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 blockchains 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 limitations in scalability and a 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 on the alternative NFT architecture introduced with 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 similar projects. The contract minted a *CryptoKitty*, an 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 was derived. Feeding this metadata into an image generator produces an image from the genome string, which is used to encode parameters such as eye and skin colour, ear type, etc. The innovative aspect of this project is that two CryptoKitties can be "bred" to generate a new one with a genome string that derives from the parent's genome. Dapper Labs established genome mechanics that produced traits randomly, which led to different rarity levels for individual kitties. This new approach translated into a peak of popularity and transactions submitted in Ethereum, which led to network stoppages stemming from scalability and throughput limitations of the network [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 centred 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. Flow uses a fundamentally different approach and claims to present the same level of NFT functionalities as Ethereum and other NFT-ready blockchains.

This article introduces the Flow blockchain and its architecture, followed by the introduction and analysis of a pair of simple implementations of an 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 to NFT-based projects.

This article continues in Section II with an overview of 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

NFTs were introduced about a decade ago with the Quantum project [3] and were popularised through Ethereum and the smart contract technology it introduced. Though NFTs existed in a small temporal window, the research community did produce a significant number of relevant publications that used them 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 traceability by mirroring the lifecycle of an NFT within a blockchain with the real product. This tokenisation trend continues with [8], where a similar strategy is used to propose an 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, identification number, etc., to individualise the token by encoding them into the metadata.

Others opted for a higher-level approach and presented an analysis based on the architectural aspects of NFTs rather than using 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 for tracing and value transfer applications. [13] and [14] presented *systematisation of knowledge (SoK)* articles about this technology, but they did not approach any architectural aspects. All publications mentioned thus far used Ethereum and Hyperledger for the examples provided, and none even mentioned Flow as an alternative. In that regard, at 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. [15], [16], and [17] provided the few mentions of Flow in academic examples, with [17] 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. It assumes a general knowledge of blockchain technology from the reader; as such, we omit basic details about blockchain technology.

A. Consensus Protocol

Early blockchains implemented *Proof-of-Work (PoW)* as a consensus protocol, where nodes solve cryptographic puzzles towards getting the privilege of publishing the next block and getting any rewards included. High levels of electric energy waste with PoW computations made this protocol unpopular. Computations executed to solve these puzzles have no use whatsoever, and the popularity of PoW blockchains such as Bitcoin exacerbated the energy waste issue.

The blockchain community reacted by proposing alternative consensus mechanisms to PoW. Reputation-based protocols, like *Proof-of-Authority (PoA)* and proportional protocols such as *Proof-of-Elapsed Time (PoET)* and *Proof-of-Stake (PoS)*, were among the most popular suggestions [18].

Dapper Labs developed Flow in 2020, around the time when PoS was becoming a popular alternative. Ethereum also switched to a PoS protocol through the Paris fork, but did so only in 2022. As such, Flow joined a growing number of public blockchains by adopting the PoS consensus protocol as well.

B. Flow Node Roles

Flow uses an innovative four-node type architecture to pipeline transaction processing in the network, which it claims to provide superior scalability and throughput. This new architecture 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 on *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 proof. SPoCKs are *Boneh-Lynn-Shacham (BLS)* signature scheme-based proofs created by the founders of Flow [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 on 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 defines a syntax 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 but functionally similar to Ethereum. Both use the **contract** keyword to define the main contract structure, use constructor functions that execute automatically once during deployment, and can extend their functionalities by importing external contracts and interfaces.

The similarities stop here. Cadence manages storage automatically, unlike Solidity; therefore, it does not require the implementation of default destructor functions in contracts. Smart contracts deployed in Flow remain in an "updatable" state initially. While in this state, contract owners can update their code. Once the developer(s) are satisfied with the contract's performance, they can *lock* it to prevent future changes, which allows for cleaner deploys and optimises blockchain storage. Ethereum 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 to be used instead. *Transactions* require a valid digital signature and funds allocated to pay for gas costs. Transactions that attempt to modify digital objects in storage must have a digital signature from the account owner, since only he/she has access to it. *Scripts* are "free" to execute because they do not consume gas. As such, scripts 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-orientated programming languages. Cadence also establishes the mechanism that guarantees the uniqueness of resources in the network. Once created, a resource can only exist in one location at a time; therefore, it cannot be copied; it can only be moved or destroyed. Only functions defined in smart contracts can create resources. Cadence limits the scope of the **create** keyword used to create a resource to a smart contract. Creating resources changes the state of the blockchain, which means that resource-creating functions in contracts need a transaction to execute. So far, resources are exclusive to Flow, which is a limiting factor to establishing interoperability with other blockchains.

4) Cadence Types: Cadence uses basic types, similar to Solidity, for elements such as integers, strings, floats, etc. The nomenclature is slightly different than in Solidity. For example, an unsigned 256-bit integer in Solidity is denoted by **uint256**, while the Cadence equivalent uses **UInt256**. Complex elements, like resources, events, structs, etc., have complex types that result from concatenating the contract name with the element. Pairing the contract name with the resource name creates a unique identifier by default. This ensures type uniqueness within Flow projects since contracts in the same storage space require different names. To uniquely identify a resource in the blockchain, this nomenclature extends to include the address of the deployed contract as a prefix to the type. `A.0ae53cb6e3f42a79.FlowToken.TokensWithdrawn` exemplifies the type of a withdraw event emitted by a FlowToken contract deployed at address 0x0ae53cb6e3f42a79.

5) Capabilities and References: Cadence provides a fine-tuned access control mechanism through an object-capability model. Capabilities are values that represent the authorisation to access and operate on a digital object. Capabilities in Cadence divide into *storage* and *account* types. Capability issuing is limited to the owner of the resource or account targeted and is stored in a special `\public` domain area available for public access but restricted to store capabilities. Sec. III-E1 provides additional details in these domains. After publishing a capability to the `public` domain, another user can "borrow" it. *Borrowing* in Cadence retrieves a reference to a stored object through a previously published capability. References in Cadence are akin to memory pointers in general-purpose programming languages. Flow uses references to protect stored objects from unauthorised transfers, modifications, and deletions by providing read access to stored objects without allowing access to the object itself.

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 that 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 are used to create references to stored resources and structs with different "versions." 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 with '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, like Ethereum, defines standards to standardise fungible and non-fungible token mechanics in smart contracts. 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 uses the *FungibleToken* standard [23] to regulate fungible token applications. Ethereum's equivalent standard is *ERC-20*. Non-fungible tokens have a corresponding *NonFungibleToken* standard [24] that mirrors Ethereum's *ERC-721* standard. These standards 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 to an account address instead of a contract address. The amount of storage space available per account is proportional to the account balance in FLOW tokens. The current rate is 1 FLOW token required per 100 MB of storage used. Flow accounts are digital objects stored in the blockchain. 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 with the account is split into contract and general-purpose. Contract storage stores the code from compiled smart contracts, and the general-purpose storage area saves 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* referred to in Sec. III-C5 limited to read-only access; and a `\private` domain used for capability storage also, but this one restricted to the owner.

Cadence uses UNIX-style paths to store digital objects, using the storage domain as the first element. `\storage\ExampleNFT` indicates an "ExampleNFT" resource stored in the main storage domain, and `\public\ExampleNFTCap` refers to an "ExampleNFTCap" capability published into the public domain.

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 complicates the storage process, as more objects are sent to storage. Dapper Labs developed Flow with the concept of digital collectibles in mind. The expectation in this blockchain is 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, is standardised in Flow through the *NonFungibleToken* standard, and each collection is limited to storing resources of one type. Collections need 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. Collections in Flow behave akin to directories in UNIX.

IV. Development of a NFT Smart Contract in Cadence

The present section assumes a minimal knowledge from the reader about general blockchain technology and NFT development in Solidity. As such, we opt to omit the implementation details from this blockchain and focus the text on the unfamiliar Cadence version. Fig. 1 presents a schematic representation of the contract developed. It displays required standard dependencies and the functions and fields required by this inheritance.

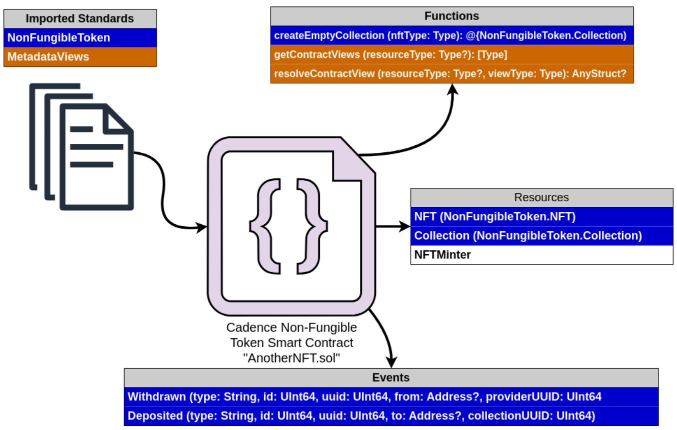

Figure 1. Structural organisation of an NFT smart contract in Cadence

A. Imported Standards

The simplest standardised version of a Cadence NFT smart contract requires the import of two standards, namely, the *NonFungibleToken* and *MetadataViews* interfaces. We omitted references to the *MetadataViews* because this import is an imposition from the *NonFungibleToken* import. Flow is particularly suited for applications with high NFT throughput, such as digital collectibles. The *MetadataViews* standard is used mostly to support metadata operations, which are used extensively in Flow to compose digital collectible objects similar to trading cards. This functional aspect falls outside the bounds of this study; therefore, we implemented the required functions from this standard as stubs. Fig. 1 uses a colour scheme to indicate which standard required the function, resource, and event indicated in the contract structure.

1) Resources: The *NonFungibleToken* standard requires the implementation of NFTs and a collection resource as minimal requirements. But Fig. 1 shows a third resource in the contract, namely a *NFTMinter* used to abstract, simplify, and secure the minting of new NFTs. New resources can only be created with a contract function due to Cadence restricting the "**create**" keyword to the context of the contract. Transactions are not able to invoke this internal action by themselves. Cadence restricts the creation of new resources to contract functions only. A secure and simple approach to resource creation is to define the create function inside of another resource and create and save it to the owner's account storage in the contract constructor. Only the owner can create new NFTs, since he/she is the only one with access to the account storage. This effectively decouples the creation of NFTs from the contract itself.

2) Functions: The *NonFungibleToken* standard requires the implementation of a *createEmptyCollection* function to obtain a collection resource, but it also requires similar implementations in the NFT and even in the collection resource itself. Standard NFT contracts in Flow provide multiple access points to create new collections. The *createEmptyCollection* function at the contract level requires a type as input because it is not possible to infer the storage type at that level. Versions at the resource level infer the storage type from the resource housing it.

3) Events: A contract that imports the *NonFungibleToken* inherits the *Withdrawn* and *Deposited* events by default. In Cadence, like in Solidity, importing standards implements and emits imported events automatically. Events imported from standards do not require explicit implementation in contracts, unlike functions and resources. Whenever an ExampleNFT moves in or out of storage, the events in Fig. 1 emit automatically.

V. Results and Comparison

A. Blockchain Comparison

Flow implements a different architecture, and it is purpose-orientated, unlike a general blockchain like Ethereum. Table I presents an objective comparison between the architectural differences between the solutions.

The technologies are similar in their general approach: both implement a fungible token as cryptocurrency and use it as gas towards optimising virtual machine computations. The term *gas* was introduced in Ethereum, but Flow implements the same exact logic, though they refer to these costs as simple *transaction fees*. Ethereum spent the first 9 years using PoW to achieve node consensus, like Bitcoin, but the Paris fork from September 2022 implemented the *Ethereum Improvement Proposal 3675 (EIP-3675)*, which switched the consensus protocol to PoS [25].

Both blockchains define smart contract programming languages to produce code that can only execute in the respective distributed virtual machines. The languages have distinct syntaxes but produce digital elements with similar behaviours and functionalities. Flow justifies their higher throughput, low operational costs, and higher scalability on their four-node pipelining architecture. Essentially, Flow doubles the roles in Ethereum's two-node architecture to establish a more efficient, but also more complex, node architecture.

Flow is the cheapest and fastest of both, with a block rate between 12 and 30 times higher and with transactions 46,154 times cheaper than Ethereum's. Yet, Ethereum's popularity and maturity still trump Flow's operational

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 $

advantages. At the time of this writing, Ethereum's daily volume of transactions is up to 2.5 times higher than Flow's. It appears that popularity and ecosystem maturity influence technology adoption more than efficiency and cost, at least on a short-term basis. It is important to notice that Ethereum has been available for twice as much time as Flow, and Flow had to compete with other newly created blockchains at a level that Ethereum never had to. But from the performance point of view, the superiority of Flow is clear.

B. *Contract Implementation Details*

We developed two smart contracts in Solidity and Cadence implementing the *ERC-721* and *NonFungibleToken* standards, respectively, and used them to execute the following sequence of macro actions:

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

This sequence of actions was relatively straightforward to execute in Solidity. Flow had some overhead that had to be dealt with first, namely the creation of additional accounts and the creation of a collection resource, as indicated in Sec. III-E2 in each account, unlike in Ethereum. Though it is possible to save an NFT directly into a storage path in Flow, we opted to include collections to illustrate the overhead impact in Flow's version of the process.

C. *Cost Comparison*

We began the comparison exercise by analysing how much gas was consumed in each of the steps considered. Ethereum gas calculations are easier since development frameworks often include gas calculation tools that keep track of the balance of the accounts throughput through a series of transactions. We used Hardhat, a popular development framework for Ethereum, for this purpose and a built-in *gas reporter* tool to determine the gas expenditure. Table II displays the results:

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

The *gas reporter* was connected to a cryptocurrency pricing oracle to obtain the gas costs in EUR in real time from off-chain data sources. The ETH price is currently very volatile. We present these gas values mainly to illustrate the difference to a cheaper option such as Flow, rather than to infer their absolute value. Flow's development framework does not have a comparable feature to Hardhat's *gas reporter*, but determining the balance of an account is easy. We calculated these fees directly by subtracting account balances between steps. But in Flow there is another simple method to determine costs. Flow imposes a base dependency to all standardised contracts to a contract named *FlowFees*. This contract is required in every Flow standard, including the *NonFungibleToken* used, to automate fee computations. The contract also emits a *FeesDeducted* event every time any fees are deducted from an account. We can capture these *FeesDeducted* events and retrieve the amount paid as fees from the argument to validate our computations. Table III presents the fee calculated for the Flow case:

1) Analysis: A quick analysis reveals that, even with some overhead included, Flow is cheaper to operate than Ethereum. Table III reveals that most transactions cost the same value of 0.00001 FLOW. This is the minimal fee established in Flow [26], which can mean that the transaction cost was actually lower than that. Flow's fee system is based on 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. We used $TxCost = (ExecFee + IncFee) \times SrFee$ to calculate the total cost of each transaction. When TxCost amounts to less than the minimal fee, TxCost ceils to it instead.

Table III
FLOW TOKEN BALANCE OF EACH ACCOUNT IN THE EXERCISE

Tx	FLOW token balance of accounts					
	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

Currently, the inclusion fee is fixed at 0.000001 FLOW, thus less than the minimum fee, and the surge factor is included in the transactional costs formula, but it is not yet available on the mainnet. This means that the computational effort required by a transaction is the most influential factor to determine the cost fee. But Table III shows that these do not exceed the minimum fee per transaction established in most cases. Contract deployment and creating new accounts are the most expensive operations. The deployment transaction cost is proportional to the size of the contract in question, which for our case amounts to 6730 bytes of storage. Flow's documentation only specifies storage costs for data in general but not for contract storage. But it estimates the cost of a deployment of a 50 KByte contract to be 0.00002965 FLOW. This fee is coherent with the logic so far, considering the small size of our contract. The documentation also confirms that account creation is indeed the most expensive of all of Flow's base operations, though a lot of that cost derives from the requirement that new accounts are created with a balance of 0.001 FLOW, and that has to be paid by the account creator. Without that value, the account creation process costs 0.00002 FLOW per account, or two minimal fees.

The similar exercise in Solidity required fewer transactions, but it cost almost 72 €. Flow required extra operations to bring the system up to par, but even with that, the total amounts to 2.00216 FLOW (around 1.72 €), a reduction by a factor of 42 compared to Ethereum. The cost in Flow 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 so that these could pay for transactions without dropping the balance lower than the required minimum. Given how much cheaper transaction costs are in flow, this could have been a lower value without risking the operation. On a cost basis, Flow is clearly the preferable option.

D. Storage Comparison

Storage is a critical aspect in blockchain applications. Blockchains replicate block data within the nodes of the network, which makes data writes in this context expensive and complex. Considering the importance of blockchain data storage, particularly for NFTs that use this (meta)data to establish properties, we run a similar analysis towards determining the storage space consumption in the exercise thus far.

1) Storage in Ethereum: Ethereum uses a *contract-based* approach to store data. This means that all contract data is stored *referenced* to the contract deployment address. Ethereum defines an "astronomically large array" indexed from the deployed contract address as the storage space of the contract. This 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. Contracts store parameters sequentially from slot index 0. But mappings, due to their dynamic nature, distribute their data throughout the storage space. Mapping values are stored in a position obtained from a digest from a hash function applied to the mapping key concatenated with a positional index, which means that mapping data is not stored in sequential position in the storage array. This complicates the storage analysis since Ethereum nor Solidity provides a direct way to determine the storage used. So, we had to devise a method that takes this into account to calculate storage values. We used a custom script that checks storage slots for data, i.e., non-zero values, and included our findings in Table IV:

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

Hardhat provides a *size-contracts* feature that returns the storage occupied by each deployed contract, which simplifies finding how much space is used by them. The results returned from this showed that contract data is consuming the lion's share of storage allocation. The remaining values were found by running a storage scanning script to determine how many slots were 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 mappings. All mappings in the ExampleNFT contract are only used for the purpose of establishing token ownership, which is confirmed in Table IV. The NFT only "exists" in transactions 01 and 02, since that is when the contract mappings show storage values. Mapping data gets deleted once the token is burnt.

2) Storage in Flow: Storage calculations are simpler in Flow. Flow accounts have a storage parameter that is trivial to obtain, similar to the balance. We use a script that automates the storage calculation process to determine each account's storage space used after each step. Table V presents our findings:

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

The emulator account displays an unusually large used storage, but this is not relevant for our exercise. To speed up and simplify development, Flow provides its emulator environment with a series of standards already deployed into this account storage area, hence the large value at startup.

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 presents a more granular storage where it discriminates saved data up to the individual byte.

Regarding the size of the NFT construct itself, Ethereum is the better option in the sense that it produces a smaller data footprint. 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, also to illustrate the differences. An Ethereum NFT computes at 94 bytes of storage, including a 32-byte string as metadata, while a programmatically smaller NFT from Flow requires 163 bytes. Flow has a significantly more storage overhead than Ethereum, but it still maintains a higher transactional throughput nonetheless. Though Flow does require more storage in absolute values, the price for storage in Flow, both in storage and transaction fees, is much less than in Ethereum, which diminishes this disadvantage.

Overall, Flow is still the better option in this case. The high volatility and value of ETH price make storing data in Ethereum too expensive. Also, this chain has a slower block rate than Flow, which means a lower rate of data writes. Flow does write more data to achieve similar functionality but does so in an inexpensive and faster fashion, which skews the preference to its direction.

VI. Conclusion

This paper presents an implementation analysis for two distinct architectures used to create NFT-based contracts. Ethereum and Flow were chosen for this exercise due to the former's role as a general-purpose and popular blockchain used as a reference and the latter, a specific blockchain created to solve known throughput and scalability issues with Ethereum.

Despite being significantly younger than Ethereum, Flow provides enough features to produce NFT-capable contracts. The structural analysis in Sec. III reveals Flow as a more complex, but also more configurable blockchain. From a programmatic point of view, Cadence is syntactically more complex; contracts consume more time to develop, but the deployment process is simpler and faster and produces smaller deployable code than the Ethereum equivalent. The cost aspect benefits Flow as well, though the limitations of Ethereum in that aspect fall outside of the technological scope considered. Flow's lack of relative popularity works in its favour by providing a cheaper and realistic alternative to NFT-based applications since FLOW's price is not as affected by the same speculative forces that risk making Ethereum too expensive to use.

Though Flow offers clear advantages, the majority of NFT-based applications in the blockchain ecosystem still use Ethereum. The 7 years that Ethereum has on Flow are a significant gap to overcome. Ethereum was the only realistic option for smart contract development for a long time, which boosted its popularity to levels difficult to replicate. Nevertheless, Flow does present a rich, albeit limited, application ecosystem focused on digital collectibles, which is the type of application suitable for an NFT-intensive blockchain.

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/