Analysis of a Non-Fungible Token centric Blockchain Architecture and Comparison with a General Purpose Blockchain

1st Ricardo Lopes Almeida
Università di Camerino
Università di Pisa
Camerino and Pisa, Italia
ricardo.almeida@unicam.it

2nd Fabrizio Baiardi
Dipartimento di Informatica
Università di Pisa
Pisa, Italia
fabrizio.baiardi@unipi.it

3rd Damiano Di Francesco Maesa
Dipartimento di Informatica
Università di Pisa
Pisa, Italia
damiano.difrancesco@unipi.it

4th Laura Ricci
Dipartimento di Informatica
Università di Pisa
Pisa, Italia
laura.ricci@unipi.it

***Abstract*—*Non-Fungible Tokens (NFTs)* are among the most recent and promising additions to the blockchain universe. The usefulness of this concept triggered several public blockchain to offer support in this regard, starting with Ethereum, one of the popular and flexible blockchains. This support manifests itself through the ability of defining these tokens with smart contract programming. Ethereum defined itself as the reference for NFT development, establishing in the process token standards that are widely used today. This support also triggered the exploration of this concept through the development and deployment of NFT-based projects in the network, which created a rich application ecosystem, but it also revealed limitation in scalability and lack of sufficient throughput in the network. One attempt to solve these issues resulted in the creation of Flow, a new blockchain launched in 2020 that was built from scratch with NFT support in mind. This paper focuses in the alternative NFT architecture Flow introduces, namely, how it is structured, the main differences with a general solution such as Ethereum, and how a simple NFT contract implemented in this new paradigm compares with a similar implementation in Ethereum.**

***Index Terms*—Blockchain, Non-Fungible Tokens, Ethereum, Flow, Solidity, Cadence**

I. Introduction

The inspiration for Flow resulted from an experience from its creators, *Dapper Labs*, with the *CryptoKitties*, one of Ethereum's first NFT projects. This project extended the NFT concept with a new usability layer absent from similar projects. The contract minted a *CryptoKitty*, a NFT representing a digital cat-like creature, and each kitty token was characterised by a unique genome parameter, an internal 256-byte string from which the metadata of the token were derived from. The token metadata was then fed into an image generator that showed the kitty's characteristics. Parameters such as eye color, skin color, ear type, etc were encoded in portions of the genome string. The innovative aspect of this project was that two CryptoKitties could be "bred" to generate a new one with a genome string that derived from the parent's genome. Dapper Labs established the genome mechanics such that new traits were acquired somewhat randomly and some traits were rarer than others. This new approach translated into a peak of popularity and a surge in transactions submitted in Ethereum, and that end up exposing the scalability and throughput limitations of Ethereum [1].

Dapper Labs initially tried to solve these issues from within the Ethereum blockchain, but at some point it became clear that the blockchain needed architectural modifications to be able to overcome these throughput limitations. As such, instead of trying to "fix" Ethereum, Dapper Labs launched Flow in 2020 instead [2], a new blockchain solution developed from scratch centered around supporting NFTs and related mechanics. Flow presents several key differences from Ethereum, from how nodes behave in the network, the consensus algorithm used, to how data is stored and accessed in the chain, as well as similarities, such as defining and using a native cryptocurrency token to regulate blockchain operations (*gas* in Ethereum) and smart contract support. Yet, Flow claims to present the same level of NFT functionalities as Ethereum and other similar NFT ready blockchains, but approaching the concept from a fundamental different point.

This article introduces the Flow blockchain and its architecture, followed by the introduction and analysis of a pair of simple implementations of a NFT minter smart contract, one in Cadence, Flow's smart contract programming language, and another in Solidity, Ethereum's equivalent. Then we compare both architectures and implementations towards determining the merits of Flow's claims as a viable and optimised alternative from NFT-based projects.

The rest of this article is structured as follows: Section II overviews the publications relevant to this work. Section III provides an introduction to Flow, the blockchain central to this exercise. Section V compares implementations of NFT smart contracts in both Cadence and Solidity, as well as the supporting blockchain architectures, namely Flow and Ethereum. This article concludes with Section VI.

II. Related Works

Non-Fungible Tokens were introduced about a decade ago with the Quantum project and got a boost when Ethereum and smart contracts appeared [3]. Even considering such small temporal window, the research community did produce a significant number of relevant publication that used NFTs in some capacity. The authors in [4], [5], and [6] explore a tokenisation approach to manage real estate, where houses, apartments, land plots, etc. are abstracted by NFTs. Real estate properties are unique and individual, similar to NFTs, and the blockchain mechanics that regulate these tokens are very similar to how real estate markets operate. [7] follows a similar approach where they use NFTs to abstract pharmaceutical products and use a *digital twin* approach to ensure the traceability by mirroring the lifecycle of a NFTs within a blockchain with the real product. This tokenisation trend continues with [8], where a similar strategy is used to propose a energy management system for microgeneration. The authors developed a blockchain-based environment where NFTs abstract actors in the system, i.e., solar panels, battery packs, wind turbines, consumers, utility companies, etc. and the values exchanged in the system, i.e., electric energy and money, are abstracted with fungible tokens instead (cryptocurrencies). [9] provides another example where an event management system uses NFTs to abstract event tickets, taking advantage of the same uniqueness and individual elements, such as allocated seat, event name, id number, etc, to individualise the token by encoding them into the metadata.

Others opted for a higher lever approach and presented an analysis based on the architectural aspects of NFTs rather than use them as a simple building block in a solution. [10] explores the architectural aspects of specific NFT implementations in the Hyperledger environment, a framework to develop private custom blockchains. [11] and [12] present a similar high level architectural approach but with a specific scope in mind, namely, tracing and value transfer applications. [13] and [14] presented *systematisation of knowledge (SoK)* articles about this technology, but they did not approached any architectural aspects. All publication mentioned thus far used Ethereum and Hyperledger for the examples provided, and none even mentioned Flow as an alternative. In that regard, to the time of this writing, we found no academic publications using Flow as a development platform, let alone exploring if the new architectural approach could benefit their solutions. The few mentions to Flow in academic examples came from review style papers, namely [15], [16], and [17], the latter providing the most extensive explanation of all.

A. Our Contribution

This paper presents a detailed exploration of the Flow blockchain as an architectural alternative to implement NFTs. To illustrate the differences, we also use Ethereum as an example of a general-purpose blockchain for comparison. We also present a concrete example of a simple NFT contract using Cadence, the smart contract programming language used in Flow, and compare it to a Solidity version of a functionally equivalent NFT contract. This exercise finalises with the analysis of the results obtained.

III. Flow Blockchain

This section goes into detail about the functional aspects of the Flow blockchain and assumes a general knowledge of blockchain technology from the reader and therefore, basic details about blockchain workings are omitted.

A. Consensus Protocol

Early blockchains implemented *Proof-of-Work (PoW)* as consensus protocol, where nodes solve cryptographic puzzles towards getting the privilege of publishing the next block and getting any rewards included. Before Flow, this protocol fell out of favor due to high levels of energy waste that PoW requires. Computations executed to solve these puzzles have no use whatsoever and the popularity of PoW blockchains such as Bitcoin exacerbated this issue.

The blockchain community reacted by proposing alternative consensus mechanisms, such as reputation based protocols like *Proof-of-Authority (PoA)*, and proportional protocols such as *Proof-of-Elapsed Time (PoET)* or *Proof-of-Stake (PoS)* were proposed as alternatives to PoW [18].

PoS became a popular alternative in 2020, around the time when Flow was being developed, which also coincided with Ethereum announcing a future fork (The Paris fork scheduled to 2022) to switch its consensus protocol to PoS. As such, Flow was created with a PoS consensus mechanism, joining a growing number of public blockchains using the same type of consensus.

B. Flow Node Roles

The scalability and throughput claims from Flow derive from an innovative four-node type architecture used to pipeline transaction processing in the network. Flow establishes four node roles, as opposed to the 2-node architecture used by Ethereum, which sacrifices some redundancy and adds a small increase in complexity in return for gains in speed, throughput and scalability, while maintaining minimal operational costs.

The Flow network is based in *consensus nodes* that decide the presence and order of transactions in the blockchain; *collection nodes* used to enhance network connectivity and data availability for applications; *execution* *nodes* to perform the computations required in transactions; and *verification nodes* to validate outputs returned from the execution nodes [19]. Delegated computations are validated with *Specialised Proofs of Confidential Knowledge (SPoCK)*, a type of non-interactive zero-knowledge proofs based on the *Boneh-Lynn-Shacham (BLS)* signature scheme developed by Flow creators for this specific purpose [20].

C. Cadence Language

Cadence is the programming language to write smart contracts in Flow, as well as scripts and transactions required to interact with deployed contracts in the network. Ethereum uses Solidity exclusively to write smart contracts. To interact with a deployed contract in Ethereum, users have to rely in Ethereum's API and interact with it using scripts written in general purpose languages such as Javascript and Python. Flow simplifies this by using Cadence for both smart contract development and to interact with deployed instances in the network. Cadence implements a *Resource Oriented Programming Paradigm* through a strongly static type system, with built in access control mechanics that can be further specialised through a *capability-based* security scheme that can narrow the scope of users allowed to interact with a given resource. Cadence syntax was inspired by modern general-purpose programming languages such as *Swift*, *Kotlin*, and *Rust* [21] [22].

1) Smart Contracts: Smart contracts in Flow serve the same purpose as in Ethereum and other blockchains with similar support. Flow contracts are syntactically different from Solidity, but functionally they are very similar: both use the **contract** keyword to define the main contract structure, use constructor functions that executed automatically once during deployment and can extend their functionalities by importing external contracts and interfaces. But, unlike Solidity, Cadence does not require the implementation of default destructors because storage management is automatically managed. Smart contracts deployed in Flow remain in an "updatable" phase initially. During it, the code can be updated by the creator(s). Once the developer(s) are satisfied with the contract's performance, they can *lock* it to prevent future changes. This allows for cleaner deploys and optimise blockchain storage, unlike Ethereum that saves any new versions of an existing contract in a new block, which can clutter the blockchain with useless code.

2) Interacting with Flow Smart Contracts: In Flow, if an interaction is limited to read operations, i.e., the script instructions **do not change** the state of the blockchain, then a *script* should be used. If the instructions do **change the state** of the blockchain, by saving, modifying or deleting data, a *transaction* needs be used instead. *Transactions* require a valid digital signature and funds allocated to pay for gas costs. Flow restrict modifications to digital objects in storage to only the owner(s) of such object, therefore transactions that do these changes require a digital signature from the owner. *Scripts* on the other hand, are "free" to execute because they do not consume gas, therefore they do not require signatures and are used to read public parameters in a contract.

3) Resource Oriented Paradigm: Cadence establishes its programming paradigm through a special digital object named *Resource* which was inspired by Rust's *linear types*. From a functional point of view, Cadence resources are similar to objects in Object-Oriented programming languages. The difference resides in the control exercised by the language to ensure that Resources are unique in a blockchain environment. Once created, a resource cannot be copied; only moved or destroyed. Since resources cannot be copied, they also can only exist in one location at a time, often saved in an account's storage area. Only functions defined in smart contracts can create resources. Cadence limits the scope of the **create** keyword used to create an resource to a smart contract. Flow smart contracts are deployed with resource creation functions which then are invoked using transactions, since creating a resource changes the state of the blockchain. So far, resources are exclusive to Flow, which is a limiting factor to establish interoperability with other blockchains.

4) Cadence Types: Cadence was developed as a type-safe language and this is part of the strategy to individualise and uniquely identify each digital object in the blockchain. Cadence uses basic types, similar to Solidity, for basic data elements, such as integers, strings, floats, etc. The nomenclature is slightly different but consistent with Solidity. An unsigned 256 bit integer in Solidity is preceded by **uint256**, but in Cadence the equivalent type is **UInt256**. Complex types, e.g, resources, events, structs,etc., have complex types resulting from the concatenation between the name of the implementing contract and the name of the element. Since elements such as resources, structs and events can only be created through a contract that contains their implementation details, Flow uses this to its advantage. Pairing the contract name with the resource name creates a unique identifier by default. This ensures type uniqueness withing a Flow projects since contracts saved in the same account storage require different names. To uniquely identify a specific resource in the blockchain, this nomenclature is extended to include the address of the deployed contract prefixed to this identifier. This strategy can uniquely identify every complex type in the blockchain. For example event emitted by transactions are displayed as `A.0ae53cb6e3f42a79.FlowToken.TokensWithdrawn`.

This is a complex type concatenating the address of the deployed contract (0x0ae53cb6e3f42a79), the name of the contract (FlowToken) and the name of the event (TokensWithdrawn).

5) Capabilities and References: Cadence supports capability-based security through the object-capability model to provide a fine-tune access control mechanism.

Similar to traditional capabilities, these are a value that represents the authorisation to access and operate on a digital object.

Capabilities in Cadence divide into *storage* and *account* capabilities depending on the object targeted by it. Capability issuing is limited to the owner of the resource or account targeted and are stored in a special storage `\public` domain area open for public access but restricted to capability storage only. Sec. III-E1 details these concepts thoroughly. After publishing a capability to the `public` domain, another user can "borrow" it. *Borrowing* is an official action in Cadence that retrieves a reference to a stored object using a previously published capability. References in Cadence are akin to memory pointers in general purpose programming languages. Retrieving a reference to a resource returns a pointer of sorts whose functionalities are defined in the borrowed capability. References are used to read data from stored objects without accessing the object directly, thus protecting it against deletion and unauthorised transfers.

6) Access Control: Cadence implements access control in two levels of granularity. At a coarser level, contracts and objects limit the access to their inner constituents, e.g., functions, parameters, structs, etc., using the **access** keyword, with the scope of access inside parenthesis. **access(all)** grants public access. **access(E)**, where E is an entitlement restricts access to only users with such entitlement. **access(account)** limits the scope of access to other functions and objects saved in the same account, while **access(contract)** narrows the scope to the same type of elements, but limited to the same contract. The most restricted level is **access(self)**, which restricts access to functions and parameters defined within the element affected.

a) Entitlements: Entitlements complement the second level of access by providing granular access control to individual members (fields and functions) of a composite object, namely a resource or a struct. Entitlements can be used to create references to stored resources and structs with different "versions", i.e., the digital object referenced is the same but the sets of fields and functions available are different for each entitlement. Cadence uses *auth(entitlement)* to specify the entitlement to use during the resource retrieval process and '&' to identify a reference to a resource, while'@' is used to denote the resource itself. For example, consider a *Resource R* with two fields: *access(A) name* and *access(B) age*, where A and B are custom entitlements defined in the contract where *Resource R* is defined. With a resource R in storage, retrieving auth(A) &R returns a reference to R where only *name* is accessible. Conversely, retrieving auth(B) &R returns a reference to the same resource, but with only *age* available. Entitlements can be combined using 'and' or 'or' logic: *access(A, B) name* sets *name* to require entitlements A **and** B, while *access(A | B) name* changes the requirement to either entitlement A **or** B.

D. Token Standards

Flow standardise its fungible and non-fungible projects by defining contract standards, similarly to Ethereum. Flow developers followed Ethereum's approach in publishing contract standards as interfaces, but they also took the opportunity to name them more suggestively. As such, Flow regulates fungible tokens applications with the *FungibleToken* standard [23], Ethereum's *ERC-20* equivalent. Non fungible tokens have a corresponding *NonFungibleToken* standard [24] that mirrors Ethereum's *ERC-721* standard. These standards have the same function as in Ethereum, i.e., they guarantee the implementation of parameters and functions defined in the standard, which Flow uses to ensure interoperability within the application ecosystem.

E. Account-based Storage Model

Flow implements an *Account-centric Storage Model*, different from the *Contract-centric Storage Model* used in Ethereum. Storage locations are indexed from an account address and the amount of storage space per account is proportional to the amount of FLOW token in balance, currently at a rate of 1 FLOW token per 100 MB of storage. An account in Flow is, essentially, a digital object that needs to be saved in the blockchain itself. This requirement establishes that accounts in Flow require a minimum of 0,001 FLOW to be operational, the amount required to sustain the basic structure of an account in the blockchain. This is also the fee to create a new account in Flow, though a newly created account comes with 0.001 FLOW already in its balance. At the time of this writing, FLOW price oscillates between 0.5 and 1$ per token.

1) Storage Domains and Paths: An account in Flow is a digital object identified by an address, contains a *balance* and a set of *public encryption keys* used to validate transactions signed by the account owner. The storage area associated to the account is split into contract storage, where smart contracts are stored, and a general purpose storage area to save other digital objects, such as NFTs and other resources.

The general storage space is divided into three domains: a `\storage` domain only accessible by the account owner and where all data is actually written into; a `\public` domain to store the *capabilities* in Sec. III-C5, therefore limited to read-only access; and a `\private` domain used for capability storage also, but this one restricted to the owner.

Object are stored using UNIX-style paths, with the storage domain as the first element. For example, `\storage\ExampleNFT` indicates an "ExampleNFT" resource stored in the main storage domain, while `\public\ExampleNFTCap` indicates a capability published as "ExampleNFTCap" in the public domain that can be used to retrieve a reference to the ExampleNFT in `\storage`.

2) Resource Collections: The storage system described in Sec. III-E1 is not flexible if a large number of resources are to be stored. Like UNIX paths, storage paths in Flow need to be unique in each account, which can complicate the storage process as more objects are sent to storage. Flow was developed around the concept of digital collectibles, with the expectation that accounts would hold large numbers of NFTs or other resources. To solve this limitation, Flow uses a special resource called *Collection*. Essentially, a collection is a resource that can save and hold other resources in storage, are standardised in Flow through the *NonFungibleToken* standard and each collection is limited to store resources of one type. Collections needs to be created and saved in a unique storage path, but after that, other resources go in and out of a collection using *deposit* and *withdraw* functions, using token identification numbers to identify the token in question instead of having to create a new and unique storage path for each new resource. In UNIX analogy, collections behave akin to directories.

IV. Development of a NFT Smart Contract in Cadence

The central exercise of this paper focused in the development of a Cadence and a Solidity simple NFT smart contract following each blockchain's standard for non-fungible tokens, while keeping the details and functionalities as close as possible in both contracts. We assume at this point that the reader has minimal knowledge on how NFTs are implemented in Solidity and Ethereum, therefore we omit the details of this implementation while shifting the focus to the Cadence version. Fig. 1 presents a schematic representation of the contract developed displaying the required standard dependencies and the functions and fields required by this inheritance.

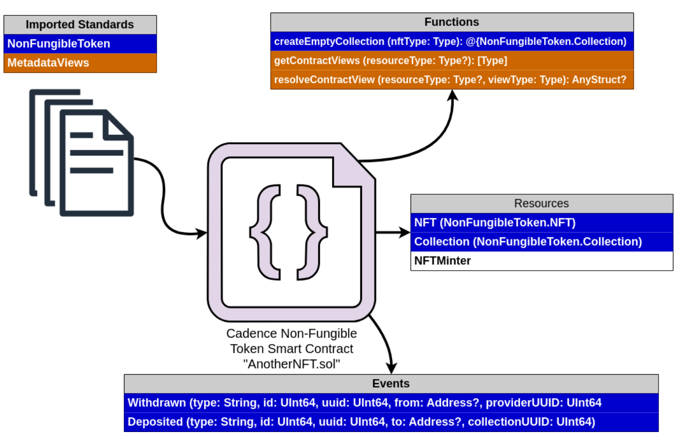

Figure 1. Structural organisation of a NFT smart contract in Cadence

A. Standards Imported

The simplest standardised version of a Cadence NFT smart contracts requires the import of two standards, namely, *NonFungibleToken* and *MetadataViews* interfaces. We omitted references to the *MetadataViews* because this standard does not has a corresponding standard in Ethereum. Flow was developed to support digital collectible projects, which emulate real world collectible systems, like trading cards for example. This emphasises the metadata operations associated to the NFTs produced, hence Cadence produced a standard to deal specifically with NFT metadata operations. There's a dependence between the *NonFungibleToken* standard and *MetadataViews*, which means that a basic NFT contract has to import both, but the *MetadataViews* requirements are minimal and are not consequential to this comparison.

We used a color scheme in Fig. 1 to indicate which standard required the implementation of the function, resources and events indicated in the contract. The *MetadataViews* requirement imposes the implementation of a pair of functions to set and retrieve metadata related items in the contract. These functions are "empty" as they simply return an empty element of the value defined to minimise their influence in the contract.

1) Resources: Only the *NonFungibleToken* standard imposes the implementation of a NFT and a collection resources, the latter to ease the storage of multiple resources, as it was mentioned in Sec. III-E2. A NFT and a Collection definition are the minimal requirements from Flow in this regard. But Fig. 1 shows a third resource in the contract, namely a *NFTMinter*. This resource is used to abstract, simplify and secure the minting of new NFTs from this contract. A NFT minter is not required by any of the standards, but the contract becomes unusable without one, so it is a sort of implicit requirement. This happens because new resources can only be created with a contract function due to Cadence restricting the use of the "**create**" keyword to the context of the contract and within a function. Transactions can not invoke this internal action. A secure approach to create new NFT resources is to define the function that creates new ones inside of a NFT minter resource and having this minter resource created and saved into the contract deployer storage in the contract constructor. This restricts the creation of new NFTs to the owner of the storage space where the minter was saved, i.e., the contract deployer, while making it impossible to create new NFTs from the contract itself, thus decoupling this action from the contract itself.

2) Functions: The *NonFungibleToken* standard requires the implementation of a *createEmptyCollection* function to obtain a collection resource. Unlike NFTs, this contract provides multiple access points to create new collections. The *NonFungibleToken* standard also imposes a similar function in the NFT resource definition and the collection resource itself. This may sound redundant, but in standardised NFT contracts in Flow it allows collections to create other collections. The *createEmptyCollection* function in Fig. 1 requires a type as input because it is not possible to infer the type of any resource to store at the contract level, therefore this needs to be provided. The versions of this function implemented in the NFT and collection resources do not need it because they can infer the type to store from the resource itself.

3) Events: A contract that imports the *NonFungibleToken* inherits the *Withdrawn* and *Deposited* events by default. In Cadence, like in Solidity, events are made available and emitted automatically by the logic imported from the standards, i.e., there is no need to define them explicitly in the contracts, unlike functions and resources. The events indicated are emitted automatically whenever a NFT resource is moved in or out of an account storage.

V. Results and Comparison

A. Blockchain Comparison

The Flow blockchain is significantly different from general purpose blockchains available publicly. To reinforce this conclusion, we took Ethereum, the most popular blockchain used in the academic context, as the reference point to produce Table I where the two solutions were compared side by side regarding the most relevant characteristics towards NFT implementations.

Table I
Ethereum-Flow Technology comparison

Parameter	**Non-Fungible Token Architecture**	
	Ethereum	**Flow**
Year	2013	2020
Cryptocurrency	ETH	FLOW
Programming Language	Solidity	Cadence
Consensus Algorithm	2013-2022: PoW, 2022-: PoS	Proof-of-Stake (PoS)
Network Nodes Roles Types	1 - Execution Client	1 - Collector Node
		2 - Consensus Node
	2 - Consensus Client	3 - Execution Node
		4 - Verification Node
Token Standards	ERC-20 (fungible token)	FungibleToken
	ERC-721 (non-fungible token)	NonFungibleToken
Data Storage Architecture	Contract-based	Account(User)-based
Block Rate (Average)	12 - 15 seconds per block	0.5 - 1 seconds per block
Daily transaction volume (2024)	1 - 1.25 million transactions\day	0.5 - 1 million transactions\day
Cost per transaction (average)	5.5 gwei (∼0.39 $)	∼0.00000845 $

The two technologies are similar in their general blockchain approach: both implement a cryptocurrency token and use it as a regulatory mechanism to optimise the functioning of their respective virtual machines. Both implement the concept of gas, a fraction of the cryptocurrency implemented, that needs to be paid beforehand to execute transactions that change the state of the blockchain. Ethereum coined the term *gas* to refer to cryptocurrency used to pay virtual machine computational costs. Flow implements the same exact logic, though they do not use the *gas* term as much. Both blockchains use PoS as the consensus protocol, tough this is a recent upgrade for Ethereum. This chain spent the first 9 years using PoW, like Bitcoin, but it went through the Paris fork in September 2022, which implemented the *Ethereum Improvement Proposal 3675 (EIP-3675)* [25] that switched the consensus protocol to PoS from there on.

The similarities stop here. From here, Flow and Ethereum provide very different programming languages for smart contract development. Both languages produce code that can be executed in the respective distributed virtual machines but using different syntaxes, albeit producing elements with similar behaviours and functionalities. Flow claims that its higher throughput, low operational costs and higher scalability are due to their four node pipelining architecture. Essentially Flow splits the two-node architecture from Ethereum into half, doubles the roles in the blockchain but this translates into a more efficient, but also more complex, blockchain. Both blockchains provide official standards to regulate token based development and mechanics, but the obvious incompatibility between the technologies produced a pair of token standards for each chain, among others.

Flow is the cheapest and fastest of both. Flow's block rate is between 12 and 30 times higher than Ethereums and Ethereum transactions are, exactly 46154 times more expensive than Flow's. Yet, Ethereum's popularity and maturity still trump over Flow's operational advantages. At the time of this writing, Ethereum daily volume of transactions can be as 2.5 times higher than in Flow while being objectively worse to operate. This is not an endorsement of Ethereum, or Flow for that matter, but a factual statement: popularity and application ecosystem maturity influence technology adoption more that efficiency and cost, at least in a short term basis. It is important to notice that Ethereum has been available for twice as much time as Flow, and Flow made its debut in a time where many other blockchains were being created as well. Flow has yet to distinguish itself from other, more established NFT supporting blockchains. But from the performance point of view, the superiority of Flow is clear, especially when compared with the "older" technology of Ethereum.

B. Contract Implementation Details

We developed two smart contracts implementing the NFT standard in both Ethereum and Flow, i.e., implementing the *ERC-721* and *NonFungibleToken* standards respectively. Both contracts were used after to execute the following sequence of macro actions:

1) Deploy the NFT contract in a network
2) Mint a NFT into a user account
3) Transfer the NFT from one user account to another
4) Burn the NFT

This sequence of actions were relatively straightforward to execute in the Solidity case. In Flow, there was some overhead to deal with first. Other than deploying a large contract, account creation is the most expensive atomic operation [26], both in gas and storage costs. Ethereum does not even has such function because of how fundamentally different Flow and Ethereum accounts are, but give the effort required to create one, we included this configuration step in the Flow analysis. Also, as it was indicated in Sec. III-E2, Flow uses collection resources to simplify NFT storage. Though it is possible to save a NFT directly into a storage path, we opted to include this step as well to illustrate the impact that these overhead actions common in Flow have in the complete process.

C. Cost Comparison

We begun the comparison exercise by analysing the operational costs associated to each operation indicated, namely, how much gas was consumed per each step considered. Obtaining the gas spent for a specific set of steps is trivial. Ethereum development frameworks often include gas calculation tools that essentially keep track of the balance of the accounts involved in the process and computes the differences. We used Hardhat as the Ethereum development framework and the built-in *gas reporter* tool to determine the gas expenditure indicated in Table II:

Table II
GAS CONSUMPTION REPORT FROM HARDHAT'S GAS REPORTER TOOL.

Solc version: 0.8.24		Optimizer enabled: false		Runs: 200	Block limit: 6718946 gas	
Methods						
Contract	Method	Min	Max	Avg	# calls	eur (avg)
ExampleNFT	burn(uint256)	-	-	29592	1	0.8091
ExampleNFT	safeMint(address, uint256, string)	-	-	121859	2	3.3312
ExampleNFT	safeTransferFrom(address, address, uint256)	-	-	58401	2	1.5973
Deployments		-	-	2420549	36 %	66.205
ExampleNFT		Totals		2630401		71.943

We configured Hardhat's *gas reporter* to connect to a cryptocurrency pricing oracle, an external application to the blockchain that can be queried for off-chain data, to obtain the gas costs in EUR in real time. ETH price currently is very volatile, but the values obtained are enough to illustrate the difference, especially compared to a much cheaper option of Flow. This notion was already presented in the last row in Table I, where the average price difference is quite apparent. Flow's development framework does not offer a comparable tool to Hardhat's *gas reporter*. As such we calculated these fees directly by subtracting account balances between steps. Additionally, Flow imposes a base dependency to all standardised contracts to a special contract named *FlowFees*. This contract is implemented in every Flow standard, including the *NonFungibleToken* standard used, to automates a series of fee based computations and to emit a *FeesDeducted* event every time any fees are payed while executing instructions from the contract that implements the standard. The event is emitted with the amount paid as argument and we also captured these events to validate the balance differences calculated. We used these strategies to determine the gas costs for the previous exercise with the Cadence contract. The calculations are indicated in Table III:

Table III
FLOW TOKEN BALANCE OF EACH ACCOUNT IN THE EXERCISE

FLOW token balance of accounts						
Tx	**Emulator-account**		**account01**		**account02**	
	Balance	Difference	Balance	Difference	Balance	Difference
00	9.99600		0		0	
01	9.99396	-0.00204	0.001	0.00100	0.001	0.00100
02	7.99392	-2.00004	1.001	1.00000	1.001	1.00000
03	7.99390	-0.00002	1.001	0.00000	1.001	0.00000
04	7.99388	-0.00002	1.00099	-0.00001	1.00099	-0.00001
05	7.99386	-0.00002	1.00099	0.00000	1.00099	0.00000
06	7.99385	-0.00001	1.00098	-0.00001	1.00099	0.00000
07	7.99384	-0.00001	1.00098	0.00000	1.00098	-0.00001

Transactions
00 – New service-account created
01 – Emulator test accounts created (2)
02 – Emulator test accounts funded with 1.0 FLOW
03 – Deploy ExampleNFTContract into emulator
04 – Create a NonFungibleToken.Collection in each account
05 – Mint an ExampleNFTContract.NFT into account01 Collection
06 – Transfer ExampleNFT from account01 to account02
07 – Burn the ExampleNFTs from account02

1) Analysis: We included the overhead operations from Flow in this analysis and even with it, Flow is substantially cheaper to operate than Ethereum. The first conclusion from Table III is that most transactions cost the same value of 0.00001 FLOW (less than 0.00001€ at the time of this writing). This is the minimum fee in Flow [26], which may mean that the operation may have been even cheaper. Flow's fee system is based in three fee factors: an *inclusion fee (IncFee)* to pay for the process of including the transaction into a block, transporting information, and validating signatures in the network; an *execution fee (ExecFee)* to pay for FVM computations and operating on data storage, and a *Surge fee (SrFee)* applied dynamically to modulate network usage and avoid surges. The total transaction cost can be calculated with: $TxCost = (ExecFee + IncFee) \times SrFee$.

Currently, the inclusion fee is fixed to 0.000001 FLOW, thus less than the minimum fee, and the surge factor is planned but not yet implemented. This means that, currently, transaction fees in Flow are mostly influenced by the computation effort required. But Table III shows that these do not exceed the minimum fee per transaction established in most cases. The most costly operations are the contract deployment and the new accounts creation, the latter substantially larger. The deployment transaction cost is proportional to the size of the contract in question. The contract used in this exercise occupies 6730 bytes of storage. Flow's documentation does not indicates the exact cost of storage per unit of memory consumed, but it estimates the cost of a deployment of a 50 KByte contract to be 0.00002965 FLOW. Our contract is substantially small, therefore the fee requested is coherent with the logic so far. The documentation also confirms that account creation is indeed the most expensive of all of Flow's base operations, but even in this case, a large part of that cost derives from the requirement for a minimum balance of 0.001 FLOW for accounts. This balance is included in the total cost of the transaction, though it is actually a transfer of funds between the account paying for the account creation process, and the new account. Without that value, the account creation process is actually twice the minimum transaction fee, i.e., 0.00002 FLOW (the cost indicated in Table III refers to the creation of two accounts).

The comparison between the experiments from a cost perspective is quite extreme. The Solidity exercise required less transactions and yet it totalled almost 72 €. Flow required a series of extra operations to bring the system up to par, namely, to create a pair of extra accounts and respective collection resources, but even so, the total amount required only 2.00216 FLOW, which amounts to 1.72 €, a reduction by a factor of 42 compared to Ethereum. The Flow cost is actually a very conservative estimation of sorts. It includes 2.0 FLOW that were transferred to the two extra accounts to increase their balance above the minimum of 0.001 FLOW required to sustain the minimum of storage space and pay for transactions, which could be a much lower value and still allow the exercise to complete. On a cost basis, Flow is clearly the preferable option.

D. Storage Comparison

Storage is a critical aspect to consider in blockchain applications. Writing data into a blockchain block is an expensive operation because that data is going to be replicated in network nodes. This consideration has pushed blockchain to develop efficient data manipulation methods to optimise blockchain operation. This aspect is visible in smart contract development, whose code tends to be small and very optimised, when compared to other, non-distributed, applications. Another blockchain aspect that derives from this restriction is the complexity of the process of writing data into a block. Typically, this requires a digitally signed transaction and a gas cost paid upfront.

Considering the importance of storing data in the blockchain, particularly to NFTs that use this data (metadata) to establish their uniqueness, we run a similar analysis to the one depicted in Sec. V-C but towards determining the storage space considerations for the project.

1) Storage in Ethereum: Ethereum stores data in a *contract-based* approach. The means that all data related to a given contract is stored *referenced* to the address where the contract is deployed. Ethereum defines an "astronomically large array" indexed from the deployed contract address as the storage space of the contract. The array has 2^{256} potential slots, which is a number close to the number of atoms in the visible universe, hence the "astronomical" adjective, and each index in the array can store up to 32 Bytes [27]. The "potential" derives from Ethereum only storing non-zero values, i.e., if a contract parameter has value 0, it does not count to the total storage used by the contract. Contract parameters are stored sequentially from index 0 but mappings, due to their dynamical nature, distribute their data throughout the storage space. Mappings use a key-value scheme and the index of a mapped element in storage by the hash of the key concatenated with a positional argument. This means that if a contract contains mappings or dynamic arrays, the data stored under its address is actually spread around the storage space. Which also complicates the storage analysis since Ethereum nor Solidity provides a direct way to determine the storage used, so we had to be creative and devise a method of our own for that effect. Following the logic used by Ethereum to store data, we wrote a script that checks storage slots for data, i.e., non-zero values. Table IV presents our findings.

Table IV
Storage analysis of the Solidity NFT contract

Tx	Storage used by the ExampleNFT Solidity contract 0x5FbDB2315678afecb367f032d93F642f64180aa3			
	Contract Size	**Simple Storage**	**Mapping Storage**	**Total**
00	10458	96	0	10554
01		128	94	10680
02		128	94	10680
03		128	0	10586

Transactions
00 - Deploy ExampleNFT Contract into the emulator
01 - Create an ExampleNFT into account01
02 - Transfer ExampleNFT from account01 to account02
03 - Burn the ExampleNFT from account02

The lion share of used storage is clearly the deployed contract. The Hardhat framework used in this case actually simplifies this one step by providing a *size-contracts* feature that return the storage occupied by each deployed contract. The remaining values were found by running a storage scanning script to determine how many slots are in use and the amount of data stored in each. The *Simple Storage* column refers to contract parameters such as *totalSupply*, *nextTokenId*, etc, while the *Mapping Storage* refers to values stored in Solidity mappings specifically, which are used to establish the ownership logic in NFT contracts. Solidity establishes the existence of a NFT as a set of related mapping entries. All mappings in the ExampleNFT contract considered are only used for this purpose, which is confirmed in Table IV. The NFT only "exists" in transactions 01 and 02 and that is reflected in the storage used in mappings. As soon as the token is destroyed (burned) this data is deleted.

2) Storage in Flow: Compared to Ethereum, Flow simplifies this step greatly. It does not provide a tool per se, but determining the total storage used by an account is a trivial operation in Flow. As such, we developed a simple script to return the storage used by each account involved in the exercise and run it after each step. Table V presents our findings.

The emulator-account displays an unusual large used storage but this is not relevant for our exercise. To speed up and simplify development, Flow provides its emulator

Table V
FLOW token balance of each account in the exercise

Storage used by Flow accounts (Bytes)						
Tx	**Emulator-account**		**account01**		**account02**	
	Storage (Bytes)	Difference	Storage (Bytes)	Difference	Storage (Bytes)	Difference
00	428655		0		0	
01	429290	635	1007	1007	1007	1007
02	429607	317	1007	0	1007	0
03	436337	6730	1007	0	1007	0
04	436613	276	1588	581	1588	581
05	436769	156	1751	163	1588	0
06	436894	125	1588	-163	1751	163
07	437035	141	1588	0	1588	-163

Transactions
00 – New service-account created
01 – Emulator test accounts created (2)
02 – Emulator test accounts funded with 1.0 FLOW
03 – Deploy ExampleNFTContract into emulator
04 – Create a NonFungibleToken.Collection in each account
05 – Mint an ExampleNFTContract.NFT into account01 Collection
06 – Transfer ExampleNFT from account01 to account02
07 – Burn the ExampleNFTs from account02

environment with a series of standards already deployed into the emulator-account storage area, hence the unusual value at startup. We only make use of the *NonFungible-Token* one, so most of that is not being used even.

3) Analysis: Flow presents itself as a more efficient alternative to save data in a blockchain. Ethereum saves data in 32-byte chunks arranged sequentially from the deployed contract address. To optimise storage, when possible, Ethereum saves two values into the same slot, if both are 16 bytes or less, by splitting one 32-byte slot into two of 16, but overall, apart from contracts, Ethereum saves data "discretely", i.e., in 32 or 16-byte chunks, except for strings, which are saved in UTF-8 using 2 bytes per character. Flow on the other hand presents a more granular storage, where data is discriminated to the byte.

Regarding the size of the NFT construct itself, Ethereum is the better option in the sense that it produces a smaller data footprint. It is important to note that the NFT created in Flow was blank, i.e., without any additional metadata other than the bare minimum required by the standard, while the version in Ethereum was created with the recipient account address as metadata, stored as a UTF-8 string, and even so, Ethereum NFT was computed at 94 bytes of storage, while a programmatically smaller NFT from Flow required 163 bytes. Flow has a significantly more storage overhead than Ethereum, as in the data required for minimal functionality, but it still maintains a higher transactional throughput nonetheless. Though Flow does require more storage in absolute values, but price for storage in Flow, both in storage and transaction fees, is much less than in Ethereum, which makes this apparent disadvantage meaningless.

Overall, Flow is still the better option in this case. Storage in Ethereum is a thorny issue in great part because of how expensive ETH has become as well. Also, this chain has a slower block rate than Flow, which means a lower rate of data writes, since these require transactions finalised in blocks. Flow does write more data to achieve similar functionality, but does so in a inexpensive and faster fashion, which balances the preference to its side.

VI. Conclusion

This paper presents an implementation analysis for two distinct architectures to create NFT-based contract capable of minting and transferring these tokens between users. Ethereum and Flow were chosen for this exercise due to former's role as a reference, general-purpose, and very popular blockchain, the latter being a blockchain created with the specific purpose of solving known issues with the Ethereum chain, namely, by supporting high throughput and scalable NFT projects.

Flow is quite younger than Ethereum and, because of this, is still under some development and missing a wider spectrum of applications when compared to a more mature Ethereum. Nevertheless, Flow provides more than enough capabilities to produce a NFT-capable contract. If anything, the structural analysis done in Sec. III reveals Flow as the more complex, but also more configurable blockchain. From a programmatic point of view, Cadence is syntactically more complex than Solidity and contracts consume more time to develop, but the deployment process is simpler and faster than the Ethereum equivalent. The cost aspect favors Flow as well, though the limitations of Ethereum in that aspect fall outside of the technological scope. ETH high price is a much more consequence of speculative action than from an intrinsic value as a requirement to execute transactions in the network. Flow's lack of relative popularity when compared with Ethereum works in its favor by providing a much cheaper and realistic alternative to NFT-based applications since FLOW's price is not yet affected, at least not as much, from the same speculative forces that risk making Ethereum simply too expensive to use for regular users.

Flow presents a strong case for a much cheaper alternative to Ethereum for NFT-based applications. It is also provides a self-contained programming language for smart contracts in Cadence. Flow uses the same language to create smart contracts, scripts and transactions used to interact with the former. From a developer standpoint, it is a simpler approach than Ethereum, which uses Solidity for smart contract development only, but interacting with contracts deployed in the blockchain, which includes all NFT-related mechanics, requires an additional, third party framework, such as Hardhat, Truffle, Remix, etc.

From a storage point of view, Flow is more complex to understand, but easier to use. The UNIX-like approach to storage paths makes it easier to understand due to its similarity with regular operating systems. Ethereum's storage model is simpler but hard to operate with, especially when complex structures such as mappings are stored. The contract-based approach is more transparent. Realistically, as long as one knows the address of the deployed contract, it is possible to read all data stored in the contract, if not encrypted. The process is not simple but it is doable. Flow's account-based storage approach presents a cleaner approach. Owning digital objects in

Flow is a easier concept to understand because of the exclusivity of the storage area associated to an account. Only the owner can access this area, regardless of where the contract that created the digital resource is actually saved.

Though Flow offers clear advantages, Ethereum still has the lion share of the NFT-based applications in the blockchain ecosystem. The 7 years that Ethereum has on Flow are a significant gap to overcome, especially considering that Ethereum was the only realistic alternative to smart contract development for a significant part of that period, which boosted its popularity to levels difficult to replicate. Also, Flow is still in relative development. Some of the concepts indicated in this article were introduced or updated significantly in the recent Crescendo upgrade [28] from September 2024, such as entitlements, capabilities and access control mechanics. Even with all this considered, Flow does present a rich, albeit limited, application ecosystem mostly focused on digital collectibles, which is the type of applications that inspired the creation of this blockchain in the first place.

References

[1] bbc.com. Cryptokitties craze slows down transactions on ethereum. [Online]. Available: https://www.bbc.com/news/technology-42237162

[2] R. Gharegozlou. Introducing flow, a new blockchain from the creators of cryptokitties. [Online]. Available: https://medium.com/dapperlabs/introducing-flow-a-new-blockchain-from-the-creators-of-cryptokitties-d291282732f5

[3] J. Exmundo. (2023, 3) Quantum, the story behind the world's first nft. [Online]. Available: https://nftnow.com/art/quantum-the-first-piece-of-nft-art-ever-created/

[4] N. N. Hung, K. T. Dang, M. N. Triet, K. V. Hong, B. Q. Tran, H. G. Khiem, N. T. Phuc, M. D. Hieu, V. C. Loc, T. L. Quy, N. T. Anh, Q. N. Hien, L. K. ṅd Bang, D. P. Nguyen, N. T. Ngan, X. H. Son, and H. L. Huong, "Revolutionizing real estate: A blockchain, nft, and ipfs multi-platform approach," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 14416 LNCS. Springer Nature Switzerland, 2023, pp. 68–73. [Online]. Available: http://dx.doi.org/10.1007/978-3-031-48316-5_10

[5] D.-E. Bărbuță and A. Alexandrescu, "A secure real estate transaction framework based on blockchain technology and dynamic non-fungible tokens," in *2024 28th International Conference on System Theory, Control and Computing (ICSTCC)*. IEEE, 2024, pp. 558–563.

[6] A. Sharma, A. Sharma, A. Tripathi, and A. Chaudhary, "Real estate registry platform through nft tokenization using blockchain," in *2024 2nd International Conference on Disruptive Technologies, ICDT 2024*. IEEE, 2024, pp. 335–340.

[7] F. Chiacchio, D. D'urso, L. M. Oliveri, A. Spitaleri, C. Spampinato, and D. Giordano, "A non fungible token solution for the track and trace of pharmaceutical supply chain," *Applied Sciences (Switzerland)*, vol. 12, pp. 1–23, 2022.

[8] N. Karandikar, A. Chakravorty, and C. Rong, "Blockchain based transaction system with fungible and non-fungible tokens for a community-based energy infrastructure," *Sensors*, vol. 21, p. 32, 2021.

[9] F. Regner, A. Schweizer, and N. Urbach, "Nfts in practice-non-fungible tokens as core component of a blockchain-based event ticketing application completed research paper," in *Proceedings of the 40th International Conference on Information Systems*, 2019, pp. 1–17.

[10] S. Hong, Y. Noh, and C. Park, "Design of extensible non-fungible token model in hyperledger fabric," in *SERIAL 2019 - Proceedings of the 2019 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2019, pp. 1–2.

[11] L. Yang, X. Dong, Y. Zhang, Q. Qu, and Y. Shen, "Generic-nft : A generic non-fungible token architecture for flexible value transfer in web3," *TechRxiv*, pp. 1–7, 2023.

[12] M. Bal and C. Ner, "Nftracer: A non-fungible token tracking proof-of-concept using hyperledger fabric," *arXiv: 1905.04795v1*, pp. 1–9, 2019. [Online]. Available: http://arxiv.org/abs/1905.04795

[13] G. Wang and M. Nixon, "Sok: Tokenization on blockchain," in *Proceedings for the 2021 IEEE/ACM14th International Conference on Utility andCloud Computing (UCC '21) Companion (UCC '21 Companion)*. ACM, 2021, pp. 1–9.

[14] K. Ma, J. Huang, N. He, Z. Wang, and H. Wang, "Sok: On the security of non-fungible tokens," *arXiv:2312.0800v1*, vol. 1, pp. 1–15, 2023. [Online]. Available: http://arxiv.org/abs/2312.08000

[15] Q. Wang, R. Li, Q. Wang, and S. Chen, "Non-fungible token: Overview, evaluation, opportunities and challenges," *arXiv*, 2021. [Online]. Available: http://arxiv.org/abs/2105.07447

[16] Q. Razi, A. Devrani, H. Abhyankar, G. S. Chalapathi, V. Hassija, and M. Guizani, "Non-fungible tokens (nfts) - survey of current applications, evolution, and future directions," *IEEE Open Journal of the Communications Society*, vol. 5, pp. 2765–2791, 2024.

[17] B. Guidi and A. Michienzi, "From nft 1.0 to nft 2.0: A review of the evolution of non-fungible tokens," *Future Internet*, vol. 15, 6 2023.

[18] S. Bouraga, "A taxonomy of blockchain consensus protocols: A survey and classification framework," *Expert Systems with Applications*, vol. 168, pp. 1–17, 2021. [Online]. Available: https://doi.org/10.1016/j.eswa.2020.114384

[19] A. Hentschel, Y. Hassanzadeh-Nazarabadi, R. Seraj, D. Shirley, and L. Lafrance. Flow: Separating consensus and compute - block formation and execution. [Online]. Available: http://arxiv.org/abs/2002.07403

[20] T. Ben, Y. Riad, and S. Wahby, "Flow: Specialized proof of confidential knowledge (spock)," 2020. [Online]. Available: https://eprint.iacr.org/2023/082

[21] F. blockchain development team, "Flow prime," Flow, technical report, 2020. [Online]. Available: https://flow.com/primer

[22] (2023) The cadence programing language. [Online]. Available: https://cadence-lang.org/docs/language/

[23] J. Hannan, B. Müller, D. Shirley, B. Karlsen, A. Kline, G. Sanchez, and D. Edincik. Flow fungible token standard interface contract. [Online]. Available: https://github.com/onflow/flow-ft/blob/master/contracts/FungibleToken.cdc

[24] ——. The flow non-fungible token standard interface contract. [Online]. Available: https://github.com/onflow/flow-nft/blob/master/contracts/NonFungibleToken.cdc

[25] M. Kalinin, D. Ryan, and V. Buterin. Eip-3675: Upgrade consensus to proof-of-stake. [Online]. Available: https://eips.ethereum.org/EIPS/eip-3675

[26] D. Labs. Flow developers - fees. [Online]. Available: https://developers.flow.com/build/basics/fees

[27] S. Marx. Understanding ethereum smart contract storage. [Online]. Available: https://programtheblockchain.com/posts/2018/03/09/understanding-ethereum-smart-contract-storage/

[28] D. Shirley. Upgrading flow: How crescendo exceeded expectations. [Online]. Available: https://flow.com/post/how-crescendo-exceeded-expectations