SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

Metafusion

BLOCKCHAIN AND DITRIBUTED LEDGER TECHNOLOGIES

Professors:
Claudio Di Ciccio

Students:
Francesco Palandra
Andrea Sanchietti
Tommaso Sgroi
Luca Sorace

Academic Year 2023/2024

Contents

1 Preface

1.1 Purpose of this document

This document aims to explain the functionalities, features, and implementation of **MetaFusion**, a one-of-a-kind **DApp** that allows users to generate images harnessing the power of **Stable Diffusion** [1]. It is effectively a DApp for the exchange of **Non-Fungible Tokens** (NFTs) based on **Ethereum** [2], designed to provide a system of prompts and collectible cards. Through the use of Blockchain, MetaFusion ensures **security** and **transparency** in both transactions and generation, eliminating the risk of fraud and duplication.

1.2 Overview

The world of NFTs has truly gained popularity in recent years! The ability to certify and trace digital asset ownership to a single user has paved the way for numerous applications. One of the most popular and symbolic ones at the moment is the opportunity to buy and sell images to be the sole owner of them. All this is made possible thanks to programmable Blockchain, the technology capable of recording transactions and, along with them, executing small pieces of code. Over the past years, various **standards** have been created to work with NFTs, but perhaps the most important and popular are **ERC20** [3] and **ERC721**[4]. They are the keystone for tracking the ownership of a unique NFT within the Blockchain.

MetaFusion is a standalone DApp for creating, exchanging, and selling NFTs. So, what makes our DApp different from any other trading platform? Here at MetaFusion, we've developed a system that allows our users to create their NFTs by harnessing the power of Stable Diffusion. Each user can collect Prompts and combine them as they wish to generate unique images, yet verifiable and reproducible at the same time.

Our project is divided into 5 main components:

1. **Contracts:** we have five of them, and they represent the core of the DApp. They track the ownership of NFTs and provide users with methods to manage their tokens.

2. **Oracle:** Its task is to read images and prompt generation requests from the blockchain, generate them, upload the generated data to the decentralized IPFS system, and finally publish the Content Identifier (CID) on the blockchain.

3. **Frontend:** It is the portal through which users can interact with the blockchain to manage their NFTs.

4. **Tracker:** Its task is to read every event published by the contracts and save them to a database, thus speeding up the frontend during data fetching. As you can gather, we chose to track events to avoid overloading the contract with code for retrieving certain information, such as all NFTs owned by a specific account or all NFTs for sale. This information is not stored in memory on the blockchain, and the only way to retrieve it from the frontend would otherwise be to fetch all events from the blockchain and process them to find the needed information at that moment. A possible solution could have been to exploit **ERC721Enumerable's** capabilities [5], but we preferred to avoid this standard as it overloads the gas fees for almost every operation.

5. **Web-API:** It has access to the database and exposes endpoints to the client to visualize the state of the blockchain.

1.3 Team members and main responsibilities

The tasks we have completed during the project are:

1. Application and functionality design
2. Research and preparation of technologies for DApp development
3. Design and implementation of smart contracts
4. Design and implementation of the Oracle, tracker, and web API
5. Design and development of the frontend
6. Testing and debugging of components
7. Code review and adding comments.

Each of us participated in tasks 1), 2), 6), and 7). Below are the remaining duties carried out by each team member:

- Francesco Palandra: Development of the frontend and the connection between the frontend, the web API, and the blockchain.
- Andrea Sanchietti: Implementation of smart contracts, the Oracle, and the web API. Connection between the frontend, the web API, and the blockchain.
- Tommaso Sgroi: Implementation of smart contracts, the tracker, and the web API.
- Luca Sorace: Implementation of smart contracts, the database, and the tracker.

2 Background

2.1 Small historic view

The blockchain is a technology that aims to build a distributed and decentralized system to store immutable and verifiable transactions. Each transaction is stored inside a block, which is a structure containing the hash of the previous one, creating a long chain of links.
The initial proposal of a blockchain comes from the article "Bitcoin: A Peer-to-Peer Electronic Cash System"[6], made by a group of individuals under the name "Satoshi Nakamoto". That article talks about a system where people can exchange money without a central authority. To achieve this, the article suggests using asymmetric cryptography and signatures to control the ownership of the money and the "Proof of work" algorithm to achieve consensus among all nodes and eliminate the double-spending problem. Bitcoin implements **Script** [7], a stack-based - *so, not touring complete* - scripting system for transactions.

Following Bitcoin's steps, in 2015 Ethereum was born to solve the limitations of Bitcoin. Ethereum's selling point was the support of smart contracts, proper snippets of code, this time touring complete, that can be executed by every node in the blockchain. Thanks to smart contracts Tokens and Decentralized Apps *(Called "DApps")* started to emerge in the Ethereum blockchain, making it rise in popularity.

In 2022 a big upgrade called **"The Merge"** happened in the Ethereum blockchain. Of the novelties, the most important one was the consensus protocol switching from **Proof of Work** to **Proof of Stake**, reducing by 99% the energy used to run the blockchain.

2.2 Consensus

Consensus is a mechanism mainly used to let all nodes of a blockchain agree on the validity of all transactions. Different blockchains use different consensus algorithms, however, the most important ones are:

- **Proof of work (POW)**: POW adds a nonce field to the block that's included in the hash. Each node, to propose a block, tries setting that nonce to a randomized value, hashes the whole block, and checks if the output is lower than a certain threshold. If so, the block is approved.
  This algorithm makes computing the POW hard, but easy to verify, so only a few nodes are able to broadcast a winning block. Branches (where two nodes win at the same time) are possible but solved probabilistically by just waiting for enough blocks to be released on the same branch. Six blocks are generally enough to say that a main branch has emerged among the other ones.

- **Proof of stake (POS)**: At each slot, a set of validator is chosen to propose a block, based on the number of tokens that they own. They may be branches anyway with a low probability, in this case, the winning block is the one that forms the chain with the greatest weight of attestations.
- **Byzantine fault tolerance (BFT)**: Consensus is achieved even between malfunctioning or malicious nodes. BFT is very fast, but it requires a few amount of nodes. It is used in blockchains like **Algorand** [8], but only among a limited amount of proposers that are randomly chosen.

2.3 Types of blockchain

Blockchains can be classified based on their permission level regarding the consensus part, or based on their visibility.

Consensus permissions classification:

- **Permissionless blockchain**: They are fully decentralized blockchains. Based on the principle of the "trustless", they let every node participate in the consensus process granting everyone with full anonymity.
- **Permissioned blockchain**: They are closed networks where an administrator selects some nodes that will manage consensus and data validation.

Visibility classification:

- **Public blockchain**: Every node can propose new blocks and view the old ones. Bitcoin and Ethereum are examples of a public blockchain.
- **Private blockchain**: They're often used as enterprise-distributed solution. The Blockchain access is restricted only to some pre-selected nodes.
- **Hybrid blockchain**: A blockchain combining the advantages of the public and private ones, to fit the specific needs of the project. Since it's a sort of "had-oc" solution, the internals may vary from blockchain to blockchain.

2.4 Token standards

In Ethereum, using smart contracts, it is possible to implement Tokens, which represent any kind of asset. Different token standards differ in their proprieties. These standards are available to Solidity's programmers (the programming language used in smart contracts) as frameworks that can be included in a smart contract.

- **ERC-20 tokens** [3]: Each token is fully interchangeable with other equivalent ones, so it is "fungible". These tokens are widely used in Ethereum blockchain to represent coins or other kind of assets.
- **ERC-721 tokens (NFT)** [4]: Each token is unique and represents virtual collectible assets. Owning an NFT of an asset is considered equivalent to owning that said asset, this is why this is a secure and transparent way to manage ownership. Contrary to ERC-20 tokens, these cannot be interchanged with other equivalent ones, so they're "non-fungible". A variant of the ERC-721 token is the **ERC-721Enumerable** [5], which stores useful information that can be used to fetch the existent tokes, in exchange for a higher gas demand.
- **ERC-1155**: These tokens can be both fungible and not fungible, allowing more flexibility for Solidity's programmers.

2.5 Application's domain

The blockchain is currently used in many kinds of applications. Its main features, like the verifiability and security of transactions, the decentralized network, and the lack of a central authority, offer a solid way to store events and transactions regarding logistics, supply chains, finance, private identities, and voting systems. Thanks to the power of smart contracts is moreover possible to implement logic directly inside the blockchain, in a way that it's still secure and verifiable, completely deleting the need for intermediaries.

To make an example, the transaction log is really useful to track money exchange in a financial environment. Or, another example, the anonymity behind the transaction's author is fundamental for creating a voting system

It's to keep in mind that blockchain is a fairly new technology that still has to develop. Many companies have already put an eye on and are already investing in this technology, however, it still needs some time before being deployed in more applications worldwide.

3 Presentation of the Context

3.1 Aim of MetaFusion

MetaFusion has been designed with a fundamental purpose: to provide users with an interactive and engaging environment for **creating and exchanging unique digital assets** in the form of NFTs. Our main goal is to enable users to explore their creativity through the combination of Prompts, thereby creating unique and personalized digital artworks.

Beyond the mere economic transaction, MetaFusion aims to provide a fun and **stimulating experience**, transforming the process of creating and exchanging NFTs into an engaging and exciting activity. Additionally, we recognize the importance of investing in the blockchain and the NFT ecosystem.

MetaFusion offers users the opportunity to invest in unique digital assets, providing a chance for portfolio diversification in a continuously growing sector. In this way, MetaFusion goes beyond a simple financial transaction, offering an environment that combines creativity, entertainment, and investment opportunities, transforming the concept of NFTs into a comprehensive and rewarding experience.

3.2 What is the reason for using a Blockchain, and which specific type should be employed?

The use of Blockchain represents a fundamental choice for MetaFusion, our platform dedicated to the creation and exchange of NFTs. Blockchain, particularly the ERC721 protocol, plays a crucial role in **certifying and tracing the ownership** of digital assets within our platform. It provides an immutable and transparent ledger of transactions, ensuring unequivocal proof of the legitimacy and authenticity of NFTs.

Blockchain technology allows for the decentralization of information, eliminating the need for a central authority and ensuring that each transaction is secure, transparent, and traceable. Furthermore, the use of Blockchain paves the way for a wide range of innovative applications, enabling users to participate in a continually evolving NFT ecosystem.

In summary, the adoption of Blockchain in MetaFusion not only ensures the security and integrity of transactions but also forms the foundation for a reliable, authentic NFT experience that meets the growing needs of enthusiasts and digital investors.

By following the Wust Gervais schema [9], we identified the perfect type of blockchain for our application.

As you can see, we have chosen a **permissionless blockchain**, as our system requires state storage, accepts multiple writers, cannot rely on a Trusted Third Party

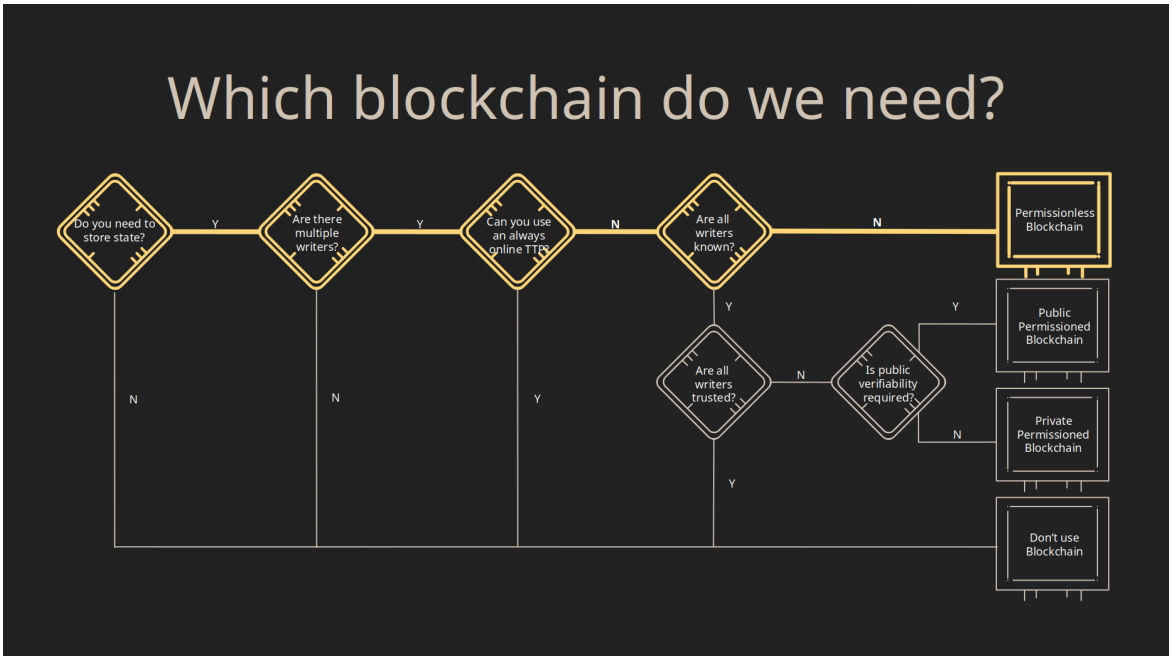

Figure 1: The Gervais diagram

(TTP), and not all writers are known.

We deliberately selected **Ethereum** as the development platform for our DApp. The decision to use Ethereum, a public blockchain, was driven by several key factors that reflect our commitment to ensuring maximum transparency, accessibility, and community engagement. As a public blockchain, Ethereum offers an open and accessible ledger to everyone, allowing users to easily verify transactions and NFT ownership. This transparency reinforces user trust and underscores the authenticity of digital creations within MetaFusion.

Furthermore, Ethereum's public architecture facilitates the participation of the entire community. Users and investors can actively contribute to the growth and evolution of MetaFusion's NFT ecosystem, leveraging the potential of an open and collaborative platform. The choice of Ethereum reflects our desire to create an inclusive and dynamic environment where the community can contribute to the ongoing success of the platform. In this way, MetaFusion becomes not only a space for the creation and exchange of NFTs but also a meeting point for a broad and diverse community interested in digital assets.

4 Software Architecture

This section is structured as follows:

1. An introduction to the general architecture of our DApp
2. A close up to the smart contracts architecture
3. A description of the oracle and its interactions with the blockchain
4. An explanation of what is the purpose of the tracker, and which are the events tracked by it.
5. A brief explanation of the web API.

4.1 DApp Architecture

As previously mentioned, our application is divided into **5 main parts**: Contracts, Oracle, Tracker, web API, and frontend. The following image (Figure 2) illustrates the overall architecture.

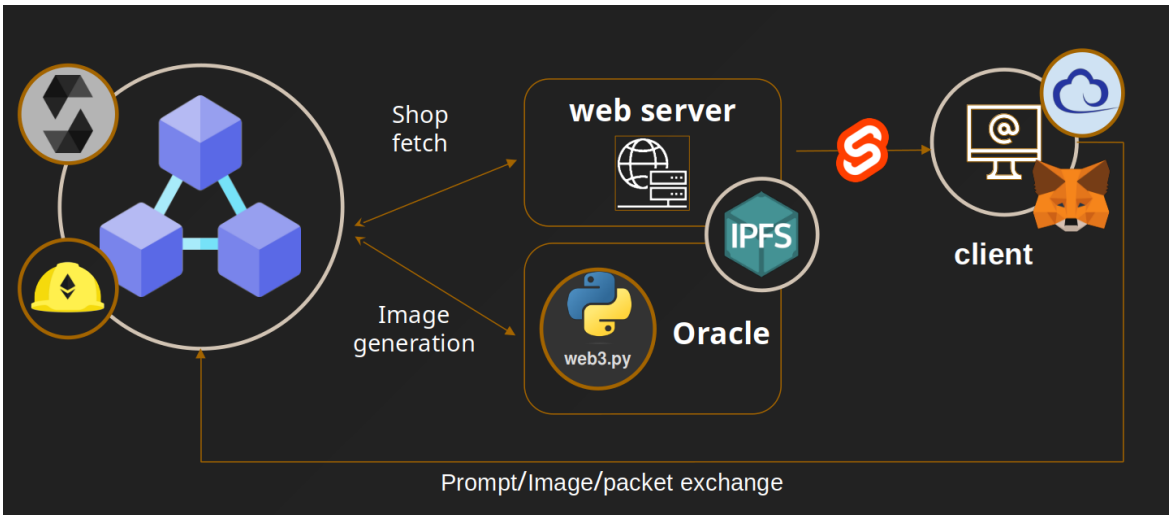

Figure 2: The general architecture of MetaFusion

On the left side, we have the blockchain, simulated through **HardHat** [10], along with the contracts. In the center, there is the oracle at the bottom and a server at the top, serving as both a tracker and a web API. Finally, on the right side, we have the client, which connects with **Metamask** [11] and interacts with the blockchain through **ethers.js** [12].

Now, we will proceed to explain the functionalities and interactions of each element in more detail.

4.2 Smart contracts

We use smart contracts to forge packets, open them to generate packet IDs, generate image IDs, and destroy them; we also use smart contracts to buy and sell our tokens. *Packets, Prompts, and Cards extend a contract called Sellable*, which implements ERC721. We chose this standard because of its properties:

1. **Non-Fungible Nature:** Each token within an ERC-721 contract is distinct and has a unique identifier. Unlike fungible tokens (e.g., ERC-20 tokens), ERC-721 tokens are not interchangeable on a one-to-one basis.

2. **Ownership and Transferability:** ERC-721 tokens represent ownership of a specific asset, and ownership can be transferred between Ethereum addresses. Each token has an owner, and ownership is changed through the `transferFrom` function.

3. **Approval Mechanism:** To transfer a token on behalf of another address (e.g., in a marketplace scenario), the owner must explicitly approve the transfer using the `approve` function. This approval mechanism enables secure and controlled token transfers.

Due to the nature of our application, we have opted to use the **"Composition over inheritance"** design pattern to handle tokens of packets, prompts, and cards differently, all while avoiding their exposure to the client.
The primary contract, *MetafusionPresident* (named so because it serves as the **exclusive executor and handler** of the contracts), incorporates objects for packets, prompts, and cards, and implements methods for minting, burning, listing, and buying assets. The president also set constants as fees and parameters like the number of prompt types and prompts in a packet. The Sellable contract itself extends ERC-721 and exposes all the common functions used by our NFTs.

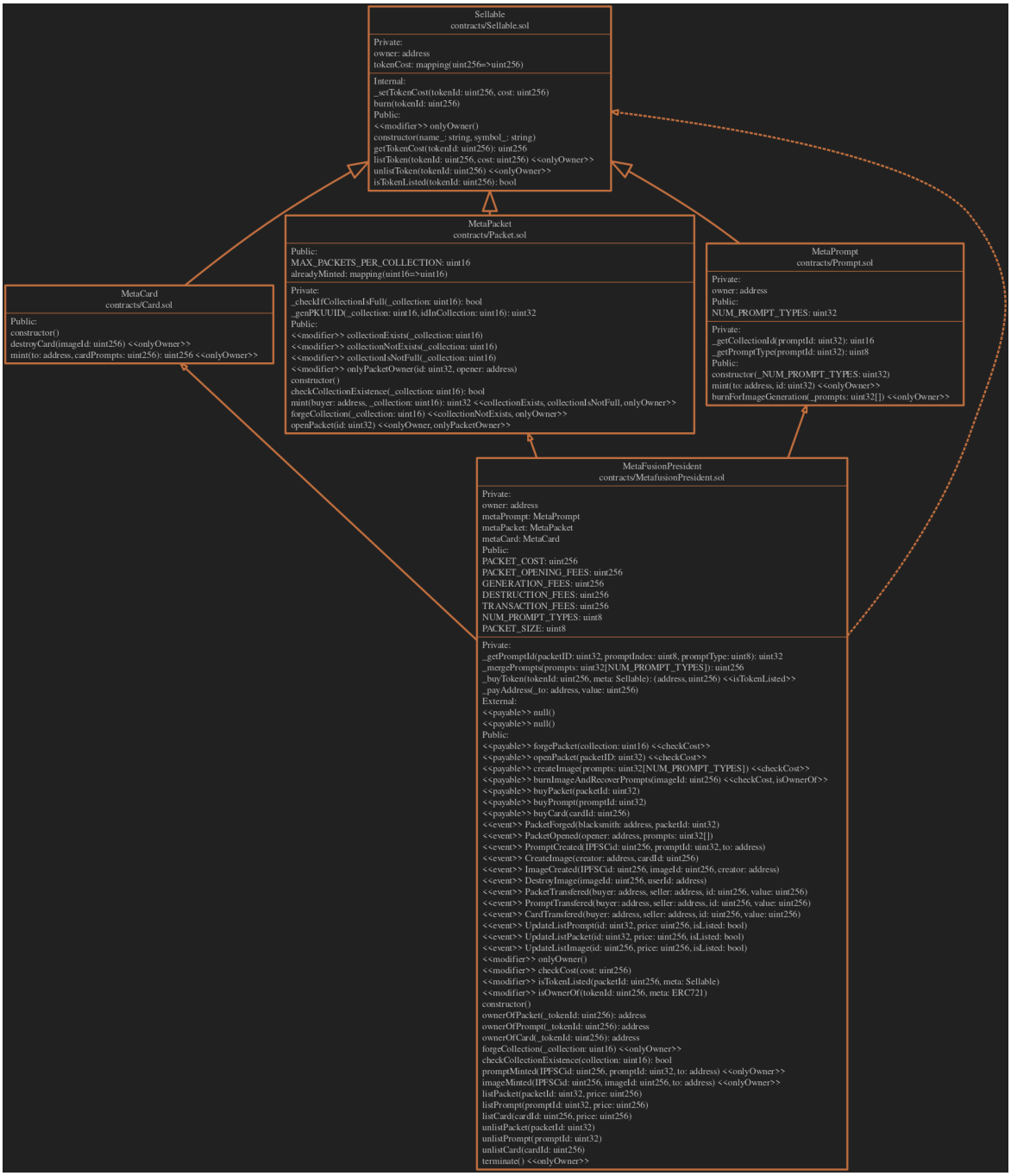

Figure 3: Architecture of the contracts.

This implementation allows us to ***restrict access to the NFTs contracts*** *to only the President, which ensures when operations are valid and, if not, reverts them with the most appropriate error.* The "MetafusionPresident" contract is also delegated to **emit the specific events** when some action is performed over the NFTs, such as buy, sell, or mint. Those events are a crucial feature to inform the oracle, the tracker, and everybody on the blockchain of an action performed by a client. **This is a solid and secure design.**

4.2.1 Token lifecycle

The *lifecycle* of our assets is summarized in figure 4. The first thing that a user can do is buy (1) a packet. This packet is then added to the user's collection (2, 3) and the user can open it to reveal (4) the prompts hidden inside (5). Prompts can then be combined to generate (6) Cards (also referred to as Images(7)). The user can list (8) each one of these assets and let someone else buy (9) them. Cards can also be burned (6) to get back the prompts used to generate them.

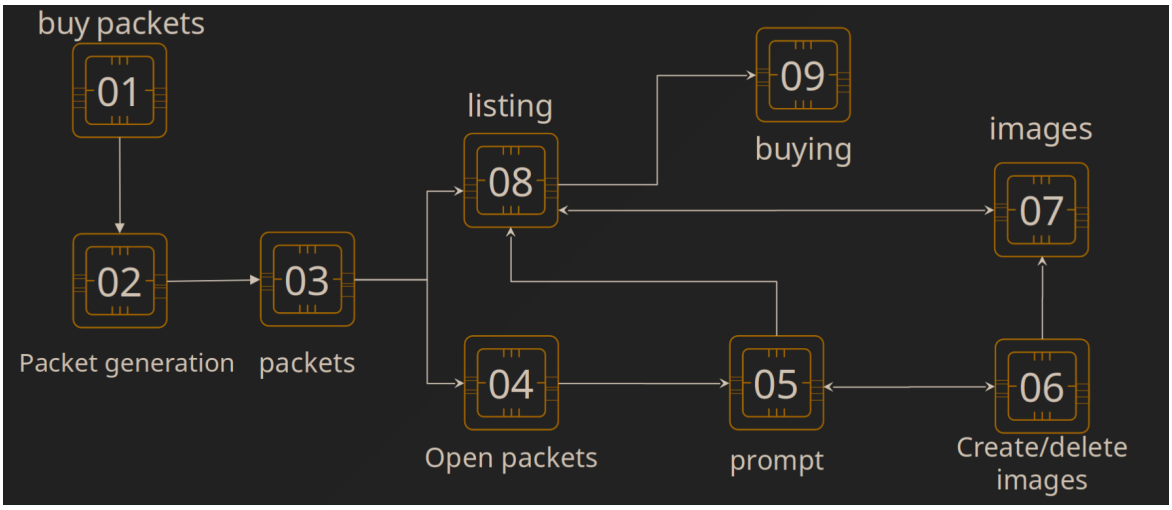

Figure 4: Lifetime of our assets.

4.2.2 Use case scenarios

The following figures (6, 5) represent the use case scenario for the MetaFusionPresident, where a user performs various actions on his tokens, the owner creates new collections, and the Oracle publishes new Prompts and Cards.

Figure 5: Use case scenario for Token exchange

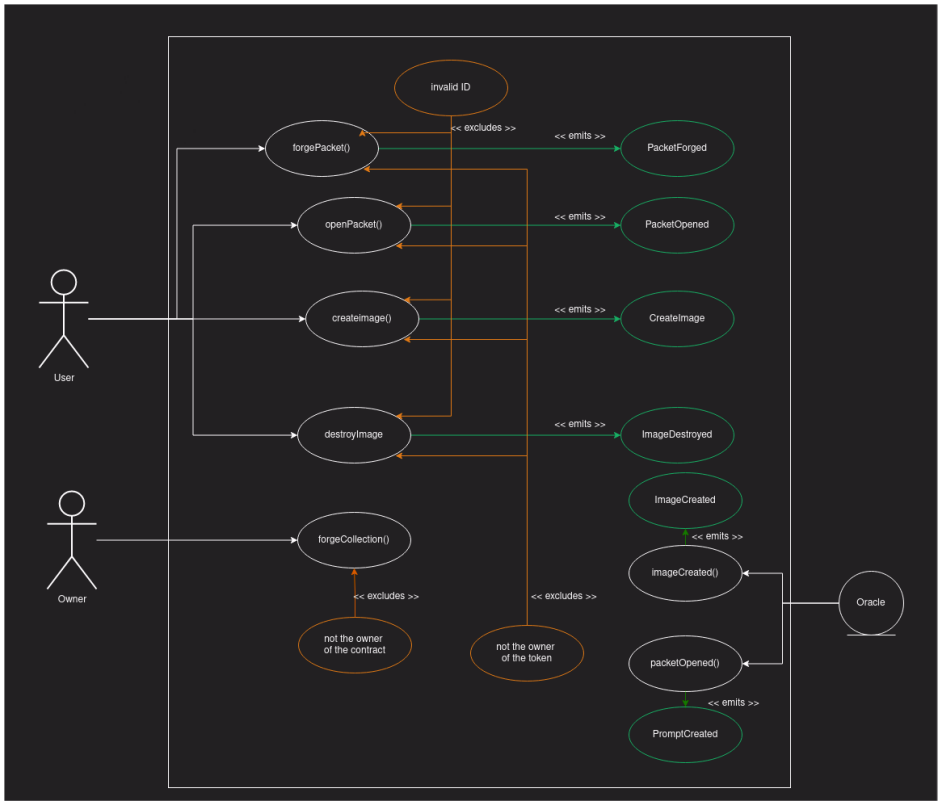

Figure 6: Use case scenario for minting and burning of our Tokens

4.3 ID system in MetaFusion

Something that we are very proud of is the **Token ID system**. We wanted to **reduce** as much as possible the **overhead** given by the necessity of storing on the Blockchain which Prompts are used for generating a Card. *We decided to exploit the ERC721 Token ID system to store information and at the same time have a unique ID for each one of the generated Tokens.*

4.3.1 Packet IDs

To understand our ID system, we have to first explain how we decided to build Packet IDs.

In MetaFusion, we have NFT **collections** (or editions). The owner of the contract is the only one who has access to the function that generates new collections. We store a mapping inside the Packet contract that maps a collection to the number of Packets already minted for that collection. If a collection is mapped to 0, it means that the collection is still not available, while every number n greater than 0 means that $n-1$ packets have been minted for that specific collection. When the owner calls the $forgeCollection$ function, the Packet contract sets to 1 the value in the mapping associated with the new collection.

Since we have both the collection ID and the number of packets already minted for

that collection, we can simply create the ID of a packet by concatenating these two numbers in an uint32

Figure 7: The structure of the Packet id

Since we have a limit of 750 packets per collection and the number of possible collections is $2^{16} = 65536$, we will not have problems with this ID system for packets.

4.3.2 Prompt IDs

When a user decides to open a Packet, we have to generate **8 Prompts** and each one of them must have a pseudo-random type ID that indicates the **Prompt type** (0 for character, 1 for hat, 2 for handoff, 3 for color, 4 for eyes and 5 for style).
We decided to cut the 3 most significant bits from the Packet ID that is being opened to store a sequential integer that indicates the position of the Prompt inside the packet (we need numbers from 0 to 7, so 3 bits are perfect), and the bits from the 13th to the 16th to store the type of the Prompt (3 bits for 6 possible type values).

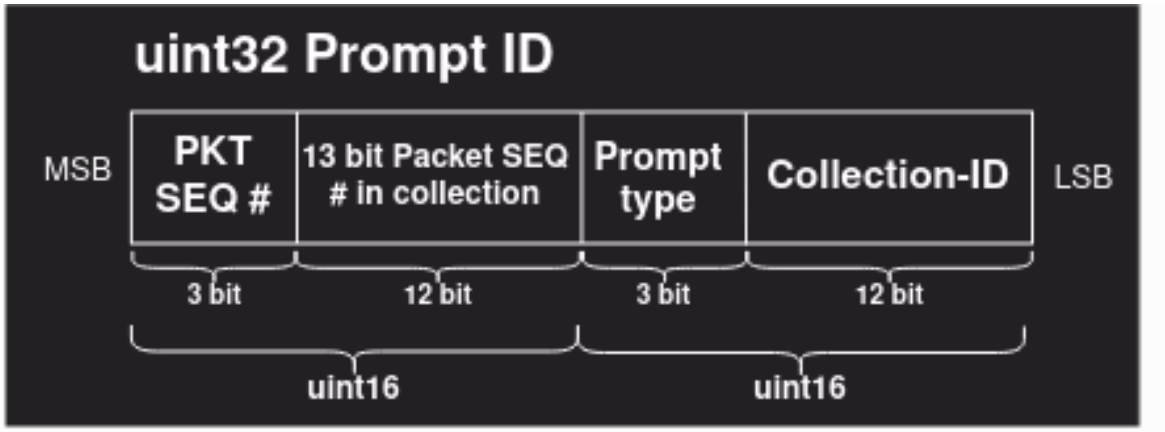

Figure 8: The structure of the Prompt id

This reduces the number of possible collections to $2^13 = 8192$ which is still a very high number. The total number of bits necessary to store the ID of a Prompt is then 32, just like the Packet ID.

4.3.3 Card IDs

A Card is made of 6 prompts (one for each possible Prompt type). To generate a Card, the character Prompt is mandatory, while the other prompts are not strictly required. The ID of a Card is then calculated by concatenating the IDs of the prompts used for generation. This gives us a $6 * 32 = 192$ long ID that is then completed with a 64 bit long pseudo-random seed, yielding a **256-bit long ID** that can be conveniently stored in an uint256.

Figure 9: The structure of the Image id

When a Card is minted, we also burn the Prompts used to generate the ID, as we have all the information needed to mint again them when the user decides to destroy the Card. This is useful as we don't have to keep in memory a mapping from Prompt ID to a boolean that says if the Prompt is already been used, and simultaneously acts as a safety measure to prevent the exchange of Prompts that have been used to generate an image (we refer to used prompts as **freezed**).

4.4 Events

Our system relies on events. They are useful for sharing data and informing the Oracle and the Tracker of what is happening on the chain; they build a history of transactions that are stored forever and can be accessed by anybody. This makes our system fault-resistant since is possible to replicate all transactions to achieve the final consistent status of the system, even in the face of failures or disruptions. Events serve as a reliable and transparent mechanism for broadcasting changes and updates, ensuring that both the oracle (4.5) and the tracker (4.6) can stay synchronized with the latest state of the blockchain. The historical record of transactions, perpetually stored and accessible to anyone, not only enhances fault resilience but also promotes transparency, and trust within the decentralized ecosystem. This robust foundation allows stakeholders to verify the integrity of the system and contributes to the overall security and reliability of our blockchain-based solution.

Our events are:

1. **PacketForged:** a user has brought a new packet from the contract, it returns the address of the wallet that brought the packet and the ID of the packet.

2. **PacketOpened:** a user has opened a packet and a fixed number of prompt's NFT are generated; the events return the list of IDs and the address of the opener.

3. **PromptCreated:** When the packet is opened, this event is emitted (generated through the oracle) and returns the IPFS hash, prompt token, and the address of the owner.

4. **CreateImage:** When the prompts are merged to forge a card, this event is emitted and returns the owner of the new card and its token.

5. **ImageCreated:** Emitted after the oracle has created the image, it returns the IPFS hash and the image token (ID).

6. **DestroyImage:** Emitted when a user wants to destroy his image recovering prompts from it, it returns the token of the burned image and the user address.

7. **PacketTransfered:** emitted after a user has successfully brought a Packet from another user, it returns the buyer, the seller, the Packet token, and the Packet's cost.

8. **PromptTransfered:** emitted after a user has successfully brought a Prompt from another user, it returns the buyer, the seller, the packet token, and the packet's cost.

9. **CardTransfered:**emitted after a user has successfully brought a Card from another user, it returns the buyer, the seller, the packet token, and the packet's cost.

10. **UpdateListPacket:** emitted when a user just listed or unlisted a Packet. This event returns the ID of the listed Packet, a boolean that describes the listing status (true for listed, false for unlisted), and the price set for the Packet.

11. **UpdateListPrompt:** emitted when a user just listed or unlisted a Prompt. This event returns the ID of the listed Prompt, a boolean that describes the listing status (true for listed, false for unlisted), and the price set for the Prompt.

12. **UpdateListImage:** emitted when a user just listed or unlisted an Image. This event returns the ID of the listed Image, a boolean that describes the listing status (true for listed, false for unlisted), and the price set for the Image.

4.5 Oracle

The Oracle is a centralized entity that serves requests for image and prompt generation. Since randomness, image generation, and strings are natural enemies of smart contracts, developers must rely on components that are capable of pulling information from the blockchain and pushing real-world data into it. Our oracle continuously checks new blocks and, whenever he finds an event associated with a generation request, he generates data, uploads it on the IPFS, and finally pushes the IPFS Content ID (CID) into the Blockchain.

The Oracle listens for two events emitted by users when they mint prompts:

1. **PacketOpened:** when this event occurs, the Oracle uses the prompts that are shared in the emitted event to extract random Prompts, load them on IPFS, and finally call a President's function on the blockchain that emits the event **PromptCreated(IPFSCid, promptId, minter)**.

2. **CreateImage:** When a user creates an Image on the blockchain, he creates a mere Token on the blockchain. The Oracle has to read the prompts used to create the token and generate an image with stable diffusion. Then he uploads the image on IPFS and calls the President's function that emits the event **ImageCreated(IPFSCid, imageId, minter)**.

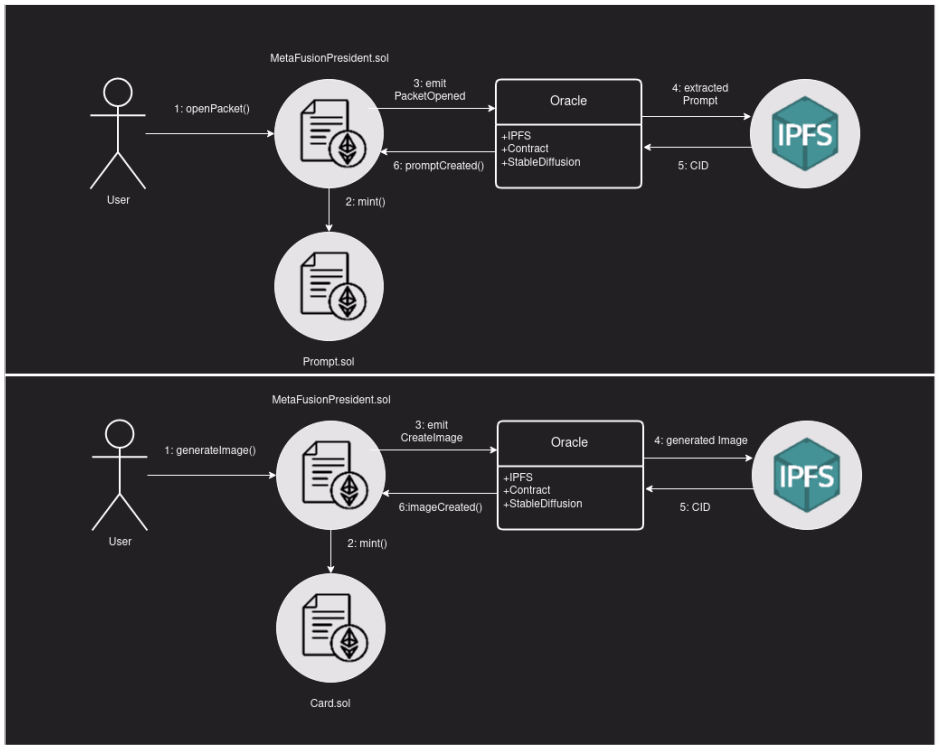

Figure 10: Oracle collaboration diagram.

4.6 Tracker

The tracker is our Python program that uses the Web3.py [13] library to listen and filter events on the blockchain. Its purpose is to fetch MetaFusion custom events, extract data, and store them in a centralized SQL database (DB). The purpose of the DB is to use it as a cache, without fetching data from the blockchain or IPFS all the time. It is possible to rebuild the entire database simply by reading the transactions from the blockchain and the linked data from IPFS; since we don't rely on any centralized system.
The tracker works as follows: connect to IPFS, connect to the MetafusionPresident smart contract, initialize the event's filters, connect to the local DB, and listen to the chain's events.
The events that the tracker selects are all those that concern user interactions:

1. create NFTs
2. Buy, list and unlist NFTs
3. Transfer NFTs ownership
4. Destroy NFTs

By doing so, we can *track all updates* and the client can view what is happening using our platform updating our centralized database. Every event has a custom name that tells which action is performed on the chain and gives all useful data involved. The tracker keeps track of almost all the events that are mentioned in section 4.4.

Here 11 we sketched a general collaboration diagram for the tracker. Every time a user performs an action that alters the state of the blockchain, an event is emitted.

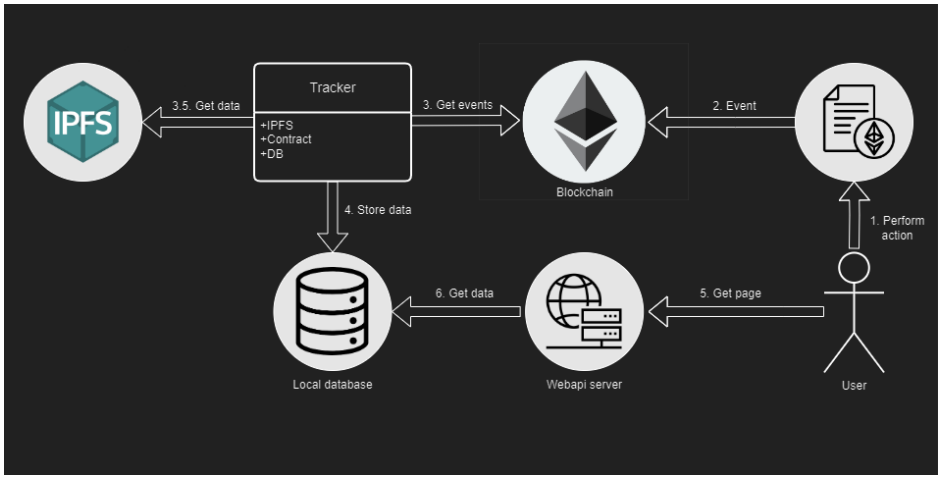

Figure 11: Tracker's collaboration diagram.

This event is watched by the tracker who alters the state of the database. When the caller of the contract is the Oracle, the tracker also downloads the data that have

just been uploaded to IPFS.

Events are read sequentially, so we have the certainty that the state in the database cannot reach an inconsistent status.

4.7 Web API

At this point we can talk about the web API; its role is to expose some REST API endpoint for interacting with the cache database retrieving resources. It responds to the frontend's requests providing all information about users (the NFTs he owns) and Packets, Prompts, Cards NFTs.
The web API uses Fast API framework [14], we expose 4 types of API: **User**, **Packet**, **Prompt**, **Card**.

5 Implementation

5.1 Tools

5.2 Smart contract

Our smart contract was built and thoroughly tested using the Hardhat development environment, a powerful tool that provides comprehensive testing and deployment environment for Ethereum smart contracts. Hardhat allows us to deploy and interact with our smart contract on a virtual Ethereum blockchain, facilitating efficient testing of all integrations and interactions in a controlled environment.

Hardhat played a crucial role in the development process, particularly due to its robust testing capabilities. Given the complexity of our token generation logic, which involves intricate functionalities and functions within the smart contract (refer to the earlier discussion on ERC-721 tokens and the specific characteristics of our contract), Hardhat proved instrumental in ensuring the correctness and reliability of our code.

To enhance our testing procedures and ensure the solidity of our smart contract, we employed the Chai testing library in conjunction with the Ether library for TypeScript. Chai provides a set of expressive assertions, allowing us to write clear and concise tests to validate the behavior of our smart contract functions. The integration of the Ether library, specifically designed for TypeScript, streamlined the development process, enabling us to leverage strong typing and benefit from the advantages of TypeScript in our Ethereum smart contract development.

5.2.1 Code

Note: we added comments to every element of the code. However, for compression reasons, we decided to remove them from the following section.

Now let's take a brief overview of the contract implementation:

```
contract MetaFusionPresident {

    /////////////// CONSTANTS ///////////////
    address immutable private owner;  // Owner of the contract

    fees and constants declaration ...

    /////////////// VARIABLES ///////////////
    MetaPrompt private metaPrompt;
    MetaPacket private metaPacket;
    MetaCard   private metaCard;
```

```
    /////////////// EVENTS ///////////////

    events declaration ...

    /////////////// CONSTRUCTOR///////////////
    constructor() {
        owner = msg.sender; // set the owner
        metaPacket = new MetaPacket();
        metaPrompt = new MetaPrompt(NUM_PROMPT_TYPES);
        metaCard = new MetaCard();
    }
```

5.2.2 Sellable contract

Before going into details with Packet, Prompt, and Card, we must see the Sellable contract, which is extended by all NFTs contracts. It exposes the basic common features.

```
contract Sellable is ERC721 {
    /////////// CONSTANTS ///////////
    address immutable private owner;
    /////////// VARIABLES ///////////
    mapping(uint256 => uint256) private tokenCost;
    /////////// MODIFIERS ///////////
    modifier onlyOwner() {
        require(msg.sender == owner, "You're not the owner!");_;
    }
    /////////// CONSTRUCTOR ///////////
    constructor(string memory name_, string memory symbol_)
        ERC721(name_, symbol_) {
        owner = msg.sender;
    }
```

5.2.3 Instantiate Packet contract

```
contract MetaPacket is Sellable {
    /////////////// CONSTANTS ///////////////
    uint16 constant MAX_PACKETS_PER_COLLECTION = 750;
    ////////////// VARIABLES  ///////////////
    mapping (uint16 => uint16) public alreadyMinted;
    /////////////// MODIFIERS ///////////////

    ...

    /////////////// CONSTRUCTION ///////////////
    constructor() Sellable("MetaPacket", "PKT") { }
```

5.2.4 Instantiate Prompt contract

```
contract MetaPrompt is Sellable {
    address immutable private owner;
    uint32 public immutable NUM_PROMPT_TYPES;
    constructor(uint32 _NUM_PROMPT_TYPES)
        Sellable("MetaPrompt", "PRM") {
    NUM_PROMPT_TYPES = _NUM_PROMPT_TYPES;
}
```

5.2.5 Instantiate Card contract

```
contract MetaCard is Sellable {
    constructor() Sellable("MetaCard", "MCD") { }
    ...
}
```

5.2.6 Important functions

Looking at all functions will be extremely time-consuming, considering the large domain of the application, so we'll take a look only at the important ones.

5.2.7 Forge Packet

Forge Packet: from the president, the user buys a new packet, so the delegated contract instantiates a new Packet's NFT:

```
// MetafusionPresident contract function
function forgePacket(uint16 collection) public payable
    checkCost(PACKET_COST) {
    // Forge a metaPacket.
    uint32 packetUUid = metaPacket.mint(msg.sender, collection);
    emit PacketForged(msg.sender, packetUUid);
}
```

```
// Packet contract function
function mint(address buyer, uint16 _collection)
        public collectionExists(_collection)
        collectionIsNotFull(_collection) onlyOwner returns(uint32){
    uint16 id = alreadyMinted[_collection];
    uint32 packetUUid32 = _genPKUUID(_collection, id);
    uint256 packetUUid = uint256(packetUUid32);
    _safeMint(buyer, packetUUid);
    alreadyMinted[_collection]++;
    return packetUUid32;
}
```

5.2.8 Open Packet

Open Packet: When the user decides to burn a Packet, he invocates a function of the MetaFusionPresiden that uses the contract responsible for packets to destroy the Packet Token. Then the President generates the prompt IDs and mints them by calling the Prompt contract:

```
// MetafusionPresident contract function
function openPacket(uint32 packetID) public payable
checkCost(PACKET_OPENING_FEES){
    // destroy the packet.
    metaPacket.openPacket(packetID);
    // mint the prompts
    uint32[] memory prompts = new uint32[](PACKET_SIZE);
    for (uint8 i = 0; i < PACKET_SIZE; i++) {
        // pseudo random prompt type
        uint256 idhash = uint256(keccak256(
            abi.encodePacked(packetID, i, block.timestamp)));
        uint8 prompt_type = uint8(idhash % NUM_PROMPT_TYPES);

        // mint the prompt
        uint32 promptId = _getPromptId(packetID, i, prompt_type);
        metaPrompt.mint(msg.sender, promptId);
        prompts[i] = promptId;
    }
    // emit event
    emit PacketOpened(msg.sender, prompts);
}
```

Opening the packet the NFT is destroyed.

```
// Packet contract function
function openPacket(uint32 id) public onlyOwner
onlyPacketOwner(id, tx.origin) {
    super.burn(uint256(id));
    // simply use ERC-721 _burn func and set price to 0
}
```

5.2.9 Create Card

Create Card: The user can burn some of his Prompt Tokens to mint a new image. Here we use the IDs of the prompts to generate the ID of the Card. We also calculate a pseudo-random seed to add to the ID.

```
function createImage(uint32[NUM_PROMPT_TYPES] memory prompts)
public payable checkCost(GENERATION_FEES) {
    // check if the user sent the exact number of prompts
    require(prompts.length == NUM_PROMPT_TYPES, string(
      abi.encodePacked("You shall pass the exact number of prompts:",
```

```
      NUM_PROMPT_TYPES)));

    // burn the prompts.
    uint32[] memory prompts_array = new uint32[](NUM_PROMPT_TYPES);
    for (uint32 i = 0; i < NUM_PROMPT_TYPES; i++) {
        uint32 prompt_id = prompts[i];
        prompts_array[i] = prompt_id;
    }

    metaPrompt.burnForImageGeneration(prompts_array);

    // merge prompt IDs and mint the card
    uint256 mergedPrompts = _mergePrompts(prompts);
    uint256 cardId = metaCard.mint(msg.sender, mergedPrompts);
    // emit event
    emit CreateImage(msg.sender, cardId);
}
```

5.2.10 Burn Prompt for Image Generation

The function **burnForImageGeneration** works as follows: burn prompt and mint a new image that embeds the generation's prompts with a random seed (that seed will be used to generate the image by the oracle).

```
function burnForImageGeneration(uint32[] memory _prompts)
public onlyOwner {
    require(_prompts[0] != 0, "Character prompt is missing!");
    // chack if all the prompts are valid
    for (uint8 i = 0; i < NUM_PROMPT_TYPES; i++) {
        if (_prompts[i] != 0) {
            uint16 collectionId = _getCollectionId(_prompts[0]);S
            require(tx.origin == ownerOf(_prompts[i]),
              "Only the owner of the prompts can create an image!");
            require(_getCollectionId(_prompts[i]) == collectionId,
              "The prompts must belong to the same collection!");
            require(_getPromptType(_prompts[i]) == i,
              "The prompts must be of the correct type!");
        }
    }
    // burn the prompts
    for (uint8 i = 0; i < NUM_PROMPT_TYPES; i++) {
        if (_prompts[i] != 0){
            burn(_prompts[i]);
        }
    }
}
```

5.2.11 Burn Card and Recover Prompts

Burn Card and recover Prompts: This function allows a user to destroy his Card NFT and recover the used prompt for it. To retrieve Prompts, we have to mint them again.

```
function burnImageAndRecoverPrompts(uint256 imageId) public payable
checkCost(DESTRUCTION_FEES) isOwnerOf(imageId, metaCard){
    // burn the image
    metaCard.destroyCard(imageId);
    // recover the prompts
    uint256 imageIdShifted = imageId >> 64; //remove seed
    for(uint8 i = 0; i < NUM_PROMPT_TYPES; i++){
        uint32 currentPromptId = uint32(imageIdShifted & 0xffffffff);
        if (currentPromptId != 0){
            metaPrompt.mint(msg.sender, currentPromptId);
        }
        imageIdShifted = imageIdShifted >> 32;
    }
    emit DestroyImage(imageId, msg.sender);
}
```

5.2.12 Buy Token

Buy Token: In the end we have a function to buy listed Tokens, we just show the private generalization of it, since all other specific functions call this.

```
function _buyToken(uint256 tokenId, Sellable meta) private
isTokenListed(tokenId, meta) returns (address, uint256){
uint256 token_cost = meta.getTokenCost(tokenId);
    require(msg.value >= token_cost + TRANSACTION_FEES,
        "You didn't send enough ethers!");
    address seller = meta.ownerOf(tokenId);
    _payAddress(seller, token_cost);
    meta.transferFrom(seller, msg.sender, tokenId);
    return (seller, token_cost);
}
```

5.3 Oracle

Whenever a user mints a Prompt or a Card on the blockchain, the Oracle has to generate the respective sentence or image and alert everyone that the minted Token has been associated with the generated content.
The Oracle is fully written in Python and uses **web3.py** [13] to interact with the MetaFusionPresident contact. It continuously polls the Blockchain and looks for fresh Events.
Whenever he finds an Event, he uses Python's reflections system (explained more in

detail in the 5.4) to retrieve the correct handler.

5.3.1 Oracle's Events

As anticipated, two different Events can trigger the Oracle:

- **PacketOpened(address opener, uint32[6] prompts)**: where he randomly extracts a sentence.
- **CreateImage(address creator, uint256 cardId)**: Where he retrieves the prompts and the seed from the cardId, builds a full sentence, and finally generates the image with SDXL turbo, a diffuser model available thanks to the diffusers library. During the generation process, he also sets the seed of the generation to the one extracted by the card ID. Thanks to this model, the creation of images is almost instantaneous and users can immediately visualize their Cards.

Our Oracle is a fully centralized system, providing a robust and controlled environment for handling the intricacies of content generation in response to blockchain events. One notable advantage of this centralized design is the ability to recover seamlessly in the event of a system crash. In the unlikely scenario that the Oracle encounters an unexpected failure, such as a crash, it is designed to restart from the point of the last successfully generated image.

This crash recovery mechanism ensures that the Oracle can pick up where it left off, specifically from the generation associated with the last event it processed before the crash. In practical terms, this means that if the Oracle was in the process of creating an image or sentence for a recently minted Prompt or Card, it can resume this generation task upon restarting. As a result, users might experience a temporary delay in receiving the generated content, but the overall state of the blockchain remains unaltered.

This approach enhances the reliability of the MetaFusion system, as it guarantees that even in the face of unforeseen disruptions, the Oracle can maintain continuity and consistency in content generation. Users can trust that their interactions with the platform remain reproducible, and the system as a whole is resilient to Oracle-related incidents.

5.3.2 IPFS

We used a local-only instance of IPFS [15] to test the functionalities of our application. The IPFS node is started whenever the Oracle or the Tracker are initialized and is shared among them.
We used the Python library IPFS-Toolkit to connect and interact with the local node.

5.4 Tracker

The role of the tracker is crucial for the workflow of the frontend application. It updates the database shared with the Webapi server. The tracker must perform the following actions:

1. Listens events from the chain.
2. Updates the database keeping it consistent.

To do so, we use the web3 API library [13], which allows us to connect to a smart contract and filter events generated from it.

```
filters = [
 contract.events.PacketForged.create_filter(fromBlock="latest"),
 contract.events.PacketOpened.create_filter(fromBlock="latest"),
 contract.events.CreateImage.create_filter(fromBlock="latest"),
 contract.events.PromptCreated.create_filter(fromBlock="latest"),
 contract.events.ImageCreated.create_filter(fromBlock="latest"),
 contract.events.DestroyImage.create_filter(fromBlock="latest"),
 contract.events.PromptTransfered.create_filter(fromBlock="latest"),
 contract.events.PacketTransfered.create_filter(fromBlock="latest"),
 contract.events.CardTransfered.create_filter(fromBlock="latest"),
 contract.events.UpdateListPrompt.create_filter(fromBlock="latest"),
 contract.events.UpdateListPacket.create_filter(fromBlock="latest"),
 contract.events.UpdateListImage.create_filter(fromBlock="latest"),
]
```

Then we handle each event differently according to their implementation using custom classes. We fetch the right classes using this fancy *pythonic* script, that makes use of reflections:

```
def handle_event(event, provider, contract, IPFSClient, data, logger):
    event_name = event.event
    event_args = event.args
    event_class = get_event_class(event_name)
    kwargs = dict(event_args)
    kwargs['event'] = event_name
    event_object = event_class(**kwargs)
    event_object.handle(contract, provider, IPFSClient, data)
    event_object.log(logger)
```

Through the "data" object (of type Data) we perform actions on the database:

```
class Data:
    def __init__(self, create_db: bool=False):
        self.con = DatabaseConnection(create_db)
        if create_db:
            cd = CreateDatabase(self.con)
            cd.create()
    def get_cursor(self):
```

```
8             return self.con().cursor()
9         # more than 600 lines of methods...
```

5.5 Webapi

The Webapi server is just a facilitation to observe the state of user's transactions, it exposes endpoints to the frontend application sending the latest data. We implemented it using FastAPI framework [14] and wrote the documentation of the APIs with Swagger editor using Openapi 3.0 standard [16].

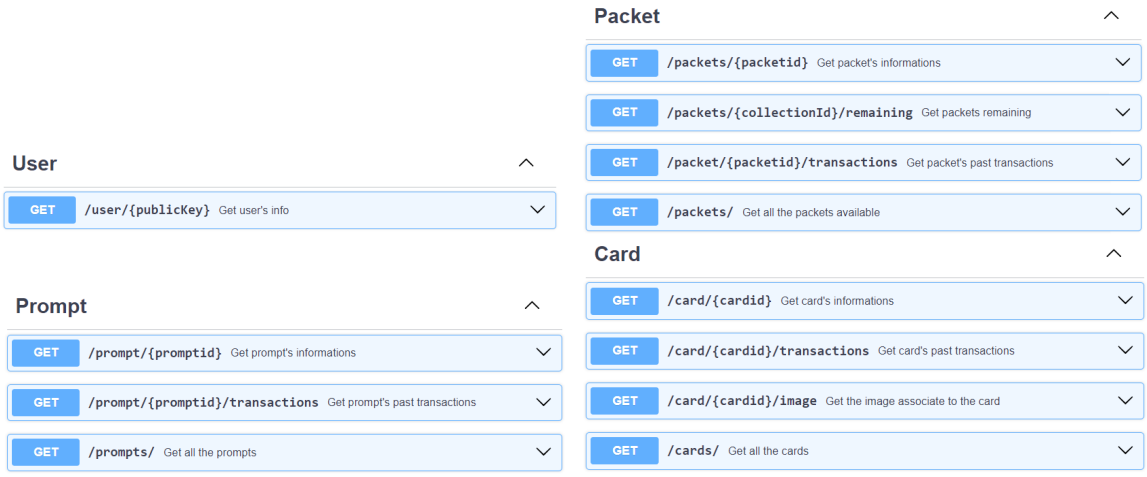

Figure 12: API endpoints

5.6 Frontend

We used Svelte[17] as a platform to develop our frontend and ether.js[12] to connect and interact with the blockchain.

5.6.1 User's Perspective

Understanding the user experience is crucial for the success of any application. In the case of MetaFusion, we aim to provide a seamless and secure interaction for users engaging with our blockchain-based NFT platform.

5.6.2 Onboarding and Authentication

Upon entering the MetaFusion ecosystem, users start by connecting their wallets, such as Metamask, to the frontend. This initial step ensures a secure and personalized experience for each user.

5.6.3 Packet Minting

The core functionality of MetaFusion lies in the creation and exchange of digital assets, primarily in the form of packets. Users can mint new packets by interacting with the MetafusionPresident smart contract. The process involves blockchain transactions, and the resulting packets become part of the user's collection.

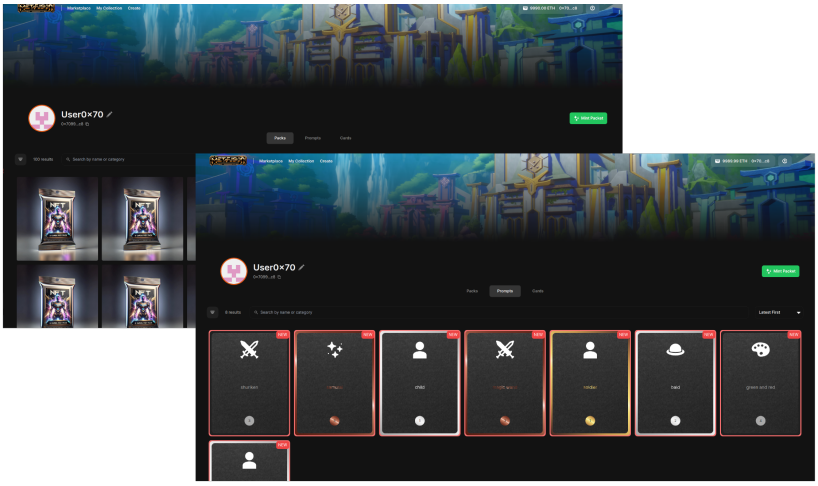

Figure 13: A view of the Packets and Prompts owned by a user.

5.6.4 Prompt Exploration and Card Creation

After acquiring a packet, users can open it to reveal prompts, which serve as the foundation for creating unique cards. The process of exploring prompts and combining them to generate cards is facilitated through user-friendly interactions on the frontend. Each step in this creative process is recorded on the blockchain, providing transparency and traceability.

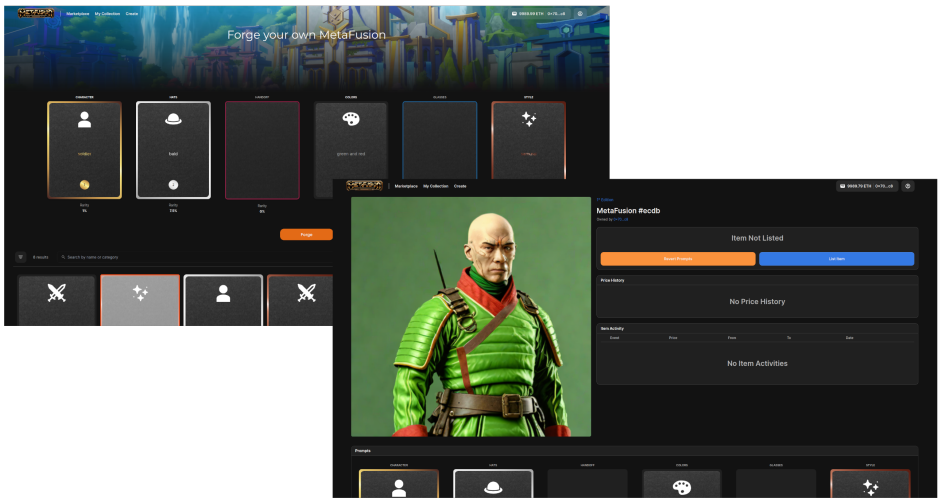

Figure 14: Creation of an image

5.6.5 Asset Listing and Trading

MetaFusion enables users to list their digital assets, including packets, prompts, and cards, for sale on the marketplace. The user-friendly interface of the frontend allows for easy management of listings, setting prices, and facilitating transactions. The events emitted during these actions are recorded on the blockchain and broadcasted through the system, ensuring all participants are informed.

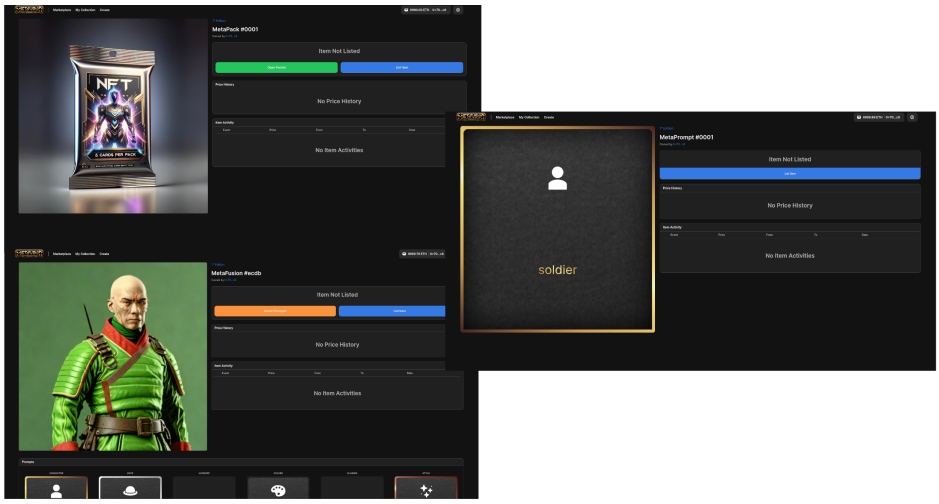

Figure 15: Closeup of our Tokens

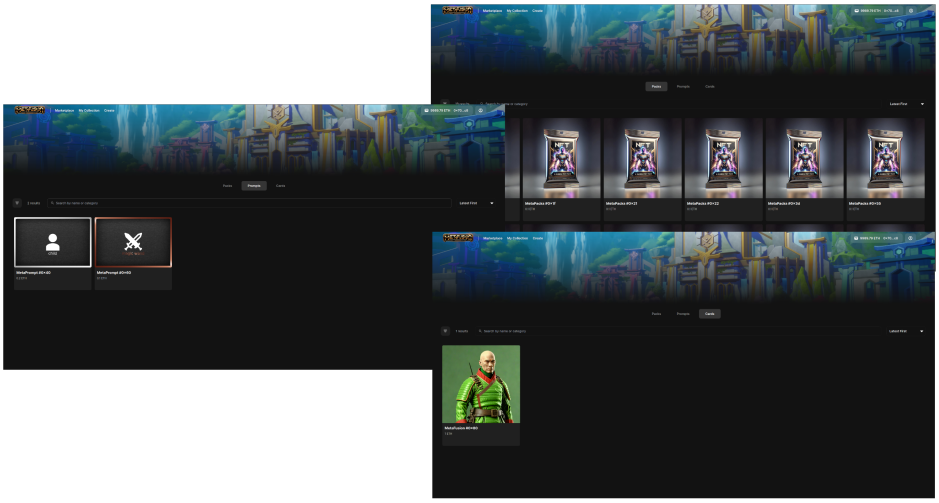

Figure 16: Our marketplace

5.6.6 Interaction with the Oracle

While users may not directly interact with the Oracle, its role in ensuring the generation of Cards and Prompts significantly impacts the user experience. Users benefit from the reliable and transparent generation of assets, contributing to the overall trustworthiness of the MetaFusion platform.

5.6.7 Real-time Updates via the Tracker

The Tracker plays a crucial role in providing users with real-time updates on blockchain events. Users can track the lifecycle of their assets, from creation to trade, through the frontend. The Tracker ensures that users have access to the latest information without the need for constant interaction with the blockchain.

5.6.8 Web API Integration

The Web API serves as the bridge between the centralized database and frontends, enabling efficient retrieval of user and asset information. Through REST API endpoints, the frontend seamlessly interacts with the cached data, offering a smooth experience for users.

6 Known Issue and Limitations

Lack of Effective Recovery System: Despite the possibility of implementing it, we have not developed an effective recovery system to reset the state in case the servers stop. This could potentially lead to issues in maintaining the system's integrity and user data in the event of unexpected server shutdowns.

Oracle Downtime Impact: If the Oracle experiences downtime, users may not immediately receive their images or prompts. This downtime could disrupt the timely generation and delivery of digital assets, affecting the user experience.

Fixed Fee Structure Concerns: The use of constant fees may result in pricing challenges for users or pose sustainability issues for the project. The generation of images and the maintenance of infrastructure can incur significant costs, and a fixed fee structure might lead to prices that are either too high for users or insufficient to cover project expenses. Further research and analysis are needed to determine an optimal fee structure.

Quality of Generated Images: The generated images may not always meet users' expectations in terms of aesthetics, as they might lack certain details. Improving the quality of generated images remains a challenge, requiring ongoing efforts to enhance the algorithm and update the model used for generation as soon as a new one is released.

7 Conclusions

In conclusion, MetaFusion represents a dynamic and engaging ecosystem for the creation and exchange of unique digital assets through the use of blockchain technology, specifically the ERC721 protocol of Ethereum. The platform provides users with the opportunity to explore their creativity by combining prompts to generate unique NFTs. The choice of Ethereum as the foundation underscores the commitment to transparency and community participation. However, the project faces some challenges, such as the need to improve the quality of generated images, optimize fee structures, and address potential service disruptions. MetaFusion aims to transform the NFT experience, offering an inclusive, stimulating environment open to the participation of a broad and diverse community of digital enthusiasts and investors.

References

[1] Stability.ai. Introducing sdxl turbo: A real-time text-to-image generation model. https://stability.ai/news/stability-ai-sdxl-turbo, 2023.

[2] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

[3] Erc20. https://docs.openzeppelin.com/contracts/4.x/erc20.

[4] Erc721. https://docs.openzeppelin.com/contracts/2.x/api/token/erc721.

[5] Erc721enumerable. https://docs.openzeppelin.com/contracts/2.x/api/token/erc721IERC721Enu

[6] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2023.

[7] Script. https://en.bitcoin.it/wiki/Script, 2023.

[8] Jing Chen and Silvio Micali. Algorand, 2017.

[9] Karl Wüst and Arthur Gervais. Do you need a blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54, 2018.

[10] Hardhat. https://hardhat.org/, 2023.

[11] Metamask. https://metamask.io/.

[12] Ether js library. https://docs.ethers.org/v5/, 2023.

[13] Web3 api library documentation. https://web3py.readthedocs.io/, 2023.

[14] Fast api framework. https://fastapi.tiangolo.com/, 2023.

[15] Ipfs. https://ipfs.tech/.

[16] IBM. Openapi 3.0 specifications. https://www.ibm.com/docs/it/app-connect/12.0?topic=apis-op 2023.

[17] Svelte kit. https://kit.svelte.dev/, 2023.

[18] Chai assert library. https://www.chaijs.com/, 2023.

[19] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger.