concept on an academic wallet.

Browser extensions are supported across several major browsers, including Microsoft Edge, Google Chrome and Firefox. While each browser introduces minor platform-specific differences, we opted to develop the extension primarily for Google Chrome. Chrome currently holds the largest global browser market share[1], offering a broad potential user base. Additionally, since Google Chrome and Microsoft Edge are both based on the Chromium open-source project, extensions developed for Chrome are also compatible with Microsoft Edge, further extending platform reach without additional development overhead.

Among the various technologies available for browser extensions development, we selected *TypeScript*[2] as the core programming language and *React*[3] as the framework for building the user interface. TypeScript offers the flexibility and web-centric capabilities of JavaScript while introducing static typing, which improves code safety, clarity, and maintainability. It also integrates seamlessly with smart contract development, as libraries exist to generate TypeScript types directly from contract definitions. React, originally developed by Facebook as an open-source JavaScript library, is widely used for building modern web interfaces. It enables efficient User Interface (UI) development and facilitates seamless interaction with application logic. Our familiarity and prior experience with both React and JavaScript also influenced our choice, enabling a faster and more reliable development process. Additionally, we employed *Vite* as the building tool for our environment, making it easier to manage and bundle the various modules of our browser extension efficiently.

8.1.2 Functionalities

The browser extension provides students with all necessary tools to access and manage their academic wallets. The primary entry point is the login functionality. To authenticate, students enter the credentials (see Figure 8.2) supplied by the university at wallet creation. During login, the extension derives the private key of an EOA from the student's ID and password using the PBKDF2 algorithm. PBKDF2 is a key-derivation function that, given a salt and a password, produces the private key of an Ethereum EOA. In our implementation, the student's ID serves as the salt, eliminating the need for a centralized salt repository and ensuring uniqueness per users. Upon deriving the private key and instantiating the EOA, the extension queries the StudentsRegister contract to retrieve the associated SCA address. If a valid address is returned, authentication succeeds and the extension proceeds to fetch the student's personal details and academic records from their smart account.

Once logged in, students can view a consolidated list of their academic records

[1] `https://en.wikipedia.org/wiki/Usage_share_of_web_browsers`

[2] `https://www.typescriptlang.org/`

[3] `https://react.dev`

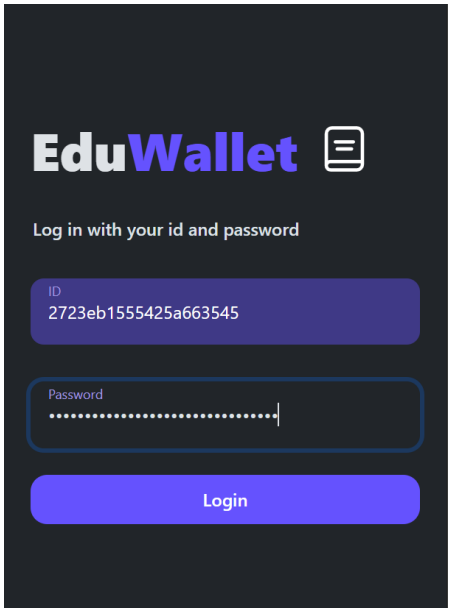


Figure 8.2: Screenshot of the browser extension login window

(Figure 8.3a) and inspect individual entries (Figure 8.3b). Records are grouped first by university, using each institution's short identifier, and then by degree program. Each entry displays the course name, course code, ECTS credits, and grade, if available. The homepage also shows the student's total accumulated ECTS. Selecting a record opens its detailed view, which adds the date of evaluation and hyperlink to the certificate stored on the decentralized storage system.

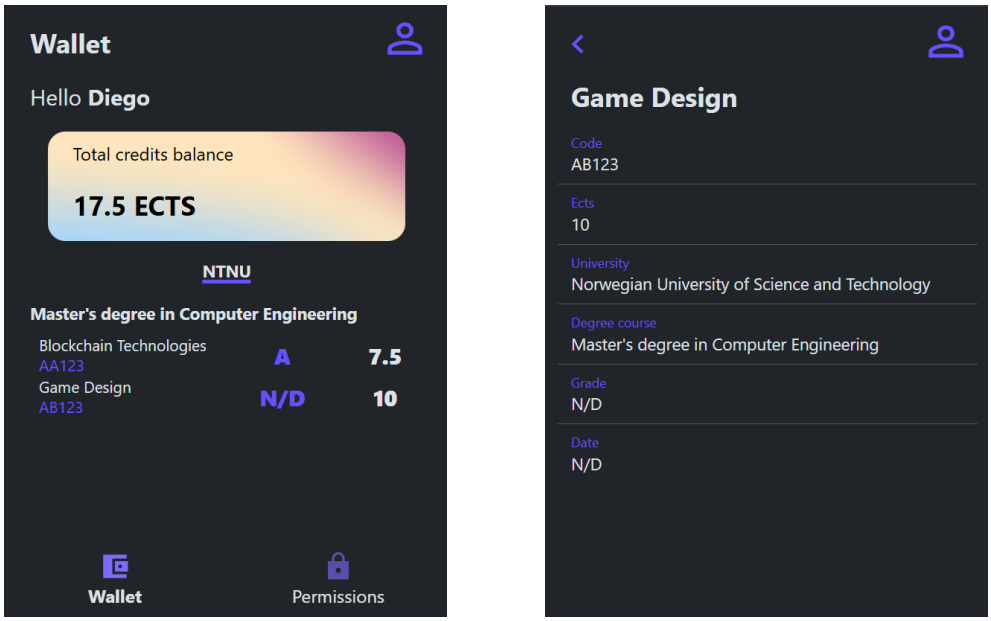
The extension also supports permission management via the lock icon. In this interface (Figure 8.4), students can review granted permission and pending access requests from universities. The permissions are categorized into three groups: requests, read permissions and the write permissions. Buttons adjacent to each entry enable students to easily grant new permissions or revoke existing ones, ensuring full control over their academic wallet.

Finally, clicking the user icon in the top-right corner opens the personal information page, where students can view their profile details (Figure 8.5).

8.1.3 Blockchain Interactions

The most straightforward way to perform on-chain operations from off-chain components is by using external libraries specifically designed to streamline these interactions. The two primary options are *web3*[4] and *ethers*[5]. We opted for *ethers* over *web3* because it is more lightweight, an essential feature for web applications that must minimize browser resource usage, and because it is more modern

[4] `https://web3js.readthedocs.io/en/v1.10.0/index.html`
[5] `https://docs.ethers.org/v6/`



(a) Homepage **(b)** Single record page

Figure 8.3: Screenshots of the browser extension showing the academic records homepage and detailed record view.

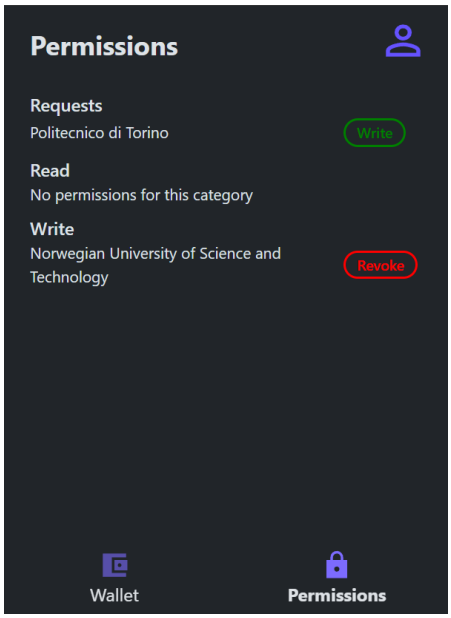


Figure 8.4: Screenshot of the extension permissions window

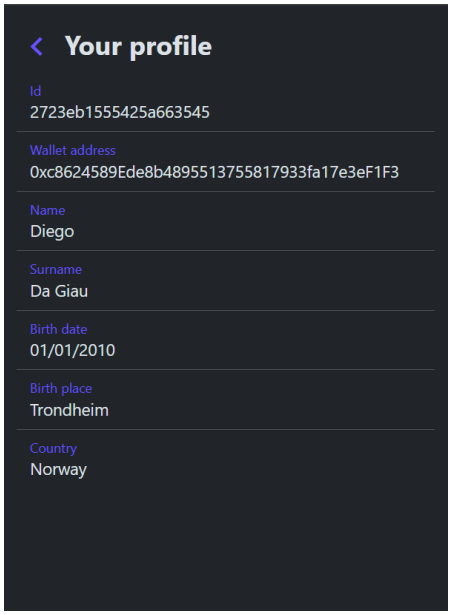


Figure 8.5: Screenshot of the browser extension user information window

and offers better TypeScript support. Another crucial library used in the development of the extension is *TypeChain*[6], a TypeScript-oriented tool that, given the Solidity code of smart contracts, extracts type definitions that can be imported directly into the application. This ensures our application remains type-safe and consistent with the structures and requirements defined in the smart contracts that comprise EW.

With these libraries in place, the browser extension needs to establish a connection to the blockchain to execute operations. Our system leverages the ethers JSON-RPC provider, which connects to a chain via its URL. The URL of the used chain is hardcoded in the extension, along with the addresses of the core contracts, namely, the StudentsRegister, EntryPoint and Paymaster (see Code listing 8.1). To deploy the extension on a different configuration, such as a layer 2 network with a new URL and different core contract addresses, these values must be manually updated in the extension's source code.

Code listing 8.1: Blockchain configuration info data variable and its type definition

```
/**
 * Configuration for blockchain network connections.
 * Defines parameters needed to connect to Ethereum networks.
 */
interface BlockchainNetworkConfig {
    /** Chain identifier for the Ethereum network. */
    readonly chainId: string,
    /** JSON-RPC endpoint URL for the Ethereum network. */
    readonly url: string;
```

[6] https://www.npmjs.com/package/@typechain/ethers-v6

```
    /** Smart contract address for the StudentsRegister contract. */
    readonly registerAddress: string;
    /** Smart contract address for the EntryPoint contract used in the account
        abstraction protocol. */
    readonly entryPointAddress: string;
    /** Smart contract address for the Paymaster contract that sponsors
        transaction gas fees. */
    readonly paymasterAddress: string,
}

/**
 * Blockchain network configuration.
 */
export const blockchainConfig: BlockchainNetworkConfig = {
    /** Chain identifier. */
    chainId: "31337",
    /** Network endpoint. */
    url: "http://127.0.0.1:8545",
    /** StudentsRegister contract address. */
    registerAddress: "0x51a240271AB8AB9f9a21C82d9a85396b704E164d",
    /** EntryPoint contract address. */
    entryPointAddress: "0xF2E246BB76DF876Cef8b38ae84130F4F55De395b",
    /** Paymaster contract address. */
    paymasterAddress: "0xB9816fC57977D5A786E654c7CF76767be63b966e",
}
```

To support the required on-chain functionalities, the browser extension performs three types of interactions:

1. Direct read-only operations
2. Read-only operations via the SCA
3. Transactions via the smart account

Direct Read-Only Operation

This is the simplest interaction and is executed, for example, during login phase, where the extension retrieves the address of the student's smart account. In this case, the extension directly invokes the corresponding function of the StudentsRegister contract, using the student's EOA as the sender.

Read-Only Operation via Smart Account

These interactions occur when retrieving permissioned data, such as academic records, accessible only to authorized universities or the student. For this, the extension invokes the dedicated function to execute read-only operations, provided by the SmartAccount contract. As described in Section 7.2, the call requires specifying:

- the address of the target contract
- the encoded function signature and parameters (calldata)

Encoding is handled using functions from the ethers library. The operation is executed using the student's EOA.

Transaction via Smart Account

This type of interaction is required for operations that alter blockchain state, such as granting or revoking permissions to universities, by accessing restricted functions. These actions require gas consuming transaction executed via the student's SCA, using UserOperations. The browser extension constructs a UserOperation by assembling the following fields:

- Sender address (the student's EOA address)
- Target contract address
- Encoded function name and parameters
- Gas and fee-related parameters
- Transferred value (always set to 0 in our case)
- Paymaster address and associated parameters
- Student's EOA signature, generated using *ethers* utilities

After constructing the UserOperation, the extension sends it directly to the EntryPoint contract using its hardcoded address. As discussed in Section 7.1.1, in a public network deployment, this operation would be relayed via a bundler instead.

8.1.4 UI Prototyping

The UI was initially designed using *Figma*, a powerful prototyping tool that enables detailed visualization and planning UI across various types of applications. The UI prototype, accessible via the link provided in Appendix A.2, played a crucial role not only in shaping the visual layout of the extension, but also in refining its functional requirements. By sketching out all application windows and placing ourselves in the shoes of end users, we were able to identify the key pieces of information the interface needed to convey and determine the most effective ways to present them.

8.2 Software Development Kit

This section describes the SDK, a TypeScript package designed to simplify integration of the EW system within university infrastructure. By abstracting low-level blockchain concerns, such as EOA key management, contract deployment and referencing, read-only queries and gas consuming transactions, the SDK lowers the barrier to entry for institutions wishing to adopt on-chain academic records.

When evaluating integration strategies, we considered two alternatives: web APIs and a stand-alone SDK. Web APIs require a centralized server to receive client requests, execute business logic, and relay blockchain interactions. In contrast, an SDK is distributed as a client-side library that runs directly within the user's environment, requiring no dedicated server infrastructure beyond standard package hosting. This approach aligns with NFR 6 in Table 5.2, which prioritizes decentralization. Moreover, an SDK offers superior scalability compared to APIs.

Since API calls traverse the public internet and share server resources, they can be affected by network latency or server overload when serving thousands of institutions worldwide. The SDK, by running locally on each institution's system, minimizes network dependencies and leverages the user's own compute resources. However, SDKs entail two main trade-offs. First, updates depend on consumer upgrading to newer package versions, whereas a centralized API can be improved transparently. Second, SDKs are inherently platform-specific: a TypeScript SDK supports only JavaScript/TypeScript environments. To address this, one must develop and maintain multiple SDK variants for different languages or platform.

Given that our off-chain components are implemented in TypeScript, and that *Node.js*[7] is widely adopted in modern back-end systems, we selected TypeScript for the SDK. The choice maximizes compatibility with university IT stacks and leverages existing expertise in the JavaScript ecosystem.

8.2.1 Working with the SDK

Once imported in the university's system, the SDK exposes a suite of functions and types that encapsulate all interaction with the on-chain academic record platform. An example of its usage is shown in Code listing 8.2, where the SDK is imported and used to register a student in the system. The registration function accepts two parameters: the university EOA wallet, instantiated from the private key provided at university enrolment using the ethers *Wallet* type; and a *StudentData* object, a type defined by the SDK that includes the student's name, surname, date of birth, place of birth, and country of birth. The function returns the student's credentials (ID and password) and the SCA address. Because all SDK calls involve blockchain queries or transactions, each function is declared asynchronous and must be awaited by the caller.

Code listing 8.2: Import of the SDK and exemplar invoke

```
// Import the SDK
import * as eduwallet from 'eduwallet-sdk';

// Register the student using the SDK
const student = await eduwallet.registerStudent(
    uni,
    {
        name,
        surname,
        birthDate,
        birthPlace,
        country,
    });
```

In addition to students registration, the SDK provides the following core functionalities for university integrators:

- Enrol a student in one or more courses.

[7] An open-source server environment used to run JavaScript and TypeScript code outside the browser (`https://nodejs.org`)

- Record a new evaluation for a student.
- Retrieve a student's personal details.
- Retrieve a student's personal details and complete academic records.
- Request permission to read from or write to a student's academic wallet.
- Verify an existing permission in a student's academic wallet.

Register a student

Universities register students by providing their own EOA and a StudentData object,v Code listing 8.3. The SDK generates a new EOA for the student using PBKDF2 algorithm, already presented in Section 8.1.2. Specifically, the SDK randomly generates a student's ID and password, which are used as inputs to derive the private key. This approach satisfies NFR 1 in Table 5.2 by avoiding reliance on third-party wallets (e.g., MetaMask). The SDK then constructs a UserOperation to invoke the registration function on the StudentsRegister contract via the university's smart account. Finally, it queries the StudentsRegister to retrieve the student's SCA address and returns it, along with the student's credentials, to the caller.

Code listing 8.3: SDK student registration function

```
/**
 * Registers a new student in the academic blockchain system.
 * Creates both a student EOA and smart account.
 * @param {Wallet} universityWallet - The university EOA with registration
 *                                    permissions
 * @param {StudentData} student - The student information to register
 * @returns {Promise<StudentCredentials>} The created student credentials and
     wallet
 *                                         information
 */
async function registerStudent(
    universityWallet: Wallet,
    student: StudentData
): Promise<StudentCredentials>
```

Enrol a student

To enrol students in courses, universities invoke the function whose signature is presented in Code listing 8.4. This function requires the university's EOA wallet, the student's SCA address, and an array of *CourseInfo* objects, each containing course code, name, degree program, and ECTS credits. The SDK converts these into the format expected by the Student contract's Result structure (Section 7.3) and submits a UserOperation to perform the enrolment.

Code listing 8.4: SDK student enrolment function

```
/**
 * Enrols a student in one or more academic courses.
 * Records course enrolments on the student's academic blockchain record,
 * establishing the foundation for future evaluations.
```

```
 * @param {Wallet} universityWallet - The university EOA with enrolment authority
 * @param {string} studentWalletAddress - The student's SCA
 * @param {CourseInfo[]} courses - Array of courses to enrol the student in (code,
 *                                 name, degreeCourse, ects)
 * @returns {Promise<void>} Promise that resolves when all enrolments are
 *                          successfully recorded
 */
async function enrolStudent(
    universityWallet: Wallet,
    studentWalletAddress: string,
    courses: CourseInfo[]
): Promise<void>
```

Record a new evaluation

Recording an evaluation requires the university's EOA wallet, the student's SCA address, and an array of *Evaluation* objects (see Code listing 8.5 for the function signature). Each object includes the course ID, grade, evaluation date, and an optional path to a certificate file. For entries that include certificates, the SDK first uploads the file to the decentralized storage system (see Section 8.3) and retrieves a reference to it. It then assembles a UserOperation containing all relevant data, including the storage reference. This UserOperation is executed via the university's smart account to invoke the evaluation function in the student's contract.

Code listing 8.5: SDK student evaluation function

```
/**
 * Records academic evaluations for a student's enrolled courses.
 * Publishes certificates to IPFS when provided and records evaluations on the
 * blockchain.
 * @param {Wallet} universityWallet - The university EOA with evaluation
 *                                    permissions
 * @param {string} studentWalletAddress - The student's SCA address
 * @param {Evaluation[]} evaluations - Array of academic evaluations to record
 * @returns {Promise<void>} Promise that resolves when all evaluations are
 *                          successfully recorded
 */
async function evaluateStudent(
    universityWallet: Wallet,
    studentWalletAddress: string,
    evaluations: Evaluation[]
): Promise<void>
```

Retrieve Student Details and Records

To fetch personal details, with (Code listing 8.7) or without (Code listing 8.6) academic records, the SDK takes the university's EOA wallet and the student's SCA address. It executes a read-only operation via the university's SCA, to invoke the appropriate view function on the student's smart account. Upon success, the SDK maps the returned data into a *Student* object, containing StudentData and an array of academic records.

Code listing 8.6: SDK function to retrieve student's personal details.

```
/**
 * Retrieves basic student information from the blockchain.
 * Only fetches personal data without academic results.
 * @param {Wallet} universityWallet - The university EOA with read permissions
 * @param {string} studentWalletAddress - The student's SCA address
 * @returns {Promise<Student>} The student's basic information
 */
async function getStudentInfo(
    universityWallet: Wallet,
    studentWalletAddress: string
): Promise<Student>
```

Code listing 8.7: SDK function to retrieve student's personal details and academic records.

```
/**
 * Retrieves student information including academic results.
 * Provides a complete academic profile with course outcomes.
 * @param {Wallet} universityWallet - The university EOA with read permissions
 * @param {string} studentWalletAddress - The student's SCA address
 * @returns {Promise<Student>} The student's complete information with academic
 *                             results
 */
async function getStudentWithResult(
    universityWallet: Wallet,
    studentWalletAddress: string
): Promise<Student>
```

Request a permission

When a university needs permission to read or write a student's academic wallet, it invokes the function shown in Code listing 8.8. This function requires the university's EOA wallet, the student's SCA address, and the desired permission type (*Read* or *Write*). It then constructs and submits a UserOperation to call the corresponding function on the student's SCA.

Code listing 8.8: SDK permissions request function

```
/**
 * Requests permission to access a student's academic wallet.
 * Universities must request access before they can read or modify student records.
 * @param {Wallet} universityWallet - The university EOA
 * @param {string} studentWalletAddress - The student's SCA address
 * @param {PermissionType} type - Type of permission requested (Read or Write)
 * @returns {Promise<void>} Promise that resolves when the permission request is
 *                          submitted and confirmed
 */
export async function askForPermission(
    universityWallet: Wallet,
    studentWalletAddress: string,
    type: PermissionType
): Promise<void>
```

Verify permissions

To verify existing permissions (Code listing 8.9), the SDK takes the university's EOA wallet and the student's SCA address. It then executes a read-only operation via the university's smart account, targeting the student smart account. The student's SCA returns the granted permission type, or *null* if none is held. The SDK forwards this result to the caller, using the appropriate TypeScript type.

Code listing 8.9: SDK permission verification function

```
/**
 * Verifies a university's permission level for a student's academic wallet.
 * @param {Wallet} universityWallet - The university EOA to check permissions for
 * @param {string} studentWalletAddress - The student's SCA address
 * @returns {Promise<PermissionType | null>} The permission level (Read or Write)
 *                                           or null if no permission
 */
 async function verifyPermission(
    universityWallet: Wallet,
    studentWalletAddress: string
): Promise<PermissionType | null>
```

8.2.2 Access On-Chain Functionalities

Since the SDK adopts the same model as the browser extension to access on-chain functionalities, the Section 8.1.3 provides a thorough explanation of the different types of interactions, as well as the tools and libraries used to execute them.

8.2.3 Input Management

All SDK functions incorporate input validation and presence checks to ensure data integrity. For example, the function responsible for recording a new evaluation in a student's academic wallet first verifies that all required parameters are provided. It then checks the validity of the student's smart account address by confirming that it begins with *0x*. Additionally, the function ensures that the evaluations array contains at least one entry and that each evaluation includes all mandatory fields.

8.3 Decentralized Storage System

This section presents our solution to one of the most significant challenges in blockchain-based systems: the high cost of on-chain storage. To address this issue, presented also through the NFR 3 in Table 5.2, we introduce an off-chain decentralized storage solution in our project. This system is used to store and retrieve certification files, such as language certificates or graduation diplomas, which require significantly more space than plain text[8]. Storing such documents directly on-chain would result in substantial gas costs, making the approach impractical.

[8]PDF files typically range from a few kilobytes to several megabytes, whereas plain text data usually occupies only a few bytes.

We chose a decentralized storage system over traditional local or cloud-based solutions to maintain the decentralized nature of our environment and to meet NFR 6 outlined in Table 5.2. Among the various decentralized options available, we selected IPFS for its ability to provide verifiable and distributed file storage. This choice is motivated by several factors: IPFS is an open source protocol with a large and active community, strong support, and widespread adoption. It also serves as the foundational layer for many other decentralized platforms, such as Filecoin[9] and Web3.Storage[10], allowing future extensions or upgrades to be implemented with minimal effort [19]. Furthermore, IPFS ensures immutability of stored files, a critical feature for academic certificates, which must remain unchanged over time.

8.3.1 Pinning files

To fully leverage IPFS, we integrated *Filebase*[11], a third-party pinning service. Pinning refers to the act of instructing a node to keep a copy of a file permanently [18], preventing it from being removed during garbage collection. Without Filebase, we would have needed to run our own local IPFS node and manage file pinning manually, an approach that introduces instead complexity, higher maintenance costs, and reduced data availability in a testing system like ours. In contrast, Filebase handles node operation and file pinning, offering an accessible solution thorough its Amazon Web Services (AWS) S3-compatible API, which simplifies file uploads to the peer-to-peer network. Notably, Filebase also provides a free tier allowing up to 5 GB of storage, which is sufficient for our needs. This is an advantage over other pinning solutions such as Web3.Storage, which lacks a fully free plan, or Pinata[12], which offers more limited options.

8.3.2 Integration in the system

As illustated in Figure 8.1, both the browser extension and the SDK interact with the storage system. The browser extension enables students to retrieve certificates associated with their academic records through the official IPFS public gateway. As shown in Figure 8.6, each certificate is presented as clickable, composed by the gateway's base URL followed by the file's CID on the IPFS network. The CID of each document is stored on-chain within the student's academic wallet, alongside other record information such as the course name (see Section 7.3). This enables students to view and download their certificates from a standard web interface.

Similarly, the SDK uses the same mechanism to retrieve certificates on behalf of universities. When uploading a file, however, it interacts directly with Filebase to ensure that the file is pinned and hosted by an active node. The SDK receives the document from the university and uses the AWS S3-compatible API to upload

[9] `https://filecoin.io`
[10] `https://web3.storage`
[11] `https://filebase.com`
[12] `https://pinata.cloud`

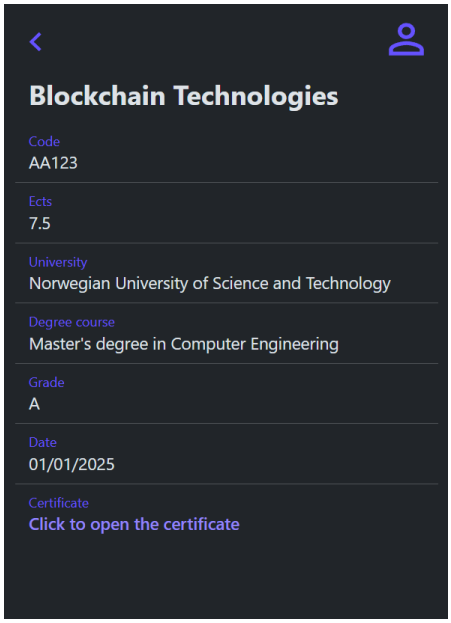


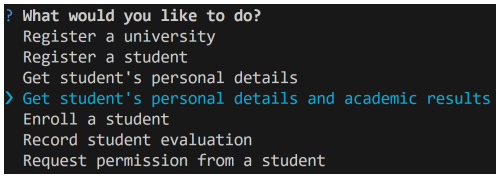
Figure 8.6: Screenshot of the browser extension presenting the page of a course. The image shows the link to the certificate saved in the decentralized storage.

it. This API requires the access key associated with the pinning account, which is managed by the EW system administrator, along with the file itself. In return, it provides the CID, which is then stored in the academic record. Section 9.4 provides a more detailed explanation of the API structure and usage.

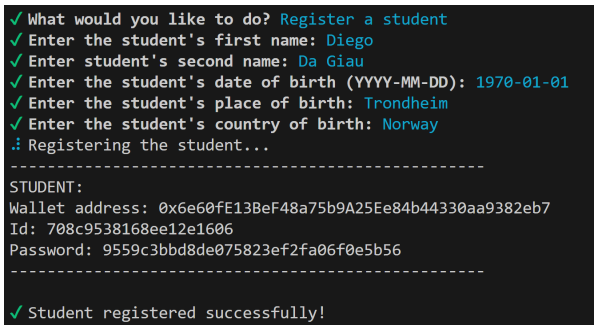
8.3.3 Security and Limitations

Academic certifies, and official documents more broadly, are legal artifacts that must always be secure and verifiable. IPFS inherently supports these properties through its use of content-based addressing. In this model, each file is identified by a CID, which is derived from the cryptographic hash of the file's content. Any alteration to the file results in a completely different CID, ensuring that tampering is immediately detectable [18]. Since the CID is stored on the blockchain at the time the certificate is issued by the university, the document's authenticity and integrity are guaranteed.

While IPFS offers strong immutability and verifiability, it lacks built-in access control. In the context of our system, we assume that certificates are publicly accessible documents. Consequently, any part in possession of a file's CID can retrieve it via the public gateway. However, if access control becomes a requirement, there are several strategies to address this limitation [25]. One option is to encrypt files before uploading them to IPFS, such that only authorized components within our system can decrypt them. Another approach is to use a private IPFS network, where access can be restricted to approved entities.



(a) Sliding menu



(b) User input and visual feedbacks

Figure 8.7: Snapshots of the CLI interface

8.4 CLI

This section describes the testing tool developed to evaluate the use of the SDK. In a real-world deployment, universities are expected to integrate the SDK into their existing LMS to interact with EW. However, given that this is a testing environment, smaller and simpler than a real-world deployment, we developed a minimal CLI to simulate the interaction between a university's system and our academic register. We opted to implement a CLI rather than a web application or desktop GUI and this decision allowed for faster development, enabling us to focus on the core functionalities of the SDK, the blockchain logic, and the browser extension, without introducing additional complexity related to graphical or UX design.

Figure 8.7 illustrates the visual aspects of the CLI. In Figure 8.7a, users can navigate through a sliding menu offering various options. When input is required, the interface prompts the user for the necessary information and provides visual feedback upon completion of the operation (Figure 8.7b).The CLI leverages the *inquirer*[13] TypeScript library to manage user interactions and uses *ora*[14] to display feedback. Specifically, *ora* is responsible for showing success and error messages, as well as animated text with spinners to indicate ongoing operations.

8.4.1 CLI Features

The CLI exposes all the features outlined in Figure 8.8. Users can:

- Submit a request to register a university in the EW system.

[13] `https://www.npmjs.com/package/inquirer`
[14] `https://www.npmjs.com/package/ora`

Figure 8.8: Use case diagram representing the functionalities provided by the CLI.

- Register a new student.
- Retrieve a student's personal details.
- Retrieve a student's details and academic results.
- Enrol a student in a new course.
- Evaluate a student.
- Request permissions from a student.
- Verify existing permissions.

Additionally, the CLI provides options to change the current university and exit the program.

Testing environment initialization

Before performing any operation, the CLI must initialize a local blockchain test network by deploying the following smart contracts:

- EntryPoint
- StudentDeployer
- UniversityDeployer
- Paymaster
- StudentsRegister

The Paymaster contract then must also be funded to sponsor users' transactions.

In public test networks or in production environments, this initialization step would be unnecessary, as the contracts would already be deployed. Their addresses would be hardcoded in the CLI, SDK and browser extension.

University registration

To apply to the EW system, a university must provide its name, country, and short name. Upon receiving these parameters, the CLI generates a random private key, which becomes the private key of the university's EOA. The EOA is then set as the active identity used to execute the subsequent operations on behalf of the university. Upon successful completion, the CLI shows the generated private key, which is essential for accessing the university's SCA. In a real LMS, this private key must be securely stored and used to initialize the EOA wallet when interact with the SDK.

To enable immediate interaction with the EntryPoint contract on the local testnet, the CLI also funds the university's EOA. This step is unnecessary in public or production networks, where the bundler covers transaction fees and is reimbursed by the Paymaster.

Register a new student

To register a student, the university provides their name, surname, date of birth, place of birth and country of birth. The CLI then invokes the SDK to create the student's smart account and credentials, which are returned to the university (Figure 8.7b). The SCA address uniquely identifies the student and must be stored by the university, as it is required for all future interactions.

The CLI also funds the student's EOA for local testing. The wallet address is obtained from the login credentials, as explained for the browser extension in Section 8.1.

Student Information Retrieval

To retrieve a student's personal information or full academic record, university must provide the student's smart account address. The CLI then returns the requested data.

Enrol and evaluate

To enrol or evaluate a student, the university must provide the smart account address and course code. Enrolment also requires the course name, number of ECTS and degree course name. Evaluation requires the evaluation date and, optionally, the path to a certificate file. Since the SDK allows enrolment in or evaluation of multiple courses at once, the CLI also supports submitting multiple records in a single command.

Permissions: Request and Verification

To access or modify student's academic records, the university must request permission. The CLI requires the student's SCA address and the type of permission (read or write). The CLI also includes the option to verify whether the university currently has read or write permission for a specific student.

Changing University and Exiting the CLI

These functionalities are unique to the CLI and are included for convenience. To change the active university, the user provides the private key of another registered university. To exit the CLI, the user selects the corresponding menu option.

Administrator Functionalities

The CLI also includes admin functionalities, as outlined in FR 1 of Table 5.1, to facilitate system testing. Specifically, the administrator can:

- Review the information submitted in university registration request.
- Approve and register universities in the system.

These functionalities are embedded within the university registration option. A university is automatically registered when its information is provided by the user during the registration process.

8.4.2 Data validation

All user input is validated using regular expressions and formatting rules. For instance, wallet addresses and private keys are validated based on length and structure[15]. Strings are validated to fall within predefined length limits. Dates must follow the YYYY-MM-DD format to avoid ambiguity and must be after January 1, 1970, as they are stored as Unix timestamps (unsigned integers). ECTS values are checked to ensure they are valid integers or floating-point numbers within acceptable limits. Since the smart contracts store ECTS as integers scaled by 100 (see Section 7.3 for further details), the CLI ensures the values will not cause overflow during storage.

[15]Private keys must start with *0x* and be followed by 64 hexadecimal characters; addresses by 42.

Chapter 9

Implementation

This chapter describes the practical implementation of the EW system, building upon the designs introduced in the previous chapters. While Chapter 7 and Chapter 8 focus on the system's architecture and the rationale behind key design decisions, this chapter highlights the concrete tools, technologies, and development processes used to bring the system to life.

Key implementations aspects include the use of account abstraction for users account, the integration of off-chain and on-chain components and the introduction of an access control system that empowers students to manage who can access their data. Special attention is given to the coordination between the academic records system and the decentralized storage layer.

9.1 Development Environment

This section outlines the tools and technologies used to author, compile, and test EW system.

The primary code editor is Visual Studio Code[1], chosen for its lightweight footprint and extensive ecosystem of extensions. To support Solidity development, we installed the Nomic Foundation's Solidity extension, which provides syntax highlighting, inline error detection, and code completion directly within the editor, thereby accelerating development and debugging.

To compile contracts and simulate a local blockchain, we utilized Hardhat[2]. Our configuration specifies:

- **Solidity compiler version**: 0.8.28 (the latest fully supported by Hardhat at the time of writing)[3]
- **EVM version**: Cancun
- **Optimizer settings**: Enabled with 1000 runs to reduce bytecode size and keep contracts (e.g., Student, EntryPoint) below the 24576-byte limit.

[1] `https://code.visualstudio.com/`

[2] `https://hardhat.org/`

[3] Versions 0.8.29 and 0.8.30 were not yet fully supported.

- **Preconfigured account**: A single funded EOA serves as deployer, funder for student and university wallets, and system administrator. Its private key is fixed to ensure deterministic contract addresses, as these are derived from the deployer's key and the number of contracts deployed by the account (nonce).

The complete Hardhat configuration file can be found in Appendix D.

For blockchain interaction in off-chain code, we rely on two core libraries:

- **ethers**: Provides providers for network connectivity, cryptographic utilities (e.g., PBKDF2), and the *Wallet* class for EOA key management and transaction signing.
- **TypeChain**: A Hardhat plug-in that generates TypeScript bindings from contracts, allowing them to be imported and used as strongly typed classes.

Finally, our smart contracts leverage external libraries to streamline common patterns:

- **OpenZeppelin Contracts**: Supplies contract implementations of access control and other standard modules.
- **AccountAbstraction**: Provides the EntryPoint contract and abstract contracts required for ERC-4337 account abstraction.

9.2 Account Abstraction and On-Chain Integration in Off-Chain Components

As described in Section 7.1.1, account abstraction redefines on-chain interactions within the Ethereum network. Whereas traditions transactions were bound to EOA, the introduction of UserOperations enables SCA to initiate and execute transactions directly on-chain. This paradigm shift also alters how off-chain components interact with and invoke on-chain logic.

In Section 8.1.3, we identified three primary modalities for invoking smart contract functionality:

1. Direct view function calls
2. View function calls via a SCA
3. Gas-consuming transactions via a SCA

The following subsections detail the code patterns used to implement each interaction modality, as well as the core components involved in the implementation of account abstraction.

9.2.1 Direct View Function Calls

To perform a direct call to the view function shown in Code listing 9.1, the system first initializes an ethers provider. It then uses the StudentsRegister__factory class generated by TypeChain to create a StudentRegister instance connected to

the on-chain contract via the *connect* method, which accepts the contract address and provider as arguments. The instance exposes all public functions, variables, and types defined in the Solidity contract. Because the target function is permissioned, the student's EOA must be connected as the transaction sender. This is achieved by invoking the factory's *connect* method again, this time passing a *Wallet*, instantiated with the student's private key and the same provider. The view function is then called as a standard method on the resulting object. The return value is automatically typed according to the contract's Solidity definition and, in this case, is a string.

Code listing 9.1: Direct call to a smart contract view function

```
import { StudentsRegister__factory } from
"@typechain/factories/contracts/StudentsRegister__factory"

// Ethereum JSON-RPC provider instance
const provider = new JsonRpcProvider(blockchainConfig.url);
// Retrieves the StudentsRegister contract instance
const studentRegister = StudentsRegister__factory
.connect(blockchainConfig.registerAddress, provider);
// Retrieves the student's smart account address from the StudentsRegister contract
const studentAccountAddress = await studentsRegister
.connect(connectedStudent).getStudentAccount();
```

9.2.2 View Function Calls via a Smart Contract Account

View functions calls via a SCA exploit the *executeViewCall* method defined in the SmartAccount abstract contract (see Code listing 9.2). This method, which can be only invoked by the SCA owner, accepts two arguments: the address of the target contract and encoded function name and parameters. By using it, SCAs can perform read-only operations without submitting gas-consuming transactions.

Code listing 9.2: *executeViewCall* method in the *SmartAccount* abstract contract.

```
/**
 * @notice Performs a view call to another contract
 * @dev Only the owner can execute view calls; uses staticcall to ensure no state
 * changes
 * @param _targetContract Address of the contract to call
 * @param _calldata The encoded function data to send to the target contract
 * @return bytes The data returned from the view call
 */
function executeViewCall(
    address _targetContract,
    bytes calldata _calldata
) external view returns (bytes memory) {
    // Only owner can execute view calls
    if (msg.sender != owner) {
        revert UnauthorizedCall();
    }
    // Static call ensures we only execute view functions
    (bool success, bytes memory returnData) = _targetContract.staticcall(
        _calldata
    );
```

```
    if (!success) {
        revert ViewCallFailed(returnData);
    }
    return returnData;
}
```

To invoke this method from TypeScript, the steps illustrated in Code listing 9.3 must be followed:

1. **Calldata encoding**: The target function's selector and parameters must be encoded into a hexadecimal string. We use the *BaseContract* class from ethers to generate this encoding in a generic way, independent of the specific contract interface.
2. **Account connection and method invocation**: Instantiate a SmartAccount contract object connected with the owner's EOA wallet (e.g., the university's EOA). Then, invoke the *executeViewCall* method on this connected SCA instance, passing the target contracts address along with the encoded calldata.
3. **Result decoding**: The call returns an encoded hexadecimal string. We decode it back into expected return type using ethers utilities.

Code listing 9.3: TypeScript code invoking *executeViewCall* on a *SmartAccount* instance.

```
// Encode the function call
const calldata = targetContract.interface.encodeFunctionData(functionName, params);
// Execute the view call through the smart account
const results = await smartAccount.connect(connectedUniversity)
.executeViewCall(targetContractAddress, calldata);
// Decode the result
const decodedResults = targetContract.interface
.decodeFunctionResult(functionName, results);
```

9.2.3 Gas-Consuming Transactions via Smart Contract Account

To execute state-changing operations, such as granting or revoking permissions, our off-chain components fully leverage the ERC-4337 account abstraction protocol. On the smart-contract side, the *BaseAccount* abstract contract provides two core methods (see Appendix B):

1. **execute**: Accepts a target address (either an EOA or another smart contract), a value (ETH to transfer) and encoded calldata, then performs a single transaction.
2. **executeBatch**: Accepts an array of such triples (address, value, calldata) and executes them automatically in a single transaction.

Both methods incur in gas costs, in contrast to the read-only *executeViewCall* from Section 9.2.2.

Account abstraction security is enforced by the *validateUserOp* function within BaseAccount, invoked by the EntryPoint contract before execution. Within *validateUserOp*, our smart account calls its own *_validateSignature* helper (see Code

listing 9.4), which in our case simply verifies that the signer of the UserOperation matches the account owner.

Code listing 9.4: Function to validate the sender's signature in the *SmartAccount* contract.

```
/**
 * @notice Validates the signature on a user operation
 * @dev Verifies that the operation was signed by the account owner
 * @param _userOp The user operation containing the signature to validate
 * @param _userOpHash The hash of the user operation that was signed
 * @return validationData 0 if signature is valid, 1 if invalid
 */
function _validateSignature(
    PackedUserOperation calldata _userOp,
    bytes32 _userOpHash
) internal virtual override returns (uint256 validationData) {
    // Verify the signature matches the owner's address
    if (owner != ECDSA.recover(_userOpHash, _userOp.signature))
        return SIG_VALIDATION_FAILED;
    return SIG_VALIDATION_SUCCESS;
}
```

On the client side, we encapsulate UserOperation handling in an *AccountAbstraction* TypeScript class (see Appendix E). Because no public bundlers are available in our local testnet, we must format and submit operations directly to the EntryPoint contract as *PackedUserOperation* objects. The workflow is as follows:

1. **Create the UserOperation**: Populate fields such as sender address, target contract address, value, calldata, gas and fee parameters, and paymaster details (see Section 7.1.1).
2. **Pack the UserOperation**: Pack and encode some UserOperation fields, such as the gas consumption ones.
3. **Sign the UserOperation**: Sign the packed payload with the sender's EOA using ethers utilities, incorporating some network information and operation format.
4. **Send the UserOperation**: Submit the signed operation via the *EntryPoint.handleOps* method. Because this submission itself is a transaction, the CLI must pre-fund users' EOAs on the testnet.
5. **Verify the result**: After execution, inspect the returned execution trace stack for any exception logs emitted by the UserOperation. The absence of such errors confirms the successful execution.

The primary limitation of our implementation lies in gas consumption estimation. In our AccountAbstraction class, all gas and fee-related parameters are hardcoded. Since the system operates in a controlled testing environment with dedicated paymaster and user accounts pre-funded with large balances, we configured these limits conservatively, setting them to high values to avoid transaction failures due to out-of-gas errors. We chose not to implement a more advanced gas estimation mechanism, as it would introduce unnecessary complexity in the context of a local setup. In a real-world deployment, this concern is mitigated by bundlers, which either expose utilities for gas estimation or handle it internally.

As a result, the interaction with UserOperations becomes seamless for the end user and developers alike.

9.3 Access Control System

One of the main principles of Web3 is the shift in data ownership and management, from centralized systems typical of Web2 architectures to a user-centric model. In EW, academic records are owned by students, as established by FR 9 in Table 5.1, and this ownership is enforced through a robust access control system. This permission system is implemented entirely in smart contracts, specifically within the Student contracts.

Access control in the Student contract is achieved using the AccessControlEnumerable abstract contract, an extension of the more general AccessControl contract, both of which are provided by the OpenZeppelin contracts library. This library allows developers to define roles, which are then used to restrict access to specific contract functions. In AccessControlEnumerable, a role is defined as the Keccak-256 hash of a string, as shown in Code listing 9.5.

Code listing 9.5: Role definition in the *Student* smart contract

```
// Role definitions for access control
bytes32 private constant READER_ROLE = keccak256("READER_ROLE");
```

Roles can be granted or revoked using methods provided by the AccessControlEnumerable library. The Student contract utilizes the following methods:

- grantRole
- revokeRole
- _grantRole
- _revokeRole

The key distinction between these methods is their access control. The functions *grantRole* and *revokeRole* can only be invoked in a transactions initiated by the entity holding the *DEFAULT_ADMIN_ROLE*, whereas *_grantRole* and *_revokeRole* are unrestricted methods. The DEFAULT_ADMIN_ROLE is defined by the AccessControlEnumerable library and must be assigned to the administrator of the roles, in our case the student. This role is granted at the time of smart account creation. The unrestricted methods are primarily used in functions involving universities (e.g., permission requests), while the permissioned methods are used by students (e.g., permission revocation).

To enforce role-based access, the contract uses the *onlyRole* function modifier (Code listing 9.6) and the *hasRole* method (Code listing 9.7). The modifier restricts access to function execution based on role possession, while the method allows for role verification at runtime. Both tools are provided by the OpenZeppelin access control library.

Code listing 9.6: Function definition using the `onlyRole` modifier.

```
function evaluate(
        EvaluationInfo[] calldata _evaluations
    ) external onlyRole(WRITER_ROLE) {
    ...
}
```

Code listing 9.7: *hasRole* method for verifying access permissions.

```
if (hasRole(WRITER_APPLICANT, _university))
```

As shown in Code listing 9.8, the permission-related functions accept or return roles in their hashed format. Therefore, off-chain components must define and compute permissions in the same way as the contract.

Code listing 9.8: Functions to request and grant permissions in the *Student* contract.

```
/**
 * @notice Allows a university to request permission to access student data
 * @dev University addresses will be added to READER_APPLICANT
 * or WRITER_APPLICANT roles
 * @param _permissionType Permission type requested (READER_APPLICANT
 * or WRITER_APPLICANT)
 */
function askForPermission(bytes32 _permissionType) external {
    // Validate permission type
    if (
        _permissionType != READER_APPLICANT &&
        _permissionType != WRITER_APPLICANT
    )
    ...
}

/**
 * @notice Grants permission to a university
 * @dev Only callable by the student (DEFAULT_ADMIN_ROLE)
 * @param _permissionType Permission type (READER_ROLE or WRITER_ROLE)
 * @param _university Address of university to grant permission to
 */
function grantPermission(
    bytes32 _permissionType,
    address _university
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    // Check if the permission exists
    if (_permissionType != WRITER_ROLE && _permissionType != READER_ROLE)
    ...
}
```

Both the browser extension and the SDK use the *id* function from the ethers library to compute the hash of role strings, as shown in Code listing 9.9. The id function takes a string and returns its Keccak-256 hash, ensuring consistency with the contract's role definition.

Code listing 9.9: Role hash generation in off-chain components.

```
/**
 * Role identifiers used for access control.
```

```
 * Uses Ethereum's id() function to generate role identifiers from string constants
     .
 */
export const roleCodes: RoleCodes = {
    /** Role identifier for read access requesters */
    readRequest: id("READER_APPLICANT"),
    /** Role identifier for write access requesters */
    writeRequest: id("WRITER_APPLICANT"),
    /** Role identifier for approved readers */
    read: id("READER_ROLE"),
    /** Role identifier for approved writers */
    write: id("WRITER_ROLE"),
}
```

9.4 Decentralized Storage System Interaction

EW employs a decentralized storage system to manage certificates linked to academic records. Both the browser extension and the SDK can retrieve files, but only the SDK is responsible for uploading them on it.

File retrieval is performed via the public IPFS gateway URL. By appending a file's CID to this gateway URL, students and universities can directly access the document. This gateway is hardcoded in both the browser extension and the SDK, as shown in Code listing 9.10. The corresponding CID for each certificate is stored on-chain and retrieved when querying academic results.

Code listing 9.10: IPFS public gateway URL configuration in off-chain components.

```
/**
 * Configuration for IPFS storage.
 * Defines parameters needed to retrieve certificates.
 */
interface IpfsStorageConfig {
    /** Gateway URL for retrieving IPFS content */
    gatewayUrl: string;
}

/**
 * IPFS storage configuration.
 */
export const ipfsConfig: IpfsStorageConfig = {
    /** IPFS gateway url. */
    gatewayUrl: "https://ipfs.io/ipfs/",
}
```

Uploading a file requires interaction with an IPFS pinning service, as described in Section 8.3.1. Our SDK uses an AWS S3–compatible API provided by Filebase (see Appendix F). The API call requires the following parameters:

- **Bucket**: Name of the virtual folder for organizing files.
- **Key**: Unique identifier for the file within the bucket.
- **Body**: The file content to be uploaded.

- **ContentType**: The type of the file (set to Portable Document Format (PDF) in our implementation).

Prior to issuing the upload request, a middleware, sourced from the official Filebase documentation[4], is attached to the API client. This middleware intercepts the response to extract the resulting CID, which the SDK then records on-chain alongside the academic record.

[4] `https://docs.filebase.com/`

Chapter 10

Results

This chapter presents the results obtained through the validation of the proposed solution, as described in the previous chapters. The validation involved testing all the developed components that interact to form EW, and analysing the outcomes produced by the testing environment. This process is fully reproducible by following the deployment steps outlined in the GitHub repository (Appendix A.1).

10.1 Components Validation

The goal of the components validation was to verify whether all the system functionalities, as outlined in Chapter 5, had been properly implemented. To perform this verification, we carried out various system-level tests focusing on the interaction among components.

The tests were conducted in the local testing environment, with the Ethereum node configured as explained in Section 9.1. Given that all elements of the EW solution are tightly integrated and designed to work in synergy, the most effective validation approach was to test them as a cohesive unit, specifically, by validating the SDK and the browser extension together.

The SDK functionalities were tested through the CLI, using its UI to select and execute various operations. The browser extension instead was tested using the Chrome browser, replicating the experience of student users. After performing operations such as the student registration and enrolment via the CLI, we used the resulting student credentials to log into the extension and verify the data stored in student's academic wallet. Through these tests, we confirmed that all FRs were fulfilled for both the SDK and the browser extension.

These functional tests revealed a few bugs in system behaviour. For instance, while testing data input through the CLI, we encountered an issue with date handling: since dates are stored as unsigned timestamps, any date before January 1, 1970, was rejected by the smart contracts, causing transactions to fail. As a result, we implemented additional checks in the off-chain components to validate and restrict date inputs to compatible values. We also noticed occasional graphical

```
Contract call:       EntryPoint#depositTo
Transaction:         0xca0d51c4477d7eea41c9fd85ed003d30801dbecb7a1996c4ca8585c87b751420
From:                0x7e5f4552091a69125d5dfcb7b8c2659029395bdf
To:                  0xf2e246bb76df876cef8b38ae84130f4f55de395b
Value:               1000000. ETH
Gas used:            45449 of 45449
Block #6:            0x429fb5353c784cc2f7f4fac31dc3a4d4809faa23e7f280fe01c29530dc1e1894
```

Figure 10.1: Transaction information from the local Hardhat node

Table 10.1: Gas prices and exchange rates for Ethereum and Polygon networks

Network	**Token**	**Gas Price (Gwei)**	**Exchange rate ETH to EUR (EUR)**	**Exchange rate ETH to NOK (NOK)**
Ethereum	ETH	2	2,303.31	26,557.16
Polygon	POL	30	0.19	2.19

glitches in the browser extension. In some cases, a student's academic record was correctly retrieved but not displayed in the interface. The data would appear only after switching views, for example, from the wallet page to the personal information page. These issues are likely caused by imperfect handling of the extension's internal state management and have been noted for future refinement.

10.2 Transactions Analysis

One of the features provided by the local Hardhat node used to test our on-chain components is the ability to analyse transactions. As shown in Figure 10.1, Hardhat enables inspection of each transaction and its related information, including the sender and receiver addresses, the amount of ETH transferred, the computational effort consumed (gas used), and the hash of the block that contains the transaction.

Using this data, we analysed the cost of transactions to assess whether the proposed solution is compatible with real-world usage. We did not give significant attention to transaction waiting times in our tests due to the limitations of a local environment. Since the node was not shared with other users and only EW-related transactions were processed, transaction delays and congestion were not representative of a real public blockchain scenario.

To convert gas costs into traditional currencies, we used gas prices from online gas trackers[1] and token exchange rates retrieved from an online converter[2]. The conversion values are summarized in Table 10.1. Gas prices are expressed in Gwei, a standard unit where 1 Gwei corresponds to 10^{-9} ETH or POL.

All transaction data collected during validation are presented in Table 10.2. In particular, by analysing the distribution diagrams in Figure 10.2, we observe that the costs are relatively contained, with the most frequent cost around 0.5

[1] `https://etherscan.io/gastracker` and `https://polygonscan.com/gastracker`

[2] `https://www.coinbase.com/it/converter`

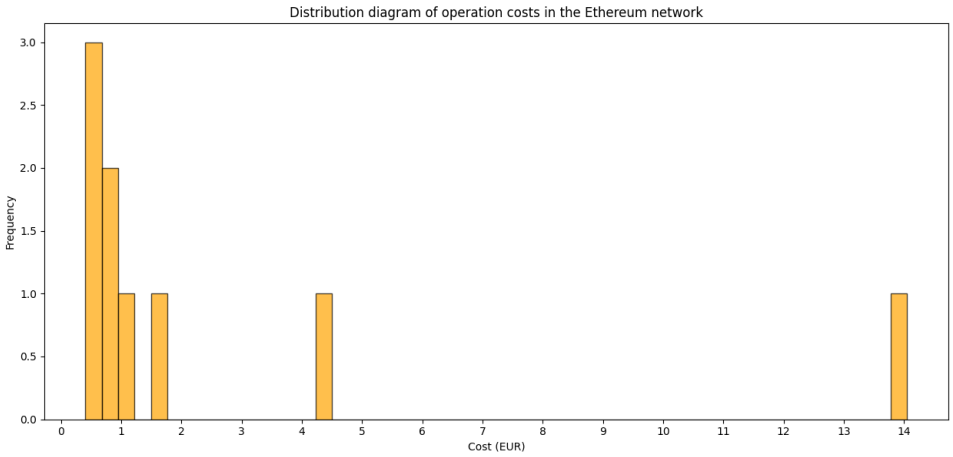
Table 10.2: Gas costs and fiat currency equivalents for EduWallet smart contract operations on Ethereum and Polygon networks.

	Operation	Gas used	Ethereum		Polygon	
			EUR	NOK	EUR	NOK
1	Deploy 4 core contracts	5775926	26.6075	306.7844	0.0329	0.3796
2	Register university	937412	4.3183	49.7900	0.0053	0.0616
3	Register student	3050503	14.0525	162.0254	0.0174	0.2005
4	Enrol in one course	228660	1.0533	12.1451	0.0013	0.0150
5	Enrol in two courses	335233	1.5443	17.8057	0.0019	0.0220
6	Evaluate one course	141137	0.6502	7.4964	0.0008	0.0093
7	Evaluate two courses	106053	0.4885	5.6329	0.0006	0.0070
8	Request permission	196625	0.9058	10.4436	0.0011	0.0129
9	Grant permission	203678	0.9383	10.8182	0.0012	0.0134
10	Revoke permission	87306	0.4022	4.6372	0.0005	0.0057

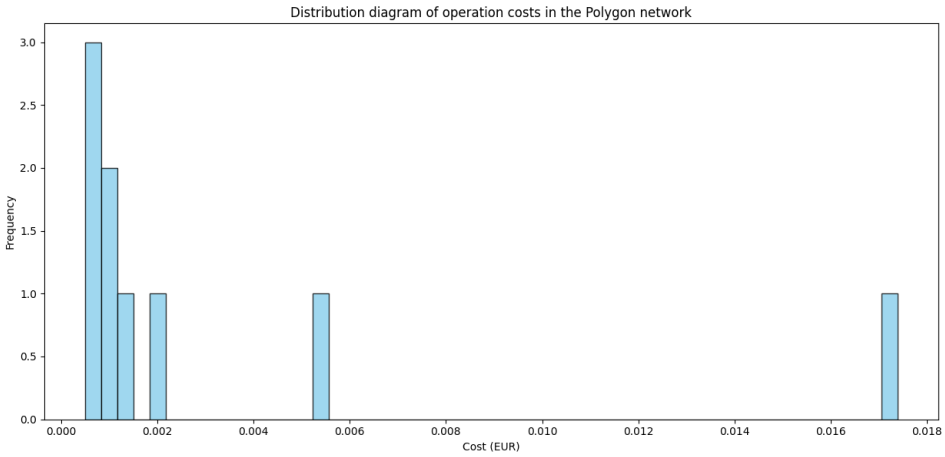
EUR (5.7 NOK) on the Ethereum network and 0.0001 EUR (0.0012 NOK) on the Polygon network. These diagrams exclude the deployment costs of the four initial smart contracts, StudentsRegister, StudentDeployer, and Paymaster, as these would typically be deployed only once in a real-world scenario and would therefore represent outliers in our analysis.

For the remaining transactions, as expected, the costs on the Ethereum network are significantly higher, approximately 800 times, compared to those on the Polygon network. For instance, registering a student, which involves deploying a dedicated smart contract, costs approximately 14 EUR (161 NOK) on Ethereum, while on Polygon, the same operation costs approximately 0.02 EUR (0.23 NOK). Despite the higher costs on Ethereum, the overall expenses of our system remain compatible with real-world usage. For example, considering the academic career of a typical student completing 40 courses, the total wallet management cost would amount to roughly 80 EUR (922 NOK) on Ethereum and only 0.10 EUR (1.15 NOK) on Polygon.

An important optimization enabled by our system is the ability to enrol in and evaluate multiple courses within a single transaction. This feature significantly improves cost-efficiency compared to processing each course in a separate transaction.



(a) Distribution diagram of operation costs in the Ethereum network



(b) Distribution diagram of operation costs in the Polygon network

Figure 10.2: Distribution diagrams of the cost to execute EduWallet operations in Ethereum and Polygon networks.

In conclusion, the validation demonstrates that our system satisfies all the FRs and NFRs outlined in Chapter 5. Furthermore, the operational costs of EW remain modest, particularly when considering deployment on the Polygon network, which offers significantly lower transaction fees compared to the Ethereum main network.

Chapter 11

Discussion

This thesis addresses a simple but impactful problem: the difficulties faced by students and universities in managing academic records. As discussed in Chapter 1 and Chapter 4, despite the increasing globalization of higher education, universities still share students' academic results through digital or paper-based documents. This leads to bureaucratic inefficiencies and wasted time. The goal of this work was to develop a blockchain-based academic records registry that offers a reliable, modernised way for students and universities to store and share academic data. This chapter reflects on the proposed solution, considers the results presented in Chapter 10, and suggests key directions for future work.

The results show that our solution successfully satisfies all defined FRs and NFRs, demonstrating its compatibility with real-world usage. The cost per student remains low on both Ethereum and Polygon networks, confirming the economic feasibility of our approach. The contribution of the EW system lies in providing a complete platform, with interfaces for both students and universities, that enables blockchain-backed academic record management in a user-friendly, "black-box" manner. Our design abstracts away the blockchain complexities, allowing users to focus solely on the core functionalities. As discussed in Chapter 3, prior projects in this domain often fall short in one or more areas: they either limit themselves to academic certificates, provide inadequate user interfaces that require students to deploy their own smart contracts, or fail to uphold Web3 principles of data ownership by giving universities sole control over records. EW addresses these limitations by leveraging emerging standards such as account abstraction to streamline interaction and decentralise ownership.

That said, our solution presents certain weaknesses. The most notable is its current limitation to a local development environment. While this setup facilitated faster development and avoided unnecessary ETH consumption during smart contract iteration, it prevented us from testing the system under real-world conditions, including network latency, congestion, and fluctuating gas prices. Another challenge lies in the system's complexity and management. EW is a large-scale project composed of multiple tightly integrated components, each with specific configurations and dependencies. Coordinating these components proved to be

demanding and may have benefitted from dedicated project management tools designed for modular, distributed systems.

A further technical challenge was the use of account abstraction. As a relatively new and rapidly evolving technology, up-to-date resources and documentation were often lacking. Moreover, its integration required a deeper, lower-level interaction with the blockchain, particularly when crafting UserOperations manually, adding complexity to the development process.

Despite these challenges, EW represents a complete solution. It demonstrates how blockchain technology can be used to securely manage students' academic records without compromising ease of interaction.

11.1 Future Work

Among the many possible enhancements for our system, we identify three future directions that offer the most significant value to the current solution. These proposals span three key areas of improvement:

1. System testing
2. Functionality expansion
3. UX and UI refinement

11.1.1 Public Network Deployment

As previously mentioned in this chapter, one of the main limitations of our solution is its deployment and testing within a local blockchain environment. A critical next step to strengthen the system's validation is deploying the smart contracts on a public test network, which provides a more realistic environment. Future developers will need to select the target network(s), either the Ethereum main network or a Layer 2 solution, and update the corresponding configuration values currently hardcoded in the system.

11.1.2 New Stakeholder: the Employer

A significant functional improvement would be the introduction of a new stakeholder: the employer. This addition would enable students to share verifiable academic credentials with prospective employers. Employers could be granted access to EW via temporary accounts and a dedicated interface, or alternatively through expiring links or QRs code, following the example of Cerberus [22] as discussed in Section 3.4. This feature would expand the platform's use beyond academia and enhance its relevance in professional settings.

11.1.3 Browser Extension Data Management

As outlined in Section 10.1, the browser extension currently experiences sporadic bugs related to data visualization, likely originating from its internal data hand-

ling mechanisms. Improving the reliability of this component would significantly enhance the student experience. A valuable enhancement would involve redesigning the extension's data management logic to eliminate these issues and ensure consistent, error-free interaction with academic wallets.

Chapter 12

Conclusion

This thesis began with a central question: *Is it possible to develop a system that facilitates academic records sharing between institutions, while also enabling students to truly own and manage their data?* Over the course of this work, we presented our proposed solution to this challenge, highlighting its strengths and acknowledging its limitations.

We began by reviewing related work in the field, identifying both useful foundations and critical shortcomings. These insights guided the definition of a comprehensive use case, capturing the multifaceted nature of the problem. From this use case, we derived a set of functional and non-functional requirements that our system needed to fulfil. We then explored the design of the on-chain components, providing an in-depth discussion of the smart contracts and the rationale behind key architectural decisions. Particular attention was given to the adoption of account abstraction, a pivotal choice that enabled us to simplify user interaction with the blockchain while maintaining security and decentralization. Following this, we detailed the design of the off-chain components, including the SDK, browser extension, decentralized storage system, and CLI, explaining how these elements interface with the smart contracts. We also illustrated how account abstraction is implemented across the system, and how decentralized storage helps reduce blockchain-related costs while preserving the system's decentralized nature. The implementation chapter offered a deeper technical look at the system's core, helping readers understand how the various modules operate in practice. We validated our solution through testing, confirming its functionality and cost-efficiency in a controlled environment. Finally, we discussed the broader implications of our work and identified promising directions for future development.

In summary, this thesis contributes a practical and innovative solution to the management of academic records, enabling universities to align with Web3 principles such as decentralization and data ownership. Beyond its application in the educational sector, our work also offers broader value to the blockchain community by demonstrating how to integrate off-chain and on-chain components in a cohesive, user-friendly system.

Bibliography

[1] T. Berners-Lee, R. Cailliau, J.-F. Groff and B. Pollermann, 'World-wide web: The information universe', *Internet Research*, vol. 2, no. 1, pp. 52–58, 1992. DOI: 10.1108/eb047254.

[2] N. Choudhury, 'World wide web and its journey from web 1.0 to web 4.0', *International Journal of Computer Science and Information Technologies*, vol. 5, no. 6, pp. 8096–8100, 2014.

[3] A. Murray, D. Kim and J. Combs, 'The promise of a decentralized internet: What is web3 and how can firms prepare?', *Business Horizons*, vol. 66, no. 2, pp. 191–202, 2023, ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2022.06.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007681322000714.

[4] D. Kristol and L. Montulli, *Rfc2109: Http state management mechanism*, 1997.

[5] D. Sheridan, J. Harris, F. Wear, J. C. Jr, E. Wong and A. Yazdinejad, *Web3 challenges and opportunities for the market*, 2022. arXiv: 2209.02446. [Online]. Available: https://arxiv.org/abs/2209.02446.

[6] P. P. Ray, 'Web3: A comprehensive review on background, technologies, applications, zero-trust architectures, challenges and future directions', *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 213–248, 2023, ISSN: 2667-3452. DOI: https://doi.org/10.1016/j.iotcps.2023.05.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667345223000305.

[7] M. Nofer, P. Gomber, O. Hinz and D. Schiereck, 'Blockchain', *Business & Information Systems Engineering*, vol. 59, no. 3, pp. 183–187, 2017, ISSN: 1867-0202. DOI: 10.1007/s12599-017-0467-3.

[8] I. Acharjamayum, R. Patgiri and D. Devi, 'Blockchain: A tale of peer to peer security', in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2018, pp. 609–617. DOI: 10.1109/SSCI.2018.8628826.

[9] S. Nakamoto, 'Bitcoin: A peer-to-peer electronic cash system', *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 2008.

[10] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer and M. Virza, 'Zerocash: Decentralized anonymous payments from bitcoin', in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[11] V. Buterin *et al.*, 'A next-generation smart contract and decentralized application platform', *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[12] G. Wood *et al.*, 'Ethereum: A secure decentralised generalised transaction ledger', *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[13] V. Buterin, 'Ethereum: Platform review', *Opportunities and challenges for private and consortium blockchains*, vol. 45, pp. 1–45, 2016.

[14] C. Sguanci, R. Spatafora and A. M. Vergani, *Layer 2 blockchain scaling: A survey*, 2021. arXiv: 2107.10881 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2107.10881.

[15] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg and E. W. Felten, 'Arbitrum: Scalable, private smart contracts', in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 1353–1370, ISBN: 978-1-939133-04-5.

[16] S. Kahtyat, 'Designing a decentralized identity verification platform', *Norsk IKT-konferanse for forskning og utdanning*, no. 3, Oct. 2024. [Online]. Available: https://www.ntnu.no/ojs/index.php/nikt/article/view/6249.

[17] A. Mühle, A. Grüner, T. Gayvoronskaya and C. Meinel, 'A survey on essential components of a self-sovereign identity', *Computer Science Review*, vol. 30, pp. 80–86, 2018, ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2018.10.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013718301217.

[18] J. Benet, *Ipfs - content addressed, versioned, p2p file system*, 2014. arXiv: 1407.3561 [cs.NI]. [Online]. Available: https://arxiv.org/abs/1407.3561.

[19] E. Daniel and F. Tschorsch, 'Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks', *IEEE Communications Surveys & Tutorials*, vol. 24, no. 1, pp. 31–52, 2022. DOI: 10.1109/COMST.2022.3143147.

[20] Y. Shakan, B. Kumalakov, G. Mutanov, Z. Mamykova and Y. Kistaubayev, 'Verification of university student and graduate data using blockchain technology', *International Journal of Computers Communications & Control*, vol. 16, Sep. 2021. DOI: 10.15837/ijccc.2021.5.4266.

[21] M. Tanriverdí, 'Publiceduchain: A framework for sharing student-owned educational data on public blockchain network', *IEEE Access*, vol. 12, pp. 51 772–51 785, 2024. DOI: 10.1109/ACCESS.2024.3385660.

[22] A. Tariq, H. Binte Haq and S. T. Ali, 'Cerberus: A blockchain-based accreditation and degree verification system', *IEEE Transactions on Computational Social Systems*, vol. 10, no. 4, pp. 1503–1514, 2023. DOI: 10.1109/TCSS.2022.3188453.

[23] V. Buterin, Y. Weiss, D. Tirosh, S. Nacson, A. Forshtat, K. Gazso and T. Hess, 'Erc-4337: Account abstraction using alt mempool', Ethereum Improvement Proposals, Tech. Rep., 2021.

[24] Z. Lin, T. Wang, C. Zhao, S. Zhang, Q. Yang and L. Shi, 'A measurement investigation of erc-4337 smart contracts on ethereum blockchain', in *2024 International Conference on Computing, Networking and Communications (ICNC)*, 2024, pp. 1164–1170. DOI: 10.1109/ICNC59896.2024.10556301.

[25] B. Guidi, A. Michienzi and L. Ricci, 'Data persistence in decentralized social applications: The ipfs approach', in *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, 2021, pp. 1–4. DOI: 10.1109/CCNC49032.2021.9369473.

Appendix A

Project Links

A.1 GitHub Repository

https://github.com/NTNU-IDI/eduwallet-eduwalletdiego

A.2 Figma Prototype Link

https://www.figma.com/design/aZrmR2thWfRGKQWDQbZE9C/EduWallet?node-id=125-95&t=gQwA5a4uDzRy8jBl-1

Appendix B

Base Account Contract

Code listing B.1: BaseAccount smart contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

/* solhint-disable avoid-low-level-calls */
/* solhint-disable no-empty-blocks */
/* solhint-disable no-inline-assembly */

import "../interfaces/IAccount.sol";
import "../interfaces/IEntryPoint.sol";
import "../utils/Exec.sol";
import "./UserOperationLib.sol";

/**
 * Basic account implementation.
 * This contract provides the basic logic for implementing the IAccount interface -
 * validateUserOp.
 * Specific account implementation should inherit it and provide the
 * account-specific logic.
 */
abstract contract BaseAccount is IAccount {
    using UserOperationLib for PackedUserOperation;

    struct Call {
        address target;
        uint256 value;
        bytes data;
    }

    error ExecuteError(uint256 index, bytes error);

    /**
     * Return the account nonce.
     * This method returns the next sequential nonce.
     * For a nonce of a specific key, use `entrypoint.getNonce(account, key)`
     */
    function getNonce() public view virtual returns (uint256) {
        return entryPoint().getNonce(address(this), 0);
    }

    /**
```

```
41      * Return the entryPoint used by this account.
42      * Subclass should return the current entryPoint used by this account.
43      */
44     function entryPoint() public view virtual returns (IEntryPoint);
45
46     /**
47      * execute a single call from the account.
48      */
49     function execute(address target, uint256 value, bytes calldata data)
50     virtual
51     external {
52         _requireForExecute();
53
54         bool ok = Exec.call(target, value, data, gasleft());
55         if (!ok) {
56             Exec.revertWithReturnData();
57         }
58     }
59
60     /**
61      * execute a batch of calls.
62      * revert on the first call that fails.
63      * If the batch reverts, and it contains more than a single call, then wrap the
64      * revert with ExecuteError, to mark the failing call index.
65      */
66     function executeBatch(Call[] calldata calls) virtual external {
67         _requireForExecute();
68
69         uint256 callsLength = calls.length;
70         for (uint256 i = 0; i < callsLength; i++) {
71             Call calldata call = calls[i];
72             bool ok = Exec.call(call.target, call.value, call.data, gasleft());
73             if (!ok) {
74                 if (callsLength == 1) {
75                     Exec.revertWithReturnData();
76                 } else {
77                     revert ExecuteError(i, Exec.getReturnData(0));
78                 }
79             }
80         }
81     }
82
83     /// @inheritdoc IAccount
84     function validateUserOp(
85         PackedUserOperation calldata userOp,
86         bytes32 userOpHash,
87         uint256 missingAccountFunds
88     ) external virtual override returns (uint256 validationData) {
89         _requireFromEntryPoint();
90         validationData = _validateSignature(userOp, userOpHash);
91         _validateNonce(userOp.nonce);
92         _payPrefund(missingAccountFunds);
93     }
94
95     /**
96      * Ensure the request comes from the known entrypoint.
97      */
98     function _requireFromEntryPoint() internal view virtual {
99         require(
100            msg.sender == address(entryPoint()),
```

```
            "account: not from EntryPoint"
        );
    }

    function _requireForExecute() internal view virtual {
        _requireFromEntryPoint();
    }

    /**
     * Validate the signature is valid for this message.
     * @param userOp          - Validate the userOp.signature field.
     * @param userOpHash      - Convenient field: the hash of the request, to check
     *                          the signature against. (also hashes the entrypoint
     *                          and chain id)
     * @return validationData - Signature and time-range of this operation.
     *                          <20-byte> aggregatorOrSigFail - 0 for valid
     *                                                          signature, 1 to
     *                                                          mark signature
     *                                                          failure,
     *                                    otherwise, an address of an aggregator
     *                                    contract.
     *                          <6-byte> validUntil - Last timestamp this operation
     *                                                is valid at, or 0 for
     *                                                "indefinitely"
     *                          <6-byte> validAfter - first timestamp this
     *                                                operation is valid
     *                          If the account doesn't use time-range, it is enough
     *                          to return SIG_VALIDATION_FAILED value (1) for
     *                          signature failure.
     *                          Note that the validation code cannot use
     *                          block.timestamp (or block.number) directly.
     */
    function _validateSignature(
        PackedUserOperation calldata userOp,
        bytes32 userOpHash
    ) internal virtual returns (uint256 validationData);

    /**
     * Validate the nonce of the UserOperation.
     * This method may validate the nonce requirement of this account.
     * e.g.
     * To limit the nonce to use sequenced UserOps only (no "out of order" UserOps)
        :
     *      `require(nonce < type(uint64).max)`
     * For a hypothetical account that *requires* the nonce to be out-of-order:
     *      `require(nonce & type(uint64).max == 0)`
     *
     * The actual nonce uniqueness is managed by the EntryPoint, and thus no other
     * action is needed by the account itself.
     *
     * @param nonce to validate
     *
     * solhint-disable-next-line no-empty-blocks
     */
    function _validateNonce(uint256 nonce) internal view virtual {
    }

    /**
     * Sends to the entrypoint (msg.sender) the missing funds for this transaction.
     * SubClass MAY override this method for better funds management
```

```
160         * (e.g. send to the entryPoint more than the minimum required, so that in
161         * future transactions it will not be required to send again).
162         * @param missingAccountFunds - The minimum value this method should send the
163         *                              entrypoint. This value MAY be zero, in case
164         *                              there is enough deposit, or the userOp has a
165         *                              paymaster.
166         */
167        function _payPrefund(uint256 missingAccountFunds) internal virtual {
168            if (missingAccountFunds != 0) {
169                (bool success,) = payable(msg.sender).call{
170                        value: missingAccountFunds
171                    }("");
172                (success);
173                // Ignore failure (its EntryPoint's job to verify, not account.)
174            }
175        }
176    }
```

Appendix C

Base Paymaster Contract

Code listing C.1: BasePaymaster smart contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

/* solhint-disable reason-string */

import "@openzeppelin/contracts/access/Ownable2Step.sol";
import "@openzeppelin/contracts/utils/introspection/IERC165.sol";
import "../interfaces/IPaymaster.sol";
import "../interfaces/IEntryPoint.sol";
import "./UserOperationLib.sol";
/**
 * Helper class for creating a paymaster.
 * provides helper methods for staking.
 * Validates that the postOp is called only by the entryPoint.
 */
abstract contract BasePaymaster is IPaymaster, Ownable2Step {
    IEntryPoint public immutable entryPoint;

    uint256 internal constant PAYMASTER_VALIDATION_GAS_OFFSET = UserOperationLib.
    PAYMASTER_VALIDATION_GAS_OFFSET;
    uint256 internal constant PAYMASTER_POSTOP_GAS_OFFSET = UserOperationLib.
    PAYMASTER_POSTOP_GAS_OFFSET;
    uint256 internal constant PAYMASTER_DATA_OFFSET = UserOperationLib.
    PAYMASTER_DATA_OFFSET;

    constructor(IEntryPoint _entryPoint) Ownable(msg.sender) {
        _validateEntryPointInterface(_entryPoint);
        entryPoint = _entryPoint;
    }

    // Sanity check: make sure this EntryPoint was compiled against the same
    // IEntryPoint of this paymaster
    function _validateEntryPointInterface(IEntryPoint _entryPoint)
    internal
    virtual {
        require(
            IERC165(address(_entryPoint))
            .supportsInterface(type(IEntryPoint)
            .interfaceId),
            "IEntryPoint interface mismatch"
```

```
        );
    }

    /// @inheritdoc IPaymaster
    function validatePaymasterUserOp(
        PackedUserOperation calldata userOp,
        bytes32 userOpHash,
        uint256 maxCost
    ) external override returns (bytes memory context, uint256 validationData) {
        _requireFromEntryPoint();
        return _validatePaymasterUserOp(userOp, userOpHash, maxCost);
    }

    /**
     * Validate a user operation.
     * @param userOp     - The user operation.
     * @param userOpHash - The hash of the user operation.
     * @param maxCost    - The maximum cost of the user operation.
     */
    function _validatePaymasterUserOp(
        PackedUserOperation calldata userOp,
        bytes32 userOpHash,
        uint256 maxCost
    ) internal virtual returns (bytes memory context, uint256 validationData);

    /// @inheritdoc IPaymaster
    function postOp(
        PostOpMode mode,
        bytes calldata context,
        uint256 actualGasCost,
        uint256 actualUserOpFeePerGas
    ) external override {
        _requireFromEntryPoint();
        _postOp(mode, context, actualGasCost, actualUserOpFeePerGas);
    }

    /**
     * Post-operation handler.
     * (verified to be called only through the entryPoint)
     * @dev If subclass returns a non-empty context from validatePaymasterUserOp,
     *      it must also implement this method.
     * @param mode          - Enum with the following options:
     *                        opSucceeded - User operation succeeded.
     *                        opReverted  - User op reverted. The paymaster still
     *                                      has to pay for gas.
     *                        postOpReverted - never passed in a call to postOp().
     * @param context       - The context value returned by validatePaymasterUserOp
     * @param actualGasCost - Actual cost of gas used so far (without this postOp
     *                        call).
     * @param actualUserOpFeePerGas - the gas price this UserOp pays. This value is
     *                                based on the UserOp's maxFeePerGas
     *                        and maxPriorityFee (and basefee)
     *                        It is not the same as tx.gasprice, which is what the
     *                        bundler pays.
     */
    function _postOp(
        PostOpMode mode,
        bytes calldata context,
        uint256 actualGasCost,
        uint256 actualUserOpFeePerGas
```

```
    ) internal virtual {
        (mode, context, actualGasCost, actualUserOpFeePerGas); // unused params
        // subclass must override this method if validatePaymasterUserOp returns a
        // context
        revert("must override");
    }

    /**
     * Add a deposit for this paymaster, used for paying for transaction fees.
     */
    function deposit() public payable {
        entryPoint.depositTo{value: msg.value}(address(this));
    }

    /**
     * Withdraw value from the deposit.
     * @param withdrawAddress - Target to send to.
     * @param amount          - Amount to withdraw.
     */
    function withdrawTo(
        address payable withdrawAddress,
        uint256 amount
    ) public onlyOwner {
        entryPoint.withdrawTo(withdrawAddress, amount);
    }

    /**
     * Add stake for this paymaster.
     * This method can also carry eth value to add to the current stake.
     * @param unstakeDelaySec - The unstake delay for this paymaster. Can only be
     * increased.
     */
    function addStake(uint32 unstakeDelaySec) external payable onlyOwner {
        entryPoint.addStake{value: msg.value}(unstakeDelaySec);
    }

    /**
     * Return current paymaster's deposit on the entryPoint.
     */
    function getDeposit() public view returns (uint256) {
        return entryPoint.balanceOf(address(this));
    }

    /**
     * Unlock the stake, in order to withdraw it.
     * The paymaster can't serve requests once unlocked, until it calls addStake
     * again
     */
    function unlockStake() external onlyOwner {
        entryPoint.unlockStake();
    }

    /**
     * Withdraw the entire paymaster's stake.
     * stake must be unlocked first (and then wait for the unstakeDelay to be over)
     * @param withdrawAddress - The address to send withdrawn value.
     */
    function withdrawStake(address payable withdrawAddress) external onlyOwner {
        entryPoint.withdrawStake(withdrawAddress);
    }
```

```
161
162     /**
163      * Validate the call is made from a valid entrypoint
164      */
165     function _requireFromEntryPoint() internal virtual {
166         require(msg.sender == address(entryPoint), "Sender not EntryPoint");
167     }
168 }
```

Appendix D

Hardhat Configuration File

Code listing D.1: Hardhat configuration file

```
import { HardhatUserConfig } from "hardhat/config";
// Import the hardhat-toolbox which bundles several useful plugins
import "@nomicfoundation/hardhat-toolbox";

const config: HardhatUserConfig = {
  // Solidity compiler configuration
  solidity: {
    version: "0.8.28", // Specify the Solidity compiler version
    settings: {
      optimizer: {
        enabled: true, // Enable the optimizer to reduce gas costs
        runs: 1000,    // Higher values optimize for when the code is executed many
                       // times
      },
      evmVersion: "cancun" // Use the latest EVM version for compatibility
                           // with newest features
    },
  },

  // Network configurations for deployment and testing
  networks: {
    hardhat: {
      hardfork: "cancun",  // Use the Cancun EVM rules for the in-memory Hardhat
                           // Network
      accounts: [
        {
          balance: "100000000000000000000000000000000", // 10^34 wei (extremely
              large
                                                        // balance for testing)
          privateKey:
          "0x0000000000000000000000000000000000000000000000000000000000000001"
          // Deterministic test account
        }
      ]
    },
    localhost: {
      url: "http://127.0.0.1:8545" // Connect to a locally running Ethereum node
    },
  },

```

```
40    // Project structure paths
41    paths: {
42      sources: "./contracts",   // Directory for smart contract source files
43      tests: "./test",          // Directory for test files
44      cache: "./cache",         // Directory for the cache
45      artifacts: "./artifacts"  // Directory for compiled contract artifacts
46    }
47  };
48
49  export default config;
```

Appendix E

AccountAbstraction

Code listing E.1: AccountAbstraction class code

```
import { ethers, TypedDataDomain, TypedDataField } from 'ethers';
import { SmartAccount__factory } from '@typechain/...';
import { blockchainConfig, logError } from 'src/conf';
import { EntryPoint } from '@typechain/...';
import { AddressLike, BigNumberish, BytesLike } from 'ethers';
import { getEntryPoint } from 'src/utils';

/**
 * Constants for EIP-712 domain.
 */
const DOMAIN_NAME = 'ERC4337';
const DOMAIN_VERSION = '1';

/**
 * Returns the EIP-712 domain for ERC-4337 user operations.
 * @param entryPoint EntryPoint contract address
 * @param chainId Chain ID
 */
export function getErc4337TypedDataDomain(entryPoint: string, chainId: number):
TypedDataDomain {
    return {
        name: DOMAIN_NAME,
        version: DOMAIN_VERSION,
        chainId: chainId,
        verifyingContract: entryPoint
    };
}

/**
 * Returns the EIP-712 types for ERC-4337 user operations.
 */
export function getErc4337TypedDataTypes(): { [type: string]: TypedDataField[] } {
    return {
        PackedUserOperation: [
            { name: 'sender', type: 'address' },
            { name: 'nonce', type: 'uint256' },
            { name: 'initCode', type: 'bytes' },
            { name: 'callData', type: 'bytes' },
            { name: 'accountGasLimits', type: 'bytes32' },
            { name: 'preVerificationGas', type: 'uint256' },
```

```
            { name: 'gasFees', type: 'bytes32' },
            { name: 'paymasterAndData', type: 'bytes' }
        ]
    };
}

/**
 * Packs paymaster data for the user operation.
 * @param paymaster Paymaster contract address
 * @param paymasterVerificationGasLimit Gas limit for paymaster verification
 * @param postOpGasLimit Gas limit for post-operation
 * @param paymasterData Additional paymaster data
 * @returns Packed paymasterAndData bytes
 */
function packPaymasterData(
    paymaster: string,
    paymasterVerificationGasLimit: number | bigint,
    postOpGasLimit: number | bigint,
    paymasterData: string
): string {
    return ethers.concat([
        paymaster,
        ethers.zeroPadValue(ethers.toBeHex(paymasterVerificationGasLimit), 16),
        ethers.zeroPadValue(ethers.toBeHex(postOpGasLimit), 16),
        paymasterData
    ]);
}

/**
 * UserOperation interface for ERC-4337.
 */
interface UserOperation {
    sender: string;
    nonce: bigint;
    initCode: string;
    callData: string;
    callGasLimit: bigint;
    verificationGasLimit: bigint;
    preVerificationGas: bigint;
    maxFeePerGas: bigint;
    maxPriorityFeePerGas: bigint;
    paymaster: string;
    paymasterVerificationGasLimit: bigint;
    paymasterPostOpGasLimit: bigint;
    paymasterData: string;
    signature: string;
}

/**
 * PackedUserOperation interface for EntryPoint contract.
 */
interface PackedUserOperation {
    sender: AddressLike;
    nonce: BigNumberish;
    initCode: BytesLike;
    callData: BytesLike;
    accountGasLimits: BytesLike;
    preVerificationGas: BigNumberish;
    gasFees: BytesLike;
    paymasterAndData: BytesLike;
```

```
    signature: BytesLike;
}

/**
 * AccountAbstraction manager for ERC-4337 user operations.
 * Handles creation, signing, packing, and execution of user operations
 * for smart accounts using the EntryPoint contract.
 */
export class AccountAbstraction {
    private provider: ethers.Provider;
    private entryPoint: EntryPoint;
    private signer: ethers.Wallet;

    /**
     * Constructs the AccountAbstraction manager.
     * @param provider Ethers provider instance
     * @param signer Wallet used for signing user operations
     */
    constructor(
        provider: ethers.Provider,
        signer: ethers.Wallet,
    ) {
        this.provider = provider;
        this.signer = signer;
        this.entryPoint = getEntryPoint();
    }

    /**
     * Creates a user operation for a smart account call.
     * @param params User operation parameters (sender, target, value, data,
     *               initCode)
     * @returns UserOperation object
     */
    async createUserOp({
        sender,
        target,
        value,
        data,
        initCode = '0x',
    }: {
        sender: string;
        target: string;
        value: bigint;
        data: string;
        initCode?: string;
    }): Promise<UserOperation> {
        const accountContract = SmartAccount__factory.connect(sender,
        this.provider);
        const callData = accountContract.interface.encodeFunctionData('execute',
        [target, value, data]);

        // Get the nonce for the smart account
        const nonce = await this.entryPoint.getNonce(sender, 0);

        // Get fee data for gas pricing
        const feeData = await this.provider.getFeeData();

        // Paymaster address from config
        const paymaster = blockchainConfig.paymasterAddress;

```

```
161         const userOp: UserOperation = {
162             sender,
163             nonce,
164             initCode,
165             callData,
166             callGasLimit: BigInt(1_000_000),
167             verificationGasLimit: BigInt(5_000_000),
168             preVerificationGas: BigInt(500_000),
169             maxFeePerGas: feeData.maxFeePerGas || BigInt(2_000_000_000),
170             maxPriorityFeePerGas: feeData.maxPriorityFeePerGas ||
171                                   BigInt(1_000_000_000),
172             paymaster: paymaster,
173             paymasterVerificationGasLimit: BigInt(2_000_000),
174             paymasterPostOpGasLimit: BigInt(2_000_000),
175             paymasterData: '0x',
176             signature: '0x',
177         };
178 
179         return userOp;
180     }
181 
182     /**
183      * Computes the hash of a user operation for signing.
184      * @param userOp UserOperation object
185      * @returns Hash string (bytes32)
186      */
187     async getUserOpHash(userOp: UserOperation): Promise<string> {
188         // Convert to packed format for hashing
189         const packedUserOp = this.packUserOp(userOp);
190         // Call the EntryPoint contract's getUserOpHash (returns bytes32)
191         return await this.entryPoint.getUserOpHash(packedUserOp);
192     }
193 
194     /**
195      * Signs a user operation using EIP-712 typed data.
196      * @param userOp UserOperation object
197      * @returns UserOperation with signature
198      */
199     async signUserOp(userOp: UserOperation): Promise<UserOperation> {
200         const chainId = parseInt(blockchainConfig.chainId);
201         const entryPointAddress = blockchainConfig.entryPointAddress;
202 
203         const packedUserOp = this.packUserOp(userOp);
204 
205         const signature = await this.signer.signTypedData(
206             getErc4337TypedDataDomain(entryPointAddress, chainId),
207             getErc4337TypedDataTypes(),
208             packedUserOp
209         );
210 
211         return {
212             ...userOp,
213             signature
214         };
215     }
216 
217     /**
218      * Converts a UserOperation to the packed format required by EntryPoint.
219      * @param userOp UserOperation object
220      * @returns PackedUserOperation object
```

```
     */
    packUserOp(userOp: UserOperation): PackedUserOperation {
        // Pack callGasLimit and verificationGasLimit into a single bytes32
        const accountGasLimits = ethers.solidityPacked(
            ['uint128', 'uint128'],
            [userOp.callGasLimit, userOp.verificationGasLimit]
        );

        // Pack maxFeePerGas and maxPriorityFeePerGas into a single bytes32
        const gasFees = ethers.solidityPacked(
            ['uint128', 'uint128'],
            [userOp.maxPriorityFeePerGas, userOp.maxFeePerGas]
        );

        const paymasterAndData = packPaymasterData(
            userOp.paymaster,
            userOp.paymasterVerificationGasLimit,
            userOp.paymasterPostOpGasLimit,
            userOp.paymasterData
        );

        return {
            sender: userOp.sender,
            nonce: userOp.nonce,
            initCode: userOp.initCode,
            callData: userOp.callData,
            accountGasLimits: accountGasLimits,
            preVerificationGas: userOp.preVerificationGas,
            gasFees: gasFees,
            paymasterAndData: paymasterAndData,
            signature: userOp.signature
        };
    }

    /**
     * Executes a batch of user operations via EntryPoint.handleOps.
     * @param userOps Array of UserOperation objects
     * @param beneficiary Address to receive transaction fees
     * @returns TransactionResponse object
     * @throws Error if transaction fails
     */
    async executeUserOps(userOps: UserOperation[], beneficiary: string):
    Promise<ethers.TransactionResponse> {
        try {
            // Sign and pack each user operation in the batch
            const packedUserOps = await Promise.all(
                userOps.map(async (op) => {
                    const signedOp = await this.signUserOp(op);
                    return this.packUserOp(signedOp);
                })
            );

            // Direct call to EntryPoint.handleOps with all operations
            const connectedEntryPoint = this.entryPoint.connect(this.signer);
            return await connectedEntryPoint.handleOps(packedUserOps,
            beneficiary
            );
        } catch (error) {
            logError('Error sending batch user operations:', error);
            throw error;
```

```
281         }
282     }
283 
284     /**
285      * Verifies the result of a user operation transaction.
286      * Throws an error if the operation failed, including revert reasons.
287      * @param receipt Transaction receipt
288      * @param targetContract Target contract for error decoding
289      * @throws Error if operation failed or logs are missing
290      */
291     verifyTransaction(receipt: ethers.TransactionReceipt,
292                       targetContract: ethers.BaseContract): void {
293         const entryPoint = this.entryPoint;
294 
295         if (!receipt || !receipt.logs) {
296             throw new Error("No receipt or logs found in transaction receipt.");
297         }
298 
299         // Find UserOperationEvent logs
300         const userOpEvents = receipt.logs.filter(log => {
301             try {
302                 const parsedLog = entryPoint.interface.parseLog(log);
303                 return parsedLog?.name === "UserOperationEvent";
304             } catch {
305                 return false;
306             }
307         });
308 
309         if (userOpEvents.length === 0) {
310             throw new Error("No UserOperationEvent found in transaction logs.");
311         }
312 
313         let parsedUserOpEvent;
314         try {
315             parsedUserOpEvent = entryPoint.interface.parseLog(userOpEvents[0]);
316         } catch (e) {
317             throw new Error("Failed to parse UserOperationEvent log: " +
318             e instanceof Error ? e.message : String(e)));
319         }
320 
321         if (!parsedUserOpEvent?.args) {
322             throw new Error("UserOperationEvent log missing args.");
323         }
324 
325         const success = parsedUserOpEvent.args.success;
326 
327         if (!success) {
328             // Find UserOperationRevertReason logs
329             const revertEvents = receipt.logs.filter(log => {
330                 try {
331                     const parsedLog = entryPoint.interface.parseLog(log);
332                     return parsedLog?.name === "UserOperationRevertReason";
333                 } catch {
334                     return false;
335                 }
336             });
337 
338             if (revertEvents.length === 0) {
339                 throw new Error("UserOperation failed but no" +
340                 "UserOperationRevertReason found in logs."
```

```
                );
            }

            let parsedRevertEvent;
            try {
                parsedRevertEvent = entryPoint.interface.parseLog(
                revertEvents[0]
                );
            } catch (e) {
                throw new Error("Failed to parse UserOperationRevertReason log: "
                + (e instanceof Error ? e.message : String(e)));
            }

            if (!parsedRevertEvent?.args) {
                throw new Error("UserOperationRevertReason log missing args.");
            }

            const revertData = parsedRevertEvent.args.revertReason;

            // Try to decode custom error
            const decodedError = targetContract.interface.parseError(revertData);
            if (decodedError) {
                throw new Error(
                    `UserOperation reverted with custom error:` +
                    `${decodedError.name}, args:` +
                    `${JSON.stringify(decodedError.args)}`
                );
            } else {
                throw new Error("UserOperation reverted with unknown" +
                "custom error.");
            }
        }
    }
}
```

Appendix F

Pinning System API

Code listing F.1: AWS API used to store files in IPFS

```
// Prepare upload parameters
const uploadParams = {
    Bucket: ipfsConfig.bucketName,
    Key: `${dayjs().valueOf()}`,
    Body: bufferFile,
    ContentType: "application/pdf",
};

// Create command for S3 upload
const command = new PutObjectCommand(uploadParams);
let cid = "";

// Add middleware to extract CID from response headers
command.middlewareStack.add(
    (next) => async (args) => {
        try {
            const response = await next(args);
            if (!response.response || typeof response.response !== 'object')
                return response;
            const apiResponse = response.response as {
                statusCode?: number;
                headers?: Record<string, string>
            };
            if (apiResponse.headers && "x-amz-meta-cid" in apiResponse.headers) {
                cid = apiResponse.headers["x-amz-meta-cid"];
            }
            return response;
        } catch (error) {
            logError('Middleware error:', error);
            throw error;
        }
    }, {
    step: "build",
    name: "addCidToOutput",
});

// Execute upload
await s3Client.send(command);
```