IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 10 December 2024, accepted 8 January 2025, date of publication 15 January 2025, date of current version 23 January 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3530269

RESEARCH ARTICLE

Emphasizing the Early Phases of the Software Development Process Before Deploying Smart Contracts

JUAN-CARLOS LÓPEZ-PIMENTEL[1], CAROLINA DEL-VALLE-SOTO[1], (Senior Member, IEEE), LEONARDO J. VALDIVIA[1], AND RAÚL MONROY[2]

[1]Facultad de Ingeniería, Universidad Panamericana, Zapopan, Jalisco 45010, Mexico
[2]Escuela de Ingeniería y Ciencias, Tecnológico de Monterrey, Atizapán de Zaragoza, Mexico City 52926, Mexico

Corresponding authors: Juan-Carlos López-Pimentel (clopezp@up.edu.mx) and Raúl Monroy (raulm@tec.mx)

This work was supported in part by the Universidad Panamericana under Grant UP-CI-2024-GDL-14-ING, and in part by the Universidad Panamericana-Amazon Web Service under Grant 023579852268.

ABSTRACT Immutability is one of the main characteristics of Blockchain. However, most software development is not static. This dilemma, among others, has caused a new branch of blockchain-oriented software engineering. This paper emphasizes the importance of the early phases of software development before deploying blockchain-based software. It follows case-based research to illustrate the implications of smart contracts designed in the early phases without including all requirements. The paper presents a digital identity case designed within a microservice architecture. We show two stages: an initial design and an upgrading requirement, which causes considerable changes in the architecture. The case is analyzed from three different perspectives: 1) Economic, finding that re-deploying smart contracts does not implicate considerable cost; 2) Computational perspective, finding that it generates various implications: smart contract purpose duplication, storage wastage, failure to recognize the original smart contract, cascade dependency repercussion, and migration problems; and 3) Interconnected effect, a simple change, required for upgrading smart contracts, generates broad collateral repercussions in both on-chain (within the blockchain) and off-chain.

INDEX TERMS Blockchain, software-development, smart contract, upgrading.

I. INTRODUCTION

Blockchain has revolutionized the digital world and provoked a new branch of software engineering known as blockchain-oriented software engineering [1], [2]. Inclusive subbranches about the software engineering testing stage have been studied [3]. Developing a blockchain software application can be challenging. Usually, it requires, among other things, a solid understanding of different paradigms of programming languages, cryptography, networking, distributed systems, and especially smart contracts. Yet software engineers could be proficient, they still require specialized tools for analyzing, designing, developing, and testing smart contracts [1], [2]. To make an analogy about deploying smart contracts within a blockchain network is like creating hardware devices: if your hardware device contains errors or limited functionalities, upgrading it is impossible, so you must implement a new version to replace the old one.

The following question arises: *Why do we need to emphasize the early phases of software development when building blockchain applications?* The following list describes some hypothetical arguments: a) smart contracts are immutable; one cannot just patch them easily if a bug or a vulnerability is found in the deployment phase; b) many smart contracts store and operate on critical and valuable assets (cryptos, NFTs); c) smart contracts offer transparency on public blockchains and are freely accessible; d) A smart contract is critical back-end code executed within a blockchain and requires interoperability with other software or systems.

The associate editor coordinating the review of this manuscript and approving it for publication was Antonio Piccinno.

The previous points make it highly attractive for malicious users to find an active or a passive attack. For example, even if the communication channel is secure or blockchain interoperability software [4] is surgically well designed, the system could be at risk if smart contracts contain vulnerabilities; this has been shown in [5], where financial losses due to smart contract attacks have been reported to increase annually since 2016. Therefore, the early phases in the software development process, especially when including blockchain applications, are critical due to either smart contracts' vulnerabilities or upgrading. Both cases imply changes in the smart contracts involved.

The application of the software development life cycle to smart contracts has attracted the attention of the research community. Some researchers have focused on evaluating and validating the quality of smart contracts within the development life cycle process [6], others on the challenges, techniques, and tools that blockchain-based applications face in the testing stage within software development [7], and some others on the need to standardize development processes due to the difficulty in updating or solving bugs when releasing a new software version involving smart contracts [8]. Other researchers have focused on reviewing the literature on smart contract design from the perspective of software engineering [8], a few have studied formal verification [9], and others have tried to understand the nature of smart contract vulnerabilities [10]. Finally, due to the importance of designing smart contracts without flaws, some researchers have focused on proposing tools to find vulnerabilities in smart contracts [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. Updating and upgrading smart contracts is a challenging task due to the immutability characteristic of blockchain. One evidence of the importance of this critical task is that some investigations have addressed this area [25], [26], [27], [28], [29], [30], [31], [32].

Updating or upgrading smart contracts not only affects the blockchain part but also has collateral effects in the off-chain architecture. To highlight this and the importance of the early phases of software development in blockchain-based software, this research follows a case-based study [33] to illustrate the implications of poorly designed or incorrect smart contracts. To do that, the paper presents a digital identity case designed within a microservice architecture. It analyzes the case from three different perspectives, namely: a) economic, via Ether currency, finding that redeploying smart contracts does not implicate considerable cost; b) computational, finding that it yields lots of issues: smart contract purpose duplication, storage wastage, failure to recognize the original smart contract, cascade dependency repercussion, and migration problems, and c) interconnected effect, finding that upgrading smart contracts involves more than just modifying the contract itself; it has broad and interconnected effects that impact both on-chain (within the blockchain) and off-chain (external systems connected to the blockchain) entities.

The following list shows the main contributions of this paper:

1) Providing higher certainty about the smart contracts' upgrading implications in the off-chain and the blockchain part to the software developer community;
2) Providing how a simple upgrading process generates high implications in the off-chain part when a blockchain application is designed within a microservice architecture;
3) A list of challenges that stem when an upgrading process is required once smart contracts have already been deployed;
4) An analysis of the economic repercussions of upgrading smart contracts concludes that a single, smart contract replacement is inexpensive.

The rest of this paper is organized as follows. Section II presents the background of this research, such as smart contracts and blockchain applications in the software development process, and also emphasizes the importance of early and late phases of software development when including blockchain technology. Section III presents a digital identity case based on microservice architecture and technical details in the off-chain and blockchain; this section emphasizes the specific part where a software engineer has to deal with smart contracts. Next, Section IV sets an upgrading scenario and the technological consequences in the off-chain part. Section V exposes some challenges of upgrading smart contracts. Then, Section VI explains the economic consequences of updating and upgrading smart contracts once they have been deployed. We discuss our research in section VII and draw our conclusions in Section VIII. Lastly, in Section IX, we formulate a list of questions to address future research.

II. BACKGROUND AND RELATED WORK

The importance of setting a rigorous development process while building blockchain applications has been reflected in the literature. This section starts explaining about blockchain and smart contracts; then, it gives a view of blockchain applications in the software development process. Our literature review considers two crucial elements: a) the importance of testing smart contracts and how this relates to vulnerability detection, and b) the challenge associated with upgrading smart contracts.

A. BLOCKCHAIN AND SMART CONTRACTS

Blockchain is a distributed database, where information is allocated in blocks. Each block is chained with the previous one by adding a hash address of the previous block, among other metadata, establishing a chain stored in a ledger. The distribution arises between different computers, called nodes, establishing a consensus algorithm to keep the same ledger locally.

Blockchain was initially developed to support Bitcoin [34]; however, over time other cryptocurrencies arose. An example of this is Ethereum [35], which, together with Bitcoin, is one

of the most successful cryptocurrencies. Ethereum introduced smart contracts, becoming one of the most sought-after technologies [36], [37].

Smart contracts refer to computer instructions stored within a block at a specific blockchain network address. They contain attributes and methods that can be treated similarly to classes in object-oriented programming. A modification in one attribute, whether directly or via the methods, results in a transaction identified with an address. Each transaction is stored in one of the blocks, which means that a set of transactions forms each block. One of the main characteristics of the blockchain network is that transactions cannot be erased.

B. BLOCKCHAIN APPLICATIONS IN SOFTWARE DEVELOPMENT

The software development process refers to the steps used to create software applications. These steps might include *requirements analysis*, *planning*, *design*, *implementation*, *testing*, *deployment*, *maintenance and support*; depending of the methodology, one iterates over these previous steps till a stop criterion is met. Developing blockchain applications consists of treating smart contract development as a rigorous engineering process with a strong focus on the first phases to ensure a contract's reliability, security, and functionality before it is deployed to a blockchain network. We refer to the early phases of software development such as *requirements analysis*, *planning*, *design*, *implementation*, and *testing*, leaving the rest *deployment*, *maintenance* and *support* for the production stage.

Sánchez-Gómez et.al. [6] reviewed the Software Development Life Cycle applied to the smart contract design and testing phase. They found a shortage of methodology for evaluating and validating the quality of the smart contract development life cycle process, namely: that software developers might implement smart contracts with serious bugs that may appear after deployment. They suggested that the scientific community should promote model-based testing because model-based software engineering makes it possible to find errors; this task is not simple since smart contract design requires expert knowledge of business, transactions, security, etc.

Lal and Marijan [7] carried out a study on the challenges, techniques, and tools that blockchain-based applications face in the testing stage within software development, finding that the key component requiring extensive and rigorous testing is smart contracts and that tools encompassing the whole stack do not exist yet, since testing these technologies requires multiple domains.

Vacca et al. [8], highlighted that smart contracts and blockchain applications are being developed through non-standard software life-cycles, and those delivered applications can hardly be updated or bugs resolved by releasing a new software version.

C. ON THE IMPORTANCE OF SMART CONTRACT TESTING AND VULNERABILITY DETECTION

Waseem et al. [38] studied the testing phase in microservices architecture-based applications, concluding that, although many researchers work in this area, it is still a challenging process.

Singh et al. [9], reviewed the literature (35 research papers) focused on the smart contracts' formal verification. They found that the most common techniques were theorem proving, symbolic execution, and model-checking. These techniques were mainly used to verify the smart contracts' security properties and functional correctness. They concluded that applying formal methods to smart contract verification is still in an infant domain; as it matures, formal methods can help solve and mitigate many vulnerabilities as they have been carried out in other domains.

Vacca et al. [8], made a literature review of 96 articles from 2016 to 2020 on software engineering, aiming at current problems in smart contracts and blockchain application development. They concluded that *further investigation* is required to understand how to apply traditional testing techniques to blockchain-oriented software; *methods* to measure specific code metrics, enabling code optimization; *guidelines for developers* (even a new programming language) to simplify the creation and understanding of smart contracts; and *patterns* to prevent developers from falling into the most common attacks.

Bhardwaj et al. [39], reviewed the literature (38 works) related to Blockchain and Security Testing from 2016 to 2019, finding some gaps: a) a new blockchain and a penetration testing classification are required; b) blockchain transactions latency issues; c) legal regulations are still required; d) cyber risks and privacy; and e) scalability. So, they created a penetration testing framework for smart contracts and decentralized applications and compared it with manual penetration testing, detecting missing vulnerabilities not reported during the regular testing process. Thus, they remarked that the testing stage is crucial since blockchain has irreversible transactions.

Zhou et al. [10], studied 13 vulnerabilities in Ethereum smart contracts and their countermeasures, finding the nonexistence of a uniform definition of vulnerabilities; sometimes, the same bugs could appear in the literature with different names.

Derived from the importance of designing smart contracts free of flaws, some researchers have focused on proposing tools focused on finding vulnerabilities in smart contracts: Sereum [11], SmartCheck [12], Osiris [13], NPChecker [14], MadMax [15], ContractWard [16], sfuzz [17], SMARTEMBED [18], DEFECTCHECKER [19], MODNN [20], [21], ContractCheck [22], ReenRepair [23] and Solvent [24]. SmartCheck, DEFECTCHECKER, ContractWard, sFuzz, SMARTEMBED, MODNN and ContractCheck tools seem more generalized in terms of coverage of vulnerabilities. In contrast, tools such as NPChecker, MadMax, Osiris,

Sereum, ReenRepair and Solvent are focused on some more specific categories of vulnerabilities.

D. ON UPGRADING SMART CONTRACTS

Immutability is one of the most important characteristics of blockchain. Consequently, it also makes updating smart contracts impossible, even when vulnerabilities have been found. Like all source programming code, smart contracts are error-prone. Therefore, software developers might question the updating or upgrading options. An upgradeable smart contract is a contract that incorporates upgrading approaches proposed by the community and is designed specifically for possible upgrading [32].

There are two general approaches for upgrading patterns developed by researchers to achieve upgradability on the Ethereum platform: data segregation patterns and proxy patterns. Wöhrer and Uwe, [25] introduced a data segregation pattern, which suggests that a contract should be written as two separate sub-contracts; a data contract and a logic contract. Upgradability is achieved by upgrading the logic contract without touching the data contract. The problem is whether the data contract must be updated in this case. The other approach uses a proxy contract to overtake the ownership of storage addresses of all versions of a given contract. Any call to the target contract would be redirected to the proxy contract, which will send all transactions to the same address it controls and achieve the upgradability goal. Three practices for upgrading smart contracts using proxy contracts are: ERC-1822 [26], ERC-2535 [27], and ERC-3448 [28]. The problem, in general, with this approach is whether the proxy contract must be updated and what will happen with the old smart contracts that are not already part of the storage addresses of the proxy smart contract.

Bui et al., [29] addressed the upgradability problem in Ethereum, finding that none of them have any security control to defend against typical attacks. They proposed a framework called The Comprehensive-Data-Proxy pattern, which uses data segregation on the top of the proxy pattern. Salehi et al. [30] also analyzed approaches addressed to the smart contract upgradability problem in Ethereum. They developed a measurement mechanism for finding the number of upgradeable contracts found on Ethereum using upgrade patterns; they found 8,225 upgradeable proxy contracts.

OpenZeppelin is a library for guiding the design of secure smart contracts, providing several patterns for implementing upgradability [40], being proxy patterns one of its stellar suggestion [41]. Amri et al. [42] reviewed the OpenZeppelin upgradeable patterns and compared them from different aspects, such as cost, performance, and security. However, they have suggested to conduct more evaluations in terms of performance, throughput, latency, and code complexity.

Chen et al. [31], focused on *smart contracts software maintenance* after smart contracts have been deployed. They argued that Ethereum blockchain contains many instances of smart contracts with vulnerabilities (maybe for copy-paste practices). They conducted a literature review of papers published from 2014-2020. Like other researchers, they concluded that the smart contract ecosystem must still be improved; for example, friendly tools to debug, test, and audit smart contracts are still required.

Even though previous strategies have been proposed to deal with the upgradeable problem and the proxy pattern remains the most widely used upgradeable approach, not all specialized developers apply it, and the truth is that once they are deployed, obsolete smart contracts prevail in the blockchain. The case presented in the following section aims to emphasize the early phases of the software development process before deploying smart contracts rather than to improve previous approaches. In addition, we show that changing smart contracts on the blockchain triggers significant ripple effects within the blockchain ecosystem and in the architecture that relies on the old smart contract, which needs to be reconfigured or updated to ensure proper integration with the new version of the smart contract.

III. THE CASE OF STUDY AND THE PROPOSED ARCHITECTURE

This section justifies the choice of microservice architecture and briefly describes such architecture. Later, it explains our digital identity case and how it has been adapted to the microservice architecture.

A. SOFTWARE DEVELOPMENT ARCHITECTURES

Monolithic architecture is a way to build software as a unit, where a central server has several responsibilities and does mostly everything. This architecture is still valid for small projects, but when organizations grow, it is not enough [43]. Microservice architecture is an alternative since it offers modularity, scalability, distributed development, and integration of heterogeneous and legacy systems [44]. While monolithic systems provide simplicity and ease of development, the microservice architecture design remains challenging, and it has become the leading design for cloud-native systems [45]; since it provides other benefits, such as better flexibility and independent deployment, [46].

Serverless architecture has become a new trendy topic when designing cloud-native systems. Unlike microservices, serverless architecture disregards management and server configuration [47]. Serverless is easier to manage and scales automatically, while microservices offer more control and flexibility over infrastructure. However, each architecture has advantages under different scenarios [47].

Furthermore, the benefits of using microservices architecture are confirmed from practitioners' point of view [48].

The software development process in the microservice architecture is not easy; its design, monitoring, and testing are more complex because it might be constituted for multiple nodes and technologies [49]. However, this distribution allows us to insert the blockchain part as a node and to make an analysis in detail.

B. MICROSERVICE ARCHITECTURE

In microservice architecture, a system is built as a collection of small, loosely coupled, independent services that communicate with each other over a network. Each service is designed to perform a specific function and can be developed, deployed, and scaled independently. This architecture permits adding technical components such as: a) an API gateway, which makes implementation and management simpler and more consistent [50]; b) containers, ensuring portability and consistency across environments, it is where services are implemented; c) storing information, each service might have its own database and d) network communication.

Figure 1 illustrates a general distributed software architecture divided into five entities: i) the user interacting from a personal computer; ii) the user interface, being accessed by the user, that obtains services from the back-end; iii) the API gateway, receiving and emitting requests by the user interface; iv) the services, obeying the requests of the API gateway and accessing directly to a smart contract within a blockchain node; and v) The blockchain node that contains the smart contracts. The network communication between these parts uses TLS/SSL over the secure hypertext transfer protocol (HTTPS). From the users' perspective, the architecture can be divided into two parts: front and back-end [51]. The front end involves the first two entities, and the back-end the rest.

C. THE DIGITAL IDENTITY CASE

The example presented here is related to digital identity, defined as the online or digital version of a person's identity. It includes personal information that can be used to identify a person, such as his complete name, email, address, date of birth, social media profiles, social security number, etc. In today's digital world, our secure online identity is essential. Digital identity can access various Internet services, such as financial information, medical records, emails, social networks, etc. More and more, our lives move online, and protecting our digital identity is becoming increasingly important. Digital identity information is commonly stored in traditional databases using encryption mechanisms to provide confidentiality, integrity, and replication processes to provide availability, among others. However blockchain technology is used under the argument that it offers some key reasons: i) Removes the need for intermediaries; ii) Reduces the risk of failures; iii) Ensures transparency; iv) Offers immutability; v) Reduces risks of fraud and tampering.

In the architecture was added a microservice called *Digital Identity* where the end user can obtain personal data, as shown in Figure 2. Additionally, the CURP microservice was added to complement the digital identity. CURP is a Spanish abbreviation of Unique Population Registration Code; it individually registers all people residing in Mexico, nationals and foreigners, and Mexicans living in other countries [52]. As you can see, the figure is compounded by six entities. The two first entities belong to the Front-End (*User* and *User Interface*). The rest of the entities belong to the Back-End (*API Gateway*, *Digital Identity*, *CURP*, and the *Blockchain* part). The figure also shows the technologies used during implementation in the bottom right part. The architecture was mounted on the Docker platform.

D. TECHNICAL DETAILS IN THE ARCHITECTURE

1) OFFCHAIN DETAILS

Offchain, in our architecture, refers to all software technology out of the blockchain. It includes the *User Interface* (UI), *API Gateway*, *Digital Identity*, and *CURP* microservices. The blockchain details are left for the following subsection.

The UI shows different options depending on the offered services, as shown in Table 1. Columns **Get** and **Set** refer (marked with x) if it is possible to consult or store such data in the microservice, respectively.

TABLE 1. Digital identity and CURP services specifying if the service corresponds to reading (get) or writing (set).

Services	Digital Identity		CURP	
	Get	Set	Get	Set
name	x	x	x	x
flastName	x	x	x	x
mlastName	x	x	x	x
day	x	x	x	x
month	x	x	x	x
year	x	x	x	x
dateCreation	x		x	
key			x	x
entity			x	x
dateRegistration			x	

Table 2 shows the technologies used in each entity over the operating system and the Node.js Web Server. Digital Identity and CURP microservices interact with the blockchain using Web3.js library. This library acts as a bridge between the microservices (coded in Node.js) and the blockchain network. It permits sending and consulting transactions. Furthermore, Web3.js can be used for large-scale deployments of smart contracts by carefully handling nonces, gas management, and rate limits to ensure successful execution.

2) BLOCKCHAIN DETAILS

We create an instance installing Ganache CLI technology. It is a fast and customizable blockchain emulator. It allows you to make calls to the blockchain without the overhead of running an actual Ethereum node. Transactions are mined instantly. Inside Ganache two smart contracts were implemented, details in the following two subsections.

3) SMART CONTRACT OF DIGITAL IDENTITY

Figure 3 shows, in the left part, a smart contract coded in Solidity Programming Language [53]. The contract (called *DIdentity*) includes attributes such as name, father and mother's last name, and date of birth. The smart contract is deployed by calling its constructor method and passing the respective attributes. The figure shows, in the right part,

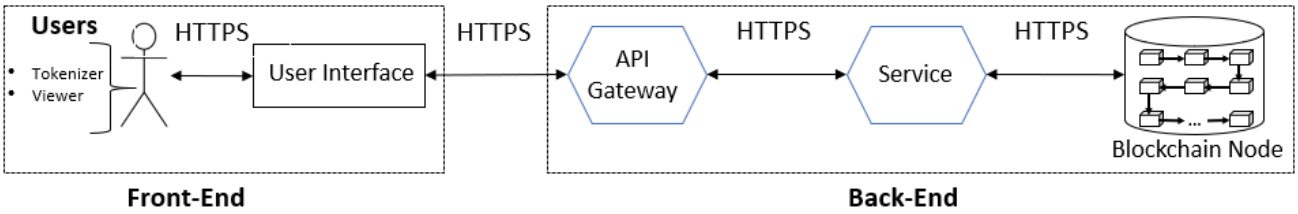

FIGURE 1. A general microservice architecture specifying two general parts: Front-end and Back-End.

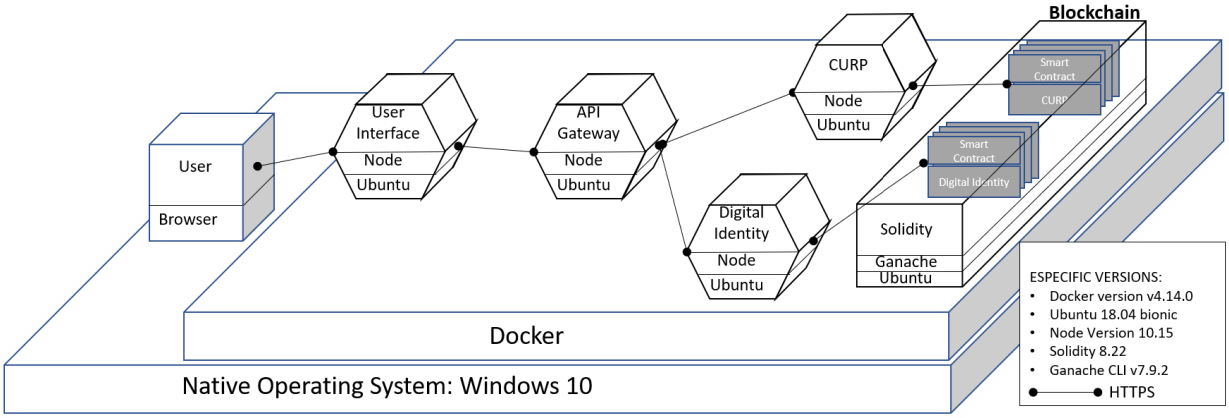

FIGURE 2. The implemented architecture based on microservices. Front-End: browser and the user interface; and the Back-End: the rest.

TABLE 2. Off-chain technologies used in the architecture.

Layer	User Interface	API Gateway	DIdentity	CURP
Framework	Express, Jade	Express API Gateway	Express	
Language programming	JavaScript	JavaScript (NodeJS)	JavaScript (NodeJS)	
Connectivity	HTTP library, AJAX	HTTP library	Web3.js	
Additional Technology	HTML, CSS	YAML		

a cloud, denoting the blockchain part and where the smart contract instances have been deployed; blue rectangles show three instances of *DIdentity*. Each instance shows its smart contract address (at the top) and the attributes' data (at the below).

4) EXTENDING DIDENTITY WITH THE CURP

With the CURP smart contract, we show how the previous smart contract might be extended without modification. To do this, the CURP case exemplifies it.

You can see in Figure 4 an example of a CURP smart contract. The figure shows the use of *DIdentity* smart contract; you can see in line 13, how an address of *DIdentity* is required. Functions of lines 19, 22, and 25 return the complete name, and the rest return the other attributes of *DIdentity* smart contract. The right part of the figure illustrates an example of an instance deployed in the cloud blockchain.

IV. UPGRADING: OFFCHAIN REPERCUSSION

Blockchain applications include other parts such as the user front-end (presentation logic), the business logic, the data access logic, and whether it is interconnected with other systems. These parts will also be affected if changes in the smart contracts are required. Let us illustrate the repercussions of upgrading smart contracts with an example in this section.

A. UPGRADING SCENARIO

We set a simple scenario in which an attribute was omitted in the initial design previously shown in Table 1. The attribute was called *gender*. Table 3 shows the type of access each microservice should hold. This new attribute is included in the initial design, which involves a list of changes in the blockchain and off-chain parts, which will be analyzed next.

TABLE 3. Specifying the ugrading service requirement *gender* in digital identity and CURP services.

Services	Digital Identity		CURP	
	Get	Set	Get	Set
gender	x	x	x	

B. CHANGES IN THE OFF-CHAIN PART

In follow-up to the technologies used to develop the microservices presented previously in Table 2, now Table 4 will describe the changes required for the new requirement.

```solidity
// SPDX-License-Identifier: jclopezpimentel
pragma solidity 0.8.22;

contract DIdentity{
    string public name;
    string public fLastName;
    string public mLastName;
    uint16 public day;
    uint16 public month;
    uint16 public year;
    // it contains the date the contract was created
     uint public dateCreation=0;

 constructor(string memory _name, string memory _fLastName,
    string memory _mLastName, uint16 _day, uint16 _month,
    uint16 _year) {
    name = _name;
    fLastName = _fLastName;
    mLastName = _mLastName;
    day = _day;
    month = _month;
    year = _year;
    dateCreation = block.timestamp;
  }
}
```

FIGURE 3. Smart contract digital identity (in Solidity Programming Language) and three instances.

TABLE 4. Involved off-chain technologies affected in the upgrading example.

Technologies		Change description
User Interface	Express, Jade, HTML, CSS	Changes in the view of the user
	JavaScript	Adding new field validation
	HTTP library, AJAX	Sending and Getting the answer of the request
API Gateway	YAML	Adding three new routes: 1 post and 2 get
DIdentity	JavaScript (NodeJS)	Adding two new routes: post and get
	Web3.js	Adding a consult and a post to the blockchain
CURP	JavaScript (NodeJS)	Adding a new route: get
	Web3.js	Adding a new access to the blockchain

As you can see, adding a simple attribute to the initial design implies a high repercussion.

One of the main characteristics of microservices architecture is the design of entities that are independent of each other. However, we have noted that our upgrading process involved a series of changes listed as follows:

i Number of microservices: in our case, there were only two microservices, and one of them, CURP, depends on Digital Identity, causing dependency. Because of that, identifying the appropriate boundaries for breaking down the application into microservices can be difficult. Deciding which functionalities should belong to each service requires careful consideration to avoid creating coarse-grained or too fine-grained services. For example, would it be better to add a new microservice for the new requirement shown in Table 3?

ii Different technologies: as you can see in table 4, each entity has its technology; coincidentally, some of them are repeated, but they could be completely different.

iii Re-testing: it involved testing individual services in isolation and testing across various services, mainly when some services depend on others. In addition, it involved testing the user interface and the API gateway again.

Software upgrading in the off-chain part might be tedious, but it is part of the software development cycle. Although complex, it does not hold the immutability characteristic that blockchain poses. The following section details it.

V. IMPLICATIONS FOR UPDATING OR UPGRADING SMART CONTRACTS

In software terms, updates provide patches and improve the program's performance. Upgrades refer to new software versions that bring new functions, tools, and significant improvements. From an economic point of view, the update is free, while the upgrade is not. However, in the case of blockchain, the immutability characteristic when smart contracts are deployed represents a main challenge in their updating and upgrading process. Some examples include bug fixes, improving security and performance, adapting compatibility, new features, etc. Updating methods in smart contracts have no implications in the off-chain part, only in the blockchain. As long as upgrading attributes or methods in smart contracts entail changes in both off-chain and blockchain, the focus of this paper.

```
1  // SPDX-License-Identifier: jclopezpimentel
2  pragma solidity 0.8.22;
3  import "./DIdentity.sol";
4  contract Curp{
5
6     DIdentity public videntity;
7         string public key;
8         string public entity;
9           uint public registration=0;
10
11
12    constructor(DIdentity _identity, string memory _key, string memory _entity) {
13      videntity = DIdentity(_identity);
14      key = _key;
15      entity = _entity;
16      registration = block.timestamp;
17    }
18
19      function getName() public view returns(string memory) {
20          return (videntity.name());
21      }
22      function getFLastName() public view returns(string memory) {
23          return (videntity.fLastName());
24      }
25      function getMLastName() public view returns(string memory) {
26          return (videntity.mLastName());
27      }
28      function getDay() public view returns (uint16){
29          return (videntity.day());
30      }
31      function getMonth() public view returns (uint16){
32          return (videntity.month());
33      }
34      function getYear() public view returns (uint16){
35          return (videntity.year());
36      }
37  }
```

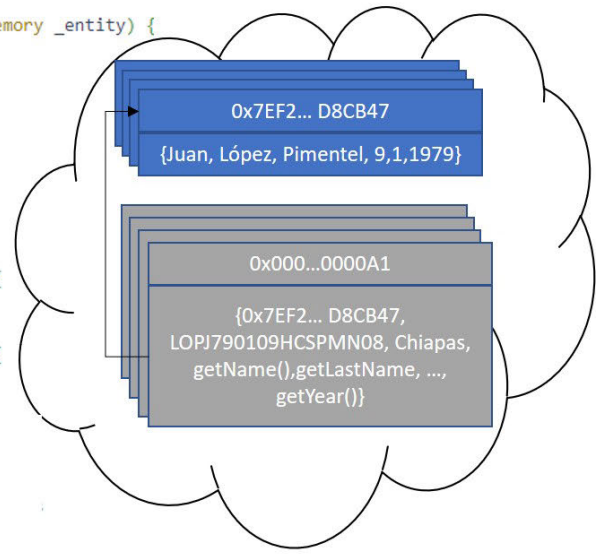

FIGURE 4. Smart contract CURP (in Solidity Programming Language) and one instance.

Upgrading new requirements as shown in Table 3 raises the question previously formulated: would it be better to add a new microservice implicating the creation of a new smart contract that could be linked to the already created smart contracts? Or would it be better to upgrade the previous smart contracts? We have chosen the path for the second question, not claiming it's the best, but necessary for our study. Our goal is to show its implications when an empirical upgrading process is developed and design patterns are not being considered, as we will describe next.

A. CHALLENGE OF UPDATING OR UPGRADING SMART CONTRACTS

Typically, smart contracts are unchangeable once deployed. This immutability characteristic means they cannot be updated or upgraded, although this brings new risks if vulnerability issues or new requirements arise.

Figure 5 illustrates an upgraded smart contract version of *DIdentity*, so-called *DIdentityV2*. This new version is very similar but adds the new attribute, *gender* (lines 9, 15, and 20 to note the difference). This new version exemplifies when changes are required because a specification was not considered in the initial design; in this case, a new required attribute is required.

Figure 6 shows instances of the smart contracts *DIdentity* and *DIdentityV2*. In the figure, blue rectangles illustrate instances of *DIdentity* shown previously in figure 3, and green rectangles illustrate instances of *DIdentityV2*.

Figure 6 can help us to explain the importance of designing correctly and completely smart contracts, which we argue as follows:

1) **Purpose duplication:** Smart contract *DIdentityV2* is a new and extended version of *DIdentity*; as you can see, the deployed smart contracts, embedded in the blockchain, store the logic and data encapsulated like a virtual object that can be located using their contract address. Making a mistake (or an omission) in its design stage would imply creating a new one and its respective instance in the blockchain. All this means generating more instances and then smart contract purpose duplication. See green instances for blue ones.
2) **Storage wastage:** Replacing a smart contract is not a simple task; it will also require knowing all implied transactions and replicating it in the new one or

```
1  // SPDX-License-Identifier: jclopezpimentel
2  pragma solidity 0.8.22;
3
4  contract DIdentityV2{
5
6      string public name;
7      string public fLastName;
8      string public mLastName;
9      bool public gender; //true will be man and false woman
10     uint16 public day;
11     uint16 public month;
12     uint16 public year;
13     uint public dateCreation=0; // it contains the date the contract was created
14
15   constructor(string memory _name, string memory _fLastName, string memory _mLastName, bool _gender,
16                 uint16 _day, uint16 _month, uint16 _year) {
17     name = _name;
18     fLastName = _fLastName;
19     mLastName = _mLastName;
20     gender = _gender; //true will be man and false woman
21     day = _day;
22     month = _month;
23     year = _year;
24     dateCreation = block.timestamp;
25   }
26 }
```

FIGURE 5. ***DIdentityV2*: a new version on the previous *DIdentity* smart contract.**

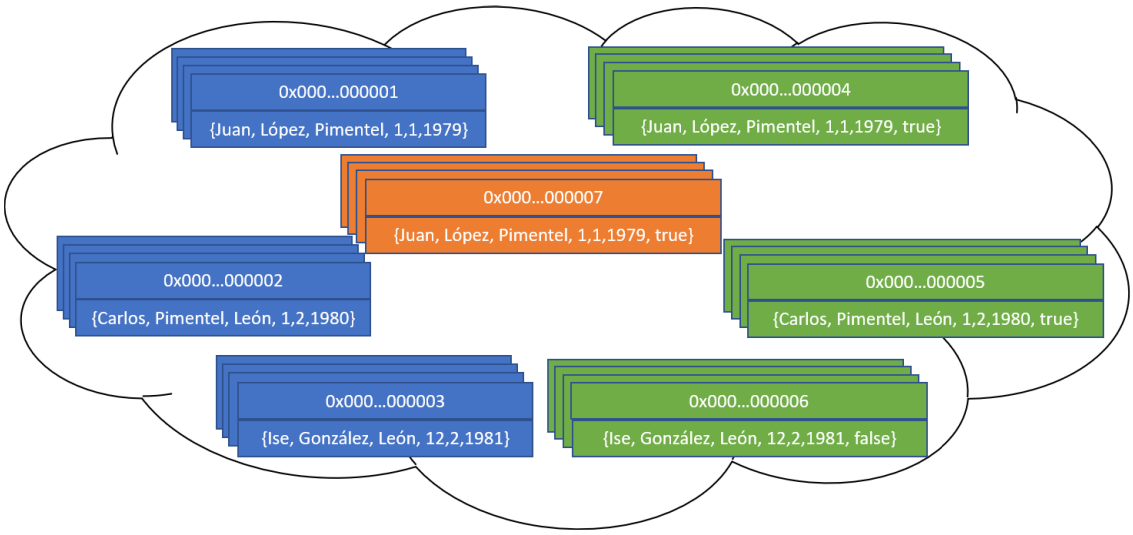

FIGURE 6. **Smart contracts duplication deployed in the blockchain with the same purpose, showing storage wastage and originality troubles.**

only replicating the last state of the older contract. Independently of the option, the older deployed smart contract cannot be erased, causing storage wastage. In our example, green instances were replicated, and blue instances caused storage wastage.

3) **Legitimacy:** How to avoid creating two or more instances of smart contracts linked to the same identity? See the orange instance contract in the figure with address $0 \times 000 \ldots 00007$ containing the same data of $0 \times 000 \ldots 00001$ and $0 \times 000 \ldots 00004$. The question here is, which contract address is the original? and what is a copy? Maybe checking the timestamp attribute could help to bow for the first one created. However, it is required to trust in another instance that would be storing the original and trusty smart contract address.
4) **Cascade dependency repercussion:** The problem is incremented when not only *VIdentity* is involved. As you can remember, CURP smart contract also uses *VIdentity* in line 13, binding a strong repercussion because now CURP smart contract would also have to be modified

(see figure 7 in the code section); and by consequence more duplication, as you can see in the figure, at the bottom part.

5) **Migration:** This is a critical process. It requires knowing all states and the ownership executor of the old smart contracts to generate a new version. This new version must consider different aspects such as cost, complexity of the smart contracts, data (whether critical or not), and a new testing plan to avoid security risks.

After listing the challenges that arise when upgrading already deployed smart contracts, the next section will provide an economic analysis of this process. This will help determine whether the cost of upgrading smart contracts might be a significant factor in deciding against using blockchain.

VI. ECONOMICAL ANALYSIS: UPGRADING THE SMART CONTRACTS

This section analyzes the economic impact generated by upgrading the case presented previously. It shall evaluate the individual cost generated by each smart contract; then, It shall analyze the cost implications if deployed in a country like Mexico with 137.2 million inhabitants. Finally, the individual upgrading cost generated by each smart contract and the collective upgrading cost considering the current Mexican inhabitants.

A. INDIVIDUAL COST OF THE SMART CONTRACTS

This section details the data and accounts used to deploy the smart contracts. It shows each transaction cost of the smart contracts' attributes and methods after a deployed or consulting execution. Data $d1$ has been used to deploy smart contract *DIdentity*:

```
d1 ={
 "name": "Juan Carlos",
 "fLastName": "Lopez",
 "mLastName": "Pimentel",
 "day": "9",
 "month": "1",
 "year": "1979"
}
```

The user account was:

```
A_1="0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"
```

After deployment, the resulting contract address was:

```
S_1="0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47"
```

Then, $d2$ was used to deploy smart contract *CURP*:

```
d2 ={
 "identity":"0x7EF2e0048f5 ... 943daF4ED8CB47",
    "key" :"0xAb8483F64d9 ... 77dD3315835cb2",
   "entity":"Chiapas"
}
```

The contract address generated after deployment was:

```
S_2 = "0x9bF88fAe8CF8BaB76041c1db6467E7b37b977dD7"
```

Finally, the following smart contract addresses were generated when VIdentityV2 and CURPV2 were deployed:

```
S_3 = "0xA831F4e5dC3dbF0e9ABA20d34C3468679205B10A"
S_4 = "0xB302F922B24420f3A3048ddDC4E2761CE37Ea098"
```

Tables 5 to 8 show the transaction costs carried out after deployment and executing a consulting process in each attribute and method. For the tables, column **Executor** indicates who has executed the transaction (a user account or a contract address); column **Operation's type** shows the type of transaction deployment or consulting; column **Method or attribute** shows the attribute name or method being executed; finally, the last two columns show the **transaction cost**: **User** presents the cost when it is executed by a user account address (inserting the smart contract in the blockchain); and Column **Contract** shows the cost when an attribute or method is consulted.

As you can see in all tables, deploying a smart contract to the blockchain is overly expensive compared with only executing a simple consulting process. On the other hand, in table 6 by consulting method *getName*(), smart contract CURP has to access to *DIdentity*, meaning indirect access; thereby, the cost is upper than those only accessing to one contract. This is similar to the other methods in the same situation, as shown in figure 8. three The formula of the transaction cost is:

$$TransactionCost = gasUsed \times gasPrice$$

The transaction costs of Tables 5 to 8 were generated using the Remix Integrated Development Environment (IDE). The *gasPrice* was 1. This means that each calculated transaction cost was equal to the gas used. The transaction cost was denominated in Wei, which refers to the smallest denomination of ether (ETH), the currency used on the Ethereum network: 1 ether (ETH) equals 1×10^{18} Wei.

B. COST DEPLOYING CURP SMART CONTRACT CONSIDERING MEXICAN INHABITANTS

This section shows the transaction costs generated to deploy the CURP smart contract and the costs generated by consulting. Note that to execute this contract, it is also required to execute *DIdentity*. Then, we execute a calculation to determine the total cost, considering the Mexican inhabitants.

The following abbreviations will express some costs obtained from Tables 5 and 6.

C_{DI}, it is the deploying cost of DIdentity obtained from table 5.

C_{DC}, it is the deploying cost of CURP obtained from table 6.

C_{IC}, it is the deploying cost of DIdentity and CURP obtained from:

$$C_{IC} = C_{DI} + C_{DC}$$

In this case:

$$C_{IC} = 1,216,351\ W\ eis$$

Let Cc be a CURP's consulting cost. If we consider a user consulting each attribute and method of its CURP:

$$\sum_{i=1}^{n} Cc_i$$

```
// SPDX-License-Identifier: jclopezpimentel
pragma solidity 0.8.22;
import "./DIdentityV2.sol";
contract CurpV2{

    DIdentityV2 public videntity;
        string public key;
        string public entity;
          uint public registration=0;


    constructor(DIdentityV2 _identity, string memory _key, string memory _entity) {
      videntity = DIdentityV2(_identity);
      key = _key;
      entity = _entity;
      registration = block.timestamp;
    }

      function getName() public view returns(string memory) {
          return (videntity.name());
      }
      function getFLastName() public view returns(string memory) {
          return (videntity.fLastName());
      }
      function getMLastName() public view returns(string memory) {
          return (videntity.mLastName());
      }
      function getGender() public view returns (bool){
          return (videntity.gender());
      }
      function getDay() public view returns (uint16){
          return (videntity.day());
      }
      function getMonth() public view returns (uint16){
          return (videntity.month());
      }
      function getYear() public view returns (uint16){
          return (videntity.year());
      }
}
```

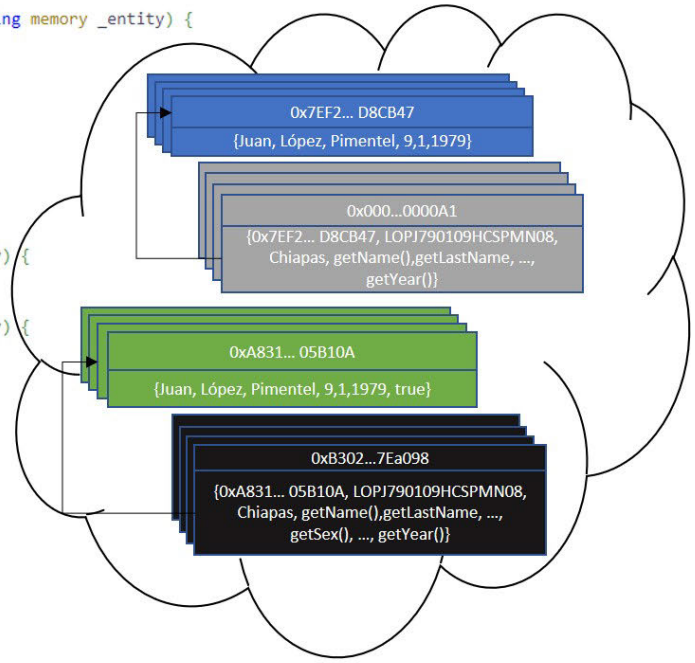

FIGURE 7. CURP version 2 (left part) and the repercussion (right part): double duplication, showing instances of *DIdentity-Curp* smart contracts and *DIdentityV2-CurpV2*.

TABLE 5. Transaction costs of smart contract DIdentity.

Executor	Operation's type	Method or attribute	Transaction cost	
			User	Contract
A_1	Deploying	Constructor	463841	
S_1	Consulting	dateCreation		2469
S_1	Consulting	name		3412
S_1	Consulting	fLastName		3435
S_1	Consulting	mLastName		3457
S_1	Consulting	day		2488
S_1	Consulting	month		2517
S_1	Consulting	year		2583

then, it adds:

$$\sum_{i=1}^{10} Cc_i = 65,986\ W\ eis$$

Considering both the deployment of DIdentity and CURP; and all costs of the CURP's consulting attributes and methods:

$$Cc_{D\&C} = C_{IC} + \sum_{i=1}^{n} Cc_i$$

TABLE 6. Transaction costs of smart contract Curp.

Executor	Operation's type	Method or attribute	Transaction cost	
			User	Contract
A_1	Deploying	Constructor	752510	
S_2	Consulting	key		3479
S_2	Consulting	entity		3434
S_2	Consulting	vIdentity		2745
S_2	Consulting	registration		2425
S_2	Consulting	getName()		9873
S_2	Consulting	getFLastName()		9962
S_2	Consulting	getMLastName()		9939
S_2	Consulting	getDay()		7958
S_2	Consulting	getMonth()		8031
S_2	Consulting	getYear()		8140

obtaining:

$$Cc_{D\&C} = 1,282,337\ W\ eis$$

Let M be the number of Mexicans being 137.2 million,[1] and considering an instance by each Mexican, the total deployed

[1] At the time the paper was written.

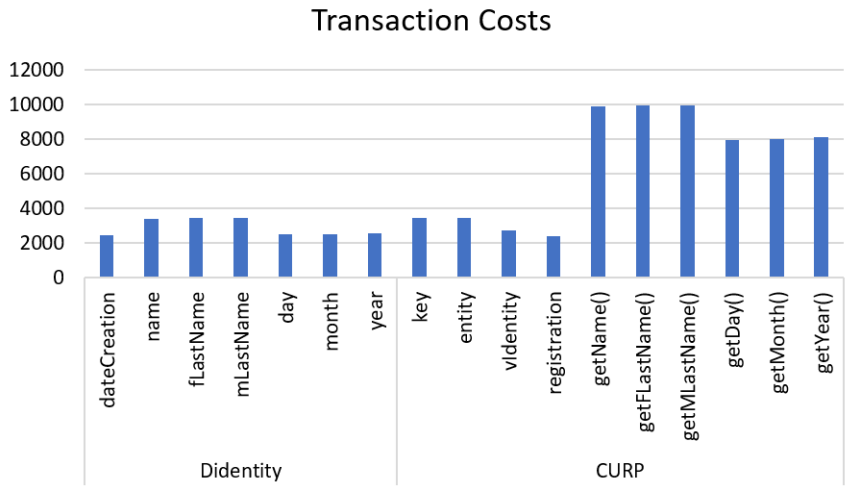

FIGURE 8. Transaction costs of smart contracts ***DIdentity*** and ***CURP*** omitting their deploying costs.

cost:

$$C_T = Cc_{D\&C} \times M$$

it was:

$$C_T = 175, 936, 636, 400, 000\ W\ eis$$

it is equal to 0.0001759366364 Ethers. Taking into account 1 Ether equals 3825.73 USD[2] the amount C_T would be 0.67 USD. Considering blockchain's benefits, we conclude that having all Mexican CURPs in a blockchain platform is not expensive. three However, it is important to consider that the gas price unit in our calculus was 1. The gas price in Ethereum is highly variable and depends on network congestion, which we have not considered.

C. THE UPGRADING COST OF RE-DEPLOYING CURPV2 CONSIDERING MEXICAN INHABITANTS

This section obtains the costs implicated by adding the upgrading process as exposed in section IV. Although the first version of smart contract CURP differs only by the new attribute for the second version, each consulting transaction cost might differ. As you can compare it in Tables 6 and 8, for example, attribute *entity* are equals, but attribute *vIdentity* are different. Thereby, it is impossible to calculate the personal migration cost C_{PM} with the following simple formula:

$$C_{PM} = (2 * Cc_{D\&C}) + Cc_{n+1}$$

being Cc_{n+1} the new attribute.

Instead, the personal migration cost C_{PM} is calculated such as the sum of the old smart contracts costs and the new one:

$$C_{PM} = Cc_{D\&C} + Cc'_{D\&C}$$

Therefore, $Cc'_{D\&C}$ is calculated similarly to $Cc_{D\&C}$, as follows:

$$Cc'_{D\&C} = C_{IC2} + \sum_{i=1}^{n+1} Cc'_i$$

where C_{IC2} is the cost generated when deployed DIdentityV2 and CURPV2, obtained from:

$$C_{IC2} = C_{DI2} + C_{DC2}$$

there, C_{DI2} and C_{DC2} are the cost generated when DIdentityV2 and CURPV2 were deployed respectively.

So, the upgrading cost might be broken down as follows:

$$\begin{aligned} C_{DI2} &= 489, 725 \\ C_{DC2} &= 822, 577 \\ C_{IC2} &= 1, 312, 302 \\ \sum_{i=1}^{n} Cc'_i &= 74, 167 \\ Cc'_{D\&C} &= 1, 386, 469 \\ C_{PM} &= 2, 668, 806 \end{aligned}$$

The Mexican migration cost C_M considering M (the number of Mexicans) is abstracted in the following formula:

$$\begin{aligned} C_M &= C_{PM} \times M \\ C_M &= 366, 160, 183, 800, 000 \end{aligned}$$

equal to 0.0003661601838 Ethers.

Again, taking into account 1 Ether equals 3825.73 USD[3] the amount C_M would be 1.40 USD. With this, we conclude that the upgrading spend for re-deploying CURPV2, considering Mexican inhabitants, is not expensive according to the current exchange. In other words, if the economic aspect of

[2]Exchange rate calculated at https://www.coinbase.com/converter/eth/usd on June 5th, 2024.

[3]Exchange rate calculated at https://www.coinbase.com/converter/eth/usd on June 5th, 2024.

TABLE 7. Transaction costs of smart contract DIdentityV2.

Executor	Operation's type	Method or attribute	Transaction cost	
			User	Contract
A_1	Deploying	Constructor	489725	
S_3	Consulting	dateCreation		2448
S_3	Consulting	name		3412
S_3	Consulting	fLastName		3457
S_3	Consulting	mLastName		3479
S_3	Consulting	day		2539
S_3	Consulting	month		2514
S_3	Consulting	year		2561
S_3	Consulting	gender		2532

TABLE 8. Transaction costs of smart contract CurpV2.

Executor	Operation's type	Method or attribute	Transaction cost	
			User	Contract
A_1	Deploying	Constructor	822577	
S_4	Consulting	key		3479
S_4	Consulting	entity		3434
S_4	Consulting	vIdentity		2767
S_4	Consulting	registration		2425
S_4	Consulting	getName()		9873
S_4	Consulting	getFLastName()		9984
S_4	Consulting	getMLastName()		9983
S_4	Consulting	getDay()		8009
S_4	Consulting	getMonth()		8028
S_4	Consulting	getYear()		8140
S_4	Consulting	getGender()		8045

upgrading smart contracts is a concern, then it should not be, according to our results three and the gas price we set in the Remix IDE.

VII. DISCUSSION

This section discusses the upgrading case developed throughout the paper from two focuses: Off-chain and Blockchain. Before that, we will give some assumptions that were not part of our scope.

A. DELIMITATION

To delimit this research, the following list describes some assumptions out of the scope of this paper:

- Consensus algorithm: we worked on the existing consensus algorithm implemented on the Ethereum network. Any update or upgrade in this aspect was not considered.
- Secure blockchain: we left out the reliability and safety of the blockchain network. We assumed it was free of attacks, so we disregarded any possible update in the mining process.
- Security between microservices: security is a crucial aspect of a system, but our analysis did not include possible vulnerabilities in the interaction between microservices and smart contracts.
- Tunneling: we assumed tunneling privacy communication in our architecture using HTTPS. We assumed this protocol was free of attacks. Any update would not affect the upgrades of smart contracts.
- Exclusively in the system development: We did not include updates or upgrades in the system and programming software; it was the technologies over which the software was mounted or developed, respectively.
- Excluding practitioner costs: it excluded the human resource cost generated for making the changes in the blockchain and the off-chain.
- Exclusively smart contract deployment: the study of our economic analysis primarily focuses on the perspective of smart contracts' deployment. Additional costs (e.g. gas fee optimizations, batching transactions) were not considered.

B. OFF-CHAIN PART: MICROSERVICE ARCHITECTURE

Software development is an iterative process. Updating and upgrading is an active part of this cycle. Changes in the first steps will have major repercussions compared to later stages, especially if implemented in a microservice approach. As follows, we state our analysis:

- High repercussion: although one of the philosophies of the microservice architecture is that each service is independent of the others, we have demonstrated high complexity, even when making simple changes. For example, a simple upgrading, as described in Section IV, involved identifying various affected parts: microservices, the API gateway, and user interfaces.
- Specialist: if these parts were developed with different technologies and programming languages, then upgrading implies having specialists with different knowledge and profiles. Additionally, it is required to add the blockchain specialist.
- Version controls: each part might have various version controls, complicating their administration.
- Off-chain processing delegation: delegating processing and validation tasks within the microservices is highly recommended. On the one hand, it could avoid unnecessary cost transactions in the blockchain; on the other, it could prevent an upgrading process in smart contracts deployed in private blockchains.
- Re-testing: for security issues, it is required to execute the testing stage again, from individual services in isolation and testing across various services, mainly when some services depend on others. Testing can be complex and time-consuming but highly required. Stress testing (multiple concurrent requests) is highly recommendable; we have found bugs while executing transactions to the smart contracts that, with simple requests, were not found.
- Handling data: blockchain applications require storing data via off-chain and blockchain; it can be more complex, especially when dealing with data consistency, transactions, and inter-service communication. For example, where must the user's addresses be stored?

C. UPGRADING ANALYSIS ON THE BLOCKCHAIN PART

Immutability is one of the more important characteristics of blockchain. However, it also makes patching or updating

smart contracts impossible, even when vulnerabilities have been found or poorly designed. Like all source programming code, smart contracts are error-prone, or perhaps software developers might want to extend their features. The importance of designing correctly and completely smart contracts is argued as follows:

1) Purpose duplication: a bug or an omission in their design stage would imply creating and deploying new smart contracts and causing the old ones to fall into disuse and stay in the blockchain.
2) Storage wastage: if a new smart contract substitutes another, the older one cannot be erased; storage wastage will occur, especially when the new one must implement all states of the older one. We coincide with Chen et al. [31], who comment that many dead smart contracts are already unused but add noise to the Ethereum blockchain.
3) Legitimacy: if a blockchain system contains two or more smart contracts with the same purpose, how do we know which is the original? In this case, an external instance will be required to store the original smart contract address.
4) Cascade dependency repercussion: some smart contracts extend or use attributes or methods of other smart contracts. A cascading repercussion exists if the latter is modified because all dependent smart contracts must be modified.
5) Validations within smart contracts: delegating validation tasks within the microservices is highly recommended instead of doing everything within the smart contracts. However, smart contracts must include their validation to avoid negative impacts when implementing microservices with errors.
6) Migration: the consequences of migrating new smart contracts after deployment can vary depending on the specific context and the nature of the migration. Here we mention some potential consequences:
    - Migration data integrity: as shown, when smart contracts are involved in an updating or upgrading procedure, they may require transferring data to the new attributes, which can be error-prone or introduce potential risks related to data integrity and consistency. It is possible to have two scenarios when migrating smart contracts: a) completely, it is required to know all states of the previous smart contracts to generate the new version, and b) partially, the complexity, in this case, is to identify exactly what parts will be migrated.
    - Executor: it must be considered the permissions of the smart contracts to be migrated to consider who can execute the new smart contracts and how to obtain the list of all transactions to replicate it in case that is required.
    - Migration cost: migrating smart contracts typically involves transaction fees on the blockchain network; these fees can vary and may be significant or not (as shown in Section VI), but they must be considered.
    - Migration complexity: as shown with the smart contract examples, migration can be complex, especially if there are dependencies or interactions with other smart contracts (e.g., *DIdentity* and *CURP*). Ensuring its correct migration process requires careful planning and re-testing process.

VIII. CONCLUSION

This paper provides higher certainty about the smart contracts' upgrading implications in the off-chain and the blockchain part to the software developer community, and organizations that could constantly be undecided if adopting or not blockchain technology without assessing whether it is truly suitable. For example, a simple upgrading process when designing a blockchain application within a microservice architecture generates high implications in the off-chain and the blockchain parts that should be considered.

When developing blockchain-based software engineering consists of treating smart contract development as a rigorous engineering process with a strong focus on *requirements analysis*, *planning*, *design*, *implementation*, and *testing* (early phases). These phases ensure the smart contracts' reliability, security, and functionality before deploying them to a blockchain network. Even so, sometimes their design could omit some attributes and/or methods that could impact enormously in the blockchain part and the off-chain, as we have discussed in Section VII, therefore and due to the immutable nature of blockchain technology, the use of upgrading design patterns is recommended.

However, our first experiments showed that the economic costs are not so high when smart contract re-deployment is caused by changing a simple attribute and involves only a few smart contracts. The previous argument is supported by an experiment with two smart contracts that required to be upgraded; it was deployed considering 137.2 million instances three and the gas price fixed to 1 unit. Note that our upgrading procedure did not follow any design pattern (as those mentioned in Section II-D); however, analyzing and comparing the costs considering design patterns would help to determine the difference for or against when using these strategies.

IX. FUTURE DIRECTIONS

Blockchain technology requires more study in some areas of software development to become consolidated and widely used. So, based on our research, the next step will be to question how software and system engineers deal with the topics we have discussed. Furthermore, we give for each question a citation of recent works addressing these research directions.

Based on the analysis of Sections IV and V:

- Migration cost: what are the complete migration costs considering offchain costs and other costs out of

transaction deployments, such as gas fee optimization and batching transactions, or those considering pattern design?, see [54].
- Security issues: what are the most common potential attacks within smart contracts that can be avoided from the off-chain part?, see [55].

Based on the analysis of Section V:

- Purpose duplication: how to clean the noise of having smart contracts in disuse on the blockchain?, see [31].
- Legitimacy: what are the mechanisms for knowing the original smart contract when having more than one for the same objective in the blockchain?, see [56].
- Migration process: what are the main mechanisms for executing automated migration in partial or complete states? How does it affect permission restrictions while considering a migration process? Smartmuv is a tool addressing lightly this issue [57].

Although our findings, Section VI, show that the economic costs were not so high, they were grounded on a case study. Comparing the upgrade costs in scenarios following the pattern design, as shown in [54], and fluctuations in the gas price would complement our findings.

REFERENCES

[1] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: Challenges and new directions," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2017, pp. 169–171.

[2] A. Al-Ashmori, S. Basri, P. D. D. Dominic, A. Muneer, Q. Al-Tashi, and Y. Y. Al-Ashmori, "Blockchain-oriented software development issues: A literature review," in *Proc. 5th Comput. Methods Syst. Softw. Eng. Appl. Inform.*, in Lecture Notes in Networks and Systems, Z. P. Radek Silhavy, Petr Silhavy, Eds., Cham, Switzerland: Springer, Jan. 2021, pp. 48–57.

[3] S. Reddivari, J. Orr, and R. Reddy, "Blockchain-oriented software testing: A preliminary literature review," in *Proc. IEEE 47th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jun. 2023, pp. 974–975.

[4] A. Augusto, R. Belchior, M. Correia, A. Vasconcelos, L. Zhang, and T. Hardjono, "SoK: Security and privacy of blockchain interoperability," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2024, pp. 3840–3865.

[5] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li, "A survey on smart contract vulnerabilities: Data sources, detection and repair," *Inf. Softw. Technol.*, vol. 159, Jul. 2023, Art. no. 107221. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584923000757

[6] N. Sánchez-Gómez, J. Torres-Valderrama, J. A. García-García, J. J. Gutiérrez, and M. J. Escalona, "Model-based software design and testing in blockchain smart contracts: A systematic literature review," *IEEE Access*, vol. 8, pp. 164556–164569, 2020.

[7] C. Lal and D. Marijan, "Blockchain testing: Challenges, techniques, and research directions," 2021, *arXiv:2103.10074*.

[8] A. Vacca, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges," *J. Syst. Softw.*, vol. 174, Apr. 2021, Art. no. 110891.

[9] A. Singh, R. M. Parizi, Q. Zhang, K.-K.-R. Choo, and A. Dehghantanha, "Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities," *Comput. Secur.*, vol. 88, Jan. 2020, Art. no. 101654.

[10] H. Zhou, A. Milani Fard, and A. Makanju, "The state of Ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support," *J. Cybersecurity Privacy*, vol. 2, no. 2, pp. 358–378, May 2022.

[11] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," 2018, *arXiv:1812.05934*.

[12] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2018, pp. 9–16.

[13] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 664–676.

[14] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in Ethereum smart contracts," in *Proc. ACM Program. Lang.*, vol. 3, Oct. 2019, pp. 1–29.

[15] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Analyzing the out-of-gas world of smart contracts," *Commun. ACM*, vol. 63, no. 10, pp. 87–95, Sep. 2020.

[16] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: Automated vulnerability detection models for Ethereum smart contracts," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1133–1144, Apr. 2021.

[17] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "SFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jan. 2020, pp. 778–788.

[18] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Trans. Softw. Eng.*, vol. 47, no. 12, pp. 2874–2891, Dec. 2021.

[19] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2189–2207, Jul. 2022.

[20] L. Zhang, J. Wang, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen, "A novel smart contract vulnerability detection method based on information graph and ensemble learning," *Sensors*, vol. 22, no. 9, p. 3581, May 2022.

[21] L. Zhang, J. Wang, W. Wang, Z. Jin, Y. Su, and H. Chen, "Smart contract vulnerability detection combined with multi-objective detection," *Comput. Netw.*, vol. 217, Nov. 2022, Art. no. 109289.

[22] X. Wang, S. Tian, and W. Cui, "ContractCheck: Checking Ethereum smart contracts in fine-grained level," *IEEE Trans. Softw. Eng.*, vol. 50, no. 7, pp. 1789–1806, Jul. 2024.

[23] R. Huang, Q. Shen, Y. Wang, Y. Wu, Z. Wu, X. Luo, and A. Ruan, "ReenRepair: Automatic and semantic equivalent repair of reentrancy in smart contracts," *J. Syst. Softw.*, vol. 216, Oct. 2024, Art. no. 112107. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224001523

[24] M. Bartoletti, A. Ferrando, E. Lipparini, and V. Malvone, "Solvent: Liquidity verification of smart contracts," 2024, *arXiv:2404.17864*.

[25] M. Wöhrer and U. Zdun, "Design patterns for smart contracts in the Ethereum ecosystem," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Phys. Social Comput. (CPSCom) IEEE Smart Data (SmartData)*, Jul. 2018, pp. 1513–1520.

[26] G. Barros and P. Gallagher. (2019). *Erc-1822: Universal Upgradeable Proxy Standard (uups)*. Accessed: Sep. 21, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1822#motivation

[27] N. Mudge. (2020). *Erc-2535: Diamonds, Multi-facet Proxy*. Accessed: Sep. 21, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2535

[28] Pinkiebell. (2021). *Erc-3448: Metaproxy Standard*. Accessed: Sep. 21, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-3448#motivation

[29] V. C. Bui, S. Wen, J. Yu, X. Xia, M. S. Haghighi, and Y. Xiang, "Evaluating upgradable smart contract," in *Proc. IEEE Int. Conf. Blockchain*, Dec. 2021, pp. 252–256.

[30] M. Salehi, J. Clark, and M. Mannan, "Not so immutable: Upgradeability of smart contracts on Ethereum," in *Proc. Int. Workshops Financial Cryptography Data Security (FC)*, vol. 13412. Cham, Switzerland: Springer, 2023, pp. 539–554.

[31] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed Ethereum smart contract development: Issues, techniques, and future challenges," *Empirical Softw. Eng.*, vol. 26, no. 6, p. 117, Nov. 2021.

[32] I. Qasse, M. Hamdaqa, and B. P. Jónsson, "Smart contract upgradeability on the Ethereum blockchain platform: An exploratory study," 2023, *arXiv:2304.06568*.

[33] R. K. Yin, *Case Study Research: Design and Methods*, vol. 5. Newbury Park, CA, USA: Sage, 2009.

[34] S. Nakamoto and A. Bitcoin, "Bitcoin: A peer-to-peer electronic cash system," *Bitcoin*, vol. 4, no. 2, p. 15, 2008.

[35] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Jan. 2014. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[36] D. Macrinici, C. Cartofeanu, and S. Gao, "Smart contract applications within blockchain technology: A systematic mapping study," *Telematics Informat.*, vol. 35, no. 8, pp. 2337–2354, Dec. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0736585318308013

[37] E. Negara, A. Hidayanto, R. Andryani, and R. Syaputra, "Survey of smart contract framework and its application," *Information*, vol. 12, no. 7, p. 257, Jun. 2021. [Online]. Available: https://www.mdpi.com/2078-2489/12/7/257

[38] M. Waseem, P. Liang, G. Márquez, and A. D. Salle, "Testing microservices architecture-based applications: A systematic mapping study," in *Proc. 27th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2020, pp. 119–128.

[39] A. Bhardwaj, S. B. H. Shah, A. Shankar, M. Alazab, M. Kumar, and T. R. Gadekallu, "Penetration testing framework for smart contract blockchain," *Peer-Peer Netw. Appl.*, vol. 14, no. 5, pp. 2635–2650, Sep. 2021.

[40] OpenZeppelin. (2024). *Proxy Upgrade Pattern*. Accessed: Oct. 4, 2024. [Online]. Available: https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies

[41] OpenZeppelin. (2024). *Proxies*. Accessed: Oct. 4, 2024. [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/proxy

[42] S. A. Amri, L. Aniello, and V. Sassone, "A review of upgradeable smart contract patterns based on OpenZeppelin technique," *J. Brit. Blockchain Assoc.*, vol. 6, no. 1, pp. 1–8, Mar. 2023. [Online]. Available: https://eprints.soton.ac.uk/491646/

[43] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *Proc. 24th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2017, pp. 466–475.

[44] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using Docker technology," in *Proc. SoutheastCon*, Mar. 2016, pp. 1–5.

[45] V. Bushong, A. S. Abdelfattah, A. A. Maruf, D. Das, A. Lehman, E. Jaroszewski, M. Coffey, T. Cerny, K. Frajtak, P. Tisnovsky, and M. Bures, "On microservice analysis and architecture evolution: A systematic mapping study," *Appl. Sci.*, vol. 11, no. 17, p. 7856, Aug. 2021.

[46] J. Christian, Steven, A. Kurniawan, and M. S. Anggreainy, "Analyzing microservices and monolithic systems: Key factors in architecture, development, and operations," in *Proc. 6th Int. Conf. Comput. Informat. Eng. (IC2IE)*, Sep. 2023, pp. 64–69.

[47] C.-F. Fan, A. Jindal, and M. Gerndt, "Microservices vs serverless: A performance comparison on a cloud-native Web application," in *Proc. 10th Int. Conf. Cloud Comput. Services Sci.*, 2020, pp. 204–215.

[48] X. Zhou, S. Li, L. Cao, H. Zhang, Z. Jia, C. Zhong, Z. Shan, and M. A. Babar, "Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry," *J. Syst. Softw.*, vol. 195, Jan. 2023, Art. no. 111521.

[49] M. Waseem, P. Liang, M. Shahin, A. D. Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *J. Syst. Softw.*, vol. 182, Dec. 2021, Art. no. 111061.

[50] S. Gadge and V. Kotwani. (Aug. 2018). *Microservice Architecture: Api Gateway Considerations*. GlobalLogic Organisations. [Online]. Available: https://www.globallogic.com/wp-content/uploads/2017/08/Microservice-Architecture-API-Gateway-Considerations.pdf

[51] I. Odun-Ayo, M. Ananya, F. Agono, and R. Goddy-Worlu, "Cloud computing architecture: A critical analysis," in *Proc. 18th Int. Conf. Comput. Sci. Appl. (ICCSA)*, Jul. 2018, pp. 1–7.

[52] CURP-DOF. (2024). *Diario Oficial De La Federación*. Accessed: Jan. 24, 2024. [Online]. Available: https://dof.gob.mx/index.php?year=1996&month=10&day=23#gsc.tab=0

[53] S. Authors. (2024). *Solidity*. Accessed: Jan. 31, 2024. [Online]. Available: https://docs.soliditylang.org/en/v0.8.24/

[54] A. Benedetti, T. Henry, and S. Tucci-Piergiovanni, "A comparative gas cost analysis of proxy and diamond patterns in EVM blockchains for trusted smart contract engineering," in *Proc. Int. Workshops Financial Cryptography Data Security (FC)*. Cham, Switzerland: Springer, Nov. 2024, pp. 207–221.

[55] I. M. Ali and M. M. Abdallah, "On off-chaining smart contract runtime protection: A queuing model approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 8, pp. 1345–1359, Aug. 2024.

[56] T. Kim, Y. Jang, C. Lee, H. Koo, and H. Kim, "Smartmark: Software watermarking scheme for smart contracts," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 283–294.

[57] M. Ayub, T. Saleem, M. Janjua, and T. Ahmad, "Storage state analysis and extraction of Ethereum blockchain smart contracts," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 1–32, Jul. 2023.

JUAN-CARLOS LÓPEZ-PIMENTEL received the master's and Ph.D. degrees in computer science from Tecnológico de Monterrey, Campus Estado de México, and the degree (Hons.) in engineering in computer systems from the Institute of Technology, Tuxtla Gutiérrez, in 2001.

Currently, he is a Professor-Researcher with the Universidad Panamericana, Campus Guadalajara. He has extensive experience in teaching, research, and coordination of research projects. He has been a Teacher at several universities in levels of bachelor's, master's, and Ph.D. in Mexico. More than 20 articles were published. He has participated in more than ten research projects. He is part of Mexican National System of Researchers Level 1. His research includes include blockchain, computer security, and distributed systems.

CAROLINA DEL-VALLE-SOTO (Senior Member, IEEE) was born in Medellín, Colombia. She received the bachelor's degree in electronics engineering with a thesis on "Design and construction of a photon counting system," the master's degree (Hons.) in science in electronics engineering (telecommunications) with a thesis named "Development of a P2P network with DNS security," and the Ph.D. degree (Hons.) in information technology and communications, with a doctoral dissertation titled "Design, implementation and comparison of a new routing protocol for Wireless Sensor Networks."

Currently, she is the Head of the Engineering Computer Academy, Universidad Panamericana, Guadalajara, Mexico. In addition, she directs and coordinates the master's in cybersecurity with the Graduate Engineering Department, Universidad Panamericana. She is a Titular D Researcher Professor and she belongs to the National System of Researchers, Level I. She is the author of more than 90 articles indexed in the Scopus citation report.

LEONARDO J. VALDIVIA received the M.S. degree in telecommunications engineering and the Ph.D. degree in embedded systems. After spending three years working on software for the automotive sector, in 2013, he started working for the railway sector, specifically in safety and security integration. Currently, he is a Professor-Researcher with the Universidad Panamericana, Campus Guadalajara. He is part of Mexican National System of Researchers Level 1. His research interests include embedded systems and blockchain applications.

RAÚL MONROY received the Ph.D. degree in artificial intelligence from Edinburgh University, in 1998, under the supervision of Prof. Alan Bundy. He has been in computing with Tecnológico de Monterrey, Campus Estado de México, since 1985. There, he is currently a (Full) Professor and he has founded a research group in advanced artificial intelligence. His research interests include the design and development of novel machine learning models, which he often applies in the domains of cyber security and public security. He is a member of the CONACYT-SNI Mexican Research System, ranked 3 (top). He is a fellow of Mexican Academy of Sciences and Mexican Academy of Computing.