Contract-based Storage versus Account-based Storage: a Comparison among Implementation Strategies for Non-Fungible Tokens

Ricardo Lopes Almeida[1], Fabrizio Baiardi[2], Damiano Di Francesco Maesa[3], and Laura Ricci[4]

[1, 2, 3, 4]Dipartimento di Informatica, Università di Pisa, Italia
[1]Università di Camerino, Italia

November 28, 2024

Abstract

Blockchain and Distributed Ledger Technology (DLT) has been on the forefront of the latest advancements in commercial cryptographic applications. These technologies produced the first instances of decentralised digital currencies, i.e., cryptocurrencies, but these are but one of many potential applications of this technology. Non-Fungible Tokens alongside with the decentralised Virtual Machine idea increased considerably the scope of possibilities offered by blockchain and DLT.

Non-Fungible Tokens (NFTs) surge as a functional inversion of "normal" cryptocurrency tokens and thus also provide a new spectrum of utilization that is still being explored in research. The non-fungibility aspect alongside with a blockchain implementation gives NFTs a "digital uniqueness" that was impossible to achieve with centralised approaches and the potential uses for such characteristics are still being investigated. NFTs can provide a clear and simpler mechanism to establish ownership of objects, digital and otherwise, in a blockchain. The potential for this technology warrants a proper investigation of these tokens.

This paper selected two commercial and public blockchains with well known NFT capabilities and use them to create, deploy and compare NFT implementations. For this purpose Ethereum, one of the most experienced, popular and general-purpose blockchains for research; and Flow, a newer, smaller, less popular but highly NFT-specialized blockchain, were chosen for this exercise.

1 Introduction

For the last 15 years, blockchain has been one of the most disruptive fields in academic research. The first blockchain applications made available to the

wider public were cryptocurrencies, fully digitalized currencies that could be used to abstract products and services since they had a fiat price associated to it and, therefore, value. But a new way to conduct finances was just the beginning. In parallel with cryptocurrency projects, researchers and enthusiasts begun to experiment with a similar but functionally different token-based concept. Shortly after the release of Bitcoin, the first blockchain cryptocurrency in history, the first Non-Fungible Token project actually was based in the Namecoin blockchain, an offshoot project that sat somewhere between Bitcoin and Ethereum. The Quantum project [9] was started by digital artists Jennifer and Kevin McKoy. After creating the image depicted in Fig. 1, they wanted to be able to sell this artwork in its digital form. The main problem was to find a verifiable way to establish the provenance of that digital piece of art. After collaborating with technology experts, they decided to register the work in the Namecoin blockchain, but using a different approach than the one used to establish ownership of cryptocurrencies, which was by far the main application of the early public blockchains of the 2010s. The ownership relations established by registering the image in the blockchain and "locking" its owner to a single account address at all times prove to be transformative. In some aspects, they simply adapted the ownership mechanics of cryptocurrencies, also known as Fungible Tokens, to another type of token that could be used to represent an actual unique object, which can be digital or physical. Given the contradictory operations of these tokens when compared to the Fungible ones, it was only natural to name these as *Non-Fungible Tokens (NFTs)*.

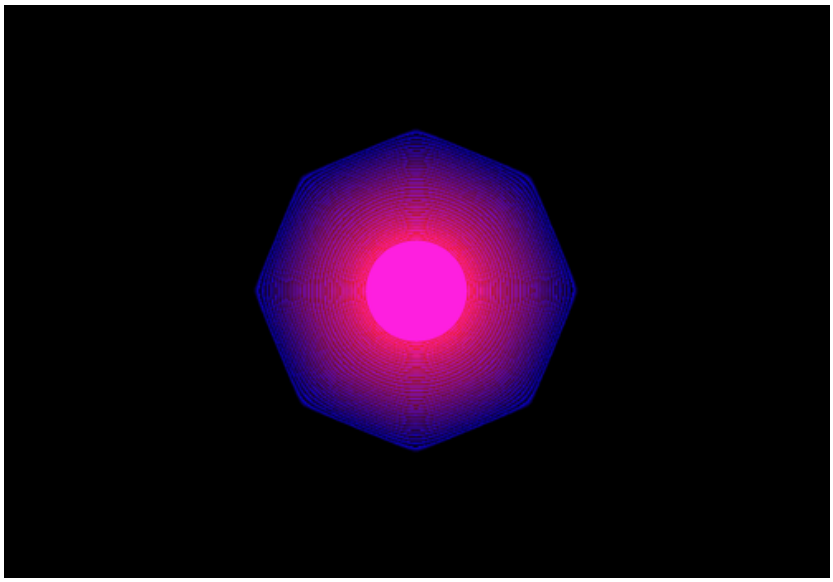


Figure 1: Quantum, the first work of art to be encoded into the metadata of an NFT [9]

Quantum was followed by other, arguably more successful, NFT projects, mostly establishing ownership of digital artworks. Popular projects such as *CryptoPunks* [26] and the *Bored Ape Yatch Club (BAYC)*, followed a similar trend as in Quantum and created finite sets of artistic NFTs that were sold directly to users, or even given away to whomever requested them, as it happen during the early days of *CryptoPunks* due to how unusual it was to buy something with crypto. Both these projects are deployed in Ethereum and transactions with these NFTs use Ethers (ETH), Ethereum's native cryptocurrency.

Both projects minted "static" NFTs in which their metadata, which is used to encode an image, either directly in the blockchain (costly) or indirectly by saving the image in a distributed repository and save its url in the NFT metadata instead (cheaper). The *CryptoKitties* NFT project [20] followed the previous two but with a slight difference. Each *CryptoKitty* was similar to a *CryptoPunk* in the sense that the image encoded in the NFT was created from an internal parameter of the NFT, and not just some random image file. The characteristics of each *CryptoKitty* were derived from an internal "genome", a 256-bit string from which substrings encoding the token characteristics (eye color, fur color, ear type, tail shape, etc.) were derived. But unlike *CryptoPunks*, *CryptoKitties* could be "bred" to generate new ones with characteristics derived semi-randomly from the parents genome. This dynamic, almost gaming, aspect of *CryptoKitties* proved to be quite attractive to collectors and enthusiast, which translated in a peak of NFT activity in Ethereum that pushed the network to and above its limits [3].

The *CryptoKitties* project exposed the limitations of the Ethereum network, particularly regarding scalability. Initially, the team behind the project (initially named *Axiom Zen* but later renamed to *Dapper Labs*) tried to address these problems from withing the Ethereum framework, but with little success. After a while, it became clear that a new blockchain architecture was needed to fully solve the issues encountered and this is how the *Flow* blockchain came to existence.

The approach taken with Flow is quite different than Ethereum from an internal perspective, but for a regular user, interacting with a Flow smart contract or NFT is not much different than with any other NFT. Just like Ethereum introduced *Solidity* as the smart contract programming language for Ethereum contracts, Flow developed *Cadence* with the exact same purpose, albeit with a significant change in how programming principles are approached. The most glaring difference between these two blockchains is how they approach data storage and ownership within the chain. Ethereum stores NFT metadata in a semi-centralised, contract-based fashion, it the sense that the central address is the one where the smart contract that regulates the NFT behavior was deployed, and all storage derives from this point. Flow has a radical approach in this sense, extending the concept of an account from just an address, as it goes in Ethereum, to use this address as a preamble to access a storage area that is unique and individual to each account, i.e., only the owner of the account has access to digital objects stored in the account's storage area. Flow also uses a similar strategy to *gas* to protect and limit the storage space associated to each

account, in which the storage space available is proportional to the amount of cryptocurrency staked in that account. Just as Ethereum uses Ether (ETH), *Flow* uses the *FLOW* token as its cryptocurrency used to pay gas fees and to sustain the account's storage area.

These differences, as well as the fact that the Flow blockchain was created precisely to address the limitations of the Ethereum blockchain regarding NFT interoperability, justify this article. In this work we detail how NFTs are implemented in each of the blockchains considered - Flow and Ethereum - in order to provide an objective comparison between them, as well as infer on how well the Ethereum limitations identified by the *CryptoKitties* project were improved in Flow.

The rest of this article is structured as follows: Section 2 provides an overview of publications relevant to our work in this article. In Section 3 we provide an introduction to concepts that are key in this work that some readers might not be as familiar as needed to fully understand this publication. Section 5 details the construction and deployment of an *ERC-721* compliant NFT Solidity smart contract in the Ethereum network, while Section 6 repeats the same process but for the Flow blockchain, which uses Cadence as the programming language to construct smart contracts. Section 7 provides a summary of our findings, as well as a comparison between the two implementations. This article concludes with Section 8.

2 Related Works

The body of research work around blockchain is quite limited and with NFTs it becomes even smaller due to how recent these technologies are. In some aspects, the technological framework of blockchain as a whole is being developed as we speak. Nonetheless, research in NFT technology and applications is an active field and has produced a significant body of work but centered on specific applications of NFTs. So much so that this in itself as justified the publication of several literature survey studies to make sense of the development of the field of NFT applications.

As example, [10], [25], and [13] provide somewhat extensive surveys of NFT applications, but they do so in a more generalised fashion, with a focus on the potential applications and expected results from the introduction of NFT-based logistics as a new approach to solve existing problems. They do a good job in enumerating the main challenges preventing a wider adoption of the technology and do present a through analysis of the main standards that promote the required interoperability in such applications.

Additionally, [33], [2], and [1] presented similar surveys but with more emphasis in the challenges and opportunities of the technology, but with lesser technical detail. The more detailed surveys are also the more recent, namely [24], and [12]. These provide a thorough synthesis of the evolution of this field, with plenty of references to concrete proposals identifying a specific instance where NFTs where used to achieve a solution.

In all of the cases considered, all emphasised the use of existing NFT standards but their mentions were limited the to the ERC standards from the Ethereum network, which is understandable given that most projects referenced in this articles are based in Solidity smart contracts deployed in the Ethereum network.Though some mention the existence of other blockchains with NFT capabilities, only [33], [24], and [12] mention the Flow blockchain specifically, with [12] providing the most detailed introduction.

2.1 Our Contribution

None of the cases considered made any mentions to the architecture behind how NFTs are implemented, either in Ethereum or any of the alternative blockchains mentioned, nor did any inference on the mechanics of data storage, specifically NFT metadata storage. As far as this writing, no other article to our knowledge was published centered on NFT storage mechanics nor considered other blockchain architectures with sufficient detail to enact a fair comparison. As such, we propose the following contributions to this research field:

1. Compared to other publications in this area, we provide a more focused and detailed introduction to the architectural aspects used to implement a smart contract capable of minting and operating with NFTs, with emphasis on the storage mechanism and control access structures used.

2. We also provide an introduction to a less popular but more application specific blockchain towards establishing a reference point for future comparisons.

3. Given that the Flow blockchain was created towards solving the technological issues that their creators encountered when implementing of the first NFTs projects in the Ethereum network, we also provide an objective comparison between the process and end result of implementing basic NFT projects in both blockchains.

3 Background

The following section provides general definitions to the main concepts that this proposal is built upon. Blockchain technology is still recent but it has matured enough and is sufficiently popular currently to a point that detailed information about it inner workings is actually quite abundant. This section intends to provide sufficient background knowledge for a reader that is not familiar with blockchain technology and related applications.

3.1 Blockchain

A blockchain is, essentially, a distributed database. A blockchain is a common data structure to a set of active nodes that compose a *peer-to-peer (P2P)*

network. Each of these active nodes has a copy of the common image of the blockchain in their internal storage. The common blockchain image refers to the data structure in which all active nodes of the P2P network agree on. Modification to this image, from here on referred to simply as blockchain, cannot be done outside of the protocol that regulate network operations and a node can only participate in that P2P network if it follows every rule established by the blockchain protocol. Protocol violators are punished, often by mechanisms baked into the protocol itself, and often removed from the network itself.

This allows blockchain to have a somewhat paradoxical nature: digital data uniqueness accomplished through data replication. The security and guarantees of uniqueness in blockchain data are proportional to the size of the network. The more active machines exist replicating **the same data structure**, the safer the data is because the higher number of potential subversions an adversary is required to do to modify blockchain data outside of the protocol.

Blockchains are append-only databases. This means that the only way to change the blockchain is to add new data to the existing common image. Deletions and modifications are nor allowed by the protocol. As a consequence, blockchains provide levels of transaction transparency that no other tool of that kind ever did: every transaction gets recorded in the blockchain in perpetuity and it is quite trivial to retrieve the historic of operations that affect a given piece of data.

Users interact with blockchains through transactions and transactions themselves are only executed if they display the correct digital signature of the "owner" of the data that can be affected by it. A simple example is a cryptocurrency transfer. Transferring any amount of cryptocurrency between two blockchain accounts changes the state of the blockchain by changing the current balance of each of the accounts affected. Or more simply, adding the transaction that moves funds from account A to account B to the blockchain is, in itself, a change of the state of the blockchain before anything else. It is important to note that this does not violate the "append-only" nature of blockchains: a blockchain account balance is the result of every transaction that ever affected it. New transactions "pile" on top of old ones but these never get deleted or modified. To allow a cryptocurrency quantity to be transferred, blockchains require only a valid digital signature for the "from" account, since it is the one that is going to get its balance diminished. Blockchain accounts are characterized by a pair of asymmetrical encryption keys, with the private one being used to digitally sign transactions in a similar fashion as with regular documents: the digital signature is the transaction text encrypted with the user's private key.

Data is appended to a blockchain with blocks and each block is a collection of transactions. A transaction is considered executed/validated once a block containing it is added to the head of the blockchain. Some applications only consider a transaction as valid once a certain number of blocks have been added on top of the block with the transaction to be validated. Consecutive blocks are connected, i.e., the "chain" in blockchain, using cryptographic hashes. Every subsequent block displays an hash digest that is calculated by concatenating the data in the current block with the hash digest of the previous block and

applying the hash function to the combination. Fig 2 illustrates this mechanic. A blockchain starts with a genesis block, which is the only one that does not contain a reference to a previous one.

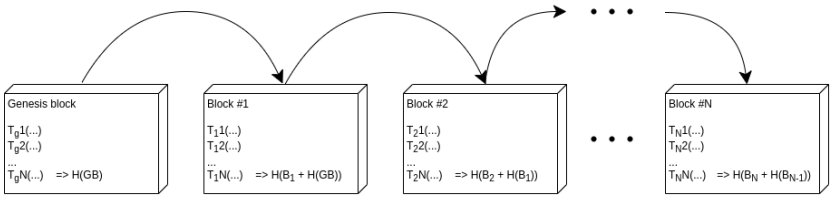


Figure 2: Example of how a generic blockchain is composed of individual blocks cryptographically connected with hash digests.

3.2 Consensus Protocols

Adding a new block is a critical step of the protocol. Only one active node can do it in each round and different protocols have different strategies to select the node allowed to publish the new block. After that, the remaining nodes update their state of the database with the new block added on top, and that becomes the latest common image until the next round. As an example, Bitcoin uses *Proof-of-Work (PoW)* to select which node gets to add the new block. New blocks are added every 10 minutes and active nodes, also known as miner nodes, try to solve a cryptographic puzzle during that period. This is a computationally intensive operation and the first node to find a solution gets to publish the new block. In most public blockchains that support cryptocurrencies, adding a new block also includes a monetary reward in the form of a newly minted cryptocurrency value. With the popularity of Bitcoin and other PoW based cryptocurrencies, this consensus protocol became progressively problematic due to the high energy cost associated to the enormous amount (and ultimately useless) number of computations required to solve these cryptographic puzzles. In part inspired by this, the blockchain community suggested and adapted more efficient consensus protocols, such as *Proof-of-Stake (PoS)*, where the new block is awarded to a node selected from a group that had previously staked some amount of cryptocurrency. The probability of getting a block is proportional to the amount of cryptocurrency locked in a stake account.

Ethereum was created initially as a PoW blockchain but it switched to a PoS consensus protocol in 2022 due to energy efficiency reasons, but also because it is a more secure and scalable protocol than PoW [8]. Another popular consensus protocol is *Proof-of-Authority (PoA)*, where nodes stake their reputation in the blockchain as leverage. Honest behaviour is rewarded and vice-versa. Higher reputations result in higher changes of publishing a new block.

3.3 Smart Contracts

The concept of smart contract was developed initially by [27] but as a reference to the automation of general-purpose legal contracts. In the blockchain context, this term refers to a script of code, witten in the specific programming language supported by the underlying blockchain, and whose execution runs synchronously on multiple nodes of the network [34]. The aggregate of computational resources of a network can be organized by a blockchain protocol running in each member, effectively creating a *Distributed Virtual Machine* and this is the main computation platform that executes transactions that can trigger smart contract functions to execute. Due to the lack of a central organizing entity, and to prevent malicious code that can damage this machine, smart contract computations require *gas*, a usually small but significant monetary value in the form of a cryptocurrency supported by the blockchain. The gas required by a smart contract execution is paid when the transaction is submitted, which means that every smart contract execution has a finite amount of gas to execute. This prevents malicious actors from running code that could exhaust the available resources of the virtual machine (by running a script with an infinite loop, for example). When the gas allocated to a transactions runs out, the transaction reverts, i.e., the blockchain recovers the state it had before attempting to run the transaction.

3.4 Non-Fungible Tokens

Non-Fungible Tokens (NFTs) work paradoxically to regular cryptocurrency tokens. Cryptocurrencies tie an address to a single variable representing the amount of that cryptocurrency the controller of that address owns. NFTs invert this relationship in the sense that they connect the token to an address instead. This clever aspect is part of what guarantees their digital uniqueness: an NFT can have one and only one owner at any point in time. NFTs essentially abstract a piece of data stored distributively in a blockchain and associate it to a single account address that represents the owner of that data. Ownership is ensured by the assumption that the private encryption key from which the account address was derived is under the control of the account owner and no one else.

The non-fungible characteristic of this tokens derives from their inability to be interchanged. With fungible tokens, as with in any cryptocurrencies, each token has a predefined value, can be traded by another one with the same denomination without any loss in value, and can be split and combined into and with fractions of other tokens. Non-fungible tokens cannot be split or combined with others, since each NFT is unique, and implies that there's no direct and fixed correspondence between any two or more NFTs. Fungible tokens are like bank notes: each note has a fixed value, can be exchanged with another with the same value or by a combination of others of smaller values that add to it without any loss in value. Non-Fungible Tokens are akin to paintings or trading cards. Each one is unique, indivisible and usually their value is highly

affected from speculation. In the public realm, most NFT applications thus far seem to emulate the analogies considered: they either represent a unique object, such as a painting or any other unique object, or they are used in a collectible setting, in which multiple NFTs are published within the context of a collection. Each token retains its uniqueness because, like physical trading cards, each has a unique identifier that differentiates it from others, even if the remaining metadata is identical to all the other tokens in that collection or series.

The NFTs ability to abstract ownership does not limit itself to digital objects. In fact, a curious argument can be made to use digital NFTs to represent physical, real "ownable objects", which includes everything from a car and a house, to less tangible goods such as ownership of land or intellectual property rights.

NFTs are implemented and can be created (minted) in a smart contract. Blockchains that support NFTs also implement a programming language that allows the creation and deployment of smart contracts that can be used to define and mint these tokens. The actual mechanics behind this process differs significantly per blockchain. This fundamental difference from the cryptographic tokens used as currency expands the scope of applications of blockchain considerably.

NFTs achieve *interoperability* through the implementation of standards. In the Ethereum network, these standards comprise of contract interfaces which define a set of requirements, namely parameters and function signatures, that a NFT token must implement. NFTs that conform to these standards indicate it so in the smart contract that governs them and the implementation of these standards guarantees to other uses a minimum of operability that ensure a secure interface with such token. In the Ethereum network, NFTs are governed by two *Ethereum Improvement Proposals (EIP)*, namely *EIP-721* and *EIP-1155*. These proposals are abstracted into the respective *Ethereum Request for Comments (ERC)* standards *ERC-721* [6] and *ERC-1155* [23], which in turn are written as smart contract interfaces that, when implemented in another contract, ensure that the any NFTs minted with that contract contain all the internal mappings required to establish ownership relationships, as well as functions required to transfer the token or to determine its internal parameters, for example.

3.4.1 Ethereum blockchain

Ethereum is one of the more popular public blockchains today. It was announced in 2013 through the publication of a whitepaper [5], followed by an *Initial Coin Offering (ICO)* in 2014, and culminating with the official launch of the new blockchain during 2015, through the release of *Frontier*, a early, bare-bone but functional version of the project. Ethereum was launched as a *Proof-of-Work* blockchain, similarly to the only existing public blockchain at that time, Bitcoin, and implemented *Ether (ETH)* as the cryptocurrency of the blockchain. The innovation of Ethereum was its smart contract support, which created a whole new logical and programmable layer on top of it. The community reacted very

positively and soon after the first Ethereum smart contract projects begun being deployed in it. ETH is used to pay the *gas* required for their executions, as well as rewarding with newly minted ETH tokens the miner node that solved the cryptographic puzzle for a given round. But like with any other cryptocurrencies with fiat value, ETH can be used as a "normal" currency. By 2022, Ethereum, still in its version 1.0, had grew considerably, and the drawbacks of using a PoW consensus protocol is such large network were becoming quite apparent and prejudicial. As such, this network switched to a PoS consensus protocol during its upgrade to version 2.0.

Ethereum was the first public blockchain to offer smart contract support though its *Ethereum Virtual Machine (EVM)*, and it was also the blockchain where the first NFTs projects were deployed. Ethereum uses Solidity to write smart contracts which in turn are used to define the rules that implement a NFT. In Ethereum NFTs are essentially defined as a combination of synchronised records kept by the implementing contract and that establish a unique relationship between a NFT, identified uniquely by id value, traditionally named *tokenId* in Solidity smart contracts (but not mandatory), and an account address that owns that token. NFT ownership is thus defined as the ability of a user to produce a correctly digitally signed transaction that can change the state of the Ethereum blockchain. This change is mostly restricted to transferring that NFT to a different address, which changes the internal mappings that establish the chain of NFT ownership, therefore triggering the blockchain to change its state. Early NFT projects limit NFT operations to only transfers, but more recent projects take advantage of *dynamic or mutable NFTs*, which are NFTs that allow the owner to change their metadata by exposing functions that allow it [12]. For an Ethereum NFT, all records that establish its ownership are centralised in the minting smart contract. Though the actual data is replicated through a series of active nodes, conceptually, all the relationships used for ownership purposes are stored in contract itself as mappings, i.e, key-value structures (known also as *dictionaries* in other programming languages). This means that there is a central point of failure in the Ethereum architecture. If the owner of the contract, or an adversary for that matter, is able to delete the contract, the ownership information is lost irreparably. This type of conceptual centralisation, allied with a fairly high block rate (15 seconds per block in average), make Ethereum difficult to scale, a fact that became evident with the sudden surge in popularity of the *CryptoKitties* project in 2017 [3].

3.4.2 Flow blockchain

The Flow blockchain was formally launched in October 2020 through the usual *Initial Coin Offering (ICO)* that helped launch most public blockchains in recent years. Flow followed a trend that was addressing the high energy consumption associated to earlier, PoW based blockchains, and established itself with the more efficient and energy friendly PoS consensus mechanism [16].

In a PoS based blockchain, new blocks are "mined" by active nodes based on their *stake* on the network, namely, the volume of the blockchain's native

cryptocurrency the node currently holds in its account. Higher stakes mean higher probability of being awarded a block publication by the protocol that regulates the network, with the token incentives that such entitles, and vice versa. This process is more energy efficient and faster than the "classical" PoW consensus, where nodes pointlessly spend computer cycles solving cryptographic puzzles only to win a race towards its completion. Given the significant energy waste due to the popularity of PoW blockchains, such as Bitcoin, a blockchain that stays clear of such nefarious protocol is not just a smart choice but a conscientious one as well [17].

Flow was created by the same team that created the *CryptoKitties* project in Ethereum, one of the first NFTs based blockchain games in this network to adopt the ERC721 standard for Non-Fungible Tokens. Even though Ethereum was the first one to offer NFT support, it did so on top of a chain that did not envision such applications at the time of its conception. This was evident by the lack of flexibility, inherent security flaws in the code implementing the tokens and its mechanics, as well as a general complexity in writing code and operating with these constructs in the Ethereum network. Ethereum was so ill prepared to deal with the popularity of this initiative that its network went down briefly due to its inability to cope with the added network traffic [3]. Flow was developed with NFTs support as its main feature, prioritizing NFT creation and mechanics over other applications, a clear contrast over blockchains focused in cryptocurrency transactions [11].

Cadence and the Resource Oriented Paradigm Flow establishes a new computational paradigm in this context, namely a *Resource Oriented Paradigm (ROP)*. It does so through textitCadence, a smart contract programming language developed for Flow and uses syntax and rules heavily inspired in popular general purpose programming languages such as *Swift, Kotlin, TypeScript*, and *Rust* [31]. Cadence establishes ROP through a special type of object, named *Resource*, inspired by the *linear types* popularised by Rust. From a technological point of view, a Resource is akin to a structure or an object from an Object-Oriented context, offering a significant degree of freedom on the type of data that it can encode. But Cadence regulates operations through the notion that every Resource is unique in Flow. This means that Resources need to be accounted at all times, and they cannot be copied, only moved. They can be stored in decentralized storage accounts and loaded from them but, at all times, the Resource is either in storage or out of storage, never in the same state at the same time. These Resources can be created only through smart contracts functions and are exclusive to the Flow blockchain, i.e., not interchangeable with other NFTs from other blockchains.

Flow implements a four-node type architecture that implements pipelining to achieve gains in speed, throughput and scalability while keeping minimal costs at a minimum sacrifice in network redundancy. Active nodes in Flow are divided into *Consensus, Collection, Execution* and *Verification* nodes that are organise in a pipeline to optimise computations. Fig. 3 provides an overview of

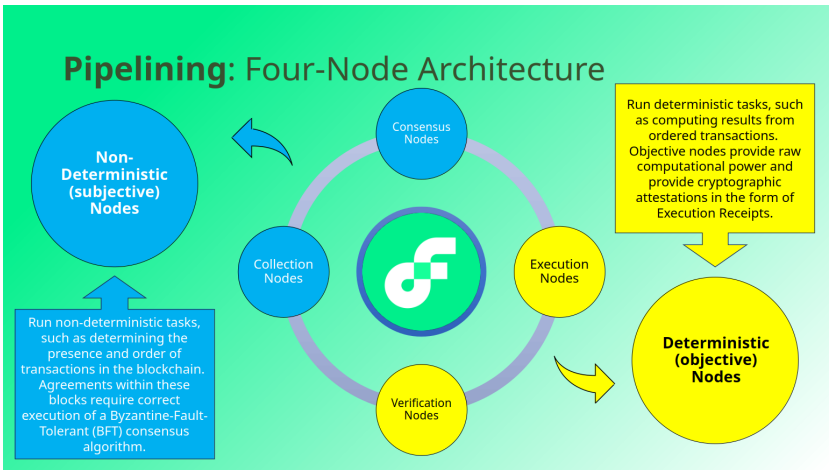


Figure 3: The four-node type architecture of Flow [18]

the architecture of Flow.

- Consensus Nodes Decide the presence and order of transactions on the blockchain.
- Collection Nodes Enhance network connectivity and data availability for applications.
- Execution Nodes Perform the computation associated with each transaction.
- Verification Nodes Responsible for keeping the Execution nodes in check.

To validate computations, the development team behind Flow developed *Specialised Proofs of Confidential Knowledge (SPoCK)*, a type of non-interactive zero-knowledge proofs based on the *Boneh-Lynn-Shacham (BLS)* signature scheme. These proofs address the *Verifier's Dilemma*, a situation where rational miners are well incentivised to accept unvalidated blocks. Miner nodes in Flow get their block reward by providing a SPoCK that can only be obtained by correctly executing all the transactions that were assigned to it [4].

3.4.3 Storage in Ethereum vs. Flow

One of the most interesting features that distinguishes Flow from other blockchains, like Ethereum, is its account-based storage model, as opposed to the contract-based one used in Ethereum, for example. In Flow, each user account has its own storage space in the blockchain. Data stored in that space is protected by the same mechanics that prevent someone from transferring a NFT that he/she does not own, transferring cryptocurrencies from another account into his/her

own etc. By default only the account controller can access and operate over its storage space. This type of storage decentralisation makes Flow more robust and potentially more scalable, since NFT transactions occur between user accounts and do not have a centralising element that needs to be changed with each transaction that affects a NFT.

Flow employs an *Account-Centric Storage Model* in which storage is the blockchain is based in an account address and the available storage space is proportional to the balance of FLOW token in that account. Due to this, Flow requires a minimum of 0.001 FLOW (around 0.0005$ at the time of this writing) in the account balance. If an account balance drops bellow this value, the account becomes limited to receive deposits and delete data until the minimum value is restored. Currently, 1 FLOW (0.5$) allows for 100 MB of on chain storage.

Since Flow saves digital objects in a private storage space associated to an account address, it uses a similar mechanic to how every operating system saves files in local storage: using file paths. Each account storage space is identified by the account address and split into three base domains:

- *Storage* This is the main private area of storage and only the account owner has any control over it, for both read and write purposes. Resources and other digital object are saved in this domain by default.
- *Public* The public domain is used to allow controlled read-only access to resources and other digital objects stored in the `\storage` domain.
- *Private* The private domain is limited to the account owners as well but it works similarly to the `\public` one in the sense it only stores capabilities and not the digital objects themselves. This domain is restricted to only the account owner, unlike the `\public` domain which can be accessed by anyone.

Digital objects stored in an account storage space are identified by a storage path obtained by the concatenation of the storage domain and an identifier. For example, an "ExampleNFT" resource saved in the user's storage domain is identified by `\storage\ExampleNFT`. Users have a control to define the identifier to be used in the storage path, as long as it is unique within the domain. This means that, in this example, any other resources saved in the same domain cannot use the "ExampleNFT" identifier anymore.

Fig. 4 displays the logical organisation of a user account in Flow. In simple terms, a Flow account is, in itself, a digital object of sorts, defined uniquely by an *address* and a set of *public keys* that are used to validate digital signatures issued by this account. The number of FLOW tokens in the account are available through its *balance* parameter. The account also has a storage area with a size proportional to the *balance* value and it is split into a *contracts* and a general *storage* area. Smart contracts deployed by this account are saved in the *contracts* area, which is publicly accessible so that anyone can verify their code and interact with them, but other digital objects, such as resources, are saved

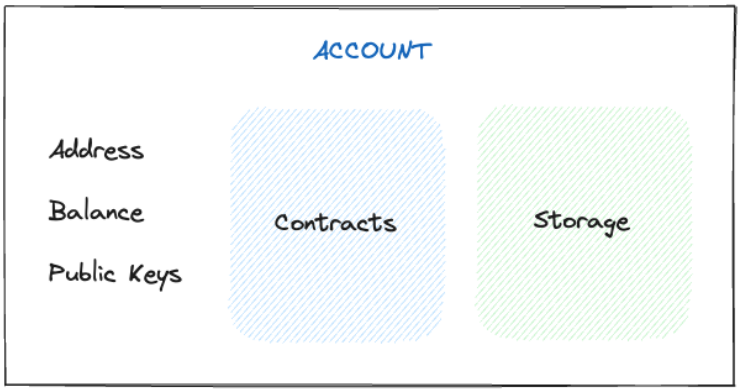


Figure 4: Logical organisation of a Flow account [28]

in the private *storage* are. Digital objects stored in this area are private by default, i.e., only accessible to the account owner, but can made public, either as a whole, or limited to certain parameters and functions, by creating and publishing *capabilities* on those resources. Sec. 3.4.3 goes into detail about this functionality used to fine tune access control in Flow.

Capabilities The `\public` and `\private` domains do not store any objects per se, they store *capabilities* instead. Capabilities are special constructs in Cadence used to delegate access to owned digital objects, i.e., stored in the user's account storage. Users other that object's owner can retrieve a *reference* to an object in someone storage from the corresponding `\public` domain, if the required capability was previously published by the object's owner. Cadence references are very similar to how memory pointers work in other general purpose languages, such as C and C++, but without allowing write access. External users use the references obtained from the `\public` domain to read parameter from the resource, as well as executing functions, as long as these do not change the resource in storage.

3.4.4 Interacting with the Flow blockchain

Similar to Ethereum, Flow functions can read and write in the blockchain, namely by executing transactions that trigger the blockchain to change its state (by changing a cryptocurrency balance, minting a new NFT, transfer a NFT to another account, etc), and as in Ethereum, functions that change the state of the blockchain need to be digitally signed by an authorized account.

In Flow, read and write operations are executed with different structures with specific names. If a block of code interacts with a deployed smart contract but it does not changes the state of the blockchain, i.e., it is restricted to data

reads, this code can be executed with a Cadence *script*. Scripts do not require signature because they do not consume gas to execute.

On the other hand, if at least one instruction in a block modifies the state of the blockchain, than a Cadence *transaction* needs to be used instead. Fundamentally, transactions are scripts that need a digital signature from an authorized account to execute. Because of it, Cadence transactions are the only context in which the transaction composer has access to a critical *AuthAccount* object that allows access to private elements of the account, such as its internal storage, FLOW token balance, etc. the assumption is that the user has read the transaction itself and knows what types of accesses it executes.

Transactions, scripts and smart contracts in Cadence all have the `.cdc` extension, unlike in Ethereum/Solidity which uses other languages, like Python or Javascript, to write scripts to interact with contract abstractions. The files differentiate themselves by how they are structured: contract files have A **contract** structure after the imports, also similar to Solidity, transactions are defined similarly, but using the **transaction** keyword and the code to execute in the transaction structure body. Scripts define a single **main** function after imports, with the code to run in its body.

3.4.5 Token standards

Token standards, for both Fungible and Non-Fungible Token, were created to establish interoperability standards so that different projects in the same network can interact among themselves, namely, cryptocurrencies and NFTs can be traded with minimal effort if users "expect" those tokens to have certain properties and functions. This enforcement is accomplished through token standards, which functionally are contract interfaces. These work very similarly to normal programming interfaces from general purpose languages, like Java for example [22], that define internal parameters and function signatures that are "forced" to be implemented if a class declares the implementation of such interface. If an NFT contract implements the ERC-721 token standard, for example, a user interacting with tokens minted by this contract can access the public internal parameter *tokenId* without having to check the actual contract to see if this parameter was implemented or not. The ERC-721 compliance of the NFT contract guarantees this, along with the remaining parameters and functions.

Ethereum Token Standards Ethereum was the first blockchain to welcome and popularize both the NFT concept, but cryptocurrencies in general through the definition of standards. Ethereum projects that followed these token standards created a interoperable ecosystem, where token, but fungible and otherwise, could be traded freely among different contracts within the Ethereum network.

- ERC-20 Fungible Token Standard The ERC-20 standard regulates the fungible tokens architecture and behaviour in the Ethereum blockchain. These tokens are mostly used as cryptocurrencies. Blockchain projects

advertise their ERC-20 compliance as a statement of compatibility with other projects in the Ethereum ecosystem. The standard itself consist in a `.sol` contract interface file written in Solidity [32].

- ERC-721 Non-Fungible Token Standard ERC-721 establishes a similar set of rule for smart contracts implementing NFTs. Implementing this interface in a contract forces it to inherit a series of key-value mappings that are used internally to implement the NFT itself, both in terms of ownership record and token metadata [6].
- ERC-1155 Multi Token Standard The non-fungible standard in Ethereum was later amended with the ERC-1155 standard to extend the ecosystem to include "semi-fungible tokens". Regular, ERC-721 tokens associate one id to one token. ERC-1155 allows for an id to be associated to a finite set of tokens. Each token with the same id is, effectively, fungible within the same set, a behaviour emulating trading card game, for example.

Flow Token Standards Flow was created with a different architecture from Ethereum and Cadence is quite different than Solidity, though both achieve similar results. But the similarities in operation with the Ethereum blockchain require that Flow established a similar set of standard for its tokens.

- Fungible Token Standard Flow's version of the ERC-20 standard is simply named *Fungible Token Standard* [29], since it was created with the express purpose of regulating fungible tokens in the network, and not as a result for a *request for comments* action. It operates very similarly to the ERC-20 counterpart: the standard itself is an inheritable contract interface in the form of a `.cdc` file [14], the extension used by the Cadence language, defining a set of parameters and functions that require implementation.
- Non-Fungible Token Standard For NFTs, Flow implements a single standard, named simply *Non-Fungible Token* [30], but it does encompass both functionalities from the ERC-721 and ERC-1155 Ethereum standards. It is, in itself, a `.cdc` [15] contract interface file that is imported from implementing contracts. NFTs have a different behaviour in Flow, mainly because of the storage architecture implemented by this blockchain. Flow NFTs are digital object stored in a specific storage path while Ethereum NFTs are, essentially, a series of synchronized key-value mappings that establish a unique relation between an address an token id. Because of this storage difference, Flow NFTs are usually stored inside collections, which are a different, non-unique, type of resource that has the capacity of "holding" multiple NFTs from the same type, i.e., minted from the same contract. This is used primarily to simplify the handling of storage paths, but it essentially mimics the behaviour from the ERC-1155 without an additional standard required.
- MetadataViews This standard it is used to extend the token functionalities regarding the processing of its metadata, as it is somewhat implied by the

standard name. This standard is a much more complex and functional version of the ERC721URIStorage. In fact, Ethereum splits its support to NFT metadata processing in several standards (ERC721URIStorage, ERC721Burnable, ERC721Metadata, ERC721Consecutive, etc.) which need to be imported to expand the support of the contract regarding token metadata processing. Flow simplifies this with one, albeit significantly more complex, standard that includes essentially all the functionalities that Ethereum splits into several standards. It is important to reference this standard, since it is going to be imported by the contract in question, but in order to keep the Solidity and Cadence projects as comparable as possible, we only address the fundamental requirements of the standard, with no additional metadata processing functionalities added to keep the two projects at the same relative level. Flow does put more emphasis in metadata manipulation due to how central NFTs are to this blockchain. At the moment of this writing, Flow has many collectible projects live in its mainnet which do extensive use of this standard to provide tokens that are rich in metadata features. But this commercial aspect of NFT projects are outside of what is intended with this study and, as such, metadata functionalities were kept to a minimum in the Flow implementation of the NFT contract.

4 Non-Fungible Tokens

NFTs here!

5 Solidity-Ethereum NFT Implementation Details

5.1 General Architecture of a Solidity NFT Smart Contract

Constructing a ERC721 compatible smart contract in Solidity begins with importing the contract interface standards, specifically the `.sol` file implementing the contract interface. These can be imported from a local path, but in recent years, informal online repositories such as *OpenZeppelin* [21] simplify this process greatly by making these standard contract interfaces available for installation as if they were normal npm modules. As such, "installing" these contracts is as easy as installing an npm module. The `.sol` files are copied to an internal *node_modules* folder and, from there, are imported into the contract, so in a way they are still being imported locally.

We begun this project by implemented a ERC-721 compliant NFT minting contract, restricting it to the minimal functionalities required by the standards imported. The idea is to restrict its functionalities to the most basic ones since these are also reflected in other architectures, specifically in the Flow

blockchain. which would allow us to to an objective comparison between smart contracts written in different languages and deployed in architecturally different blockchain but offering similar capabilities. Though architecturally, Flow and Ethereum are quite different blockchains, both NFT implementations using the official standards create common elements to both implementations, such as token unique identifiers and token balances as internal parameters, *balanceOf* and *ownerOf* functions, and *Transfer* events.

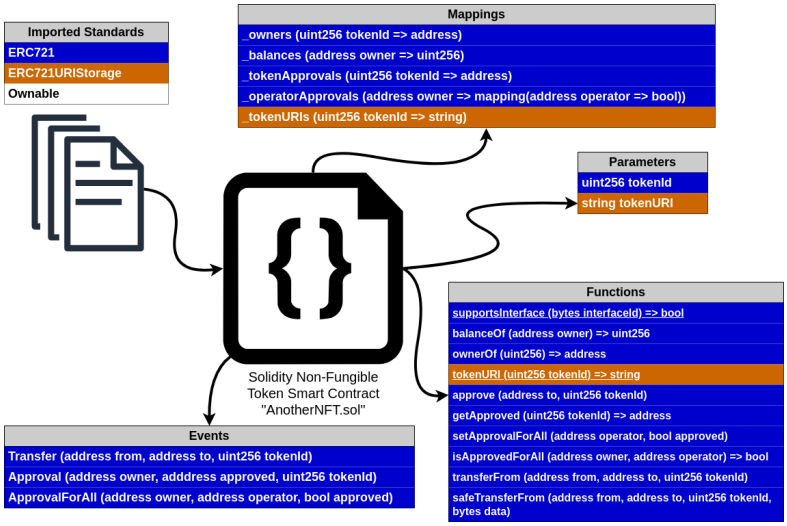


Figure 5: General organisation of a Solidity NFT minting contract

Fig. 5 references the general architecture followed in the implementation of the example NFT contract. We use a color scheme to identify the parameters, functions, event and mappings (exclusive to Solidity contracts) that are inherited from the standards. In Solidity, importing these interfaces in another contract makes the structures and functions indicated in Fig. 5 available without having to explicitly declare them in the contract. Section 5.1.1 goes into more detail about this.

For this purpose, we imported three Solidity interfaces:

1. **ERC721** This is the most important standard to implement an Ethereum NFT and that is visible in the amount of functions, parameters, mappings and events inherited from this standard alone.

2. **ERC721URIStorage** This contract interface is used to establish a more "standardised" way to deal with NFT metadata. It does so by establishing a mapping specific to store that metadata (*_tokenURIs* mapping), as well as function to retrieve the NFTs metadata.

3. **Ownable** This interface is used to simplify the establishment and verification of ownership for the contract itself only, hence why no other structures

or functions get inherited from this standard. The ownership mechanics for any NFT minted are processed by functions and mappings inherited from the ERC-721 standard. It is possible to write the contract without it, but it makes it unnecessarily complicated.

5.1.1 Interoperability derived from Standards

If a user interacts with an ERC-721 compliant Solidity NFT contract, the user can invoke the *balanceOf* function without any need for him/her to check the contract code first to ensure that the function is implemented. In fact, the majority of the cases, this function and others are simply omitted because the interface does not has them marked as **override**, which means that the implementing contract does not need to have them explicit in its code to have them available for usage. Others, such as `supportsInterface` from the ERC721 standard and `tokenURIs` from the ERC721URIStorage one are marked for overwrite, and as such they need to be explicit in the implementing contract. Overwritten functions are underlined in Fig. 5. Interface functions marked as **virtual** can be overwritten.

5.1.2 Mapping-based Ownership System

Token ownership in the Ethereum network is established by the `\_owners` mapping. When a NFT is minted, a new entry is added to this mapping creating a one-to-one relationship between an account address and a unique `tokenId`. The `tokenId` is unique within the ecosystem defined by the NFT contract. In other words, the NFT is defined by the pair contractName-tokenId.

The NFT metadata is defined through the `_tokenURIs` mapping, but this one is the one inherited by default from the ERC721Storage standard. This standard is not mandatory to implement a NFT in the Ethereum network. Realistically, NFTs can be created without any metadata set, though the usefulness of such construct is limited. Also, a mappings of this type can be added to the NFT contract without importing the ERC721URIStorage, and they can be used to store any kind of string, just like the `_tokenURIs` mapping. But using standards simplifies development and introduces a degree of certainty to other users familiar with them.

The `balances` mapping is self explanatory and the mappings suffixed with "Approvals" are used to delegate token access to other users other than the user. If the user wishes, he/she can use the "approval" functions to set permissions to allow other users to transfer the NFT and execute other owner-exclusive functions. Approval functions change the state of the blockchain, therefore require signed transactions to be executed. Each approval function is owner protected, i.e., they all begun by testing if the address requesting changes to the approval mappings is the owner of the token referred. If that is not the case, the transaction reverts.

5.1.3 Internal Parameters

In its most basic form, a standard Ethereum token, i.e., ERC721 compliant token, is defined solely by a `tokenId`, an unsigned 256-bit integer. As mentioned in Sec. 5.1.2, NFT metadata is not strictly required and a NFT contract can be implemented without one. The ERC721 standard guarantees solely the existence of the unique token id and, therefore, the uniqueness of the token in the network.

5.1.4 Events

Blockchain events are the main information mechanism to detect transaction results and contract changes in the network. Obtaining an output from functions than do not return any values, such as nft mints and token transfers, from any blockchain is quite difficult because one has to look for the correct output from the agglomerate of all the outputs of the various distributed accesses that the blockchain is attending. For a popular and sizeable blockchain, such as Ethereum, that task becomes prohibitive. Events solve this limitation. Event logging is processed differently and these can be indexed, which allows for their fast retrieval and for software tool to pool for these at intervals (event listeners). The ERC721 defines three events by default, one to be emitted whenever a token is transferred(*Transfer*) and two for whenever the approval mappings are modified. These events require arguments to be emitted, which allows for the transmission of specific information with more generalist events. For example, the *Transfer* event, when captured, also indicated the two addresses involved in the transfer and the `tokenId` of the token transferred. The emission of these events is triggered by ERC721 functions, so the whole process is automated and inherited from the contract interface.

5.1.5 Functions

Functions such *balanceOf* and *ownerOf* are self-explanatory and the approval related functions were covered in Sec. 5.1.2. The only function inherited from the ERC721URIStorage standard, *tokenURIs* is also quite simple: it receives a `tokenId` as argument and returns the string defined as metadata for the token identified. The *supportsInterface* is used to determine if the contract in question does conform to the ERC721 standard (or any other standards indicated by the contract), and it is not just something the developer is labeling the contract with, but the expected contract interface was not correctly implemented.

Finally, the last functions indicated in Fig. 5 are used to transfer the token to a `to` address. Both functions do what their name suggests, but the "safe" version does additional checks, namely if the receiving address (parameter `to`) is a contract address and, if so, test if the correct NFT receipt was returned from the contract.

6 Cadence-Flow NFT Implementation Details

Flow's version of the ERC721 standard is simply named the *NonFungibleToken* standard and is defined by a contract interface file with same name, specifically, the importable `NonFungibleToken.cdc` file. Contract interfaces in Flow/Cadence work very similarly to the ones in Ethereum/Solidity: these files define a set of default functions, internal parameters, events and resources, the latter replacing Ethereum's mappings, though in functionally different manner.

The implementation of the Flow/Cadence NFT example was as similar as possible to the Ethereum implementation from Sec. 5, towards providing the most objective comparison possible. In this sense, the Cadence NFT was implemented following the corresponding Flow standard to Ethereum's ERC721, namely the *NonFungibleToken* standard, and an auxiliary standard used to extend token functionalities, namely the *MetadataViews* standard, which provides similar functionalities to the ERC721URIStorage imported in the Ethereum solution.

Flow's version of the NFT contract distributes token functionalities in two levels: the contract itself and the resource(s) used to implement NFT mechanics. As such, representing this contract with a similar approach to the one taken for Fig. 5 results in a confusing scheme. To address this, we split the organization of the contract through the levels indicated:

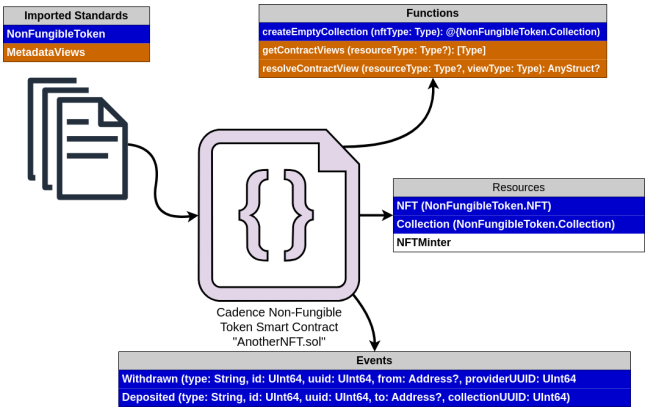


Figure 6: General organisation of a Cadence NFT minting contract

Fig 6 provides a generalist view of the NFT implementing contract. The main difference from the Solidity approach is the usage of *resources* instead of *mappings*. Other than this, the Ethereum and Flow NFT standards are quite similar, though resources are functionally more complex than mappings, thus the correspondence between these two concepts is not necessarily symmetrical even though both are used to implement token ownership in their respective blockchains.

Given the added complexity brought by Flow's usage of resources, Fig. 7

and Fig. 8 provide overviews of the two resources that require implementation due to the standardisation of the contract.

6.1 Cadence Types

Cadence is a type-safe language. Every data element in Cadence has a type, including complex elements such as resources and struts. Cadence relies in type matching as means to ensure data integrity in blockchain operations. Complex types are defined by the name of the contract that define them concatenated with the name of the resource. For example, if smart contract **ContractA** defines a resource named **ResourceA**, they type of each instance of this resource is **ContractA.ResourceA**.

6.2 Contract defined Resources

In Flow, Non-Fungible Tokens are digital objects, defined very similarly to objects in *Object-Oriented Programming* language, such as Java, Python, C#, etc., except that resources are limited to exist in one location at a time, cannot be copied, etc. The definition of a basic Non-Fungible Token digital object is shown in Fig. 7. In its simplest form, a *NonFungibleToken* compliant resource requires just an id.

The Cadence implementation is objectively more complex than the Ethereum one, based on how more complicated Flow's storage architecture is compared to Ethereum. While it was possible to represent Ethereum's NFT architecture in one figure, the same exercise for the Cadence version requires more detail to be properly understood. Cadence smart contracts define NFTs as a resource but they also typically define an adjacent resource, usually in the same NFT contract, named *Collection*.

In simple terms, a Collection is a resource that can store other resources. Collection are used to simplify the storage of multiple tokens into a single user account. In Flow tokens are saved to a logic path in an account's storage area, as it was referred in Sec. 3.4.3, that needs to be unique for every resource in storage. In other words, it is not possible to save two resources to the same storage path, just like an operating system cannot save two files to the same file path (the underlying principle is the same). This limitation can exponentially complicate storage of multiple objects, since each requires a new storage path different from the other ones.

Collections solve this by enabling the storage of multiple digital objects in the same collection, as long as they share the same type, thus requiring only one unique storage path for the collection itself. In the operating system analogy, collections are akin to directories that contain multiple files. The storage intricacies of Flow make collections almost indispensable to organise the storage of resources, and as such these were included in the NonFungibleToken standard as well.

The NFT smart contract created for this purpose, whose architecture is depicted in Fig. 6, defines three resources: the NFT itself, a NFT minter

and a Collection. The NFT and the Collection were defined according to the specifications derived from the NonFungibleToken standard and, as such, their type (indicated inside the parenthesis) derives from the standard itself. Sec. 6.1 goes into detail about Cadence use of types.

The NFTMinter is a non standardised resource that is used to simplify the minting of new NFT resources while able to maintain access to this functionality to the deployer/owner of the smart contract, as well as extending it to other users if required. New NFTs are created using the **create** keyword in a transaction (creating resources changes the state of the blockchain, therefore they cannot happen in scripts) but this operation is restricted by default to the owner of the contract, i.e, the owner of the account that contains the contract in storage. Wrapping this process in a resource, the NFTMinter, provides several advantages:

- Simplifies the creation of new NFTs by abstracting it to a function call from the NFTMinter resource.
- Provides a easy methods to delegate access by creating a publishing capabilities that can be used to delegate the creation of new NFTs to other, authorised, users.
- Increased flexibility towards additional logic to execute before creating the new resource. The function that returns the new NFT resource can be edited to include additional validations, access control, etc.

The NFTMinter is not a standardised resource, therefore its type is actually *ExampleNFT.NFTMinter* because its definition exists only in the contract named `ExampleNFT.cdc`. Since the minter is not part of any of the standards included, including one in a NFT contract is considered a good programming practice in Cadence rather than a critical resource, given that it is possible to create new NFTs without it.

6.2.1 Functions

The majority of the NFT mechanics are abstracted by the NFT resource itself, which leaves little else requiring implementation from the imported standards. The NonFungibleToken requires only a function to create empty Collections. At the contract level, this function requires a type to be provided a priori, which "locks" the collection created to accept only resources from the provided type. The remaining functions, *getContractViews* and *resolveContractView* are required by the MetadataViews standard and are used mainly to process token metadata. Since we do not intend to use them in this exercise, these were implemented as stubs.

The only function required from the NonFungibleToken standard is the *createEmptyCollection*, which returns a resource with the type *NonFungibleToken.Collection*. The '@' in the function signature indicates that the function returns a resource with the type that follows that symbol. Because this invocation occurs at the contract level, the function requires the user to provide the

type of resource that is to be stored in the collection, since this is a standardised function and as such cannot infer a priori the types of resources inferable from the contract.

6.2.2 Events

The standards define the events to emit during token transactions at the contract level. Cadence requires an explicit declaration of all the elements indicated in Fig. 6 except the events. These are inherited by default by importing the NonFungibleToken contract interface. The *withdraw* and *deposit* functions are implemented in the collection resource and the standard already includes the automatic emission of these events.

6.3 Non-Fungible Token Resource

This is the main resource in this contract. Fig. 7 presents a architecture scheme for the implementation of the NFT resource itself. This definition is part of the contract file but as a digital object, resources are significantly complex than mappings and as such deserve to be explored in detail. The ownership of each NFT in Flow is ensured by the ownership mechanism established by the blockchain, summarised by Fig. 4. Essentially, in Flow every resource needs to be "somewhere" at all times, and that implies that, at some point, resources can only exist in storage. The owner of an account that stores a NFT is also the owner of the token as well.

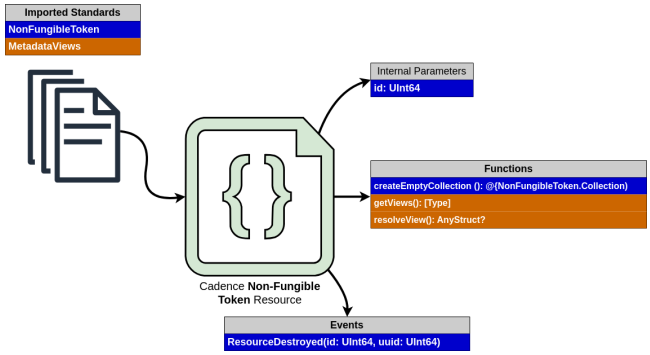


Figure 7: Organisation of a Non-Fungible Token resource in Flow

We defined the simplest NFT possible in Cadence, which consists in its unique id, a 64-bit unsigned integer, three functions that require implementation by the standard imports from the parent contract, and a event that is automatically emitted if a resource of the type of this NFT implementation, i.e., an instance of this NFT, is destroyed. Owners of resources can destroy them with the **destroy** keyword, by either invoking a contract function that does it, or running a transaction that executes this command on an owned resource. It is important to note that destroying a NFT resource in Cadence

with the **destroy** command is different than *burning* the token, which consists in transferring the resource to an unrecoverable address, as it is custom in other blockchain such as Ethereum. Flow has *Burner* contract that implements a *Burnable* interface which exposes a *burn* function. This function replicates Ethereum's burning standard by validating extensively the ownership of the token against the address that is invoking the function, but, if all validations are passed, the resource is ultimately destroyed as in deleted from the storage space where it was saved. Destroying a NonFungibleToken-standardised NFT in Flow emits the *ResourceDestroyed* event with the id of the token in question.

Function-wise, the resource required a triple of functions very similar to the requirements from the parent contract. The MetadataViews-derived function relate to the processing of the metadata of the individual resource. They also have no direct correspondence to the Ethereum version and therefore were implemented as stubs also. Finally, this resource also implements a version for the *createEmptyCollection* but this one, unlike the parent version from Sec. 6.2.1, is called from within the resource it is supposed to store. The collection resource returned from this version is already set to receive tokens with type *NonFungibleToken.NFT*.

6.4 Collection Resource

Collections use a *dictionary*, a structure used to store key-value pairs, thus similar to mappings in Ethereum. But in Cadence, dictionaries can hold resources as well. The standard imposed the use of the *ownedNFTs* dictionary to pair the id of a token with the token resource itself as value. Though collections are not restricted to save a single type of resources, this cannot violate the fact that dictionaries must have unique keys and thus cannot store tokens with the same id, even if they have different types.

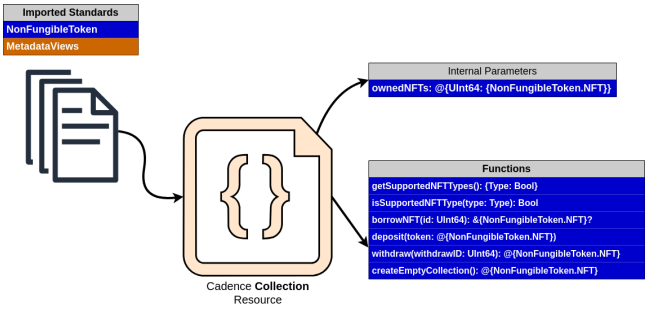


Figure 8: Organisation of a Collection resource in Flow

Data integrity in collections is validated with the *getSupportedNFTTypes* and *isSupportedType* function. These function are available for external use but they are used internally to validate if a token with a non supported type is submitted to storage.

Token transfer mechanics are abstracted and simplified by a pair of default functions from standardised collection resources, namely, *deposit* and *withdraw*. Resources are saved and loaded from storage using *save* and *load* functions from *account* objects authorised to manipulate the respective account storage. This is the process to save collection resources to an account's storage, normally implemented with a transaction. But once a collection is available from storage, further stores of other resources, most usually NFTs, is simplified by calling the respective function from the collection resource itself. *Deposit* requires a token provided as argument and does not return a value, *withdraw* requires the id of the token to retrieve and returns the resource. Cadence's version of Solidity's *transfer* function is split into a pair of function covering both directions of a transfer operation. Both functions emit the *Deposited* and *Withdrawn* indicated in Sec. 6.2.2 respectively.

Flow allows users to retrieve data from a NFT resource saved in another user's account without even getting control of the resource itself by "borrowing" a reference to a token in storage instead of the resource itself. A *resource* in Cadence works similar to a memory pointer retrievable to a resource saved in a different storage area. The *borrowNFT* function can be customised to control which parameters and functions of the original resource are returned in the reference, thus allowing for granular access control to the resource without relinquishing its ownership. This is particularly useful to read metadata fields.

Finally, the collection resource itself also includes yet another version of the *createEmptyCollection* function, this one also configured with the *NonFungibleToken.NFT* type as supported type for storage, since any invocation of this function also comes from within another collection resource already configured in this sense. This may seem redundant but it essentially provides these resources with a cloning functionality of sorts.

7 Results and Comparison

7.1 Technology Comparison

The implementation exercise depicted in Sec. 5 and Sec. 6 provided sufficient data to do a side-by-side comparison between the two blockchains technologies considered.

Table 9 summarises the technological characteristics that set these blockchain technologies apart.

7.1.1 Analysis

As it was mentioned in Sec. 3.4.2, Flow was created as a response to the limitations exposed by the earlier Non-Fungible Token projects. Ethereum had its official origin at the end of 2013, with the publication of Ethereum's first white paper [5], the first Ethereum NFT projects introduced in 2015 and, by the end of 2017, the *CryptoKitties* project had evidenced the limitations of this blockchain regarding its capacity to deal with spikes in NFT activity in the

Characteristic	NFT Technology used	
	Ethereum	Flow
Year	2013	2020
Native cryptocurrency	ETH	FLOW
Vitual Machine	EVM (Ethereum Virtual Machine)	FVM (Flow Virtual Machine)
Smart Contract Programming Language	Solidity	Cadence
Consensus Algorithm	Ethereum v1.0 (2013-2022): Proof-of-Work (PoS)	Proof-of-Stake (PoS)
	Ethereum v2.0 (2022-): Proof-of-Stake (PoS)	
Node Architecture	2 node	4 node
Node Types	1 – Execution Client	1 – Collector Node
		2 – Consensus Node
	2 – Consensus Client	3 – Execution Node
		4 – Verification Node
Fungible Token Standard	ERC-20	FungibleToken
Non-FungibleToken Standard	ERC-721	NonFungibleToken
Data storage	Contract-based	Account(User)-based
Block rate (average)	12 – 15 seconds per block	0.5 – 1 seconds per block
Daily Transaction average (2024)	1 – 1.25 million transactions per day	0.5 – 1 million transactions per day
Average Gas price per transaction	5.5 Gwei (~0.39$)	~0.00000845$

Figure 9: Technical comparison between the Ethereum and Flow blockchains

network. The team behind the *CryptoKitties* project took the 3 years that followed to develop a new blockchain concept to solve those and published their solution in 2020 as Flow, a new blockchain into the ecosystem. During this period, Ethereum begun by using the *Proof-of-Work* consensus algorithm but this chain was forked in September 2022 (the Paris fork which implemented *EIP-3675* [19]) and the mainnet consensus protocol was switched to a more energy friendly and efficient *Proof-of-Stake*. The advantages of PoS vs. PoW were well identified by 2020, so Flow was created with a PoS consensus protocol from the beginning.

Both blockchains run computations through a similar virtual machine abstraction, both named suggestively, but based in different, but comparable, node architectures. Ethereum establishes its *Ethereum Virtual Machine* through a 2 node architecture that split all EVM operations into either *executions* or *consensus* operations, with the nodes in this network being split between these two roles. Flow identifies this architectural aspect as a limiting one regarding the scalability of the network and suggested a 4-node architecture with explicit purpose of pipelining computations in their *Flow Virtual Machine* and to establish a *Resource*-based computational paradigm, as it was detailed previously in Sec. 3.4.2. This factor is evident in the average block rates for each case. Flow displays a block rate between 12 to 30 times faster than Ethereum, which results in faster transaction confirmations and computations besides having a more complex node architecture, thus supporting the claims for higher scalability from Flow.

Another architectural feature that distinguish these technologies is the storage model used to save data in the blockchain. Ethereum uses a contract-based model, where all ownership relations are implemented as key-value mappings stored in the smart contract itself, while Flow uses a account-based storage model where each user account also controls an individual storage space associated to that account and all digital objects owned by this account are saved in this space and nowhere else. Ethereum's model is simpler and easier to imple-

ment, but creates access bottlenecks by funneling all project-related transactions to go through the main smart contract, as well as establishing a ownership weak spot in the contract itself as well. All NFT ownership records are saved in the Ethereum smart contract, therefore if this contract becomes unavailable, so do all NFT records. Flow solves this limitation by establishing resource types that can exist "outside" of the implementing contract, while allowing these resources to be independently saved in a account storage space. This decoupling makes a NFT independent of its issuing contract, i.e., a NFT in Flow can exist in the ecosystem even if the original contract does not. It also removes the access bottleneck from before. The issuing Flow contract is usually accessed only to mint a NFT. Once this resource is created, it typically moved to the owner's storage space and from there can be moved to another user accounts using a standardised transfer functions that are inherited from the token standards. As long as these standard are deployed and available in the network, tokens can be manipulated even if the issuing contracts no longer exist. Ethereum provides a similar functionality regarding standard functions such as transfer, balanceOf, ownerOf, etc., but because the ownership mappings are always stored in the implementing contract, if this element is destroyed, the whole ownership chain is destroyed as well. Flow prevents this by decentralising its storage architecture into an account-based system.

Comparing both blockchains regarding their daily transaction average or the average gas price is merely illustrative because these values include a significant factor of popularity that does not translate directly into objective characteristics passible to be compared. Ethereum has a much slower block rate and very expensive gas prices when compared to Flow, but it still displays a daily transaction average that is the same as Flow in its lower end, but can be as much as 2.5 times larger. This is mainly due to popularity, which also drives price speculations, which justifies the large difference between average gas prices (An Ethereum transaction costs, in average, more expensive by a factor of 46154) as well as the increased daily transaction averages. Ethereum is 7 years ahead of Flow and it as also being used as the default blockchain for smart contract based projects, since it was the first public blockchain to offer explicit support for it. The elements that drive this price difference fall outside the bounds of this study, but it is important to node the massive difference that these may cause in blockchain usability.

Both blockchains define strict standards to regulate token mechanics in their chains, both of the fungible and non-fungible type. This creates a normalised application space where projects created by different development teams can interact with each other without any pre-arrange conventions, as long as both have the care to implement their tokens through the official standards. In both cases, these standards are abstracted into contract interfaces, that at their core are files that explicit mandatory structures and functions for any token implemented. The standards indicated in Table 9 were discussed previously in greater detail in Sec. 3.4.5.

7.2 Cost Analysis

Another important aspect from smart contract supporting blockchains is characterising how these manage the delegation of distributed computations, which is typically regulated with a cost structure based on charging *gas* to encourage users to deploy optimised code and to prevent abuses from malicious/erroneous code. Both Ethereum and Flow implement a *gas* fee structure used to pay for transaction costs. At a fundamental level the two systems are very similar, namely, the blockchain charges small fractions of a native cryptocurrency token to pay for computational costs.

For this study, the contracts indicated in Sec. 5 and Sec. 6 were deployed in an emulator environment. The *Hardhat* Solidity development framework was used to analyse the Ethereum contract, namely, using its internal EVM emulator, while Flow streamlines this aspect of development, providing a built-in emulator environment with the command line interface, which was used for this purpose as well.

For each case, a service/emulator account was used for contract deployment and other admin-level operations, such as minting a new NFT, while other, general-purpose accounts were used to emulate independent accounts that could receive, transfer, and burn NFTs. Each blockchain was analysed cost wise for a set of the most common operations while dealing with an NFT-based project, namely:

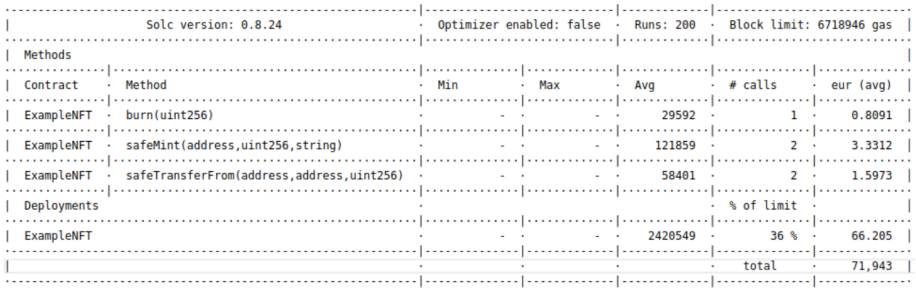
1. Deploy the NFT contract
2. Mint a new NFT into a user account
3. Transfer the NFT between user accounts
4. Burn the NFT

7.2.1 Ethereum Fee System

Ethereum defines its gas in *gwei*, a denomination of ETH, where 1 $gwei = 1 \times 10^{-9}$ ETH. The total transaction fee is split into two fee components: a *base* fee and a *priority* fee. The base fee is defined at the protocol level and is proportional to the computations that are required by the transaction. The priority component is added to the base one to make the transaction more attractive to validator and thus increase its probability of being included in the next block [7]. The total amount to pay is calculated with:

$$total\,transaction\,fee = unit\,of\,gas\,used \times (base\,fee + priority\,fee)$$

The Hardhat development framework used in this exercise provides a built-in *gas reporter* tool that provides a concise summary of the gas consumed by a sequence of operations defined in a script. The operations indicated in Sec. 7.2 were included in an script and Table 10 displays the results returned from the 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			% of limit	
ExampleNFT	-	-	2420549	36 %	66.205
			total	71,943

Figure 10: Gas consumption report from Hardat's gas reported tool

The report returned provided a global overview of how much each transaction costs but does not indicates which account paid the costs. The deployment of the contract and minting of NFT were paid by the service account while the remaining operations were paid by the account that owned the NFT. Overall, Ethereum is a notoriously expensive blockchain to operate. The ETH token is often subject of speculative periods which elevate its price above how useful the token really is. One one end, Ethereum provides a more standardised environment, given that most academic research on blockchain technology has largely preferred this blockchain. But from a practical implementation point of view, the current price to pay for basic NFT operability is prohibitive.

7.2.2 Flow Fee System

Flow implements a similar system to Ethereum to regulate distributed computations in the virtual machine. Flow does not define a gas unit per se, given that FLOW does not establishes any sub units for the moment, but it does define a minimum transaction fee of 0.000001 FLOW, unlike Ethereum. FLOW tokens are significantly cheaper than ETH (< 1 € per FLOW vs > 3000 € per ETH in 2024), so it makes sense to establish a minimum value for this blockchain. Flow transactions fees are also component based. The total transaction cost depends from three fee components:

- An Inclusion fee to pay for the inclusion of the transaction into a block, transporting information within the network and validating transaction signatures. Currently this value is fixed to the minimum transaction value of 0.000001 FLOW.
- An Execution fee to pay for FVM computations, namely, interpreting lines of code, reading and writing in storage, creating accounts, etc.
- A Surge factor applied dynamically depending on network usage. This factor is yet to be implemented in the protocol but it is intended to be used to modulate network usage spikes: when the network is too busy, this factor increases to encourage users to delay transactions, while a low factor during network idle times can be use to encourage user activity.

The total costs for a Flow transaction can be calculated with:

$$total\ transaction\ cost = (execution\ fee + inclusion\ fee) \times surge\ factor$$

Unfortunately, Flow command line interface does not provides a gas reporter or similar tool, but interacting with this blockchain is significantly more verbose than with Ethereum. Also, Flow implements a standard for fee processing, namely the FlowFees contract. This contract is deployed in mainnet, testnet and by default in an emulator instance. Gas fees are paid directly to this contract and every time a successful fee payment is registered, the contract emits a specific event, namely *FlowFees.FeesDeducted*, with the total fee paid, the inclusion fee and the execution fee as arguments. Though not ideal, it does provide a easy method to determine fee costs.

Alternatively, it is relatively easy to query the Flow blockchain to obtain FLOW balances and even the current storage level, which relates to the total costs by the storage scheme used by Flow that was described in Sec. 3.4.3, for each account involved. As such, the cost exercise was repeated for the Flow blockchain. Table 11 presents a study on how the main balance and internal storage levels fluctuate during the lifecycle of a NFT in Flow

Transactions	FLOW token balance of accounts								
	Emulator account 0xf8d6e0586b0a20c7			account01 0x179b6b1cb6755e31			account02 0xf3fcd2c1a78f5eee		
	Balance (FLOW)		Storage used (Bytes)	Balance (FLOW)		Storage used (Bytes)	Balance (FLOW)		Storage used (Bytes)
00 – New service account created	9999.99600		428655	0		0	0		9
01 – Emulator test accounts created (5)	9999.99090	-0.00510	429740	1085	0.001	0.00100	1007	1007	0.001	0.00100	1007	1007
01 – Emulator test accounts funded with 1000.0 FLOW	4999.99088	-5000.00002	429884	144	1000.001	1000.00000	1011	4	1000.001	1000.00000	1011	4
02 – Deploy ExampleNFTContract into emulator	4999.99086	-0.00002	436615	6731	1000.001	0.00000	1011	0	1000.001	0.00000	1011	0
03 – Create a NonFungibleToken.Collection in each account	4999.99081	-0.00005	437440	825	1000.00099	-0.00001	1592	581	1000.00099	-0.00001	1592	581
04 – Mint an ExampleNFTContract.NFT into each account Collection	4999.99071	-0.00010	438219	779	1000.00099	0.00000	1755	163	1000.00099	0.00000	1755	163
05 – Transfer ExampleNFT from account01 to account02	4999.99070	-0.00001	438363	144	1000.00098	-0.00001	1592	-163	1000.00099	0.00000	1812	57
06 – Transfer ExampleNFT from account02 back to account01	4999.99069	-0.00001	438504	141	1000.00098	0.00000	1755	163	1000.00098	-0.00001	1755	-57
07 – Transfer all ExampleNFT from all the other accounts to account01	4999.99065	-0.00004	439140	636	1000.00098	0.00000	2018	263	1000.00097	-0.00001	1592	-163
08 – Transfer all ExampleNFT from account01 back to the original accounts	4999.99061	-0.00004	439707	567	1000.00094	-0.00004	1755	-263	1000.00097	0.00000	1755	163
09 – Burn all ExampleNFTs from the emulator accounts	4999.99056	-0.00005	440695	988	1000.00093	-0.00001	1592	-163	1000.00096	-0.00001	1592	-163

Figure 11: Fee and storage consumption in Flow

The main conclusion from analysing Table 11 is that the majority of transactions are priced to the minimum value of 0.000001 FLOW. This particular exercise included a service or emulator account and five additional user accounts. One interesting aspect to note is that the emulator account has costs with all operations, even when the account is not included in the transaction. This is because NFT operations, such as transfers, are run from the implementing contract. Since it was the emulator account that deployed the main contract, anytime someone invokes an NFT function, the contract executions are paid by the deployer. Individual users, emulated as additional account, also incur in costs when they sign the transactions, as expected. But overall, it appears that in this simple example, the total transactions costs never exceeded the minimum value, and as such, the majority of the transactions were charged as that. In cases where it seems that a higher fee was paid, such as the 0.000005 FLOW paid by the emulator account during the final burn operation, this is actually 5 minimum transaction fees, one per each time the burn function was executed in the main contract. Given how much more cheap FLOW is compared to ETH, and how, apparently, for simple NFT interactions only the minimum fee gets charges, Flow appears to be a much more attractive economically then Ethereum.

8 Conclusion and Future Works

This paper presented a comparison between two blockchain architectures that implement the same Non-Fungible Token concept in fundamentally different ways, but achieving similar functionality. Ethereum was used as the reference blockchain for this purpose, given its popularity in academic research, extensive documentation and mature ecosystem, thus serving as a general purpose blockchain for this matter. Oppose to it was Flow, a much younger blockchain that was developed specifically to address the limitations that a general purpose blockchain, such as Ethereum, may present when dealing with NFT mechanics.

This analysis suggests that, where NFTs are concerned, Flow offers a better, more specialised architecture to implement projects based in this concept. The account-based storage model used in Flow moves into a more decentralised approach to the whole framework as compared with the contract-based approach from Ethereum. NFTs in Flow are digital resources independent from the contract that mints them.

Cost wise, the differences are more apparent. There is some irony in the fact that currently, the popularity of Ethereum is making the price of its native cryptocurrency, ETH, quite volatile and even more expensive when compared with other public blockchains with similar functionalities. ETH price is currently more influenced by external speculative events than moved by the utility of the token in the application ecosystem. When compared to FLOW, this difference is extreme. If price is a constrain in the project, Ethereum becomes quite expensive quickly.

This difference was expectable, given that Flow justifies its existence to the difficulties encountered by its creators when deploying one of Ethereum's first NFT projects. Overall, Flow offers better performance and it is easier to operate than Ethereum since the blockchain was created around the concept of resource, which are perfect abstractions for NFTs. Flow achieves this at a considerable increase in complexity, both in Cadence, the language used to write smart contracts in Flow, and in the organisation of the network itself. Given that Ethereum has a more general purpose appeal, it would be interesting to compare how Flow could fare regarding non-NFT applications. As far as we could determine, Flow's virtual machine is as capable as Ethereum's, but without proper experimental results, we need to limit Flow's superiority to NFT applications.

References

[1] Omar Ali et al. "A Review of the Key Challenges of Non-Fungible Tokens". In: *Technological Forecasting and Social Change* 187 (2023), pp. 1–13. ISSN: 00401625. DOI: 10.1016/j.techfore.2022.122248.

[2] Hong Bao and David Roubaud. "Non-Fungible Tokens: A Systematic Review and Research Agenda". In: *Journal of Risk and Financial Man-*

agement 15 (5 May 8, 2022), pp. 1–9. ISSN: 1911-8074. DOI: 10.3390/jrfm15050215.

[3] bbc.com. *CryptoKitties craze slows down transactions on Ethereum*. URL: https://www.bbc.com/news/technology-42237162 (visited on 06/22/2022).

[4] Tarak Ben, Youssef Riad, and S Wahby. *Flow: Specialized Proof of Confidential Knowledge (SPoCK)*. 2020. URL: https://eprint.iacr.org/2023/082.

[5] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. 2014.

[6] William Entriken et al. *ERC-721: Non-Fungible Token Standard*. URL: https://eips.ethereum.org/EIPS/eip-721 (visited on 11/29/2023).

[7] Ethereum.org. *Gas and Fees*. URL: https://ethereum.org/en/developers/docs/gas/ (visited on 11/25/2024).

[8] Ethereum.org. *Proof-of-Stake (PoS)*. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/ (visited on 10/24/2024).

[9] Jex Exmundo. *Quantum, The Story Behind the World's First NFT*. Mar. 2023. URL: https://nftnow.com/art/quantum-the-first-piece-of-nft-art-ever-created/.

[10] Saeed Banaeian Far et al. "A Review of Non-fungible Tokens Applications in the Real-world and Metaverse". In: *Proceedings of the 9th International Conference on Information Technology and Quantitative Management*. Vol. 214. Elsevier B.V., 2022, pp. 755–762. DOI: 10.1016/j.procs.2022.11.238.

[11] Roham Gharegozlou. *Introducing Flow, a new blockchain from the creators of CryptoKitties*. URL: https://medium.com/dapperlabs/introducing-flow-a-new-blockchain-from-the-creators-of-cryptokitties-d291282732f5 (visited on 2022-06-20).

[12] Barbara Guidi and Andrea Michienzi. "From NFT 1.0 to NFT 2.0: A Review of the Evolution of Non-Fungible Tokens". In: *Future Internet* 15 (6 June 2023). ISSN: 19995903. DOI: 10.3390fi15060189.

[13] Badis Hammi, Sherali Zeadally, and Alfredo J. Perez. "Non-Fungible Tokens: A Review". In: *IEEE Internet of Things Magazine* 6 (1 Mar. 2023), pp. 46–50. ISSN: 2576-3180. DOI: 10.1109/iotm.001.2200244.

[14] Joshua Hannan et al. *Flow Fungible Token standard Interface contract*. URL: https://github.com/onflow/flow-ft/blob/master/contracts/FungibleToken.cdc (visited on 10/28/2024).

[15] Joshua Hannan et al. *The Flow Non-Fungible Token standard Interface contract*. URL: https://github.com/onflow/flow-nft/blob/master/contracts/NonFungibleToken.cdc (visited on 10/28/2024).

[16] Alexander Hentschel, Dieter Shirley, and Layne Lafrance. *Flow: Separating Consensus and Compute*. URL: http://arxiv.org/abs/1909.05821 (visited on 06/22/2022).

[17] Alexander Hentschel et al. *Flow: Separating Consensus and Compute - Block Formation and Execution*. URL: http://arxiv.org/abs/2002.07403 (visited on 06/22/2022).

[18] Alexander Hentschel et al. *Flow: Separating Consensus and Compute - Execution Verification*. URL: http://arxiv.org/abs/1909.05832 (visited on 06/23/2022).

[19] Mikhail Kalinin, Danny Ryan, and Vitalik Buterin. *EIP-3675: Upgrade consensus to Proof-of-Stake*. URL: https://eips.ethereum.org/EIPS/eip-3675 (visited on 11/25/2024).

[20] Dapper Labs. *CrytoKitties: Collectible and Breedable Cats Empowered by Blockchain Technology*. Tech. rep. Dapper Labs, 2017, pp. 1–9.

[21] openzeppelin.com. *Build Secure Smart Contracts in Solidity*. URL: https://www.openzeppelin.com/solidity-contracts (visited on 10/29/2024).

[22] Oracle. *The Java Tutorials: What is an Interface?* URL: https://docs.oracle.com/javase/tutorial/java/concepts/interface.html (visited on 10/28/2024).

[23] Witek Radomski et al. *ERC-1155 Multi-Token Standard*. URL: https://eips.ethereum.org/EIPS/eip-1155 (visited on 11/29/2023).

[24] Qaiser Razi et al. "Non-Fungible Tokens (NFTs) - Survey of Current Applications, Evolution, and Future Directions". In: *IEEE Open Journal of the Communications Society* 5 (2024), pp. 2765–2791. ISSN: 2644125X. DOI: 10.1109/OJCOMS.2023.3343926.

[25] Wajiha Rehman et al. "NFTS: Applications and challenges". In: *2021 22nd International Arab Conference on Information Technology, ACIT 2021*. Institute of Electrical and Electronics Engineers Inc., 2021. ISBN: 9781665419956. DOI: 10.1109/ACIT53391.2021.9677260.

[26] NFTnow.com staff. *A Guide to CryptoPunks NFTs: Pricing, How to Buy, and More*. Feb. 2024. URL: https://nftnow.com/guides/cryptopunks-guide/.

[27] Nick Szabo. *Formalizing and Securing Relationships on Public Networks*. 1997. URL: https://firstmonday.org/ojs/index.php/fm/article/download/548/469.

[28] Flow development team. *Flow API documentation - Accounts*. URL: https://developers.flow.com/build/basics/accounts.

[29] Onflow Developer Team. *Flow Fungible Standard*. URL: https://github.com/onflow/flow-FT (visited on 10/29/2024).

[30] Onflow Developer Team. *Flow Non-Fungible Standard*. URL: https://github.com/onflow/flow-nft (visited on 10/29/2024).

[31] *The Cadence Programing Language*. 2023. URL: https://cadence-lang.org/docs/language/ (visited on 12/02/2023).

[32] Fabian Vogelsteller and Vitalik Buterin. *ERC-20: Token Standard*. URL: https://eips.ethereum.org/EIPS/eip-20 (visited on 10/29/2024).

[33] Qin Wang et al. "Non-Fungible Token (NFT): Overview, Evaluation, Opportunities and Challenges". In: *arXiv* (Oct. 24, 2021). URL: http://arxiv.org/abs/2105.07447.

[34] Weiqin Zou et al. "Smart Contract Development: Challenges and Opportunities". In: *IEEE Transactions on Software Engineering* 47 (10 2021), pp. 2084–2106. ISSN: 19393520. DOI: 10.1109/TSE.2019.2942301.

Todo list