Alethea Protocol

On-chain Architecture Version 2.4.1_5/9/23

Table of Contents

Version History

- Version 0.0.1 Draft
    - Defined base data components (tokens) based on the “Alethea Economy Paper”
    - Defined main functional components (helpers) based on the “Alethea Economy Paper” and “Briefing Document”
- Version 0.1
    - Redesigned how tokens and helpers interact
    - Defined functional requirements for Alethea Token
    - Added “References” section referencing main standards and improvements related to the Alethea token
    - Defined data structures for AI Pod ERC721 token and iNFT records
- Version 0.2
    - Added non-functional requirements for Alethea Token
    - Added implementation overview for Alethea Token
    - Added implementation details for Alethea Token
    - Rebranding: use “Alethea Token” instead of “Alethea ERC20 Token” to mitigate possible confusion since the token implements much more than pure ERC20
- Version 2.0
    - Added Tiny ERC721 token requirements, implementation overview and details, added illustration for 32-bits IDs tight packing
    - Added iNFT requirements, implementation overview and details
    - Added iNFT Linker requirements, implementation overview and details
    - Added Fixed Supply Sale requirements, implementation overview and details
    - Improved “Introduction” section, added protocol overview illustration
    - Added “About” section
- Version 2.1
    - Renamed AI Personality to Personality Pod
    - Added Mintable Sale description
    - Introduced Revenants (“Revenants by Alethea AI” / REV) token
- Version 2.2
    - Introduced SophiaBeing (“Sophia beingAI iNFT” / SOP) token
    - Introduced "Arkive" Mystery Box (iNFT Assets by Alethea AI / ASSET) token
    - Improved and extended Access Control descriptions, extracted it into a standalone section
    - Added the description for Upgradeable Access Control
    - Added the description for a rationale for custom Access Control implementation and its comparison with OpenZeppelin implementation
    - Fixed few typos in the text
    - Added IntelliLinkerV2 contract functional, non-functional requirements, implementation overview and details
    - NFT Airdrop Helper (PersonalityDrop, ArkiveDrop) contract functional, non-functional requirements, implementation overview and details

    - NFT Staking (PersonalityStaking) contract functional, non-functional requirements, implementation overview and details
    - OpenSeaFactory (Personality Minter, Arkive Minter) contract functional, non-functional requirements, implementation overview and details
- Version 2.3
    - Introduced Soulbound Tokens design considerations, requirements, implementation details
    - Removed the “Future of the Protocol” section
- Version 2.3.1
    - Added Soulbound Tokens explanatory diagrams
- Version 2.4
    - Added more detailed description of the fusion process
    - Added the detailed description of how to create an iNFT

Introduction

Alethea Protocol, also known as Alethea iNFT Protocol, or iNFT Protocol, embeds AI into NFTs to fully realize the promise of programmable intelligent scarcity. The goal is to turn NFTs into intelligent NFTs (iNFTs), embedded with interactive, intelligently generative capabilities and capable of sense-making and possibly human-level intelligence in the coming decades.

This document concentrates on the blockchain part of the protocol, also known as on-chain protocol architecture.

General Considerations

The protocol is modeled via *tokens* – tradable entities, ERC721 compatible and ERC20 compatible smart contracts, and *helpers* – helper smart contracts which provide interactions with and between tokens in a decentralized manner.

Tokens

Tokens are tradable, transferable entities, representing a value. Basic design approach – minimalism and immutability, tokens embed as less information as possible and, ideally, have immutable data structures.

Non-fungible tokens hold the essential data model of the protocol.

Current design approach is to store both owner information and essential token data related to the protocol mechanics in the token data structure. Some data may still be stored off-chain and in the registry smart contracts (registry helpers).

In Althea Protocol v2.0 tokens are: Alethea ERC20 (ALI), Personality Pod (ERC721), and Alethea NFT (ERC721) based collections (Revenants, etc.).

Helpers

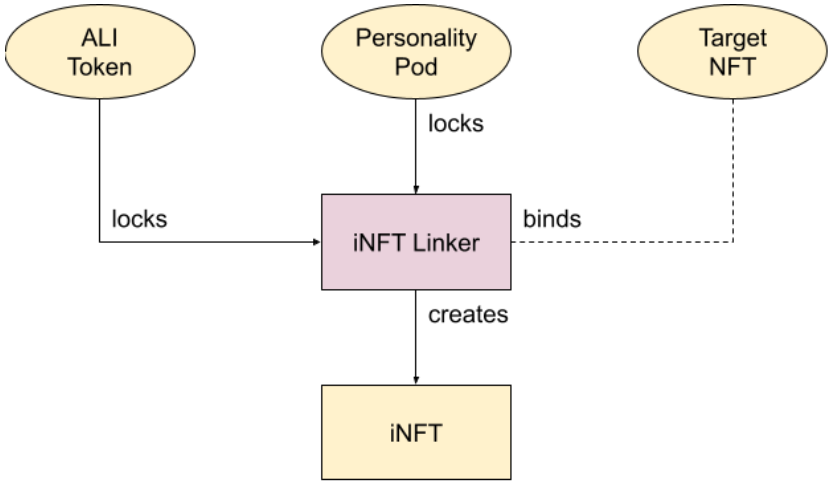
Helpers power token sales, pools, exchanges, auctions, upgrades and other token interaction logic – i.e. token data modification, ideally, without token data structure modifications. Helpers also provide storage for token data structures.

Helper smart contracts are the backbone of off-chain interaction, serving as bridges between massive off-chain and limited on-chain protocol state changes.

Helpers may not be necessarily immutable, they may be upgraded – i.e. some of them may get deprecated and disabled, other ones may appear and introduce new features or modify existing features.

In Althea Protocol v2.0 helpers are: iNFT, iNFT Linker, and Fixed Supply Sale.

The diagram below depicts several core components of the protocol in a simplified way.



Protocol Core Overview

Diagram 1. High level overview of the protocol core. ERC20/ER721 tokens are depicted as ovals, helper smart contracts – as rectangles.

Access Control

Access Control is a base parent contract for the most smart contracts of the protocol. It provides an API to check if a specific operation is permitted globally and/or if a particular user has a permission to execute it.

It deals with two main entities: features and roles. Features are designed to be used to enable/disable public functions of the smart contract (used by a wide audience). User roles are designed to control the access to restricted functions of the smart contract (used by a limited set of maintainers).

All public functions are controlled with the feature flags and can be enabled/disabled during smart contact deployment, setup process, and, optionally, during contract operation.

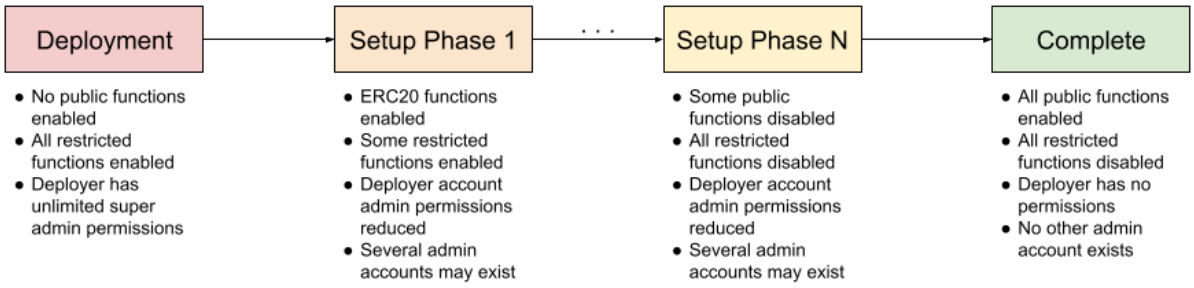
Restricted access functions are controlled by user roles/permissions and can be executed by the deployer during smart contract deployment and setup process.

After deployment is complete and smart contract setup is verified the deployer enables the feature flags and revokes own permissions to control these flags, as well as permissions to execute restricted access functions.

It is possible that smart contract functionalities are enabled in phases, but eventually smart contract is set to be uncontrolled by anyone and be fully decentralized.

It is also possible that the deployer shares its admin permissions with other addresses during the deployment and setup process, but eventually all these permissions are revoked from all the addresses involved.

Following diagram summarizes stated below:



Token Deployment Stages/Phases

Diagram 2. Token deployment and setup phases. Token evolves from the fully controlled in the initial phases of the setup process to the fully decentralized and uncontrolled in the end.

Special Permissions Mapping

Special permissions mapping, `userRoles`, stores special permissions of smart contract administrators and helpers. The mapping is a part of Access Control and is inherited by the smart contracts using it.

The value stored in the mapping is a 256 bits unsigned integer, each bit of that integer represents a particular permission. We call a set of permissions a role. Usually, roles are defined as 32 bits unsigned integer constants, but extension to 255 bits is possible.

Permission with the bit 255 set is a special one. It corresponds to the access manager role `ROLE_ACCESS_MANAGER` defined on the Access Control smart contract and allows accounts having that bit set to grant/revoke their permissions to other addresses and to enable/disable corresponding features of the smart contract (to update zero address role – see below).

“This” address mapping is a special one. It represents the deployed smart contract itself and defines features enabled on it. Features control what public[1] functions are enabled and how they behave. Usually, features are defined as 32 bits unsigned integer constants, but extension to 255 bits is possible.

Access Control is a shared parent for other smart contracts which are free to use any strategy to introduce their features and roles. Usually, smart contracts use different values for all the features and roles (see the table in the next section).

[1] Not to be confused with Solidity `public` function modifier. Publicly accessed functions are functions designed to be used by any user, with or without any special permission(s).

Access manager may revoke its own permissions, including the bit 255 one. Eventually that allows an access manager to let the smart contract “float freely” and be controlled only by the community (via DAO) or by noone at all.

Comparing with OpenZeppelin

Both our and OpenZeppelin AccessControl [3] implementations feature a similar API to check/know "who is allowed to do this thing".

Zeppelin implementation is more flexible:

- it allows setting unlimited number of roles, while current is limited to 256 different roles
- it allows setting an admin for each role, while current allows having only one global admin

Our implementation is more lightweight:

- it uses only 1 bit per role, while Zeppelin uses 256 bits
- it allows setting up to 256 roles at once, in a single transaction, while Zeppelin allows setting only one role in a single transaction

Upgradeability

Upgradeable Access Control is an Access Control extension supporting the OpenZeppelin UUPS Proxy upgrades. Smart contracts inheriting from the `UpgradeableAccessControl` can be deployed behind the ERC1967 proxy and will get the upgradeability mechanism setup.

Upgradeable Access Control introduces another “special” permission bit 254 which is reserved for an upgrade manager role `ROLE_UPGRADE_MANAGER` which is allowed to and is responsible for implementation upgrades of the ERC1967 Proxy.

Alethea ERC20 Token (ALI)

Token Summary

- Symbol: ALI
- Name: Artificial Liquid Intelligence Token
- Decimals: 18, minimal indivisible amount is 10^{-18} ALI
- Initial total supply: 10,000,000,000 ALI
- Initial supply holder (initial holder) address: `0x6B0b3a982b4634aC68dD83a4DBF02311cE324181`
- Not mintable[2]: new tokens cannot be created
- Burnable: existing tokens may get destroyed
- DAO Support: supports voting delegation

[2] Minting capability is revoked after token smart contract is deployed

Functional Requirements Summary

1. Fully ERC20 compliant according to “EIP-20: ERC-20 Token Standard”
2. Initial token supply is minted to an initial holder address
3. Token holders should be able participate in governance protocol(s) and vote with their tokens (see below – “Voting Delegation Requirements”)
4. Functional improvements required: ERC-1363 Payable Token[3], EIP-2612 permit, and EIP-3009 Transfer With Authorization

Voting Delegation Requirements

1. Token holders posses voting power associated with their tokens
2. Token holders should be able to act as delegators and to delegate their voting power to any other address – a delegate
3. Any address may become a delegate; delegates are not necessarily token owners
4. Voting power delegation doesn’t affect token balances
5. It should be possible to retrieve voting power of any delegate for any point in time, defined by the Ethereum block number (block height)
6. Delegators should be able to revoke their voting power delegation
7. Voting delegation requirements are inspired by the Compound’s COMP token; refer to Compound documentation[4] for more information not directly covered in the requirements

ERC20 Improvements Required

1. Support for atomic allowance modification, resolution of well-known ERC20 issue with approve [6]
2. Tokens owner should be able to set ERC20 allowance to “unlimited” value – the value which is not updated when token transfer is performed (like in 0x ZRX token), effectively allowing to save gas for transfers on behalf
3. Support for ERC-1363 Payable Token – ERC721-like callback execution mechanism for transfers, transfers on behalf and approvals; allow creation of smart contracts capable of executing callbacks in response to transfer or approval in a single transaction
4. Support for EIP-2612: permit – 712-signed approvals – allow token holders to approve token transfers without having an ETH to pay gas fees
5. Support for EIP-3009: Transfer With Authorization – allow token holders to send tokens without having an ETH to pay gas fees

Implied Limitations

1. Token holders are responsible not to send their tokens to non-ERC20-compliant addresses (external and/or smart contracts) via standard ERC20 interface; an attempt to do so will succeed and tokens sent may get lost forever

[3] ERC-1363 is chosen instead of ERC-777 for its simplicity
[4] https://compound.finance/docs/governance#comp

2. Voting power delegation cannot be chained: an address may delegate only voting power associated with its balance, but not the voting power delegated to it
3. Token owners are delegators but not delegates by default; they should delegate their voting power to themselves and become delegates in order to use their own voting power for voting

Non-functional Requirements

Token functions execution gas consumption depends on the token state.

Lowest gas consumption is reached for basic ERC20 functions (transfers, approvals, minting and burning) when the addresses involved don't use voting delegation. Gas consumption increases when delegation is involved.

Governance related functions (delegation and delegation on behalf) consume less gas when the balances of the delegates involved are zero, gas consumption increases when the balances are not zero.

ERC-1363 gas consumption depends on the target smart contract's callback, therefore we measure the minimum value when target callback does nothing but only event logging.

The table below summarizes gas requirements for the token operations.

Function	Gas Usage	
	No Delegation	Delegation Involved
ERC-20		
Transfers	61,696	142,005
Transfers on Behalf	71,647	151,953
Transfers on Behalf (Unlimited Allowance)	64,483	144,792
Approvals	48,520	
Atomic Approvals [ISBN:978-1-7281-3027-9]		
Atomic Approval Increase	48,876	
Atomic Approval Decrease	31,830	
ERC-1363		
Transfers	68,779	101,920
Transfers on Behalf	78,729	111,864

Transfers on Behalf (Unlimited Allowance)	78,729	111,864
Approvals	57,389	
EIP-2612		
Permits	79,095	
EIP-3009		
Transfers with Auth	77,179	157,498
Receptions with Auth	77,218	157,537
Mint/Burn Addons		
Minting	91,302	138,549
Burning	76,568	109,706
Burning on Behalf	86,060	119,198
Burning on Behalf (Unlimited Allowance)	78,940	112,078
Voting Delegation		
Function	Gas Usage	
	Empty Balances	Non-empty Balances
Delegations	50,691	113,898
Delegations on Behalf	80,586	143,790

Table 1. Gas usage requirements for ALI Token.

Token smart contract deployment is expected to cost about 3,348,175 gas.

Implementation Overview

The token smart contract is a composition of reference ERC-20, ERC-1363, EIP-2612, EIP-3009, voting delegation implementations with unlimited allowance and atomic allowance improvements added. Relevant tests from the reference implementations fully adopted to guarantee no modifications in the functionality expected.

Following reference implementations were used as a source of inspiration and tests:

- Atomic allowance:
  - https://github.com/OpenZeppelin/openzeppelin-contracts
- Unlimited allowance:
  - https://github.com/0xProject/protocol
- Voting delegation:

    - https://github.com/compound-finance/compound-protocol
    - https://github.com/OpenZeppelin/openzeppelin-contracts
- ERC-1363:
    - https://github.com/vittominacori/erc1363-payable-token
- EIP-2612:
    - https://github.com/Uniswap/uniswap-v2-core
- EIP-3009:
    - https://github.com/centrehq/centre-tokens
    - https://github.com/CoinbaseStablecoin/eip-3009

Token smart contract implementation is based on the custom AccessControl (see Access Control section).

Implementation Details

This section describes the Token smart contract implementation details.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the Token smart contract:

Solidity Name	Notation	Initial Value	Description
`totalSupply`	S_T	10 billion[5]	keeps track of total token supply

Table 2. Summary of the state variables in the Token smart contract.

Mappings

Mappings store special permissions, token balances, allowances and special permissions. Following mappings are defined on the Token smart contract:

Solidity Name	Description
`tokenBalances` `* balanceOf()`	stores token balances ERC20 balanceOf [1]
`transferAllowances` `* allowance()`	stores spending allowances ERC20 allowance [1]
`votingDelegates`	stores voting delegates info of the delegators
`votingPowerHistory`	keeps track of delegates' voting powers

[5] Taking into account decimals value set to 18, actually stored value for total supply is 10 octillion (multiplied by 1 quintillion)

`nonces`	keeps track of nonces for EIP-2612 permit
`usedNonces` * `authorizationState()`	keeps track of used nonces for EIP-3009 functions and voting delegation on behalf
`userRoles`	special permissions mapping

Table 3. Summary of the mappings in the Token smart contract.

Features and Roles

Features control the behavior of publicly available functions, such as token transfers and burning. In most cases enabling/disabling a feature allows to enable/disable corresponding functionality. Features are stored in `userRoles` mapping for the smart contract itself `userRoles[address(this)]`.

The table below summarizes the features defined in Token smart contract.

Feature	Bit	Description
`TRANSFERS`	0	Enables own tokens transfers, `transfer`
`TRANSFERS_ON_BEHALF`	1	Enables transfers on behalf, `transferFrom`
`UNSAFE_TRANSFERS`	2	Disables recipient ERC20-compliance check
`OWN_BURNS`	3	Enables burning own tokens, `burn`
`BURNS_ON_BEHALF`	4	Enables burning on behalf, `burn`
`DELEGATIONS`	5	Enables voting delegation, `delegate`
`DELEGATIONS_ON_BEHALF`	6	Enables voting delegation on behalf, `delegateWithAuthorization`
`ERC1363_TRANSFERS`	7	Enables ERC-1363 transfers, `transferFromAndCall`
`ERC1363_APPROVALS`	8	Enables ERC-1363 approvals, `approveAndCall`
`EIP2612_PERMITS`	9	Enables EIP-2612 permits, `permit`
`EIP3009_TRANSFERS`	10	Enables EIP-3009 transfers, `transferWithAuthorization`
`EIP3009_RECEPTIONS`	11	Enables EIP-3009 receptions, `receiveWithAuthorization`

Table 4. Summary of the Token smart contract features.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include minting, burning, etc. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the Token smart contract.

Role	Bit	Description
`TOKEN_CREATOR`	16	Allows minting tokens, `mint`
`TOKEN_DESTROYER`	17	Allows burning any tokens, `burn`
`ERC20_RECEIVER`	18	Disables recipient ERC20-compliance check
`ERC20_SENDER`	19	Disables ERC20-compliance check when performed by sender in `ERC20_SENDER` role
`ACCESS_MANAGER`	255	Reserved, defined in `AccessControl` smart contract, allows modifying features and roles

Table 5. Summary of the Token smart contract special permissions (roles).

Public Functions

The table below summarizes public functions designed to transfer tokens between accounts, burn tokens and authorize other addresses to perform these operations on token owner behalf.

Function Name	Description
`transfer`	ERC20 transfer, transfers own tokens Checks if the receiver is an EOA or `ERC1363Receiver` – if feature `UNSAFE_TRANSFERS` is not enabled
`transferFrom`	ERC20 transfer on behalf, transfers own tokens, or transfers tokens on behalf of address which approved the transfer Checks if the receiver is an EOA or `ERC1363Receiver` – if feature `UNSAFE_TRANSFERS` is not enabled
`safeTransferFrom`	Same as `transferFrom`, but always checks if the receiver is either an EOA or `ERC1363Receiver` smart contract
`unsafeTransferFrom`	Same as `transferFrom`, but never checks if the receiver is ERC20 compliant
`transferFromAndCall`	ERC-1363 family of transfer functions, expecting

	receiver to be a `ERC1363Receiver` smart contract
`approve`	ERC20 approve, approves transfers on behalf and burning tokens on behalf of the account which invoked `approve`
`transferWithAuthorization`	EIP-3009 transfer, transfers the tokens owned by signer Checks if the receiver is an EOA or `ERC1363Receiver` – if feature `UNSAFE_TRANSFERS` is not enabled
`receiveWithAuthorization`	EIP-3009 transfer, transfers the tokens owned by signer, can be executed only by the receiver Checks if the receiver is an EOA or `ERC1363Receiver` – if feature `UNSAFE_TRANSFERS` is not enabled
`cancelAuthorization`	EIP-3009 cancellation, cancels the EIP-3009 request to transfer tokens, or cancels the voting delegation request
`approveAndCall`	ERC-1363 family of transfer functions, expecting receiver to be a `ERC1363Spender` smart contract
`increaseAllowance`	Increases ERC20 approval value by some amount
`decreaseAllowance`	Decreases ERC20 approval value by some amount
`permit`	EIP-2612 permit, approves transfers on behalf and burning tokens on behalf of the account which signed the permit
`delegate`	Delegates voting power of the transaction sender
`delegateWithAuthorization`	Delegates voting power on behalf of the address which signed the delegation request, EIP712 compliant
`burn`	Burns own tokens, or burns tokens on behalf of address which approved burning

Table 6. Summary of public functions.

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the Token smart contract.

Function Name	Description

`mint`	Mints tokens
`burn`	Burns tokens on behalf of any address, independently of if it approved operation or not

Table 7. Summary of the functions with restricted access.

Note that Token smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

Voting Delegation

Voting delegation implemented similarly to Compound voting delegation, taking into account the non-fixed total supply nature of the token (total supply may decrease over time by design).

For better code readability, in order to follow best practices and naming conventions mappings', events' and functions' names were changed. Table below summarizes the change.

Alethea Name	Compound Name
`votingDelegates`	`delegates`
`votingPowerHistory`	`checkpoints`
`votingPowerHistoryLength`	`numCheckpoints`
`totalSupplyHistory`	`_totalSupplyCheckpoints`
`usedNonces`	`nonces`
`VotingPowerChanged`	`DelegateVotesChanged`
`votingPowerOf`	`getCurrentVotes`
`votingPowerAt`	`getPriorVotes`
`totalSupplyAt`	`getPriorTotalSupply`
`delegateWithAuthorization`	`delegateBySig`

Table 8. Summary of naming change for voting delegation related functions, events and mappings.

Note: `nonces` mapping was not just renamed, but also functionally changed allowing the use of random nonces instead of sequential. This is similar to EIP-3009, the same EIP-3009 `cancelAuthorization` function can be used to cancel the signed delegation on behalf.

Non-fungible Tokens (ERC721)

Non-fungible tokens[6] are, essentially a mapping between token ID and token owner address:

$$M_1: Token\ ID \Rightarrow Token\ Owner\ Address$$

All basic token operations like transfers, minting and burning are essentially achieved by modifying M_1:

- Zero-value (non-existent technically) mappings denote non-existent tokens
- Token creation creates an entry in the mapping
- Token destruction removes an entry from the mapping (makes it zero-value)
- Token transfer changes the value of the mapping

Token enumeration is an optional, but quite important feature, allowing light clients[7] to easily[8] enumerate tokens in existence, display them in the UI.

Global enumeration allows to iterate tokens independently of their ownership, while local enumeration allows the iteration over the tokens owned by a particular address.

Since mappings in Solidity are one-way and non-enumerable, to support enumeration of the tokens additional storage is required:

- Global enumeration – array of all the existing tokens:

$$A_1: [Token\ IDs]$$

    - To support token burning this array requires additional mapping to store global token indexes:

$$M_2: Token\ ID \Rightarrow Global\ Index$$

- Local enumeration – array of all the tokens belonging to a particular owner:

$$A_2: Owner\ Address \Rightarrow [Token\ IDs]$$

    - To support token transfers this array requires additional mapping for storing local (owner) token indexes:

$$M_3: Token\ ID \Rightarrow Local\ Index$$

[6] Non-fungible tokens are also known as NFTs, ERC721 tokens, ERC721
[7] Web3 clients with the limited/throttled access to Ethereum node (ex.: MetaMask connected via Infura)
[8] Without a need to parse transfer transaction logs (event) to reconstruct the list of existing tokens

- If token ID is short, i.e. is shrinked to 96 bits or less[9], M_3 mapping is eliminated by concatenating local index into M_1 mapping:

$$M_1: Token\ ID \Rightarrow Local\ Index \cup Token\ Owner\ Address$$

- If token ID is "tiny", i.e. is shrinked to 48 bits or less, M_2 mapping is also eliminated by concatenating global index into M_1 mapping:

$$M_1: Token\ ID \Rightarrow Global\ Index \cup Local\ Index \cup Token\ Owner\ Address$$

Storage and Gas Consumption

Having defined ERC721 token data structures, we derive storage consumption for basic operations on tokens, such as minting, transferring and burning.

On-chain Minting

On-chain minting writes (and creates) into 3 – 5 storage slots[10]:

1. Write to M_1
2. Write (append) to A_1
3. Write to M_2 – if token supports burning, and if token ID is longer than 48 bits
4. Write (append) to A_2
5. Write to M_3 – if token ID is longer than 96 bits

On-chain Transfer

On-chain transfer writes into 3 – 4 storage slots:

1. Write to M_1
2. Write (append) to A_2 – new owner
3. Write (move and shrink) to A_2 – previous owner
4. Write to M_3 – if token ID is longer than 96 bits

On-chain Burning

On-chain burning deletes from 3 – 5 storage slots:

1. Delete from M_1
2. Delete (move and shrink) from A_1
3. Delete from M_2 – if token ID is longer than 48 bits

[9] Default token ID size is 256 bits
[10] One storage slot is 256 bits

4. Delete (move and shrink) from A_2
5. Delete from M_3 – if token ID is longer than 96 bits

No Enumeration

Removing support for the enumeration allows for slight optimization:

- Global enumeration is replaced with token total supply value:

$$A_1 : Total\ Token\ Supply$$

- Local enumeration is replaced with token balances mapping of particular owners:

$$A_2 : Owner\ Address \Rightarrow Token\ Balance$$

- M_2 and M_3 structures are eliminated

Minimalistic Implementation

Removing support for all kind of balance getters results in the minimum storage costs possible:

- Removing support for totalSupply() eliminates A_1
- Removing support for balanceOf(address) eliminates A_2

Tables 9 and 10 summarize the above.

Function	Non-functional Req. Modifications			Functional Req. Modifications	
	Long[11] Token ID	Short[12] Token ID	Tiny[13] Token ID	No Enumeration	Minimalistic
Mint	5 (+)	4 (+)	3 (+)	3 (+)	1 (+)
Transfer	4 (*)	3 (*)	3 (*)	3 (*)	1 (*)
Burn	2 (*), 5 (–)	2 (*), 4 (–)	2 (*), 3 (–)	2 (*), 1 (–)	1 (–)

Table 9. Storage allocations (+), writes (*) and deallocations (–) for ERC721 with burn support.

Function	Non-functional Req. Modifications		Functional Req. Modifications	
	Long Token ID	Short Token ID / Tiny Token ID	No Enumeration	Minimalistic
Mint	4 (+)	3 (+)	3 (+)	1 (+)

[11] 256 bits
[12] 96 bits
[13] 48 bits

Transfer	3 (*)	3 (*)	3 (*)	1 (*)

Table 10. Storage allocations (+) and writes (*) for a token without burn support.

Tiny ERC721

Tiny ERC721 is an ERC721 base implementation inherited by most Alethea ERC721, including Personality Pod and other ERC721 serving as target NFTs for iNFT linking.

Functional Requirements

1. Fully ERC721 compliant token according to “EIP-721: ERC-721 Non-Fungible Token Standard”, with both optional ERC-721 extensions included (metadata and enumerable)
2. Supports token enumeration extension (optional ERC-721 interface extension)
    a. Global enumeration: it should be possible for a light client to easily enumerate tokens in existence, without the need to parse token transfer logs (events)
    b. Local enumeration: it should be possible for a light client to easily enumerate tokens owned by any given address, without the need to parse token transfer logs (events)
3. Supports token metadata extension (optional ERC-721 interface extension), including token URI
    a. Authorized address(es) should be able to set Token URI individually for existing tokens
    b. Authorized address(es) should be able to pre-set Token URI individually for tokens not yet in existence to be accessible after they are minted
    c. Authorized address(es) should be able to set / pre-set Token URI globally for all the tokens which don’t have their individual URIs defined; globally set URI should render the individual token URI as
    $$Token\ URI_i = Base\ URI \cup Token\ ID_i,$$
    where $Base\ URI$ is a part of URI common for all the tokens and index i denotes the changeable part of the URI
4. Authorized address(es) should be able to mint tokens in sequential batches, that is to mint several tokens in a row
5. Token holders should be able to approve token transfers without having an ETH to pay gas fees in a way similar to EIP-2612 – via EIP-712 signature for `approve()` – `permit()`
6. Token holders should be able to approve an address to transfer all their tokens without having an ETH to pay gas fees in a way similar to EIP-2612 – via EIP-712 signature for `setApprovalForAll()` – `permitForAll()`

Non-functional Requirements

AI Personality token ID space is limited to 32 bits and leverages “Tiny Token ID” optimization, and batch minting optimization, allowing to mint up to about 4 billion tokens:

$$Token\ ID \in \left[1,\ 2^{32} - 1\right]$$

Token functions execution gas consumption depends on the token state. The table below summarizes gas requirements for the token operations.

Function	Gas Usage	
	To EOA	To ERC721 Receiver
Transfers	101,353	105,959
Transfers on Behalf	89,652	94,258
Transfers on Behalf by Approved Operator	104,165	108,771
Approvals	48,603	
Operator Approvals	46,299	
Permits	79,236	
Operator Permits	76,953	
Mints	124,071	129,122
Batch Mints (40 tokens)	32,412	37,190
Burns	54,314	
Burns on Behalf	42,613	
Burns on Behalf by Approved Operator	57,126	

Table 11. Gas usage requirements for Tiny ERC721.

Token smart contract deployment is expected to cost about 2,876,691 gas.

Implementation Overview

The token smart contract implements core ERC721 interface with optional interfaces defined in the ERC721 specification – metadata and token enumeration.

32 bits token ID space allows for storage optimization related to enumeration: 8 token IDs (32 bits each) fit exactly into one storage slot (256 bits) of the enumeration arrays.

Similar to Alethea ERC20 Token implementation, TinyERC721 smart contract implementation is based on the custom AccessControl (see Access Control section for details).

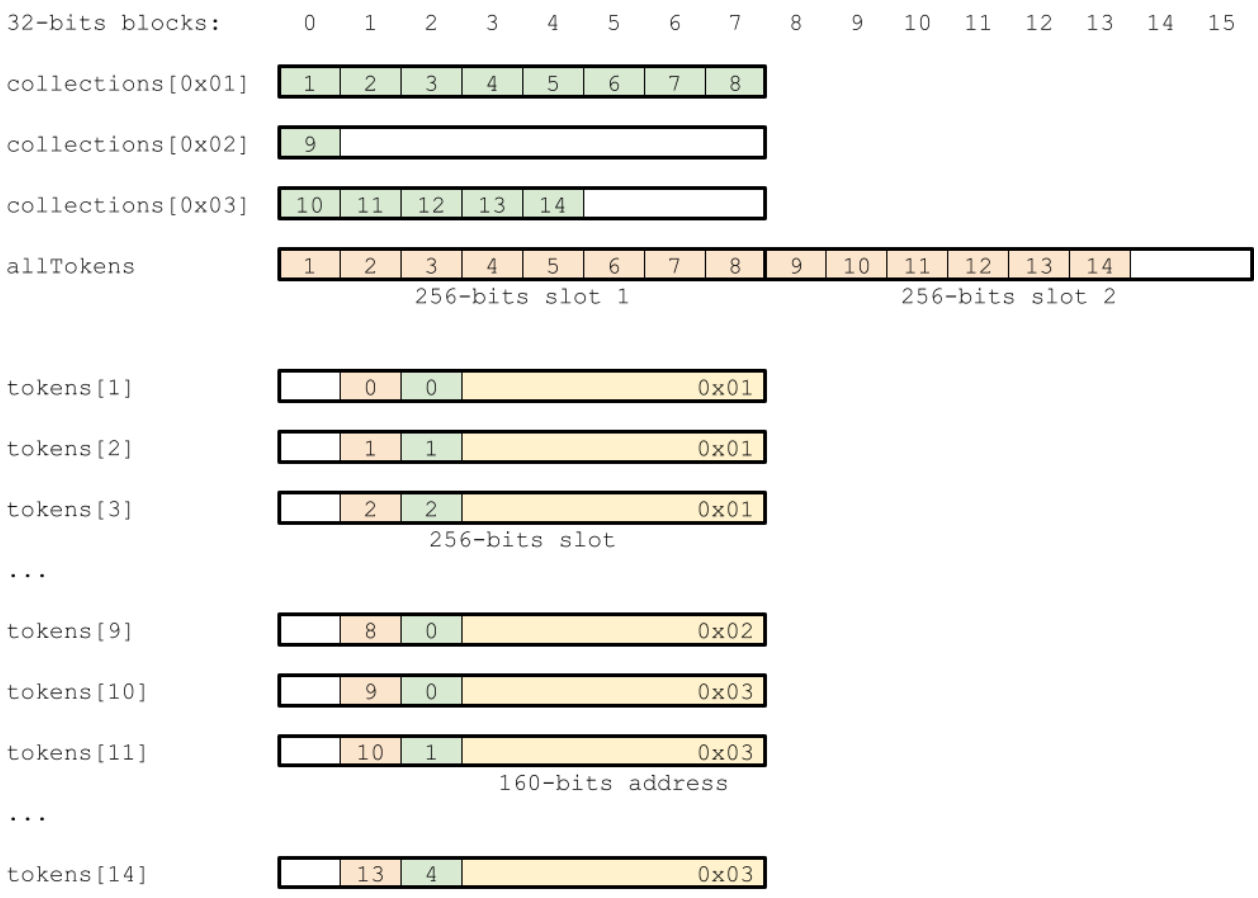
Implementation Details

This section describes the Token smart contract implementation details.

Token ID Space

Token ID space is limited to 32-bits which allows batch minting optimization, especially for batches of size multiple of eight – 8 token IDs fit exactly into the 256-bits slot in local (`uint32[] collections` – token IDs owned by a single owner) and global (`uint32[] allTokens` – all token IDs) enumeration arrays.

Token ID indexes within the local and global collections are appended to the token owner address in the `tokens` mapping, occupying 224 of 256 bits in a storage slot. Diagram below illustrates the idea.



TinyERC721 Storage Layout Example

Diagram 3. Storage allocation example for 14 tokens owned by 3 addresses (yellow). Address `0x01` owns 8 tokens, address `0x02` owns one token, and address `0x03` owns 5 tokens. Local (green) and global (orange) enumerations store token IDs, their indexes are counted from zero. Storage slots are marked with a bold border.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the Token smart contract:

Solidity Name	Description
`name` `* name()`	Descriptive name for the NFT collection represented by ERC721 smart contract, immutable, set upon deployment
`symbol` `* symbol()`	Abbreviated name for the NFT collection represented by ERC721 smart contract, immutable, set upon deployment
`baseURI` `* baseURI()`	Used to construct `tokenURI(uint256)` for tokens which don't have their URI explicitly set, mutable, can be updated with `setTokenURI(uint256, string)` function

Table 12. Summary of the state variables in the Token smart contract.

Arrays

Arrays are used to enable token enumeration support and store local and global token IDs enumerations. Both arrays store 32-bits elements. Pushing several consequent IDs into these arrays is optimized with `ArrayUtils.push32()` assembly.

Solidity Name	Description
`collections`	Stores local token IDs enumerations for each token owner
`allTokens`	Stores global token IDs enumeration

Table 13. Summary of the tightly packed arrays in the Token smart contract.

Mappings

Mappings store special permissions, token ownership info, allowances, and auxiliary info. Following mappings are defined on the Token smart contract:

Solidity Name	Description
`tokens` `* ownerOf()`	stores token ownership information and supports ERC721 `ownerOf` function
`approvals`	stores approved operators for individual tokens, supports

* getApproved()	ERC721 getApproved function
approvedOperators * isApprovedForAll()	stores approved operators for token owners, supports ERC721 isApprovedForAll function
permitNonces	keeps track of nonces for EIP-712 powered permit and permitForAll functions
_tokenURIs	stores individual token URIs
userRoles	special permissions mapping

Table 14. Summary of the mappings in the Token smart contract.

Features and Roles

Features control the behavior of publicly available functions, such as token transfers and burning. In most cases enabling/disabling a feature allows to enable/disable corresponding functionality. Features are stored in userRoles mapping for the smart contract itself userRoles[address(this)].

The table below summarizes the features defined in Token smart contract.

Feature	Bit	Description
TRANSFERS	0	Enables own tokens transfers, transferFrom
TRANSFERS_ON_BEHALF	1	Enables transfers on behalf, transferFrom
OWN_BURNS	3	Enables burning own tokens, burn
BURNS_ON_BEHALF	4	Enables burning on behalf, burn
PERMITS	10	Enables EIP-712 permits, permit
OPERATOR_PERMITS	11	Enables EIP-712 operator permits, permitForAll

Table 15. Summary of the Token smart contract features.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include minting, burning, etc. Roles are stored in userRoles mapping (excluding self address which stores features).

The table below summarizes the roles defined in the Token smart contract.

Role	Bit	Description
TOKEN_CREATOR	16	Allows minting tokens, mint

`TOKEN_DESTROYER`	17	Allows burning any tokens, `burn`
`URI_MANAGER`	21	Allows setting base URI, token URI, `setBaseURI`, `setTokenURI`
`ACCESS_MANAGER`	255	Reserved, defined in `AccessControl` smart contract, allows modifying features and roles

Table 16. Summary of the Token smart contract special permissions (roles).

Public Functions

The table below summarizes public functions designed to transfer tokens between accounts, burn tokens and authorize other addresses to perform these operations on token owner behalf.

Function Name	Description
`transferFrom`	ERC721 transfer, transfers own tokens, or transfers tokens on behalf of address which approved the transfer
`safeTransferFrom`	Same as `transferFrom`, but checks if the receiver is either an EOA or `ERC721TokenReceiver` smart contract; executes `onERC721Received` and validates the response
`approve`	ERC721 approve, approves single token transfer on behalf and burning a single token on behalf of the account which invoked `approve`
`setApprovalForAll`	ERC721 operator approval, approves/revokes token transfers on behalf and burning tokens on behalf of the account which invoked `setApprovalForAll`
`permit`	EIP-712 powered permit, approves single token transfer on behalf and burning a single token on behalf of the account which signed the permit
`permitForAll`	EIP-712 powered operator permit, approves/revokes token transfers on behalf and burning tokens on behalf of the account which signed the operator permit
`burn`	Burns own token, or burns token on behalf of address which approved burning

Table 17. Summary of public functions.

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the Token smart contract.

Function Name	Description
`mint`	Mints single token
`safeMint`	Same as `mint`, but checks if the receiver is either an EOA or `ERC721TokenReceiver` smart contract; executes `onERC721Received` and validates the response
`mintBatch`	Mints several tokens in a row
`safeMintBatch`	Same as `mintBatch`, but checks if the receiver is either an EOA or `ERC721TokenReceiver` smart contract; executes `onERC721Received` for every token minted and validates the response
`burn`	Burns tokens on behalf of any address, independently of if it approved operation or not
`setBaseURI`	Sets the base URI used to construct token URIs for the tokens without URIs explicitly set
`setTokenURI`	Sets individual token URI

Table 18. Summary of the functions with restricted access.

Note that Token smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

Personality Pod (ERC721 Token)

Personality Pod, also known as AI Personality, AI Persona, Persona, or Pod, comes in numerous traits of different rarities, but the information on traits is stored off-chain.

Personality Pod is Tiny ERC721, token ID space is 32 bits, Tiny ERC721 requirements and implementation design applies.

Token Summary

1. Symbol: POD
2. Name: iNFT Personality Pod
3. Decimals: 0 (NFT)
4. Initial total supply (minted on sale): up to 10,000 POD
5. Mintable: new tokens may get created
6. Non-burnable[14]: tokens cannot be destroyed

[14] Burning capability is revoked after token smart contract is deployed

Alethea NFT (ERC721 Token)

Alethea NFT represents pieces of digital art capable of binding with Personality Pod and form iNFTs.

Alethea NFT is Tiny ERC721, token ID space is 32 bits, Tiny ERC721 requirements and implementation design applies.

Alethea NFT is a generic contract used to deploy NFT collections, such as Revenants, SophiaBeing collections.

Token Summary: Revenants

1. Symbol: REV
2. Name: Revenants by Alethea AI
3. Decimals: 0 (NFT)
4. Initial total supply (minted on sale): 100 REV
5. Mintable: new tokens may get created
6. Non-burnable[15]: tokens cannot be destroyed

Token Summary: SophiaBeing

1. Symbol: SOP
2. Name: Sophia beingAI iNFT
3. Decimals: 0 (NFT)
4. Initial total supply (minted on sale): 100 SOP
5. Mintable: new tokens may get created
6. Non-burnable[16]: tokens cannot be destroyed

Token Summary: "Arkive" Mystery Box

1. Symbol: ASSET
2. Name: iNFT Assets by Alethea AI
3. Decimals: 0 (NFT)
4. Initial total supply (minted on sale): 10,000 ASSET
5. Mintable: new tokens may get created
6. Non-burnable[17]: tokens cannot be destroyed

[15] Burning capability is revoked after token smart contract is deployed
[16] Burning capability is revoked after token smart contract is deployed
[17] Burning capability is revoked after token smart contract is deployed

iNFT

iNFT smart contract keeps records in the distributed ledger which enrich an arbitrary NFT with the AI personality – Personality Pod ERC721 token storing the data used to feed GPT-3 algorithm.

Data Structure

For every iNFT binding (record) created, following data is stored on-chain:

1. Locked Personality
    a. Smart Contract Address
    b. Persona ID
2. Locked ALI Token Value
3. Target NFT
    a. Smart Contract Address
    b. Token ID

The data structure is called `IntelliBinding` in the smart contract code.

Functional Requirements

1. Authorized address(es) should be able to create iNFTs; upon creation iNFT:
    a. Binds to a “target” NFT, i.e. keeps track of the target NFT; target NFT owner becomes iNFT owner
    b. Locks Personality Pod
    c. May lock some amount of ALI tokens
2. Authorized address(es) should be able to destroy iNFTs; upon destruction iNFT:
    a. Unbinds from the target NFT, i.e. removes target NFT tracking
    b. Unlocks Personality Pod, transfers it to the target NFT owner
    c. Unlocks ALI tokens which were previously locked upon creation, transfers these tokens to the target NFT owners
3. iNFT has an owner which is defined as a target NFT owner
    a. iNFT owner changes when target NFT owner changes
    b. iNFT owner effectively owns locked Personality Pod and ALI tokens
    c. iNFT owner cannot transfer locked Personality Pod and ALI tokens
    d. iNFT owner can transfer Personality Pod and ALI tokens after iNFT is destroyed and Personality Pod and ALI tokens are unlocked
4. iNFT smart contract keeps track of total iNFTs amount created, which is
    a. Increased when iNFT is created
    b. Decreased when iNFT gets destroyed
    c. May not be equal to the total effective amount of iNFTs in existence if target NFTs are burnt
5. One-to-one mappings only:
    a. One and only one target NFT can be bound to an iNFT, one and only one Personality Pod can be locked in a single iNFT

b. One and only one iNFT can be bound to a target NFT, one and only one iNFT can lock a single Personality Pod

Implementation Overview

iNFT has some interfaces similar to ERC721. Table below summarizes these interfaces.

Function	Value/Description
name()	Intelligent NFT
symbol()	iNFT
tokenURI(uint256)	URL of the particular iNFT, doesn't delegate to target NFT
totalSupply()	Minted iNFT counter
exists(uint256)	Checks if iNFT record exists, doesn't delegate to target NFT
ownerOf(uint256)	Delegates to target NFT ERC721 ownerOf(uint256)

Table 19. Summary of iNFT interface functions which are similar to ERC721 functions.

Similar to Alethea ERC20 Token implementation, iNFT smart contract implementation is based on the custom AccessControl (see Access Control section for details).

Minting Flow

When minting, iNFT smart contract performs the following actions:

1. State validation and input verification
    a. Check: tx sender is authorized to mint
    b. Check: Personality Pod smart contract specified is ERC721
    c. Check: target NFT smart contract specified is ERC721
    d. Check: iNFT specified doesn't yet exist
    e. Check: target NFT is not yet bound
    f. Check: Personality Pod is not yet linked
    g. Check: Personality Pod ownership is already transferred to iNFT smart contract
    h. Check: target NFT exists
    i. Check: ALI value to be locked doesn't overflow total amount of ALI tokens on the iNFT balance (obligations to iNFT owners are enforceable)
2. Increase total ALI obligations counter
3. Create iNFT binding
4. Create reverse and personality bindings
5. Increase total supply

Burning Flow

When burning, iNFT smart contract performs the following actions:

1. State validation and input verification
    a. Check: tx sender is authorized to burn
    b. Check: binding exists
    c. Check: target NFT exists
2. Decrease total supply
3. Delete iNFT binding
4. Delete reverse and personality bindings
5. Transfer locked Personality Pod to iNFT owner
6. Decrease total ALI obligations counter
7. Transfer locked ALI tokens to iNFT owner

Implementation Details

This section describes the iNFT smart contract implementation details.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the iNFT smart contract:

Solidity Name	Description
`name` `* name()`	Descriptive name for the iNFT collection represented by iNFT smart contract, immutable, “Intelligent NFT”
`symbol` `* symbol()`	Abbreviated name for the iNFT collection represented by iNFT smart contract, immutable, “iNFT”
`baseURI` `* baseURI()`	Used to construct `tokenURI(uint256)` for iNFTs which don’t have their URI explicitly set, mutable, can be updated with `setTokenURI(uint256, string)` function
`totalSupply` `* totalSupply()`	Keeps track of the amount if iNFTs created by the iNFT smart contract, increased on mint, decreased on burn
`aliContract`	ALI ERC20 Token smart contract address, immutable, set upon the deployment
`aliBalance`	Cumulative amount of ALI tokens locked within the iNFT smart contract; sum of all individual ALI obligations to iNFT owners; iNFT ALI balance cannot be lower than this value

Table 20. Summary of the state variables in the iNFT smart contract.

Mappings

Mappings store special permissions, binding information, and auxiliary info. Following mappings are defined on the iNFT smart contract:

Solidity Name	Description
`bindings`	stores iNFT binding information, see `IntelliBinding`
`reverseBindings`	stores iNFT IDs for target NFTs
`personalityBindings`	stores iNFT IDs for locked Personality Pods
`_tokenURIs`	stores individual iNFT URIs
`userRoles`	special permissions mapping

Table 21. Summary of the mappings in the iNFT smart contract.

Features and Roles

iNFT doesn't have any public functions, and therefore doesn't introduce any features controlling them. iNFT mutable functions are accessed via iNFT Linker, Sale, and other helper smart contracts.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include minting, burning, etc. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the iNFT smart contract.

Role	Bit	Description
`MINTER`	16	Allows creating iNFTs, `mint`, `mintBatch`
`BURNER`	17	Allows destroying iNFTs, `burn`
`URI_MANAGER`	21	Allows setting base URI, token URI, `setBaseURI`, `setTokenURI`
`ACCESS_MANAGER`	255	Reserved, defined in `AccessControl` smart contract, allows modifying features and roles

Table 22. Summary of the iNFT smart contract special permissions (roles).

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the iNFT smart contract.

Function Name	Description
`mint`	Creates single iNFT

`mintBatch`	Creates several iNFTs in a row
`burn`	Destroys single iNFT
`setBaseURI`	Sets the base URI used to construct iNFT URIs for the iNFTs without URIs explicitly set
`setTokenURI`	Sets individual iNFT URI

Table 23. Summary of the functions with restricted access.

Note that iNFT smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

iNFT Linker

iNFT Linker, also known as IntelliLinker, Linker, iNFT Linker V1, or Linker V1, is a helper smart contract responsible for creating and destroying iNFTs, handling iNFT creation fees, and determining target NFT eligibility to be bound to iNFT.

Fusion

Fusion, also known as iNFT Linking, Binding, or Linking is a process of enriching an NFT with AI Personality. We call the NFT to be enriched, a target NFT.

In its essence, Fusion moves an AI Pod from the user's wallet into the iNFT smart contract and binds the AI Pod to the target NFT by creating the iNFT record, effectively “creating” (or minting) an iNFT.

Optionally, fusion also requires some amount of ALI tokens to be moved and locked in the same iNFT smart contract.

The process is reversible: target NFT owner can “destroy” (burn) the iNFT, effectively releasing the AI Pod from iNFT smart contract and transferring it back[18] to the target NFT owner.

In order for the fusion to happen, the user interacts with the helper smart contract named IntelliLinker. IntelliLinker takes an AI Pod from the user, and (optionally) some ALI tokens.

Thus, in order to perform these operations (on user’s behalf), the IntelliLinker requires a user to explicitly grant the permissions to take these assets. Moreover, if an AI Pod is staked, it needs to be unstaked first, since technically, staked AI Pod doesn’t belong to a user, but to a staking smart contract.

[18] Note: target NFT owner may change after the fusion and new owner would receive the AI Pod when unfusing

Functional Requirements

1. Linking is a process of transferring Personality Pod, and, optionally, ALI tokens from a user to iNFT smart contract, and creating a binding record with a target NFT; transferred items are locked within iNFT
2. Unlinking is a process of transferring previously locked Personality Pod, and, optionally, ALI tokens from iNFT smart contract to iNFT owner, and destroying the binding record
3. It should be possible to link Personality Pod, and, optionally, ALI tokens, to the target NFT, effectively creating an iNFT
4. Authorized address(es) should be able to set linking price, fee (ALI), and treasury address on the linker; if set,
    a. linking price is charged from the user who links iNFT
    b. linking fee is sent to the treasury address (configurable)
    c. linking price amount exceeding the fee is locked within iNFT
5. Linking fee cannot exceed linking price
    a. If linking fee is equal to linking price, all the ALI tokens charged are transferred to the treasury, no tokens are locked in the iNFT created
6. Authorized address(es) should be able to restrict target NFT smart contract addresses supported for linking with iNFT
7. It should be possible to unlink Personality Pod, and, optionally, ALI tokens, from the target NFT, effectively destroying an iNFT

Non-functional Requirements

iNFT Linker may create iNFTs with IDs starting from 2^{32}. Smaller IDs are reserved for the Fixed Supply Sale (see below in Fixed Supply Sale section).

Implementation Overview

Similar to Alethea ERC20 Token implementation, Linker smart contract implementation is based on the custom AccessControl (see Access Control section for details).

Although linker's link/unlink functions are both controlled by the feature flags and can be disabled, unlink function is designed to be publicly accessible at any time, its feature flag should be enabled and permission to disable it should be revoked forever once the protocol is live.

Linker keeps track of the iNFT IDs available to be used by storing the next free iNFT ID and automatically incrementing it. Next free iNFT ID can be updated manually if required.

Linking Flow

When linking, the Linker performs the following actions:

1. State validation and input verification
    a. Check: linking is enabled
    b. Check: sender owns Personality Pod supplied

    c. Check: target NFT contract address is whitelisted
2. Send linking fee to the treasury (if fee is set)
3. Send the linking price amount exceeding the fee to the iNFT smart contract
4. Send Personality Pod to the iNFT smart contract
5. Create iNFT
6. Increment next free iNFT ID

Unlinking Flow

When unlinking, the Linker performs the following actions:

1. State validation and input verification
    a. Check: unlinking is enabled
    b. Check: tx sender is iNFT owner
2. Destroy iNFT

Implementation Details

This section describes the iNFT Linker smart contract implementation details.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the Linker smart contract:

Solidity Name	Description
`linkPrice`	Amount of ALI tokens charged when creating an iNFT
`linkFee`	Amount of ALI tokens sent to the treasury
`feeDestination`	Treasury address
`nextId`	Next free iNFT ID to be created, automatically incremented when iNFT is created
`aliContract`	ALI ERC20 Token smart contract address, immutable, set upon the deployment
`personalityContract`	Personality Pod smart contract address, immutable, set upon the deployment
`iNftContract`	iNFT smart contract address, immutable, set upon the deployment

Table 24. Summary of the state variables in the Linker smart contract.

Mappings

Mappings store special permissions, and auxiliary info. Following mappings are defined on the Linker smart contract:

Solidity Name	Description
`whitelistedTargetContracts`	stores target NFT smart contract addresses allowed to be bound to when creating iNFTs
`userRoles`	special permissions mapping

Table 25. Summary of the mappings in the Linker smart contract.

Features and Roles

Features control the behavior of publicly available functions, such as linking, and unlinking. In most cases enabling/disabling a feature allows to enable/disable corresponding functionality. Features are stored in `userRoles` mapping for the smart contract itself `userRoles[address(this)]`.

The table below summarizes the features defined in Linker smart contract.

Feature	Bit	Description
`LINKING_ENABLED`	0	Enables linking, `link`
`UNLINKING_ENABLED`	1	Enables unlinking, `unlink`, `unlinkNFT`
`ALLOW_ANY_NFT_CONTRACT`	2	Allows binding to any NFT independently if it is whitelisted in `whitelistedTargetContracts` or not

Table 26. Summary of the Linker smart contract features.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include setting link price, fee, whitelisting target NFT contracts, etc. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the Linker smart contract.

Role	Bit	Description
`LINK_PRICE_MANAGER`	16	Allows setting link price, fee, treasury address, `updateLinkPrice`
`NEXT_ID_MANAGER`	17	Allows updating next free iNFT ID, `updateNextId`
`WHITELIST_MANAGER`	18	Allows whitelisting target NFT smart contracts, `whitelistTargetContract`

`ACCESS_MANAGER`	255	Reserved, defined in `AccessControl` smart contract, allows modifying features and roles

Table 27. Summary of the Linker smart contract special permissions (roles).

Public Functions

The table below summarizes public functions designed to link/unlink iNFTs.

Function Name	Description
`link`	Creates iNFT
`unlink`	Destroys iNFT
`unlinkNFT`	Destroys iNFT bound to specific NFT

Table 28. Summary of public functions.

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the Linker smart contract.

Function Name	Description
`updateLinkPrice`	Updates link price, fee, and treasury address
`updateNextId`	Updates next free iNFT ID
`whitelistTargetContract`	Whitelists or delists target NFT smart contract

Table 29. Summary of the functions with restricted access.

Note that Linker smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

iNFT Linker V2

iNFT Linker V2, also known as IntelliLinkerV2, or Linker V2 is an upgradeable version of iNFT Linker. Its initial release also adds one additional functional requirement allowing it to restrict target NFT contracts which can be unlinked.

Linker V2 inherits Linker V1 functional and non-functional requirements, architecture, and implementation basics, adding only little difference, highlighted below.

Functional Requirements

1. Linking is a process of transferring Personality Pod, and, optionally, ALI tokens from a user to iNFT smart contract, and creating a binding record with a target NFT; transferred items are locked within iNFT
2. Unlinking is a process of transferring previously locked Personality Pod, and, optionally, ALI tokens from iNFT smart contract to iNFT owner, and destroying the binding record
3. It should be possible to link Personality Pod, and, optionally, ALI tokens, to the target NFT, effectively creating an iNFT
4. Authorized address(es) should be able to set linking price, fee (ALI), and treasury address on the linker; if set,
    a. linking price is charged from the user who links iNFT
    b. linking fee is sent to the treasury address (configurable)
    c. linking price amount exceeding the fee is locked within iNFT
5. Linking fee cannot exceed linking price
    a. If linking fee is equal to linking price, all the ALI tokens charged are transferred to the treasury, no tokens are locked in the iNFT created
6. Authorized address(es) should be able to restrict target NFT smart contract addresses supported for linking with iNFT
7. **Authorized address(es) should be able to restrict target NFT smart contract addresses supported for unlinking from iNFT**
8. It should be possible to unlink Personality Pod, and, optionally, ALI tokens, from the target NFT, effectively destroying an iNFT

Non-functional Requirements

By default iNFT Linker V2 may create iNFTs with IDs starting from 2^{33}. Smaller IDs are reserved for the iNFT Linker V1 and Fixed Supply Sale (see below in Fixed Supply Sale section). It is possible, however, to reassign `nextId` field after deployment to inherit Linker V1 iNFT ID space which starts from 2^{32}.

Implementation Overview

Linker V2 smart contract implementation is based on the custom Upgradable AccessControl (see Access Control section and Upgradeability subsection for details). This is different from Linnker V1, which is non-upgradeable.

Same as Linker V1, Linker V2 keeps track of the iNFT IDs available to be used by storing the next free iNFT ID and automatically incrementing it. Next free iNFT ID can be updated manually if required.

Linking Flow

When linking, the Linker V2 performs same set of operations as Linker V1:

1. State validation and input verification

    a. Check: linking is enabled
    b. Check: sender owns Personality Pod supplied
    c. Check: target NFT contract address is whitelisted
2. Send linking fee to the treasury (if fee is set)
3. Send the linking price amount exceeding the fee to the iNFT smart contract
4. Send Personality Pod to the iNFT smart contract
5. Create iNFT
6. Increment next free iNFT ID

Unlinking Flow

When unlinking, the Linker V2 performs the following actions (note an additional state check):

1. State validation and input verification
    a. Check: unlinking is enabled
    b. Check: tx sender is iNFT owner
    c. Check: target NFT contract address is whitelisted
2. Destroy iNFT

How to Create an iNFT

1. Make sure you have an AI Pod you're going to fuse into iNFT in your wallet.
    a. If your AI Pod is staked, unustake it.
    b. If you don't have one – you can buy it on OpenSea.
    c. Always check the AI Pod contract/collection address in Ethereum mainnet: `0xDd70AF84BA86F29bf437756B655110D134b5651C`
2. (Optional) Make sure you have ALI tokens required for fusion in your wallet.
    a. Initial version of the protocol doesn't require any ALI tokens and this step can be skipped.
    b. You can buy ALI tokens on Uniswap
    c. Always check the ALI token address in Ethereum mainnet: `0x6B0b3a982b4634aC68dD83a4DBF02311cE324181`
3. Note that fusion binds AI Pod to any NFT, independently of the target NFT ownership. Whoever owns the target NFT owns the attached AI Pod. If you're not gifting the AI Pod, make sure you own the target NFT you're fusing with.
4. Note that fusion may be reversible or irreversible. It depends on the target NFT collection (defined by its smart contract address). If you plan to sell the AI Pod in the future, bind it to some other NFT or use it in some other way, make sure the target NFT contract you're binding with, supports unlinking.
    a. Call `isAllowedForLinking` view function on the `IntelliLinker` contract to check if target contract can be linked with
    b. Call `isAllowedForUnlinking` view function on the `IntelliLinker` contract to check if fusion is reversible
    c. Following target NFT collections are at the moment of writing:

   i. FameLadySquad, `0xf3E6DbBE461C6fa492CeA7Cb1f5C5eA660EB1B47`, linking, unlinking
   ii. The Doge Pound, `0xF4ee95274741437636e748DdAc70818B4ED7d043`, linking, unlinking
   iii. PudgyPenguins, `0xBd3531dA5CF5857e7CfAA92426877b022e612cf8`, linking, unlinking
   iv. UninterestedUnicorns, `0xC4a0b1E7AA137ADA8b2F911A501638088DFdD508`, linking, unlinking
5. Grant the `IntelliLinker` a permission to access (technically speaking, take away) your AI Pod.
   a. Use `approve` function on the `PersonalityPodERC721` to grant access for a single AI Pod
   b. Use `setApprovalForAll` function on the `PersonalityPodERC721` to grant access for all your AI Pods (can be useful when you fuse multiple iNFTs)
6. Bind the AI Pod to the target NFT using the `link` function on the `IntelliLinker` contract. The AI Pod will be moved from your wallet into the `IntelligentNFTv2` contract.
   a. Read the `Minted` and `Linked` events on the link transaction to get linked iNFT data
   b. Call `bindings`, `reverseBindings`, and `personalityBindings` mappings on the `IntelligentNFTv2` contract to read linked iNFT data
   c. Use `unlink` function on the `IntelliLinker` contract to unfuse the iNFT (if operation is supported)

Implementation Details

Linker V2 inherits the same state variables, mappings, features, roles, public and restricted functions as Linker V1.

There is slight difference in `whitelistedTargetContracts` mapping which stores both linking and unlinking whitelist values as integer (instead of storing only one as boolean):

1. Lowest bit (zero) stores linking whitelist flag
2. Next bit (one) stores unlinking whitelist flag

The signature for `whitelistTargetContract` restricted function is slightly changed and accepts two boolean flags instead of one:

1. Linking whitelist flag (`allowedForLinking`)
2. Unlinking whitelist flag (`allowedForUnlinking`)

These boolean flags are packed into an integer value and stored in `whitelistedTargetContracts` mapping as integers.

iNFT Linker V3: Custom iNFT Feature

iNFT Linker V3, also known as IntelliLinkerV3, or Linker V3 implements a “Custom iNFT Feature", switching the usage model from strict (only whitelisted ERC721 contracts can be used to create iNFTs) to permissive (only blacklisted ERC721 contracts cannot be used to create iNFTs).

Linker V3 inherits Linker V1 and V2 functional and non-functional requirements, architecture, and implementation basics, adding only little difference, highlighted below.

Functional Requirements

1. Linking is a process of transferring Personality Pod, and, optionally, ALI tokens from a user to iNFT smart contract, and creating a binding record with a target NFT; transferred items are locked within iNFT
2. Unlinking is a process of transferring previously locked Personality Pod, and, optionally, ALI tokens from iNFT smart contract to iNFT owner, and destroying the binding record
3. It should be possible to link Personality Pod, and, optionally, ALI tokens, to the target NFT, effectively creating an iNFT
4. Authorized address(es) should be able to set linking price, fee (ALI), and treasury address on the linker; if set,
    a. linking price is charged from the user who links iNFT
    b. linking fee is sent to the treasury address (configurable)
    c. linking price amount exceeding the fee is locked within iNFT
5. Linking fee cannot exceed linking price
    a. If linking fee is equal to linking price, all the ALI tokens charged are transferred to the treasury, no tokens are locked in the iNFT created
6. **Authorized address(es) should be able to switch the linker between strict and permissive modes**
    a. **Strict: only target NFT contracts explicitly whitelisted are allowed**
    b. **Permissive: all target NFT contracts except for explicitly blacklisted are allowed**
7. Authorized address(es) should be able to restrict target NFT smart contract addresses supported for linking with iNFT
8. Authorized address(es) should be able to restrict target NFT smart contract addresses supported for unlinking from iNFT
9. It should be possible to unlink Personality Pod, and, optionally, ALI tokens, from the target NFT, effectively destroying an iNFT

Non-functional Requirements

iNFT Linker V3 inherits iNFTs IDs from iNFT Liner V2. iNFT Linker V3 must not be deployed stand-alone, but only as an upgrade on top of iNFT Liner V2. It should reuse the same storage.

Implementation Overview

Linker V2 smart contract implementation is based on the custom Upgradable AccessControl (see Access Control section and Upgradeability subsection for details). This is different from Linnker V1, which is non-upgradeable.

Same as Linker V1, Linker V2 keeps track of the iNFT IDs available to be used by storing the next free iNFT ID and automatically incrementing it. Next free iNFT ID can be updated manually if required.

Linking Flow

When linking, the Linker V2 performs same set of operations as Linker V1:

1. State validation and input verification
    a. Check: linking is enabled
    b. Check: sender owns Personality Pod supplied
    c. Check: target NFT contract address is whitelisted
        i. **Or any target NFT contract is allowed globally**
    d. Check: target NFT contract address is not blacklisted
2. Send linking fee to the treasury (if fee is set)
3. Send the linking price amount exceeding the fee to the iNFT smart contract
4. Send Personality Pod to the iNFT smart contract
5. Create iNFT
6. Increment next free iNFT ID

Unlinking Flow

When unlinking, the Linker V2 performs the following actions (note an additional state check):

1. State validation and input verification
    a. Check: unlinking is enabled
    b. Check: tx sender is iNFT owner
    c. Check: target NFT contract address is whitelisted
        i. Or any target NFT contract is allowed globally
    d. Check: target NFT contract address is not blacklisted
2. Destroy iNFT

How to Create an iNFT

1. Make sure you have an AI Pod you’re going to fuse into iNFT in your wallet.
    a. If your AI Pod is staked, unustake it.
    b. If you don’t have one – you can buy it on OpenSea.

   c. Always check the AI Pod contract/collection address in Ethereum mainnet: `0xDd70AF84BA86F29bf437756B655110D134b5651C`
2. (Optional) Make sure you have ALI tokens required for fusion in your wallet.
   a. Initial version of the protocol doesn't require any ALI tokens and this step can be skipped.
   b. You can buy ALI tokens on Uniswap
   c. Always check the ALI token address in Ethereum mainnet: `0x6B0b3a982b4634aC68dD83a4DBF02311cE324181`
3. Note that fusion binds AI Pod to any NFT, independently of the target NFT ownership. Whoever owns the target NFT owns the attached AI Pod. If you're not gifting the AI Pod, make sure you own the target NFT you're fusing with.
4. Note that fusion may be reversible or irreversible. It depends on the target NFT collection (defined by its smart contract address). If you plan to sell the AI Pod in the future, bind it to some other NFT or use it in some other way, make sure the target NFT contract you're binding with, supports unlinking.
   a. Call `isAllowedForLinking` view function on the `IntelliLinker` contract to check if target contract can be linked with
   b. Call `isAllowedForUnlinking` view function on the `IntelliLinker` contract to check if fusion is reversible
   c. Following target NFT collections are disabled for unlinking at the moment of writing:
      i. **The Revenants, `0xc2D6B32E533e7A8dA404aBb13790a5a2F606aD75`**
      ii. **Sophia Being, `0x75c804fFb01b16B7592a0B9644835244E2140728`**
5. Grant the `IntelliLinker` a permission to access (technically speaking, take away) your AI Pod.
   a. Use `approve` function on the `PersonalityPodERC721` to grant access for a single AI Pod
   b. Use `setApprovalForAll` function on the `PersonalityPodERC721` to grant access for all your AI Pods (can be useful when you fuse multiple iNFTs)
6. Bind the AI Pod to the target NFT using the `link` function on the `IntelliLinker` contract. The AI Pod will be moved from your wallet into the `IntelligentNFTv2` contract.
   a. Read the `Minted` and `Linked` events on the link transaction to get linked iNFT data
   b. Call `bindings`, `reverseBindings`, and `personalityBindings` mappings on the `IntelligentNFTv2` contract to read linked iNFT data
   c. Use `unlink` function on the `IntelliLinker` contract to unfuse the iNFT (if operation is supported)

Implementation Details

Linker V3 inherits the same state variables, mappings, features, roles, public and restricted functions as Linker V2.

There is slight difference in `whitelistedTargetContracts` mapping which stores **linking / unlinking whitelist, and linking / unlinking blacklist** values as integer:

1. Lowest bit (zero) stores linking whitelist flag
2. Next bit (one) stores unlinking whitelist flag
3. **Next bit (two) stores linking blacklist flag**
4. **Next bit (three) stores unlinking blacklist flag**

The signature for `whitelistTargetContract` restricted function is slightly changed and accepts four boolean flags instead of two:

1. Linking whitelist flag (`allowedForLinking`)
2. Unlinking whitelist flag (`allowedForUnlinking`)
3. **Linking blacklist flag (`forbiddenForLinking`)**
4. **Unlinking blacklist flag (`forbiddenForUnlinking`)**

These boolean flags are packed into an integer value and stored in `whitelistedTargetContracts` mapping as integers.

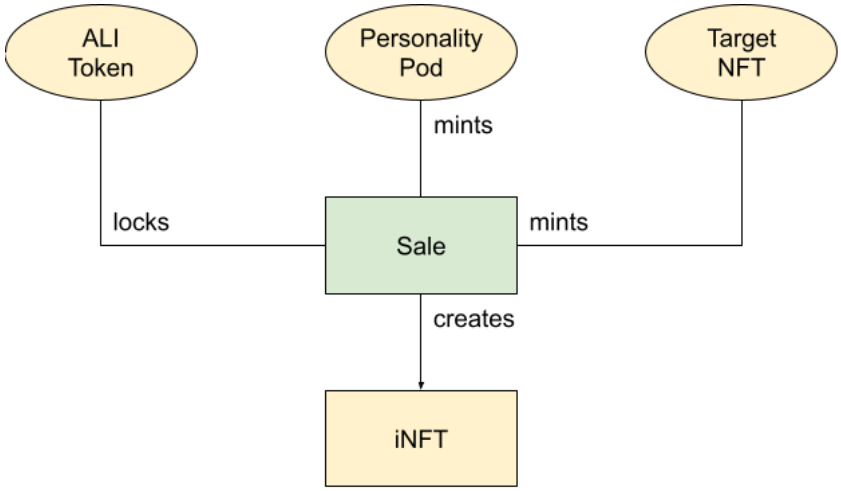
Fixed Supply Sale

Fixed Supply Sale smart contract powers “10,000 sale campaign”, and enables the minting and purchase of 10,000 iNFTs (also known as items on sale, or items). It mints target NFTs, Personality Pods, it can lock ALI tokens, and creates iNFTs. Sale smart contract is reusable, it can be used to run several similar campaigns.

The design has some functional similarity with iNFT Linker. The table and diagram below highlights functional similarities and differences.

Functionality	iNFT Linker	Fixed Supply Sale
Creates iNFTs	Yes	
Destroys iNFTs	Yes	No
Charges	ALI Token	ETH
Mints target NFTs	No	Yes
Mints Personality Pods	No	Yes
Locks ALI Tokens	Yes	

Table 30. Functional comparison of iNFT Linker and Fixed Supply Sale.



Fixed Supply Sale Integration

Diagram 4. Fixed Supply Sale integration with tokens and iNFT. ERC20/ER721 tokens are depicted as ovals, helper smart contracts – as rectangles.

Functional Requirements

1. Authorized address(es) should be able to set up the sale (initialize the sale) with the following mandatory parameters defined:
    a. Item Price, wei – ETH price of a single iNFT on sale
    b. Items on Sale, number – how many iNFT is put on sale (ex.: 10,000)
    c. Sale Duration, start/end dates – time period when items are on sale
2. The sale state[19] is defined as active if:
    a. Item Price is not zero, and
    b. Items Available on Sale is not zero, and
    c. Current time is within Sale Duration
3. Optionally, authorized address(es) should be able to set up the following optional parameters for the sale:
    a. Batch Limit, number – maximum number of items which can be bought in a single transaction
        i. Zero value meaning is “unset”; transaction gas limit and block gas limit are effectively defining the batch limit if it is unset
    b. ALI Value, wei – ALI value to be stored in each iNFT on sale
    c. ALI Source, address – an address where ALI tokens is taken from to be stored in iNFTs sold
        i. Zero value meaning is “unset”; must be unset if and only if ALI Value is zero (unset)
4. It should be possible to buy items on sale

[19] Active sale means the sale in active state; inactive sale means the sale in inactive state

   a. if and only if the sale is active, and
   b. if and only if buyer supplies enough ETH in the transaction (Item Price);
5. It should be possible to buy items in batches (2 or more)
   a. if and only if the sale is active, and
   b. in compliance with the Batch Limit set, and
   c. if and only if buyer supplies enough ETH in the transaction: batch price is a single item price multiplied by batch size, no discounts
6. Buy and buy batch transactions are expected to fail (revert) if the requirements above are not met
   a. The transactions are expected to succeed if ETH supplied exceeds the expected value; the excess amount is returned back to sender
7. ETH obtained from buyers is accumulated on the sale smart contract
   a. Authorized address(es) should be able to withdraw accumulated ETH
8. Authorized address(es) should be able to update sale setup (reinitialize the sale) independently of the sale state
   a. Sale state may change in both directions as a result of reinitialization, or remain the same, for example
   b. Inactive due to duration expiration sale may become active again if duration is extended during reinitialization,
   c. Active sale can be updated with the new sale price and remain active, but selling items at a new price
9. Authorized address(es) should be able to set all the sale parameters or any subset of these parameters in a single transaction
10. When buying single iNFT, the sale is expected to:
    a. Mint target NFT – Alethea NFT – and assign the ownership to the buyer
    b. Mint Personality Pod, and assign the ownership to iNFT smart contract (immediately locked Personality Pod)
    c. If applicable, transfer ALI tokens to be locked to iNFT smart contract
    d. Create an iNFT bound to target NFT and holding locked Personality Pod and ALI tokens (if applicable)
    e. Assign same IDs to minted target NFT, Personality Pod and created iNFT
11. When buying a batch of items, the sale is expected to repeat the steps for buying a single item (see above) n times, where n is a batch size.

Non-functional Requirements

Initialization and Reinitialization

1. Initialization and reinitialization is done with the single function, taking the following parameters as inputs:
   a. Item Price, uint64, $[1,\ 2^{64} - 2]$, maximum price is about 18 ETH
   b. Next ID, uint32, $[1,\ 2^{32} - 2]$
   c. Final ID, uint32, $[1,\ 2^{32} - 2]$

   d. Sale Start, unix timestamp, uint32, $[1, 2^{32} - 2]$
   e. Sale End, unix timestamp, uint32, $[1, 2^{32} - 2]$
   f. Batch Limit, uint32, $[1, 2^{32} - 2]$
   g. ALI Source, address
   h. ALI Value, uint96, $[1, 2^{64} - 2]$, maximum value is about 79 billion ALI
2. "Items on Sale" is derived as Final ID – Next ID + 1. First ID on sale is Next ID, last ID on sale is Final ID.
   a. The sale can sell no more than $2^{32} - 1$ (about 4 billion) items
3. "Sale Duration" is a time interval between Sale Start (inclusive) and Sale End (exclusive)
4. Any parameters can be "skipped" during initialization/reinitialization. "Skipping" means the values remain as were previously set. "Skipping" is achieved using the special value(s) for the parameters to be omitted:
   a. Item Price: value $2^{64} - 1$, `0xFFFFFFFFFFFFFFFF`
   b. Next ID: value $2^{32} - 1$, `0xFFFFFFFF`
   c. Final ID: value $2^{32} - 1$, `0xFFFFFFFF`
   d. Sale Start: value $2^{32} - 1$, `0xFFFFFFFF`
   e. Sale End: value $2^{32} - 1$, `0xFFFFFFFF`
   f. Batch Limit: value $2^{32} - 1$, `0xFFFFFFFF`
   g. ALI Source: value `0xFFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfF`
   h. ALI Value: value $2^{96} - 1$, `0xFFFFFFFFFFFFFFFFFFFFFFFF`

Buying a Batch

Buying a batch is optimized for gas usage, meaning that:

1. Target NFTs are minted in a batch
2. Personality Pods are minted in a batch directly to iNFT smart contract and locked all together
3. Cumulative value of ALI tokens are sent to iNFT smart contract and locked
4. iNFTs are created in a batch
5. Sale smart contract state is updated only once

Implementation Overview

Similar to Alethea ERC20 Token implementation, Sale smart contract implementation is based on the custom AccessControl (see Access Control section for details).

Public functions to buy items are not protected with the feature flags – there is no need for that, as the sale can be inactivated at any time, for example, by setting item price to zero.

Buy Flow

When buying, the Sale performs the following actions:

1. State validation and input verification
    a. Check: recipient is set
    b. Check: amount to buy is correct (not zero, doesn't exceed batch limit)
    c. Check: sale is active
        i. Item price is not zero
        ii. There are items on sale left (Next ID doesn't exceed Final ID)
        iii. Current time lays within sale start/end bound
    d. Check: ETH supplied in tx covers items price
2. Transfer ALI tokens to iNFT smart contract
3. Mint target NFTs to the recipient
4. Mint AI Personalities to iNFT smart contract
5. Create iNFTs
6. Increase Next ID
7. Increase total items sold counter
8. Return the ETH amount exceeding items price back to tx sender

Withdrawal Flow

When withdrawing accumulated ETH from the sale, the Sale performs the following actions:

1. State validation and input verification
    a. Check: tx sender is authorized to withdraw
    b. Check: recipient address is set
    c. Check: there is ETH to withdraw
2. Transfer accumulated ETH to the recipient specified

Implementation Details

This section describes the Sale smart contract implementation details.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the Sale smart contract:

Solidity Name	Description
`itemPrice`	Single iNFT price
`nextId`	Next free iNFT ID to be created, automatically incremented when iNFT is created
`finalId`	Last iNFT ID to be created

`saleStart`	Sale start unix timestamp
`saleEnd`	Sale end unix timestamp
`batchLimit`	Batch limit size
`soldCounter`	Total items increased counter, increases when iNFT is sold, cannot decrease
`aliContract`	ALI ERC20 Token smart contract address, immutable, set upon the deployment
`nftContract`	Alethea NFT smart contract address, immutable, set upon the deployment
`personalityContract`	Personality Pod smart contract address, immutable, set upon the deployment
`iNftContract`	iNFT smart contract address, immutable, set upon the deployment
`aliSource`	Address of the wallet holding ALI tokens reserve
`aliValue`	ALI value to transfer from the wallet to be locked in iNFT

Table 31. Summary of the state variables in the Sale smart contract.

Mappings

Only special permissions mapping is defined for the Sale smart contract, `userRoles`, it stores special permissions of smart contract administrators and helpers. The mapping is a part of Access Control and is inherited by the Token smart contract from Access Control smart contract. See Special Permissions Mapping subsection of the Access Control section for details.

Features and Roles

Sale doesn't introduce any features controlling the public functions.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include initialization/reinitialization, ETH withdrawal, etc. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the Sale smart contract.

Role	Bit	Description
`SALE_MANAGER`	16	Allows sale initialization and reinitialization, `initialize`

`WITHDRAWAL_MANAGER`	17	Allows withdrawal of the accumulated ETH, `withdraw`, `withdrawTo`
`ACCESS_MANAGER`	255	Reserved, defined in `AccessControl` smart contract, allows modifying features and roles

Table 32. Summary of the Sale smart contract special permissions (roles).

Public Functions

The table below summarizes public functions designed to buy iNFTs.

Function Name	Description
`buySingle`	Sells single iNFT to tx sender
`buySingleTo`	Sells single iNFT to the recipient specified
`buy`	Sells several iNFTs to tx sender
`buyTo`	Sells several iNFTs to the recipient specified

Table 33. Summary of public functions.

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the Sale smart contract.

Function Name	Description
`initialize`	Updates sale parameters, may change sale state
`withdraw`	Withdraws accumulated ETH to tx sender
`withdrawTo`	Withdraws accumulated ETH to the recipient specified

Table 34. Summary of the functions with restricted access.

Note that Sale smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

Mintable Sale

Mintable Sale smart contract enables minting and purchase of individual tokens. The design is similar to Fixed Supply Sale. The only difference is that it just mints an NFT or batch of NFTs of a single type and doesn’t link them into iNFT(s).

Mintable Sale can be deployed to mint Personality Pods, Revenants, and other Alethea NFTs.

In comparison to Fixed Supply Sale, Mintable Sale doesn't use and doesn't store `aliContract`, `nftContract`, `personalityContract`, `iNftContract`, `aliSource`, and `aliValue` (see Implementation Details subsection under Fixed Supply Sale). It stores only `tokenContract` instead – Mintable ERC721 contract address used to mint NFTs.

Mintable Sale has exactly the same settings and user flows as Fixed Supply Sale.

NFT Airdrop Helper

NFT Airdrop Helper, also known as Airdrop Helper, NFT Airdrop, NFT Drop, or ERC721 Drop, is a helper smart contract allowing an airdrop for the arbitrary ERC721 contract via minting. The idea is to optimize gas costs for the party which performs an airdrop (NFT owner). Helper contract, which is allowed to mint arbitrary NFTs on the target contract, holds the list of NFTs to be minted and allows minting them by the public.

Functional Requirements

1. NFT Airdrop Helper operates on the single ERC721 contract (defined on deployment), which implements a `MintableERC721` interface and has a `safeMint(address _to, uint256 _tokenId)` function
2. Airdrop Helper allows minting of the predefined list of tokens (tokens list), which is a token IDs mapping to the predefined addresses, where each predefined token ID has a correspondent predefined token owner address:
$$token\ ID \Rightarrow token\ owner\ address$$
3. When an airdrop is enabled and tokens list is defined, it should be possible for any address to redeem a token from the predefined list to the address defined in this list
4. Authorized address(es) should be able to enable/disable an airdrop
5. Authorized address(es) should be able to define the list of tokens for an airdrop

Non-functional Requirements

1. Tokens list for an airdrop is stored off-chain
2. Airdrop helper stores Merkle root of the token list Merkle tree structure
3. When redeeming a token, it's corresponded list entry ($token\ ID \Rightarrow token\ owner\ address$) must be supplied along with the Merkle proof for that entry

Constructing the Merkle Proof

When redeeming a particular token, Merkle proof needs to be constructed and supplied together with the token ID and owner address.
Merkle tree and proof can be constructed using the `web3-utils`, `merkletreejs`, and `keccak256` npm packages:

1. Hash the token list elements via `web3.utils.soliditySha3`, making sure the correctness of packing order and data types
2. Create a sorted MerkleTree (`merkletreejs`) from the hashed collection, use `keccak256` from the `keccak256` npm package as a hashing function, do not hash leaves (already hashed in step 1); Ex. MerkleTree options: `{hashLeaves: false, sortPairs: true}`
3. For any given token the proof is constructed by hashing it (as in step 1), and querying the MerkleTree for a proof, providing the hashed token element as a leaf

Refer to the Solidity source code and JavaScript test cases for the code examples on how the proof is verified and how it should be generated.

Implementation Overview

Similar to Alethea ERC20 Token implementation, Airdrop Helper smart contract implementation is based on the custom AccessControl (see Access Control section for details).

All smart contract functions can be enabled/disabled either via feature flags (for public functions) or by granting/revoking corresponding roles (restricted functions).

Setup Flow (Restricted)

To set up the Airdrop following actions must be executed:

1. Grant Airdrop Helper a permission to mint on the target NFT contract
2. Construct the MerkleTree as mentioned in the Constructing the Merkle Proof section
3. Set the Merkle root on the Airdrop contract

Redeem Flow (Public)

To redeem the NFT following actions must be executed:

1. Construct the MerkleTree as mentioned in the Constructing the Merkle Proof section
2. Construct the Merkle proof for a token of interest, as described in the Constructing the Merkle Proof section
3. Supply token ID, address and it's Merkle proof to the Airdrop Helper to redeem the token

Implementation Details

This section describes the Airdrop Helper smart contract implementation details.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the Airdrop Helper smart contract:

Solidity Name	Description

`root`	Merkle tree root for the tokens list: (address, tokenId) pairs
`targetContract`	Mintable ERC721 contract address to mint tokens of

Table 35. Summary of the state variables in the Airdrop Helper smart contract.

Features and Roles

Features control the behavior of publicly available functions, such as redeeming. In most cases enabling/disabling a feature allows to enable/disable corresponding functionality. Features are stored in `userRoles` mapping for the smart contract itself `userRoles[address(this)]`.

The table below summarizes the features defined in Airdrop Helper smart contract.

Feature	Bit	Description
`REDEEM_ACTIVE`	0	Enables redeeming of the NFTs, `redeem`

Table 36. Summary of the Airdrop Helper smart contract features.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include setting the list of the tokens to redeem (via its Merkle root), etc. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the Airdrop Helper smart contract.

Role	Bit	Description
`DATA_MANAGER`	16	Allows setting the list of the tokens to redeem, `setInputDataRoot`
`ACCESS_MANAGER`	255	Reserved, defined in `AccessControl` smart contract, allows modifying features and roles

Table 37. Summary of the Airdrop Helper smart contract special permissions (roles).

Public Functions

The table below summarizes public functions designed to redeem NFTs.

Function Name	Description
`redeem`	Redeems NFT (mints it on the target NFT contract)

Table 38. Summary of public functions.

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the Airdrop Helper smart contract.

Function Name	Description
`setInputDataRoot`	Updates token list Merkle tree root

Table 39. Summary of the functions with restricted access.

Note that Airdrop Helper smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

NFT Staking Helper

NFT Staking Helper, also known as Staking Helper, or NFT Staking, enables NFT owners to stake their NFTs to get rewards. NFT Staking operates on a single NFT contract and records when a particular NFT was staked and when it was unstaked. NFT Staking doesn't distribute the rewards, it just tracks the data which can be used to distribute these rewards.

Functional Requirements

1. NFT Staking Helper operates on the single ERC721 contract, defined during the deployment
2. When staking is enabled, token holders should be able to stake their NFTs, and get their stake positions recorded (tokenId: "staked on" timestamp)
3. When unstaking is enabled, token holders should be able to unstake their NFTs, and get their stake positions enriched with the "unstaked on" timestamp
4. It should be possible to stake and unstake batches of tokens
5. It should be possible to retrieve number of times a particular address has staked
6. It should be possible to retrieve number of times a particular NFT was staked
7. It should be possible to retrieve stake positions for any NFT owner, or/and any particular NFT
8. Authorized address(es) should be able to enable/disable staking
9. Authorized address(es) should be able to enable/disable unstaking

Implementation Overview

Similar to Alethea ERC20 Token implementation, NFT Staking Helper smart contract implementation is based on the custom AccessControl (see Access Control section for details).

All smart contract functions can be enabled/disabled either via feature flags (for public functions) or by granting/revoking corresponding roles (restricted functions).

Stake Flow

When staking, the following actions are performed by the smart contract:

1. State validation and input verification

    a. Ensure staking is enabled
    b. Verify the token is not currently staked
    c. Verify the token belongs to the address which executes staking
2. Transfer the token into the NFT Staking contract
3. Create and save staking position record, containing token ID, owner, and unix timestamp when it was staked

Unstake Flow

When unstaking, the following actions are performed by the smart contract:

1. State validation and input verification
    a. Ensure unstaking is enabled
    b. Verify the token is currently staked
    c. Verify the token belongs to the address which executes unstaking
2. Update staking position record, adding to it unix timestamp when token was unstaked
3. Transfer the token back to its owner

Implementation Details

This section describes the NFT Staking Helper smart contract implementation details.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the NFT Staking Helper smart contract:

Solidity Name	Description
`targetContract`	Target ERC721 contract address supported for staking

Table 40. Summary of the state variables in the NFT Staking Helper smart contract.

Features and Roles

Features control the behavior of publicly available functions, such as redeeming. In most cases enabling/disabling a feature allows to enable/disable corresponding functionality. Features are stored in `userRoles` mapping for the smart contract itself `userRoles[address(this)]`.

The table below summarizes the features defined in NFT Staking Helper smart contract.

Feature	Bit	Description
`STAKING`	0	Enables staking of the NFTs, `stake`, `stakeBatch`
`UNSTAKING`	1	Enables unstaking of the NFTs, `unstake`, `unstakeBatch`

Table 41. Summary of the NFT Staking Helper smart contract features.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include the token rescue manager role, responsible for rescuing tokens accidentally sent to the NFT Staking contract. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the NFT Staking Helper smart contract.

Role	Bit	Description
`RESCUE_MANAGER`	16	Allows withdrawing ERC20/ERC721 tokens, accidentally sent to the NFT Staking contract, `rescueErc20`, `rescueErc721`
`ACCESS_MANAGER`	255	Reserved, defined in `AccessControl` smart contract, allows modifying features and roles

Table 42. Summary of the NFT Staking Helper smart contract special permissions (roles).

Public Functions

The table below summarizes public functions designed to stake and unstake NFTs.

Function Name	Description
`stake`	Stakes NFT
`unstake`	Unstakes NFT
`stakeBatch`	Stakes multiple NFTs
`unstakeBatch`	Unstakes multiple NFTs

Table 43. Summary of public functions.

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the NFT Staking Helper smart contract.

Function Name	Description
`rescueErc20`	Rescues ERC20 tokens, accidentally sent to the NFT Staking contract
`rescueErc721`	Rescues ERC721 tokens, accidentally sent to the NFT Staking contract

Table 44. Summary of the functions with restricted access.

Note that NFT Staking Helper smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

OpenSea Factory

OpenSea Factory, also known as Factory, or NFT Minter contract is a helper contract, implementing OpenSea `IFactoryERC721` interface for minting NFTs on the fly when they are bought on OpenSea.

Functional Requirements

OpenSea Factory operates on the single ERC721 contract (defined on deployment), which implements a `MintableERC721` interface and has a `mint(address _to, uint256 _tokenId)` function.

OpenSea Factory operates on “options”, each option defining some specific NFT type to be minted.

Please follow OpenSea documentation to understand the basics for the functional requirements for the Factory:

1. https://docs.opensea.io/docs/2-custom-item-sale-contract
2. https://github.com/ProjectOpenSea/opensea-creatures/blob/master/contracts/IFactoryERC721.sol
3. https://docs.opensea.io/docs/2-custom-sale-contract-viewing-your-sale-assets-on-opensea

Implementation Overview

Similar to Alethea ERC20 Token implementation, OpenSea Factory smart contract implementation is based on the custom AccessControl (see Access Control section for details). However, due to the nature of OpenSea requirements, some restricted functions, like `mint`, follow a custom mechanism for restricting access. Most of the other functions follow the standard AccessControl pattern and can be accessed with corresponding permissions.

Implementation Details

This section describes the OpenSea Factory smart contract implementation details.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the OpenSea Factory smart contract:

Solidity Name	Description
`owner`	Deployer address, required by OpenSea
`nftContract`	Mintable ERC721 contract address to mint tokens of
`proxyRegistry`	OpenSea Proxy registry, determines which address is allowed to mint the token at any given time
`options`	Number of OpenSea “options” the factory supports
`baseURI`	Base URI for constructing an option URI

Table 45. Summary of the state variables in the OpenSea Factory smart contract.

Features and Roles

Factory doesn't introduce any features controlling the public functions.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include the token rescue manager role, responsible for rescuing tokens accidentally sent to the OpenSea Factory contract. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the OpenSea Factory smart contract.

Role	Bit	Description
`MINTER`	16	Reserved, not in use
`URI_MANAGER`	17	Allows updating the Base URI field
`OS_MANAGER`	18	OpenSea manager, allows registering a Factory in OpenSea via “Transfer” event emission
`ACCESS_MANAGER`	255	Reserved, defined in `AccessControl` smart contract, allows modifying features and roles

Table 46. Summary of the OpenSea Factory smart contract special permissions (roles).

Public Functions

OpenSea Factory doesn’t introduce any mutable publicly accessible functions, all the functions are designed to be accessed exclusively by OpenSea and Factory admins.

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the OpenSea Factory smart contract.

Function Name	Description
`mint`	Mints an “option”, can be accessed only via OpenSea proxy mechanism
`fireTransferEvents`	Fires a “Transfer” event for each registered “option”
`transferFrom`	Same as mint, a hack to make things work properly with OpenSea

Table 47. Summary of the functions with restricted access.

Note that OpenSea Factory smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

About

Prepared by Basil Gorin for Alethea AI to support Alethea AI protocol version 2 release (release date October 14, 2021).

Idea and concept by Ahmad Matyana and Arif Khan.
Technical design by Ahmad Matyana and Basil Gorin.
Implemented by Basil Gorin, Komninos Chatzipapas, and Vijay Bhayani.
Reviewed by Andrei K, and Miguel Palhas.

Version 2.1 dated September 29, 2021, updated to version 2.4 on September 2, 2022.

References

1. EIP-20: ERC-20 Token Standard
   https://eips.ethereum.org/EIPS/eip-20
2. EIP-1363: ERC-1363 Payable Token
   https://eips.ethereum.org/EIPS/eip-1363
3. EIP-777: ERC777 Token Standard
   https://eips.ethereum.org/EIPS/eip-777
4. EIP-2612: permit – 712-signed approvals
   https://eips.ethereum.org/EIPS/eip-2612
5. EIP-3009: Transfer With Authorization
   https://eips.ethereum.org/EIPS/eip-3009
6. ERC20 API: An Attack Vector on Approve/TransferFrom Methods
   https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/
7. 0x: An open protocol for decentralized exchange on the Ethereum blockchain

https://0x.org/pdfs/0x_white_paper.pdf
8. Compound: Governance
https://compound.finance/docs/governance
9. EIP-721: ERC-721 Non-Fungible Token Standard
https://eips.ethereum.org/EIPS/eip-721
10. EIP-712: Ethereum typed structured data hashing and signing
https://eips.ethereum.org/EIPS/eip-712
11. Decentralized Society: Finding Web3's Soul by Vitalik Buterin et al
https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4105763
12. DID and VC: Untangling Decentralized Identifiers and Verifiable Credentials for the Web of Trust by Clemens Brunner et al
https://dl.acm.org/doi/pdf/10.1145/3446983.3446992
13. EIP-4973: Account-bound Tokens by Tim Daubenschütz, Raphael Roullet
https://eips.ethereum.org/EIPS/eip-4973
14. EIP-5114: Soulbound Badge by Micah Zoltu
https://eips.ethereum.org/EIPS/eip-5114

Appendix A. Deployed Addresses

Name	Address
AliERC20v2	0x6B0b3a982b4634aC68dD83a4DBF02311cE324181
PersonalityPodERC721	0xDd70AF84BA86F29bf437756B655110D134b5651C
IntelligentNFTv2	0xa189121eE045AEAA8DA80b72F7a1132e3B216237
IntelliLinkerERC1967	0xB9F02FC926b2ab66CAdd6eA1Ee90FB0D8698790b
TheRevenantsERC721	0xc2D6B32E533e7A8dA404aBb13790a5a2F606aD75
SophiaBeingERC721	0x75c804fFb01b16B7592a0B9644835244E2140728

Table 48. Deployed smart contracts Ethereum mainnet addresses.

Name	Address
AliERC20v2	0x3ACd26F0b5080C30c066a2055A4254A5BB05F22a
PersonalityPodERC721	0x785b1246E57b9f72C6bb19e5aC3178aEffb0Fe73
IntelligentNFTv2	0x63d49c8D35C9fB523515756337cef0991B304696
IntelliLinkerERC1967	0xbF3Df254b65e527b20c255947E627f6856ff743B
TheRevenantsERC721	0x8920Df4215934E5f6c8935F0049E9b9d8dDF3656
SophiaBeingERC721	0xC6729C6cFc6B872acF641EB3EA628C9F038e5ABb

Table 49. Deployed smart contracts Ethereum görli addresses.