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 arrives through the ability of defining these tokens through 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 issue arrived by 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 introduced by Flow, 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 other 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 how it organises itself from the architecture standpoint, 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

Academic research using Non-Fungible Tokens is as recent as the concept itself. The Quantum project that produced the first NFT happened a decade ago, therefore NFT-based research is necessarily younger. But 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 [3], [4], and [5] explore a tokenisation approach to manage real estate, where houses, apartments, land plots, etc. are abstracted by NFTs since the mechanics used to operate with NFTs in a blockchain are quite similar to how real estate markets work, and real estate properties share the uniqueness and individuality of NFTs as well. A similar approach is followed by [6] where they use NFTs to abstract pharmaceutical products and use a *digital twin* approach to ensure the traceability of a given product by mirroring the lifecycle of a NFTs within a blockchain with a series of checkpoints, as the product goes from the production line to where it is going to be distributed. This tokenisation trend continues with [7], where a similar strategy is used to propose a energy management system for microgeneration cases. 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 cryptocurrencies. Another example using the same strategy is found in [8] 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, that characterise these tickets and how they can be encoded into the metadata of an NFT.

Other 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. [9] explores the architectural aspects of specific NFT implementations in the Hyperledger environment, a framework to develop private custom blockchains. [10] and [11] present a similar high level architectural approach but with a specific scope in mind, namely, tracing and value transfer applications. Even though NFT is a recent technology, [12] and [13] presented *systematisation of knowledge (SoK)* articles about this technology, but they did not approached any architectural aspects of the technology. All publication mentioned thus far used Ethereum and Hyperledger for the examples and prototypes developed, 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 [14], [15], and [16], the latter providing the most extensive explanation.

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 going to be omitted.

A. Consensus Protocol

The first public blockchains implemented *Proof-of-Work (PoW)* consensus protocol, where nodes solve cryptographic puzzles towards getting the privilege of publishing the next block and getting any rewards that usually follow. In the years that preceded the release of Flow, this protocol fell out of favor due to the high levels of energy waste that it entitles. Computations executed in the pursuit of solving these puzzles have no use whatsoever and the popularity of PoW blockchains such as Bitcoin exacerbated this issue.

The blockchain community reacted to this by proposing alternative consensus mechanisms, such as *Proof-of-Authority (PoA)* where the nodes maintain a reputation system in the network and associating the odds of being selected to insert the next block proportional to the reputation value, *Proof-of-Elapsed Time (PoET)*, where the odds of round selection are proportional to the time the node has been waiting for selection, or *Proof-of-Stake (PoS)*, where the probability of a node being selected in a round is proportional to the amount of native cryptocurrency staked, among other less known [17].

PoS was one of the first alternatives to PoW proposed around the time when Flow was being developed. Also, around that time Ethereum announced a future fork of its chain to switch the consensus protocol to PoS in the new stream. As such, Flow was created and made available from the beginning with a PoS consensus mechanism.

B. Flow Node Roles

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

Flow differentiate its nodes into: *consensus nodes* that decide the presence and order of transactions in the blockchain, *collection nodes* to enhance network connectivity and data availability for applications, *execution nodes* that perform the computations required in transactions, and *verification nodes* to validate outputs returned from the execution nodes [18]. 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. These were developed by the Flow creators for this specific purpose [19].

C. Cadence Language

Flow's version of Ethereum's Solidity is Cadence. It is a programming language used to write smart contracts in Flow, as well as the scripts and transactions used to interact with the blockchain and deployed smart contracts. Ethereum uses Solidity only for smart contract development. Interaction with the blockchain, i.e., invoke a function, access a public parameter, etc, from a deployed contract, Ethereum API is compatible with several general purpose languages, such as Python and Javascript, for that purpose. Flow integrates all these operations in its own programming language. Cadence implements a *Resource Oriented Programming Paradigm* through a strongly static type system, with built in access control mechanics that can be further specialised, i.e., to narrow the scope of allowed users, through the utilisation of *capability-based* security. Cadence syntax was inspired by modern general-purpose programming languages such as *Swift*, *Kotlin*, and *Rust* [20] [21]. Files written in Cadence, namely smart contracts, transactions and scripts, have the `.cdc` extension. The following sections go into greater detail about the concepts introduced thus far.

1) Smart Contracts: Smart contracts in Flow serve the same purpose as in Ethereum and other blockchains with similar support. Syntactically these are quite different from Solidity contracts, but functionally they are very similar: both use the **contract** keyword to define the main contract structure, define constructor functions that executed automatically once during deployment and can extend their functionalities by importing external contracts and interfaces. They do have differences as well. For example, unlike Solidity, Cadence does not require the implementation of default destructors in contracts because storage management is automatically managed in Flow when resources or other contracts are deleted/destroyed. Smart contracts deployed in Flow stay initially in an "updatable" phase. During it, the code can be changed in the block storing it. Once the developer(s) are satisfied with the contract's performance, they can *lock* it, thus preventing any future changes. This allows for cleaner deploys and optimise blockchain storage, unlike Ethereum that simply saves any updated versions of a contract in a new block. If the developers do not care to delete the old contract, the blockchain simply keeps these older versions in storage.

2) Interacting with Flow Smart Contracts: Flow differentiates the two types of blockchain interactions with different file types: if an interaction is limited to read operations, i.e., executing the instructions in the file **does not changes** the state of the Flow blockchain, then a *script* should be used. If the set of instructions to execute **change the state** of the Flow blockchain (by saving, modifying or deleting data), a *transaction* should be used instead. The main difference between these files is that *transactions* need a valid digital signature to execute and usually require funds to be used as gas as well since Flow, just like Ethereum and others, restrict modifications to digital objects in storage to only the owners of such object, hence it is critical that the owner signs the transactions first. *Scripts* in Flow are "free" to execute, i.e., they do not consume gas, and therefore they do not require signatures to execute. Scripts can be used to read any public parameter in a contract. This can lead to potential privacy issues if a developer does not take sufficient care in protecting sensible data in the contract.

3) Resource Oriented Paradigm: Cadence establishes its programming paradigm through a digital object that has a special status among others named *Resource* which was inspired by Rust's *linear types*. From a functional point of view, Cadence Resources are similar to Objects in any Object-Oriented programming language. The main difference resides in the control exercised by the language to ensure that Resources are unique in the 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. Resources can only be created through a smart contract function. Cadence uses the **create** keyword to create an Resource and this keyword can only be used in a smart contract. A smart contract creates a resource by first establishing a function that does creates and returns a given resource, and using a transaction to invoke such function. Creating a Resource changes the state of the blockchain, therefore it requires transactions to execute. Currently only Flow uses this paradigm and it is the main method to create NFTs. As such, Flow NFTs are not yet compatible with other blockchain due to this fundamental architectural difference.

4) Cadence Types: Cadence was developed as a type-safe language and this is part of the strategy to indi-

vidualise 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 if the object targeted by it. Capabilities can only be issued by the owner of the resource or account and they get stored in a special account storage area that is open for public access but it can only store capabilities. This is identified as the *public* storage domain. Sec. III-E1 goes into detail about these concepts. After the owner publishes a capability to the public storage domain, another user can "borrow" it. *Borrowing* is an official action in Cadence that retrieves a reference to a stored object through a previously published capability. References in Cadence are akin to memory pointers in some 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 access data from stored objects without accessing the object directly, thus protecting it against deletion or unauthorised transfers.

6) Access Control: Cadence implements access control in two levels of granularity. At a coarser level, contracts and objects can limit the access to their inner constituents, e.g., functions, parameters, structs, etc., using the **access** keyword and the scope of access inside parenthesis after. **access(all)** grants public access, i.e., anyone can execute a function or access a parameter preceded by this access modifier. **access(E)**, where E is an entitlement restricts access to only users with such entitlement.

Entitlements provide 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 the references indicated in Sec. III-E2 to stored resources and structs with different "versions", i.e., references to the same digital object but with different sets of fields and functions available. 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 same contract where *Resource R* is defined. With a *Resource R* created and in storage, retrieving auth(A) &ResourceR returns a reference to *Resource R* where only *name* is accessible. Conversely, retrieving auth(B) &ResourceR returns a reference to the same exact resource, but only *age* is available now. Cadence uses *auth(entitlement)* to specify the entitlement to use during the resource retrieval process and '&' to identify a reference to a resource. When the actual resource is referenced, Cadence uses '@' instead. Entitlements can be combined using 'and' or 'or' logic: *access(A, B) name* sets *name* to require entitlements A **and** B to be accessible (since '&', the usual symbol for AND, is already used to denote a reference), while *access(A | B) name* changes the requirements to either entitlement A **or** B.

D. Token Standards

Flow standardise its fungible and non-fungible projects by defining contract standards, in similar fashion as Ethereum. Flow developers followed Ethereum's approach in publishing contract standards as interfaces, but these had the opportunity to name them with more suggestive denominations. As such Flow regulates fungible tokens in its contracts with the *FungibleToken* standard [22], similarly to Ethereum's *ERC-20* standard. Non fungible tokens have a corresponding *NonFungibleToken* standard [23] mirroring closely Ethereum's *ERC-721* standard. Projects implementing these standards, among others, have the same effect as in Ethereum, i.e., they guarantee a set of parameters and functions in the implementing contract, which in Flow are also used to ensure interoperability within Flow's 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 on chain storage. An account in Flow is, its essence, 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 the account in the blockchain. This is also the fee required to create a new account in Flow, though the newly created account is returned with 0.001 FLOW in balance. At the time of this writing, FLOW 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 and contains a *balance* and a set of *public encryption keys* that can be used to validate transactions signed by the account. The storage area associated to the account is split into contract storage, where smart contracts deployed by the account are saved, 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 used to store the *capabilities* indicated in Sec. III-C5, therefore limited to read-only access; and a `\private` domain used only for capability storage as the previous one, but this one is accessible only to the account owner.

Object are stored using UNIX-style paths, with the storage domain as the first element. For example, `\storage\ExampleNFT` indicates an "ExampleNFT" 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 actual ExampleNFT resource.

2) Resource Collections: The storage system described in Sec. III-E1 is not very flexible if a large number of resources need to be stored. Like UNIX paths, storage paths in Flow need to be unique in each account, which complicates 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 used to save and hold other resources in storage. Collections are standardised in Flow through the *NonFungibleToken* standard. For a given resource type, the rule in Flow is to use a collection to store multiple items of the same type. A collection needs to be created and saved in a unique storage path, but after that, other resources go into and out of the collection using *deposit* and *withdraw* functions, using token identification numbers to identify the token in question instead of dealing with a myriad of storage paths. Continuing with the UNIX analogy, collections work similarly to directories in this case.

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.

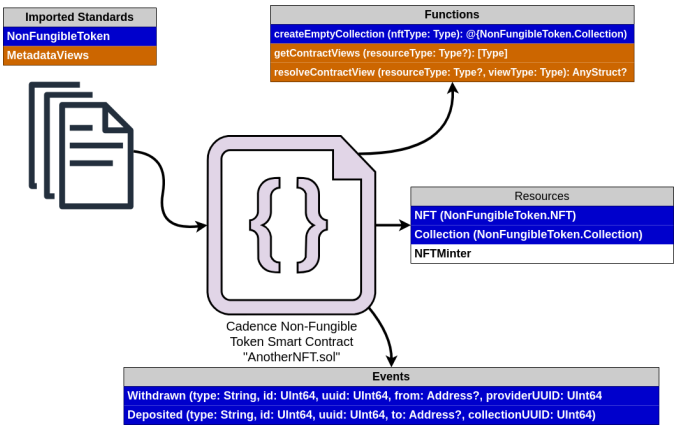

Figure 1. Structural organisation of a NFT smart contract in Cadence

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.

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)* [24] 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.

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 $

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 [25], 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 [25], 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).

VI. Conclusion

This paper presents an implementation analysis for two distinct architectures to create non-fungible tokens.

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] 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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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

[12] 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.

[13] 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

[14] 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

[15] 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.

[16] 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.

[17] 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

[18] 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

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

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

[21] (2023) The cadence programing language. [Online]. Available: https://cadence-lang.org/docs/language/

[22] 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
[23] ——. The flow non-fungible token standard interface contract. [Online]. Available: https://github.com/onflow/flow-nft/blob/master/contracts/NonFungibleToken.cdc
[24] M. Kalinin, D. Ryan, and V. Buterin. Eip-3675: Upgrade consensus to proof-of-stake. [Online]. Available: https://eips.ethereum.org/EIPS/eip-3675
[25] D. Labs. Flow developers - fees. [Online]. Available: https://developers.flow.com/build/basics/fees