UNIVERSITÀ DEGLI STUDI DI UDINE
HIC SUNT FUTURA

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE

TESI DI LAUREA IN
INTERNET OF THINGS, BIG DATA E MACHINE LEARNING

Progettazione e sviluppo di una soluzione serverless per l'esecuzione di smart contract in ambito musicale

CANDIDATO
Thomas Arghittu

RELATORE
Prof. Vincenzo Della Mea

TUTOR AZIENDALE
Paolo Casoto

Anno accademico 2023-2024

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

© 2024 Thomas Arghittu
This work is shared under the Creative Commons 4.0 License Attribution-NonCommercial-ShareAlike.

Ai miei cari nonni, perché mi hanno cresciuto ed ora sarebbero orgogliosi di me.

Sommario

L'avvento negli ultimi anni del *Web3*, la nuova era del Web, che mira a creare una rete in cui lo scambio di informazioni avviene in modo trasparente e affidabile, ha introdotto nuove tecnologie fondate su alcuni principi chiave come la decentralizzazione, la privacy, il possesso e lo scambio di asset digitali e la gestione dell'identità online senza intermediari.

Il presente elaborato si concentra sull'esplorazione della **blockchain**, la tecnologia pilastro fondamentale su cui si basa il Web3. In particolare, è presentata un'applicazione pratica nel settore della gestione dei biglietti per eventi musicali. Il sistema, basato su *Smart Contract*, sfrutta gli **NFT** (Non-Fungible Token) per garantire l'autenticità e la tracciabilità dei biglietti associati, offrendo agli utenti un servizio trasparente e affidabile.

Nell'analisi dell'implementazione, emergono i pro e contro sia della blockchain stessa che del confronto con i sistemi tradizionali, in modo da:

1. contribuire alla comprensione di potenzialità e/o limiti derivanti
2. offrire spunti per futuri sviluppi o adozioni innovative.

Indice

Elenco delle tabelle

Elenco delle figure

1

Introduzione

1.1 Contesto di interesse generale

Il **Web3**, o Web3.0, è il nome conferito all'ultima generazione di Internet, che promette di rivoluzionare il Web come lo conosciamo oggi. Originariamente, la nascita del Web moderno risale al 6 agosto 1991, quando è stata pubblicata online la prima pagina Web della storia. In questa prima fase, nel **Web1.0**, i siti e le pagine potevano essere pubblicati solo da esperti con particolari capacità tecniche ed economiche. Da questa problematica nasce l'idea del **Web2.0**, un'evoluzione che rende accessibile la pubblicazione di contenuti a qualsiasi persona dotata di almeno un dispositivo e una rete in grado di connettersi alla rete Internet. Questa è stata la più grande rivoluzione dei primi anni duemila: ha fatto la fortuna degli attuali più grandi colossi tecnologici, come *Google*, *Facebook* o *Amazon*, portando ad una topologia del Web fortemente centralizzata. Se da un lato è stato grazie a queste multinazionali che oggi è possibile usufruire di servizi digitali di qualsiasi tipo (spaziando dall'intrattenimento, alla divulgazione, etc.), dall'altro lato si sta affrontando una delle più grandi problematiche degli ultimi anni, ovvero la scarsa **privacy** degli utenti finali dovuta ad una bassa trasparenza nella gestione e nella monetizzazione dei loro dati sensibili.

Con la necessità quindi di spostare il controllo e il monopolio della rete a sempre più persone, negli ultimi anni il Web3, seguendo determinati principi[2] [4] , cerca di assicurare che tutte le informazioni e i dati che circolano siano sempre sotto il controllo degli utenti che li hanno prodotti, donando così al Web una struttura **decentralizzata**. Una delle tecnologie che hanno reso possibile la realizzazione di questo modello è la **blockchain**, che grazie alle sue potenzialità permette di sviluppare applicazioni e sistemi in grado di soddisfare i requisiti del Web3.

L'impatto che il Web3 può avere sulla popolazione è significativo, in quanto la responsabilità del mantenimento delle proprie informazioni è legato all'utente e non delegato a terzi, il che prevede maggiore attenzione e meno comfort (almeno nella prima fase) legati all'esperienza utente dei servizi.

Questi aspetti fondamentali garantiscono quindi un sistema sicuro, resistente ad attacchi maligni, che allo stesso rispetta la privacy di ogni utente e che non ha bisogno di terze parti per poter essere gestito.

1.2 Definizione del problema e degli obiettivi

Ciò che questa tesi propone è un analisi, sia dal punto di vista implementativo che dal punto di vista prestazionale, di una *dAPP*, ovvero una applicazione distribuita sviluppata sopra la blockchain, seguendo i principi del Web3. Nello specifico si tratta di un'estensione di un classico portale per la vendita di biglietti di eventi (concerti, teatri, mostre, etc.), che permette a tutti gli utenti della piattaforma di ottenere dei collezionabili al momento dell'acquisto di un biglietto, ovvero degli oggetti virtuali (paragonabili alle figurine) conservati all'interno di un **wallet** virtuale e personale che permettono l'accesso, secondo determinate regole e condizioni, a benefici utilizzabili all'interno della piattaforma stessa (quali sconti per eventuali acquisti futuri, etc.). Ogni collezionabile, chiamato **TicketNFT**, è unico (all'interno della blockchain ospitante l'applicazione), corrisponde all'evento specifico con il quale è stato ottenuto ed è assegnato ad un proprietario che, secondo la propria volontà, può decidere di scambiarlo con un altro utente per accedere a benefici differenti. Anche i benefici ottenibili sono collezionabili, chiamati **BenefitNFT**, non interscambiabili.

Nella prima parte del presente lavoro è fornita una panoramica di tutti gli aspetti teorici legati all'applicazione, in modo da avere tutte le conoscenze per poter seguire la sezione riguardante gli aspetti realizzativi (come scelte progettuali, pro e contro di diverse soluzioni, ecc.) e per saper interpretare le analisi condotte su costi e performance. La parte finale contiene alcuni schermate del prototipo sviluppato durante l'attività di tirocinio svolta dal candidato.

2

Aspetti teorici

2.1 Web3 e tecnologie blockchain

In questa sezione viene fornita una panoramica completa del Web3, approfondendo i concetti chiave, le tecnologie sottostanti e le potenzialità.

2.1.1 Sistemi distribuiti e decentralizzazione

Il concetto di **sistema distribuito** è un paradigma informatico che sta alla base del Web3. Si occupa della progettazione e dell'implementazione di sistemi composti da molteplici **nodi indipendenti** che collaborano tra di loro per raggiungere un obbiettivo comune. Questi nodi possono essere dislocati in diverse località geografiche e interagiscono tramite una **rete di comunicazione**.

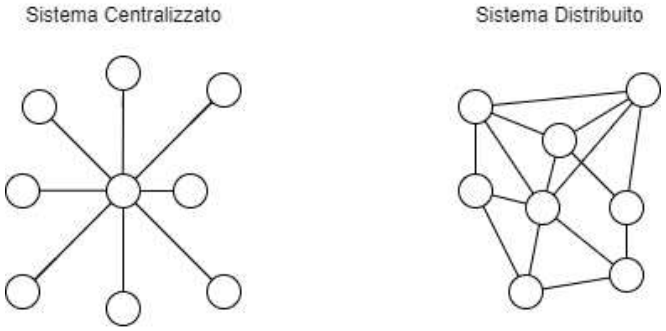

Figura 2.1: Sistema centralizzato e sistema distribuito a confronto

Decentralizzazione

La **decentralizzazione** rappresenta una caratteristica distintiva dei sistemi distribuiti, ovvero l'assenza di una autorità centrale in grado di controllare gli altri nodi connessi alla rete, garantendo una distribuzione del controllo e della responsabilità. I vantaggi della decentralizzazione sono:

- **Aumento della robustezza** del sistema, resistente a guasti o malfunzionamenti di singoli nodi;
- **Miglior scalabilità** che rende possibile aggiungere nuovi nodi per aumentare le potenzialità del sistema (come potenza di calcolo o disponibilità delle risorse);

- **Condivisione delle risorse**, sia di tipo hardware o software.

Trasparenza

Per migliorare l'usabilità di un sistema distribuito, è fondamentale che l'utente non percepisca che i suoi processi e le sue risorse sono situati in luoghi diversi. Ci sono principalmente due tipi di trasparenza che riguardano gli utenti:

- di **accesso**: per poter accedere alle risorse indipendentemente dalla loro locazione o dal nodo in sui sono distribuite;
- di **replicazione**: in cui non sono consapevoli della ridondanza dei dati e della esecuzione delle operazioni in modo coerente, indipendentemente dal nodo a cui vengono inoltrate.

Scalabilità

Un altro aspetto fondamentale è la **scalabilità**, ovvero la possibilità di far crescere il sistema aggiungendo nuovi nodi senza la compromissione delle prestazioni. Ci sono diversi modi per poter misurare la scalabilità:

- rispetto alla **dimensione**, legata al numero di utenti e risorse connessi;
- rispetto alla **posizione geografica**, riferendosi al fatto che utenti e risorse possono essere fisicamente su luoghi parecchio distanti tra di loro.

Problematiche legate ai sistemi distribuiti

L'adozione di questo paradigma, tuttavia, ha sollevato numerose sfide dovute alle problematiche incontrate durante la progettazione di tali sistemi, che rimangono ancora irrisolte.

- **Algoritmi distribuiti**: la diversa natura topologica, rispetto ai classici sistemi centralizzati, richiede lo sviluppo di nuovi algoritmi per la gestione di risorse (come la ricerca o lo scambio di dati), che devono mantenere performance e complessità accettabili al fine di rendere i sistemi distribuiti usabili;
- **Comunicazione**: la connessione in reti estese è intrinsecamente inaffidabile. Diverse problematiche, come la sincronizzazione dei nodi e la latenza di comunicazione dovuta ai mezzi trasmissivi, rendono difficile la coordinazione di operazioni tra nodi, il che prevede di implementare protocolli che introducono *overhead*.

Sistemi distribuiti e Web3

Il Web3 quindi cerca di evolvere la rete odierna seguendo il paradigma dei sistemi distribuiti, supportando applicazioni in grado di operare in maniera dislocata, garantendo un accesso alle informazioni in qualsiasi parte del pianeta. Esempi di tipologie di applicazioni sono:

- **Finanza decentralizzata** (DeFi): piattaforme finanziarie *peer-to-peer* che consentono transazioni senza intermediari (E.G.: senza istituti bancari che gestiscono i movimenti);

- **Organizzazioni autonome decentralizzate** (DAO): strutture di *governance* che consentono agli utenti di prendere decisioni collettive (E.G.: sulla gestione di organizzazioni o progetti);

- **Mercati peer-to-peer**: piattaforme per lo scambio diretto di beni e servizi tra utenti

2.1.2 Blockchain

La tecnologia innovativa che si trova alla base del Web3 è la **blockchain**. Si tratta di un registro dati composto da blocchi, ovvero **record di informazioni**, collegati in successione tra di loro in modo sicuro tramite l'utilizzo della **crittografia**. La caratteristica principale di questo sistema è l'immutabilità dei dati, ovvero la garanzia, derivata dal carattere tecnologico con cui è implementata, di avere informazioni prive di modifiche non consensuali tra gli utenti della blockchain.

Rientra nella famiglia dei **registri distribuiti** (*distributed ledger*), ovvero sistemi distribuiti basati sulla condivisione (tramite la replicazione) di un registro dati sui nodi (dislocati geograficamente). Ogni record salvato su un blocco è chiamato **transazione** e rappresenta una qualsiasi modifica dello stato della blockchain. La replicazione massiva di questo registro permette di mantenere la qualità dei dati nonostante non ci sia nessuna copia "ufficiale" centralizzata, mantenendo tutti i nodi allo stesso livello di importanza. La creazione di un nuovo blocco è quindi accettata in modo democratico dall'intera rete durante un processo detto **mining** (vedi sez. *Validazione dei blocchi 2.1.2*) e ciascun tentativo di modifica del registro da parte di nodi malintenzionati è rilevabile facilmente (in quanto per essere credibili questi dati dovrebbero essere modificati in tutte le copie di ciascun nodo della rete).

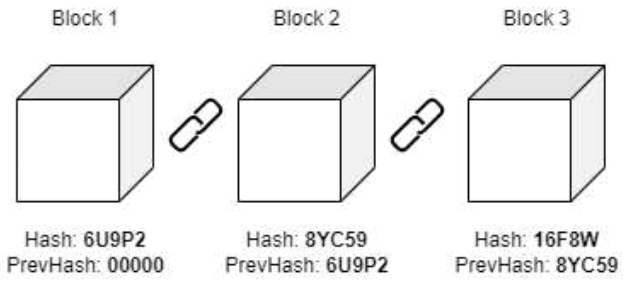

Figura 2.2: Rappresentazione Blockchain

Il primo blocco è comunemente chiamato **genesi**: è l'origine della catena e normalmente non contiene transazioni al suo interno, ma informazioni legate alla blockchain (come data e ora di creazione, messaggio o dichiarazione degli sviluppatori, eventuali parametri della blockchain, etc.). I blocchi successivi invece, come mostrato in *figura 2.2*, mantengono un riferimento a quello che li precede salvandosi l'identificativo tra i propri metadati (vedi sez. *Funzione di hash e impronte digitali 2.1.2*).

Funzione di hash e impronte digitali

Nel campo della crittografia, una **funzione di hash** è un algoritmo che dato un input di lunghezza arbitraria restituisce in output una stringa di lunghezza fissa, chiamata *hash digest* o *impronta digitale*. La funzione di hash ha diverse proprietà che la descrivono:

- **Irreversibilità**: data una impronta digitale, è computazionalmente difficile risalire al suo input (in altre parole è impossibile ricostruire il messaggio originale o, come in questo caso, al blocco sui cui viene calcolata);
- **Collisioni**: nonostante la funzione di hash sia **suriettiva** (in quanto il dominio ha cardinalità infinita e il codominio finita), la probabilità che due input distinti producano la stessa impronta digitale (che si verifichi quindi una collisione) è bassa. La percentuale è calcolata su parametri come lunghezza delle stringhe di output (normalmente 128, 256 o 512 bit);
- **Efficienza**: in termini di complessità e utilizzo delle risorse del calcolatore, è una funzione efficiente da calcolare per ciascun tipo di input;
- **Determinismo**: dato un determinato input, la funzione deve sempre restituire lo stesso output.

Formalmente, la funzione di *Hash* viene descritta in questo modo:

$$Y_m = H(x)$$

dove $H(m)$ dipende dalla funzione scelta. Il più famoso algoritmo di hash standardizzato è lo **SHA (Secure Hash Algorithm)** ed è presente in diverse versioni, tra cui *SHA-1, SHA-256* o *SHA-512*, ognuna con diversi livelli di sicurezza e velocità di elaborazione.

Nelle blockchain, la sicurezza e robustezza del sistema sono garantite proprio da questa tipologia di algoritmi: durante la fase di *mining*, la funzione di hash viene calcolata processando in input l'intero blocco, producendo una stringa che rappresenta l'identificativo del blocco stesso. L'output quindi è influenzato anche dal riferimento (o hash digest) del blocco precedente, che a sua volta è influenzato da quello prima ancora e così via, a catena. In *figura 2.2* il meccanismo viene messo in evidenza ponendo enfasi sui campi **Hash** e **PrevHash**: modificando, anche in minima parte, un solo blocco della catena, anche tutti i successivi risulterebbero modificati rendendo visibile il tentativo di manomissione.

Struttura del blocco

Un blocco è composto da due componenti principali: **header** e **body**. Nel primo, ci sono salvati tutti i metadati per la gestione del blocco stesso:

Versione	02000000
Hash del blocco precedente (PrevHash)	E87C17C45768w7e1643fsd5481sd3f4131df681
Merkle root	697we168t4v1a4rv3v1e3r43c4er14ca8c4168a
Timestap	358b0553
Bits	535f0119
Nonce	48750933
Numero di transazione	64

Tabella 2.1: Esempio Header di un blocco della Blockchain

Il campo *Versione* indica la versione del software utilizzato, il campo *PrevHash* contiene l'*hash digest* di 256 bit del blocco precedente, il campo *Merkle root* rappresenta l'*hash* di tutti gli *hash* delle transazioni (vedi sez. *Alberi di Merkle 2.1.2*), il *Timestamp* rappresenta il *time-stamp* dell'ultima transazione (in

formato *Unix hex*), i campi *Bits* e *Nonce* che rispettivamente contengono dei valori di hash target **minimo** e **calcolato** (utilizzati nell'algoritmo di consenso per la creazione di nuovi blocchi, vedi sez. *Validazione dei blocchi 2.1.2*) e infine il *Numero di transazione* che identifica il numero di transazione del blocco. Tutte le transazioni vere e proprie invece rientrano nella parte del body. Il numero varia in base alle dimensioni di ogni transazione stessa che dipende a sua volta dal numero di input e output.

Alberi di Merkle

Sempre sotto l'aspetto della sicurezza dei dati, gli **alberi di Merkle**[5] giocano un ruolo fondamentale, questa volta a livello di blocco. Si tratta di una struttura dati ad albero in cui le foglie contengono l'*hash digest* di una singola transazione (notare che le transazioni stesse non fanno parte di questo albero). Ogni nodo interno invece, mantiene l'*hash digest* della somma dei propri figli.

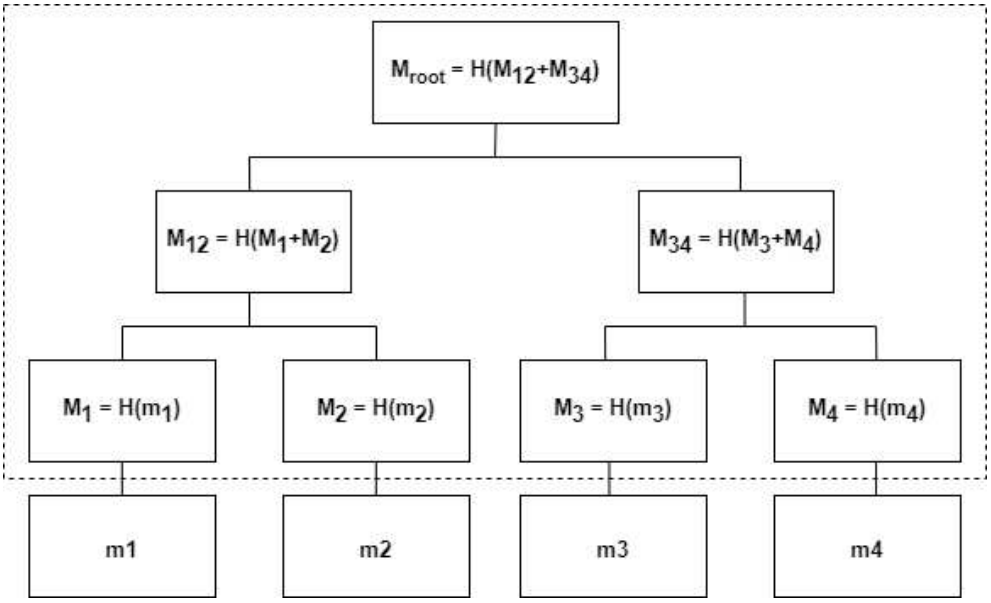

Figura 2.3: Rappresentazione albero di Merkle

Il nodo radice (o *Merkle root*) funge quindi da *impronta digitale* di tutte le transazioni contenute e risulta utile nel verificare se una transazione appartiene o meno ad un determinato blocco o se è stata modificata o meno (in particolare grazie agli *alberi di Merkle* vengono ridotti i dati necessari alla verifica). Preso come riferimento il modello della figura *2.3*, se si volesse dimostrare ad esempio che la transazione *m3* non è stata modificata, basterebbe conoscere i valori dei nodi *M4* e *M12*, ricostruire il *Merkle root* e confrontarlo con quello contenuto nell'header del blocco. Formalmente, per verificare una singola transazione in un blocco con n transazioni, servono solo $log_2(n)$ comparazioni [6].

Validazione dei blocchi

Essendo la blockchain un **sistema dinamico**, ovvero in cui lo stato può variare continuamente, le nuove transazioni (che descrivono la modifica dei dati) sono salvate su nuovi blocchi. Come anticipato nelle sezioni precedenti, il processo di creazione di blocchi è chiamato **mining** e coinvolge diversi nodi della rete. Diverse blockchain utilizzando diversi metodi di mining: la scelta dipende dagli sviluppatori stessi. Principalmente, esistono due tipi di algoritmi del consenso: **proof-of-work** e **proof-of-stake**.

Il primo, abbreviato con *POW*, è un algoritmo in cui i nodi della rete, detti **miner**, competono per risolvere un problema matematico complesso:

1. **Creazione del pseudo-blocco**: ogni *miner* all'inizio del processo sceglie un tot numero di transazioni (tra quelle pendenti) da inserire nel blocco e aggiorna i campi dell'header;

2. **Risoluzione del problema**: ognuno cerca di trovare un valore da inserire nel campo *Nonce* (vedi sez. *Struttura del blocco 2.1.2*) che, combinato con i restanti dati del blocco, è usato per calcolare la funzione di *hash* il cui output soddisfa certe condizioni di difficoltà. Solitamente queste condizioni possono essere un certo numero di zeri iniziale oppure trovare un *hash digest* inferiore al campo *Bits* (sempre dell'header);

3. **Prova del lavoro**: il primo miner che trova un *Nonce* che risolve il problema, trasmette il blocco alla rete. Gli altri nodi possono verificare facilmente che il *Nonce* proposto risolve effettivamente il problema;

4. **Aggiunta alla blockchain**: nel caso in cui il blocco sia valido, viene aggiunto alla blockchain (e il miner riceve una ricompensa, vedi sez. *Miners e auto-sostenimento 2.1.2*). In caso contrario, il processo riprende dal punto *2.*

Questo metodo utilizza la forza bruta per cercare un valore che possa soddisfare i requisiti, in quanto è computazionalmente impossibile trovare quel valore partendo direttamente dal *hash digest* in tempi ragionevoli. I vantaggi sono la **sicurezza**, in quanto è difficile per un singolo nodo ottenere il controllo della maggioranza della potenza di calcolo (51%), e la **decentralizzazione**, in quanto non richiede alcun ente centrale. La questione critica, invece, risulta essere il **consumo energetico**, in quanto sono richieste grandi quantità di energia che portano ad un impatto ambientale significativo. Inoltre, le transazioni per secondo (**TPS**) sono limitate dato la lentezza di queste operazioni.

Il secondo algoritmo invece, abbreviato con *POS*, sceglie i validatori in base alla quantità di denaro che possiedono e mettono in gioco (*stake*):

1. **Selezione dei validatori**: più denaro è messo in gioco, più probabilità c'è di essere scelti come nodi validatori del nuovo blocco di transazioni. Questo è dovuto alla filosofia secondo cui chi investe di più in un progetto blockchain contribuisce maggiormente alla sua crescita, garantendole una maggiore longevità, meritando quindi ricompense per il proprio impegno;

2. **Creazione e validazione**: il nuovo validatore scelto crea un nuovo blocco e lo propone alla rete;

3. **Verifica**: gli altri nodi verificano il blocco proposto. In caso di validità viene aggiunto alla catena e il validatore riceve anche qui una ricompensa.

Questo algoritmo mira a migliorare il grande problema dell'impatto ambientale del precedente metodo, in quanto non richiede grossi consumi energetici. È inoltre un sistema maggiormente **scalabile**, in quanto riesce a gestire più transazioni al secondo e aumenta i numeri di nodi partecipati (dovuti alla diminuzione di necessità di hardware costoso per competere con gli altri) favorendo la **decentralizzazione**. La scelta dei validatori però, potrebbe cadere sempre tra i nodi più ricchi portando ad una sorta di **accentramento del potere**.

Miners e auto-sostenimento

Non essendoci un ente centrale interessato a gestire e mantenere tutta l'infrastruttura di una blockchain, esiste un meccanismo di auto-sostenimento in cui i *miner* sono incentivati a sostenere la rete, ovvero le **ricompense**. Vengono offerte sotto forma di **criptovalute**, ovvero delle valute virtuali che operano proprio su blockchain e che hanno acquistato valore economico negli ultimi anni: l'ammontare è costituito da una quantità predeterminata e dalle commissioni associate alle transazioni incluse nel blocco. I validatori quindi, in fase di *mining*, hanno interesse a scegliere transazioni pendenti con delle *fee* maggiori, mentre gli utenti, nel caso in cui volessero far approvare il prima possibile una transazione, dovranno pagare di più.

Questo meccanismo garantisce che ci siano sempre nodi interessati a validare le transazioni e ad aggiungere nuovi blocchi, assicurando la continuazione e l'integrità della blockchain senza necessità di un'autorità centrale.

Wallet

I **wallet** di una blockchain sono strumenti digitali che permetto agli utenti di gestire le proprie criptovalute o altri asset digitali. Permettono di ricevere, inviare e monitorare le proprie transazioni, fungendo da end-point per l'interazione umana. Sono costituiti da una coppia di chiavi crittografiche:

- **Indirizzo pubblico**: la chiave condivisa per la ricezione di fondi e asset (paragonabile al numero di conto bancario);
- **Indirizzo privato**: la chiave segreta che consente di accedere e gestire il wallet associato all'indirizzo pubblico. Deve essere mantenuta nascosta e al sicuro in quanto chiunque la possiede può accedere ai dati e in caso di smarrimento la possibilità di recuperarla grazie a esterni non esiste.

Possono esistere inoltre due tipi di *wallet*: **Hot Wallet**, connessi a Internet più convenienti per l'uso quotidiano, e **Cold Wallet**, non connessi e ideali per il lungo termine. Non contengono effettivamente le criptovalute o gli asset stessi, ma solamente la coppia di chiavi e qualche altra informazioni utile per la gestione delle risorse, come ammontare del saldo, etc.

I wallet interagiscono con la blockchain attraverso diverse operazioni:

1. **Creazione e Gestione delle Transazioni**: ad esempio, quando un utente vuole inviare criptovalute il wallet crea una transazione firmata digitalmente utilizzando la chiave privata. Questa transazione è poi trasmessa alla rete blockchain per essere validata e inclusa in un blocco.
2. **Monitoraggio delle Transazioni**: i wallet permettono agli utenti di monitorare lo stato delle loro transazioni. Questo è fatto interrogando la blockchain per vedere se una transazione è stata confermata e in quale blocco è stata inclusa.

Blockchain nel mondo reale

Bitcoin è probabilmente la blockchain più famosa, nonché la prima ad aver dato il via a tutte le altre. Fondata nel 2009 da un autore tuttora ignoto, il cui nome d'arte è *Natoshi Nakamoto*, adotta un

algoritmo del consenso **proof-of-work** ed è stata progettata principalmente per essere una riserva di valore e un mezzo di scambio monetario decentralizzato usufruibile da ogni parte del mondo [8].

Ethereum invece, fondata nel 2015 da *Vitalik Buterin*, si propone come alternativa eco-friendly a *Bitcoin* utilizzando (da qualche anno) un algoritmo del consenso **proof-of-stake**. Inoltre, è stata tra le prime blockchain a fare da supporto alle **DApps** (applicazioni decentralizzate) ed a permettere l'esecuzione di **Smart Contract**[1] (vedi sez. *Blockchain Ethereum e Smart Contract 2.1.3*).

2.1.3 Blockchain Ethereum e Smart Contract

Gli **Smart Contract** rappresentano un'evoluzione fondamentale nel campo della tecnologia blockchain, consentendo l'automazione di accordi contrattuali attraverso codice eseguibile. Si tratta di programmi informatici che garantiscono agli utenti coinvolti la **certezza di una corretta esecuzione** delle condizioni a cui si ha acconsentito. L'inserimento degli *Smart Contract* in una blockchain conferisce loro l'**immutabilità** del codice, la **trasparenza** e la **non interrompibilità** dell'esecuzione. In questa sezione viene fatta una panoramica su **Ethereum**, ma i concetti trattati possono essere facilmente trasportati su tutte le diverse altre blockchain in grado di eseguire *Smart Contract*.

Ethereum possiede due tipi di *wallet*: i primi, chiamati **Externally Owned Accounts** (EOA), corrispondono a quelli posseduti dagli utenti (vedi sez: *Wallets 2.1.2*) mentre gli altri, detti **Contract Accounts**, creati durante la fase di *deploy* del contratto stesso. Ogni account inoltre possiede uno stato in cui sono salvate diverse informazioni, come l'ammontare di *Ethers* (ETH) (la criptovaluta della piattaforma *Ethereum*) o il numero di transazioni create.

Virtual Machine

Per poter essere essere eseguiti, le blockchain che supportano questi *contratti intelligenti* sono dotate di un ambiente runtime, chiamato **Virtual Machine**. Gli *Smart Contract* vengono compilati in **bytecode** dal programmatore che li sviluppa e, una volta distribuiti, possono essere chiamati attraverso transazioni che alterano lo stato della blockchain. Normalmente operano in un ambiente isolato per prevenire interazioni indesiderate con altri contratti o dati sulla blockchain, mentre ogni operazione eseguita viene validata da ogni nodo della rete. La *Virtual Machine* è progettata per garantire che le operazioni siano eseguite in modo deterministico.

È composta da diversi componenti, tra cui aree di memoria **volatili**, come *stack* e *RAM* usate durante l'esecuzione delle operazioni per dati temporanei, e **persistenti**, come *storage* utilizzati per conservare informazioni permanenti.

Progettazione di Smart Contract

Il processo di sviluppo di uno *Smart Contract* non differisce molto dal classico approccio ingegneristico applicato ai software tradizionali. Tuttavia, data la loro natura immutabile, si deve porre maggior enfasi nella fase di **testing** in quanto la modifica e l'aggiornamento del codice risulta molto verboso.

1. **Definizione dei requisiti**: identificazione dell'obbiettivo, la raccolta dei requisiti (funzionali e non funzionali) e l'analisi dei casi d'uso;

2. **Progettazione e sviluppo del contratto**: progettazione, modellazione dei dati, gestione della sicurezza (molto importante in quanto si gestiscono dati sensibili) e implementazione della soluzione;

3. **Test e debugging**: fase critica in cui, attraverso i requisiti raccolti al punto 1, è verificato il corretto funzionamento logico e rilevati eventuali comportamenti anomali (come *bug*) o potenziali vulnerabilità per la sicurezza;

4. **Distribuzione sulla blockchain**: ultima fase, chiamata anche **deploy**, in cui avviene la distribuzione del *contratto intelligente* sulla blockchain principale (**mainnet**) o su una di test (**testnet**).

Esempio di codice Solidity

Il linguaggio principale per la scrittura degli *Smart Contract* è **Solidity**. Possiede i principali costrutti dei classici linguaggi di programmazione, offrendo la possibilità sia di astrarre funzionalità ad alto livello sia di agire su dettagli (come gestione della memoria) per il miglioramento delle performance.

- **Variabili di Stato**: Memorizzano i dati che persisteranno sulla blockchain.

- **Funzioni**: Blocchi di codice che eseguono specifiche operazioni e possono essere richiamati tramite transazioni.

- **Eventi**: Meccanismi che consentono ai contratti di comunicare con altre parti della blockchain, registrando importanti azioni per l'esterno.

- **Modificatori**: Funzioni speciali che alterano il comportamento di altre funzioni, ad esempio, per controllare i permessi.

Esempio di codice in *Solidity*:

```
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint256 public data;

    function setData(uint256 _data) public {
        data = _data;
    }

    function getData() public view returns (uint256) {
        return data;
    }
}
```

Questo codice Solidity definisce un contratto chiamato `SimpleStorage` che permette di memorizzare e recuperare un valore numerico.

- `data`: Una variabile pubblica di tipo `uint256` (numero intero non negativo a 256 bit) che contiene il dato memorizzato.
- `setData(uint256 _data)`: Una funzione pubblica che consente di impostare il valore di `data`.
- `getData()`: Una funzione pubblica di sola lettura che restituisce il valore attualmente memorizzato in `data`.

Gas e costi di esecuzione

Come per transazioni, anche il deploy e le chiamate a contratto hanno costi di esecuzioni per l'autosostenimento della blockchain. Entra in gioco il concetto di **gas** [7], un'unità che misura la quantità di calcolo necessaria per eseguire operazioni sulla rete: ogni operazione di un contratto richiede una certa quantità di *gas*. Gli utenti pagano utilizzando la criptovaluta della piattaforma che li ospita, compensando così i *miner* per le risorse computazionali fornite. La quantità di risorse impiegate e lo spazio occupato in memoria influenzano pesantemente il valore di *gas* della funzione, rendendo necessario un pesante processo di ottimizzazione da parte degli sviluppatori.

Il costo del *gas* (**gas price**) è un parametro che l'utente può regolare e indica quanto è disposto a pagare per **unità di gas**, determinando così la priorità della transazione che richiama lo *Smart Contract*. Un utente può impostare anche una quantità massima di gas (**gas limit**) che è disposto a spendere per una transazione singola (tutelandosi da eventuali sovrapprezzi).

Vantaggi e svantaggi Smart Contract

Gli *Smart Contract* offrono numerosi vantaggi, ma anche alcune sfide significative. Tra i pro, vi è l'automazione delle transazioni, che elimina la necessità di intermediari, riducendo così i costi e aumentando l'efficienza. La trasparenza e l'immutabilità delle blockchain garantiscono che siano eseguiti esattamente come programmati, aumentando la fiducia tra le parti coinvolte. Inoltre, la sicurezza crittografica riduce il rischio di frodi e manipolazioni.

Tuttavia, gli *Smart Contract* presentano anche alcuni contro. Uno dei principali svantaggi è l'irreversibilità delle transazioni, che può essere problematica in caso di errori nel codice del contratto. Le vulnerabilità nel codice possono essere sfruttate per attacchi, come dimostrato da incidenti passati nel settore. Infine, la regolamentazione è ancora in evoluzione, creando incertezze legali sull'utilizzo di massa di questa tecnologia.

Fungible Token & Non Fungible Token

Nelle blockchain, gli **assets digitali** oggetti a scambio tra utenti sono rappresentati da **token**. I token possono rappresentare criptovalute, punti fedeltà, diritti di proprietà, azioni di una società o persino oggetti unici come opere d'arte digitali. Possono essere divisi in due grandi famiglie, i **Fungible Token** (FT) ed i **Non-Fungible Token** (NFT).

I primi, rappresentano asset digitali completamente intercambiabili tra di loro, in quanto ogni unità di un *FT* ha lo stesso valore. Un esempio comune sono le criptovalute: nella blockchain *Ethereum*, la valuta *ETH* è definita tramite uno *Smart Contract* che ne descrive tutte le operazioni e funzionalità precedentemente descritte. Gli standard più comuni per gli *FT* su *Ethereum* sono **ERC-20** e **ERC-777**, che definiscono le regole per la creazione e il trasferimento di questi token, garantendo la compatibilità con i diversi wallet e piattaforme decentralizzate.

I *Non-Fungible Token* invece, sono asset digitali unici che non possono essere scambiati l'uno con l'altro in modo equivalente. Ogni *NFT* possiede informazioni specifiche che lo rendono unico, come metadati, immagini, video o altre forme di contenuti digitali. Sono particolarmente importanti nel mondo dell'arte digitale, dei collezionabili, dei giochi e dei beni immobili virtuali. Offrono una prova di proprietà e autenticità, rivoluzionando il modo in cui gli artisti e i creatori possono monetizzare le loro opere digitali, vendendo direttamente ai collezionisti senza intermediari. Inoltre, gli *NFT* possono essere programmati per includere le *royalties*, un meccanismo che garantisce agli artisti di ricevere una parte delle vendite future. Su *Ethereum*, gli standard più comuni sono **ERC-721** e **ERC-1155**. Le *opere d'arte digitale* sono chiari esempi di contenuti multimediali (come immagini, video, testi, etc.) rappresentati da token unici che ne garantiscono l'autenticità e la proprietà esclusiva dell'opera.

2.2 Architetture cloud

In questa sezione, si approfondiscono i concetti legati alle **architetture cloud** per la realizzazione di *backend* di applicazioni che sfruttano servizi remoti.

2.2.1 Panoramica sul Cloud Computing e tipologie di Cloud

Con il termine **Cloud Computing** si indica un modello di erogazione di servizi informatici che consente l'accesso su richiesta ad un insieme di *risorse* configurabili tramite la rete Internet. Esempi di queste risorse sono server, spazi di archiviazione, hardware per la potenza di calcolo, etc. Queste, sono fornite come servizi gestiti da fornitori esterni (**provider**) e possono essere scalate istantaneamente a seconda della domanda, differendo dal classico modello di computazione in cui le risorse hardware e software sono locali e gestite internamente da un'organizzazione.

Generalmente, il Cloud Computing offre 3 diversi tipi di servizi:

- **Infrastructure as a Service** (IaaS): fornisce risorse di calcolo di base come server virtuali, storage e reti, consentendo alle aziende di costruire e gestire le proprie infrastrutture *IT* senza l'onere di acquistare hardware fisico;
- **Platform as a Service** (PaaS): offre un ambiente di sviluppo e distribuzione che permette agli sviluppatori di creare applicazioni senza preoccuparsi della gestione dell'infrastruttura sottostante;
- **Software as a Service** (SaaS): offre applicazioni software complete accessibili tramite internet, eliminando la necessità per le aziende di installare e gestire le applicazioni sui propri dispositivi (ad esempio *Microsoft Office 365*).

I principali vantaggi offerti sono:

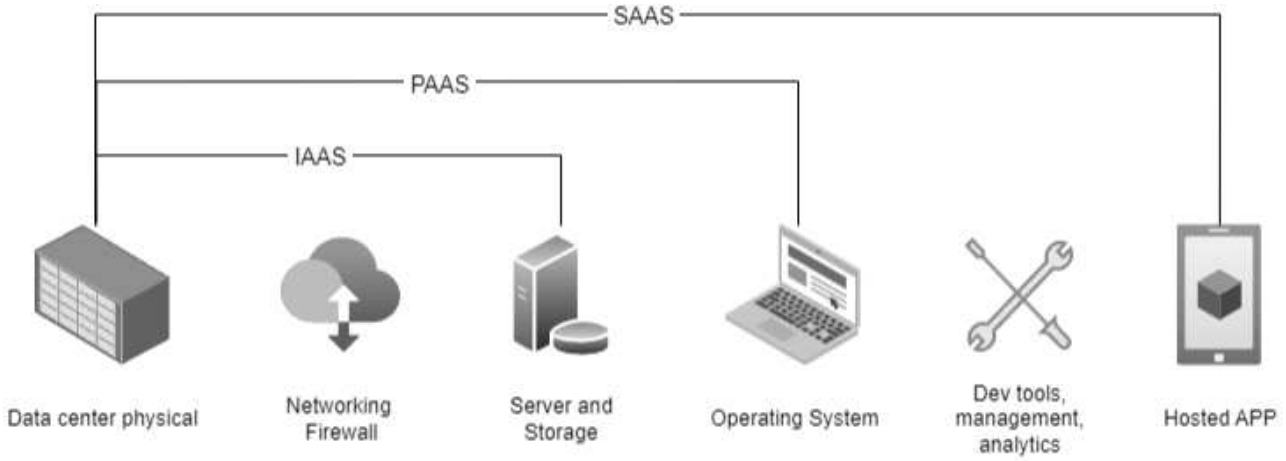

Figura 2.4: IaaS vs PaaS vs SaaS

- **Scalabilità**: che permette agli sviluppatori di aumentare o diminuire rapidamente le risorse in base alle necessità, evitando sovradimensionamenti o carenze (che impattano sul costo di sviluppo e manutenzione);
- **Elasticità**: derivante della scalabilità, che consente di adattare dinamicamente le risorse per la gestione di variazioni di carichi di lavoro;
- **Flessibilità**: la possibilità di accedere ai servizi cloud da qualsiasi luogo e dispositivo connesso a Internet, favorendo il lavoro remoto e la collaborazione globale;
- **Riduzione costi di sviluppo**: elimina la necessità di investimenti iniziali elevati in hardware e manutenzione, trasferendo invece le spese su un modello di pagamento basato sull'uso effettivo delle risorse.

L'adozione di questo modello nello sviluppo di un'applicazione richiede una sottoscrizione a un fornitore terzo che offra almeno l'infrastruttura necessaria. Tuttavia, le enormi infrastrutture necessarie hanno costi di creazione e manutenzione che superano le possibilità dei singoli sviluppatori, i quali devono quindi accettare questo compromesso.

2.2.2 Architettura Cloud a Microservizi

Si possono avere differenti approcci riguardo la progettazione di *backend* di applicazioni che sfruttano il Cloud Computing. Uno di questi è l'architettura a **microservizi**, in cui l'applicazione viene divisa in una serie di servizi indipendenti, ognuno dei quali che esegue compiti ben definiti comunicando con gli altri attraverso **API**. È un modello che offre diversi vantaggi rispetto ad uno *monolitico*, in quanto ogni microservizio può essere sviluppato e implementato indipendentemente migliorandone l'agilità e la velocità di implementazione. La separazione dei servizi inoltre, facilita la manutenzione e la fase di debugging, dato che le modifiche non influenzano necessariamente tutto il sistema.

Nel contesto del Cloud Computing, i microservizi possono sfruttare la scalabilità e l'elasticità del cloud adattando dinamicamente le risorse in base alle esigenze del carico di lavoro. Questo approccio migliora anche la **resilienza** dell'applicazione, in quanto il malfunzionamento di un singolo microservizio non compromette l'intera applicazione.

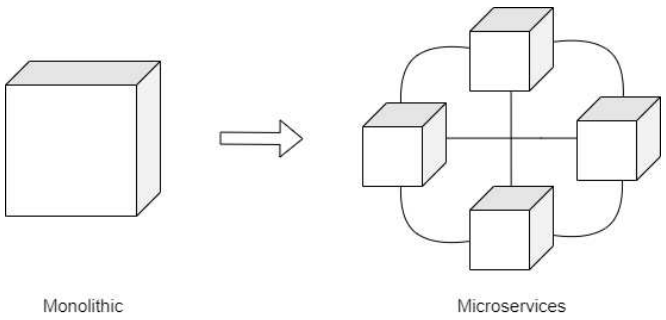

Figura 2.5: Monolithic vs Microservices

2.2.3 Architettura Cloud Serverless

Il precedente approccio tuttavia, prevede che nel team di sviluppo di una applicazione ci sia una figura che si occupi della configurazione e manutenzione di tutti i microservizi, nonché della ottimizzazione dei costi e del carico di lavoro per i server. Principalmente per questo motivo, è nata l'architettura **serverless**, che rappresenta un paradigma innovativo nel campo del Cloud Computing in quanto gli sviluppatori non si devono preoccupare di dover gestire direttamente l'infrastruttura sottostante ma solo di sviluppare l'applicazione. Il codice consiste in funzioni modulari che sono eseguite in risposta ad eventi, come richiesta *HTTP*, modifiche a database o trigger generali di servizi cloud. Il provisioning (ovvero il processo di creazione e configurazione della infrastruttura), il dimensionamento (in termini di scalabilità) la pianificazione e la gestione del server sono gestiti in maniera automatica e trasparente dai fornitori di questi servizi cloud.

È un approccio differente anche rispetto alle tre tipologie di servizi visti precedentemente: in questo caso si parla di **Function as a Service** (FaaS), in quanto il provider è responsabile della gestione del server.

Esempio di architettura serverless

Una classica architettura di applicazioni serverless comprende un insieme di componenti che lavorano tra di loro per offrire una soluzione scalabile, efficiente e flessibile:

- **API Gateway**: funge come da end-point per tutte le richieste *HTTP* (o altri protocolli). Si tratta di un servizio ad-hoc che funge da intermediario tra i client (come browser web, applicazioni mobili, ecc.) e i servizi backend. Gestisce tutte le richieste in entrata, le instrada agli end-point appropriati e applica politiche di sicurezza, autenticazione, autorizzazione, controllo del traffico, logging e monitoraggio;
- **Lambda Function**: componenti modulari di codice che implementano la logica della applicazione. Sono funzioni di tipo **stateless**, ovvero che non mantengono il proprio stato alla fine dell'esecuzione, risultando quindi indipendenti tra di loro. Vengono attivate da trigger esterni (come la ricezione di richieste da parte dell' *API Gateway*);
- **Autenticazione e Autorizzazione**: servizi che gestiscono la sicurezza delle applicazioni.

Oltre a questi servizi base, se ne possono implementare di altri, come database (**NoSQL**), storage di oggetti (per l'archiviazione di file di grandi dimensioni), etc.

Ciclo di vita Lambda Function

Il ciclo di vita di una funzione Lambda può essere suddiviso in tre diverse fasi chiave, che descrivono come la funzione viene eseguita, mantenuta e terminata:

1. **Inizializzazione**: allo scatenarsi di un determinato evento che provoca l'attivazione di un determinato trigger, inizia la fase di invocazione. Nel caso in cui non esista già un'istanza attiva della funzione, è creato un nuovo contesto di esecuzione, sono allocate le risorse necessarie e sono inizializzati gli ambienti runtime per l'esecuzione del codice. La prima volta che una funzione è invocata (o dopo un periodo di inattività) il codice della funzione deve essere caricato sull'ambiente precedentemente creato e inizializzare tutte le dipendenze affinché la funzione sia pronta ad essere eseguita (in questo caso si parla di **Cold Start**, a differenza dell'**Hot Start** in cui il codice è in cache pronto per essere ri-eseguito);

2. **Invocazione**: fase in cui il codice è eseguito, processando gli input fino alla restituzione di un output o dell'invio di un messaggio si successo / insuccesso;

3. **Shutdown**: ultima fase di vita della funzione in cui avviene il rilascio delle risorse. I dati importanti che necessitano di consistenza devono già essere stati salvati su database o file esterni, in quanto tutte le memorie volatili vengono liberate.

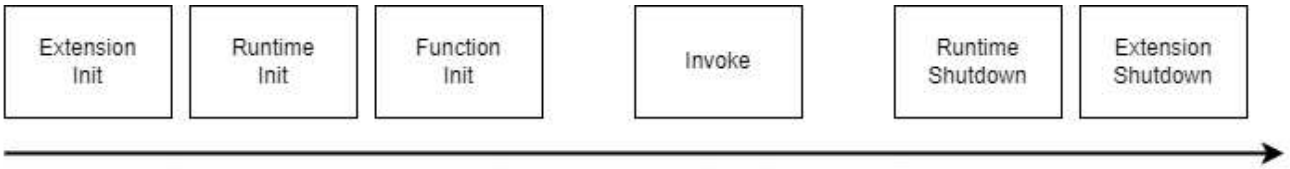

Figura 2.6: Lambda lifecycle

Vantaggi e svantaggi architettura serverless

Uno dei principali vantaggi nell'adozione di questo tipo di paradigma è la sua **scalabilità automatica**. Le funzioni possono essere scalate istantaneamente in risposta alla domanda senza bisogno di interventi manuali, portando benefici sia dal punto di vista della gestione sia dell'ottimizzazione dei costi che, generalmente, sono calcolati in base al tempo effettivo di esecuzione di codice (anziché nella allocazione continua di risorse di calcolo). Inoltre, è reso più snello il processo di sviluppo aumentando i tempi di sviluppo e le flessibilità, in quanto gli sviluppatori possono concentrasi sulla logica senza preoccuparsi del server.

Tuttavia, si possono incontrare diverse problematiche da gestire, come la **latenza** introdotta dall'avvio a "freddo" delle funzioni, la **complessità** nella gestione di applicazioni distribuite e la **dipendenza** dal fornitore di servizi cloud. Anche per la parte legata alla sicurezza richiede maggiore attenzione, poiché l'astrazione delle risorse sottostanti può complicarne la visibilità e il controllo.

2.2.4 Confronto tra Microservizi e Serverless

Di seguito viene riportata una tabella riassuntiva delle due soluzioni cloud messe a confronto.

	Microservizi	Serverless
Gestione dell'Infrastruttura	Gli sviluppatori gestiscono direttamente i server e l'infrastruttura	Il provider gestisce automaticamente l'infrastruttura, rendendo le risorse trasparenti agli sviluppatori
Sviluppo	Richiede tempo per la progettazione e l'integrazione di ciascun microservizio	Gli sviluppatori si concentrano principalmente sulla logica dell'applicazione
Costo	Pagamento continuo basato sulle risorse allocate e utilizzate	Pagamento basato sul tempo di esecuzione e sulla memoria utilizzata dalle funzioni
Scalabilità	Scalabilità manuale o automatizzata tramite altri servizi ad-hoc	Scalabilità automatica in risposta ai picchi di utilizzo
Complessità	Architettura complessa da implementare e gestire, richiede un'attenzione particolare per l'orchestrazione dei servizi	Architettura più semplice, ma può diventare complessa con l'aumentare delle funzioni serverless

Tabella 2.2: Microservices vs Serverless

La scelta di una soluzione cloud a microservizi comunque, non esclude l'adozione di un'architettura serverless, e viceversa, poiché queste due tecnologie non sono in contrapposizione ma possono essere combinate per ottimizzare flessibilità ed efficienza.

2.3 REST API

Le **REST API** (*Representational State Transfer Application Programming Interfaces*) sono un tipo di interfaccia a livello applicativo, basate sul protocollo **HTTP(s)** per consentire a diversi sistemi di comunicare tra loro: si basano sui principi dell'architettura **REST**, un modello di progettazione sviluppato da *Roy Fielding*, che consente di creare sistemi scalabili e interoperabili utilizzando le **risorse web** e le loro **rappresentazioni**.

Risorse Web

Le **risorse web** sono entità o contenuti specifici che possono essere identificati, recuperati e manipolati su un server web: nello specifico, nelle *REST API*, sono gli oggetti di interesse (come i dati o servizi) che l'*API* espone.

La richiesta e la manipolazione di queste risorse avviene tramite l'uso di indirizzo **URL** (*Uniform Resource Locator*), ovvero una stringa inviata durante la comunicazione tra client (colui che richiede) e server (colui che fornisce) che sfrutta i metodi **HTTP(s)** standard, come **GET**, **POST**, **PUT** e **DELETE**, che corrispondono rispettivamente alle operazioni di lettura, creazione, aggiornamento e cancellazione di una risorsa.

Le *REST API* sono progettate per essere **stateless**, il che significa che ogni richiesta dal client al server deve contenere tutte le informazioni necessarie per comprendere e processare la richiesta. Questo principio migliora la scalabilità e la semplicità del sistema, poiché il server non mantiene alcuno stato e non crea nessuna dipendenza tra le richieste.

Esempio e anatomia di una chiamata e risposta REST

Supponendo di avere un server che fornisce un servizio di *API* di tipo *REST* riguardo informazioni sugli utenti di un sistema collegato, una possibile richiesta per l'ottenimento dei dettagli di uno specifico utente (ad esempio con l'*ID 123*) sarebbe:

```
https://api.example.com/users/123?sort=desc&limit=10
```

Anatomia della richiesta:

1. **Protocollo**: Indica il protocollo utilizzato per la comunicazione, solitamente HTTP o HTTPS (es. `https://`).
2. **Dominio o Host**: Specifica il server su cui risiede l'API (es. `api.example.com`).
3. **Percorso**: Definisce la specifica risorsa con cui si desidera interagire (es. `/users/123`).
4. **Parametri di Query**: Aggiunti alla fine dell'URL, forniscono ulteriori dettagli o filtrano i dati richiesti (es. `?sort=desc&limit=10`).

Una possibile risorsa ottenibile in risposta invece, può essere un file in formato **json** che contiene tutte le informazioni del relativo utente:

```
{
    "id": 123,
    "name": "John Doe",
    "email": "john.doe@example.com"
}
```

Questo semplice esempio mostra come le *REST API* siano un meccanismo di comunicazione semplice e flessibile, ideale per lo sviluppo di applicazioni cloud, sia con architetture a microservizi (adatte, ad esempio, per l'interazione con ogni servizio) sia con architetture serverless (utili per il trigger di *lambda functions*).

3

Tecnologie utilizzate

3.1 Ganache: simulazione di blockchain per sviluppo e testing

Per la fase di sviluppo e testing di un sistema distribuito (*dAPP*), utilizzare una blockchain reale, come visto, risulterebbe troppo costoso e inefficace. La soluzione a questo problema è rappresentata dalle **testnet**, reti di test dove il deploy degli *Smart Contract* non comporta alcun costo e dove strumenti per il *debugging* e il *testing* sono implementati a supporto degli sviluppatori.

Ganache si presenta come un software progettato per la simulazione di blockchain: permette agli sviluppatori di eseguire una rete *Ethereum* privata sul proprio computer, consentendo la creazione e l'implementazione rapida di *Smart Contract* e la gestione di wallet fittizi per il *testing*.

Tra le varie funzionalità di *Ganache*, vi sono la possibilità di eseguire il **rollback** dello stato della blockchain tramite la creazione di **istantanee**. L'uso di questo strumento accelera significativamente il ciclo di sviluppo, riducendo i costi e i rischi associati al *testing* su una rete pubblica.

3.2 Amazon Web Services (AWS)

Amazon Web Services è una della piattaforme di cloud computing più avanzate e diffuse a livello globale. Offre una vasta gamma di servizi **on-demand**, che spaziano dall'elaborazione e archiviazione dei dati, all'intelligenza artificiale fino alla gestione di database e reti. La sua infrastruttura inoltre, permette alle aziende di tutte le dimensioni di scalare rapidamente in modo efficace, rendendolo una scelta ottimale.

AWS Global Infrastructure

AWS ha alla base una infrastruttura geo-distribuita a livello globale, strutturata in **regioni** e **zone di disponibilità**. Ogni regione è un'area geografica che contiene due o più zone di disponibilità, progettate in modo da essere isolate e indipendenti dai guasti delle *AZ* (*availability zones*) limitrofe. Offrono una connettività di rete economica ed a bassa latenza. Oltre alle regioni e alle *AZ*, *AWS* include anche le **edge locations**, ovvero dei *data center*, anch'essi distribuiti sul territorio, per la consegna rapida dei contenuti e per la ridondanza dei dati.

In questa sezione, sono analizzati i servizi utilizzati effettivamente nell'implementazione del prototipo del sistema menzionato inizialmente.

Figura 3.1: AWS Services

3.2.1 Amazon EC2: istanze virtuali

Amazon Elastic Compute Cloud è uno dei principali servizi offerti da *AWS*, progettato per fornire capacità di calcolo scalabile, che permette agli utenti di avviare e gestire istanze di diversi server virtuali di diversa tipologia, ottimizzati per diversi casi d'uso (dalla applicazioni generiche a quelle che richiedono elevate prestazioni di calcolo, memoria o capacità di archiviazione).

Scalabilità

L'aspetto chiave di *EC2* è la **scalabilità**: gli utenti possono facilmente aumentare o diminuire il numero di istanze attive in base alle esigenze del carico di lavoro, consentendo una gestione efficiente delle risorse e del costo. Le istanze possono essere configurate con diversi tipi di impostazioni e possono essere distribuite in diverse ubicazioni (regioni e zone di disponibilità), ottenendo tutti i vantaggi sopra citati.

3.2.2 Amazon APIs Gateway: end-point per chiamate REST

AWS API Gateway è un servizio completamente gestito che semplifica la creazione, pubblicazione, manutenzione, monitoraggio e protezione delle *API* su qualsiasi scala. Funziona come una "*front door*" per le applicazioni che accedono a dati, *business logic* o funzionalità dei servizi backend, come carichi di lavoro in esecuzione su *Amazon EC2*, *AWS Lambda* o qualsiasi applicazione web.

API Gateway gestisce tutte le operazioni necessarie per accettare ed elaborare fino a centinaia di migliaia di chiamate *API* contemporanee, inclusi la gestione del traffico, l'autorizzazione e il controllo degli accessi, il throttling e la gestione del versionamento.

3.2.3 AWS Lambda: esecuzione di codice senza gestione di server

AWS Lambda è il servizio messo a disposizione per creare una piattaforma **serverless**. È in grado di eseguire codice in risposta ad eventi e di gestire automaticamente le risorse di calcolo richieste da tale codice, consentendo agli sviluppatori la creazione di applicazioni e servizi senza la preoccupazione derivata dalla configurazione e mantenimento di server dedicati.

Supporta codice *Java*, Go, *PowerShell*, Node.js, C, *Python* e *Ruby*. In alternativa, fornisce anche un'**API Runtime** che consente di utilizzare qualsiasi ulteriore linguaggio di programmazione per scrivere le proprie funzioni.

Layers

Gli **AWS Lambda layers** sono meccanismi di distribuzione per librerie, runtime personalizzati e altre dipendenze delle funzioni. Un layer corrisponde ad un archivio *ZIP* che contiene tutti gli artefatti e può essere acceduto da funzioni lambda configurate appositamente. Grazie a loro, è possibile alleggerire il codice delle funzioni, alleggerendo la fase di inizializzazione e ottimizzando sia i costi sia i tempi.

3.2.4 AWS SAM: semplificazione del deployment di applicazioni serverless

AWS Serverless Application Model (*AWS SAM*) è un potente toolkit progettato per semplificare lo sviluppo e la gestione di applicazioni serverless su *AWS*. Offre una serie di vantaggi che migliorano significativamente l'esperienza degli sviluppatori:

1. **Definizione rapida dell'infrastruttura**: permette di definire rapidamente il codice dell'infrastruttura applicativa in modo compatto, descrivendo le risorse in modo efficiente e diretto, accelerando il processo di sviluppo;

2. **Gestione completa del ciclo di vita**: permette di gestire l'applicazione serverless durante tutte le fasi del ciclo di sviluppo, dalla creazione alla distribuzione, al test e al monitoraggio, ponendo il focus sullo sviluppo della logica e non nella gestione;

3. **Gestione delle autorizzazioni semplificata**: permette di usare **connettori** che facilitano la definizione delle autorizzazioni tra le risorse utilizzate, specificandole in modo chiaro e conciso;

4. **Sincronizzazione automatica delle modifiche**: permette di sincronizzare automaticamente le modifiche apportate localmente nel cloud, velocizzando i flussi di lavoro di sviluppo e test, consentendo agli sviluppatori di lavorare in modo più efficiente.

3.3 Node.js: esecuzione di JavaScript lato server

Node.js[3] è un **runtime JavaScript** open-source e multipiattaforma costruito sul motore **V8** di *Google Chrome*, progettato per eseguire codice JavaScript al di fuori di un browser. Creato da Ryan Dahl nel 2009, *Node.js* è diventato una delle tecnologie più popolari per lo sviluppo di applicazioni server-side grazie alla sua efficienza, scalabilità e flessibilità.

L'architettura di Node.js è basata su un modello di **I/O asincrono** e **non bloccante**, che lo rende particolarmente adatto per applicazioni che richiedono prestazioni elevate e la gestione di molte connessioni simultanee, come server web, API e servizi di streaming.

3.3.1 Architettura Node.js

L'architettura di *Node.js* può essere descritta attraverso i seguenti componenti e concetti principali.

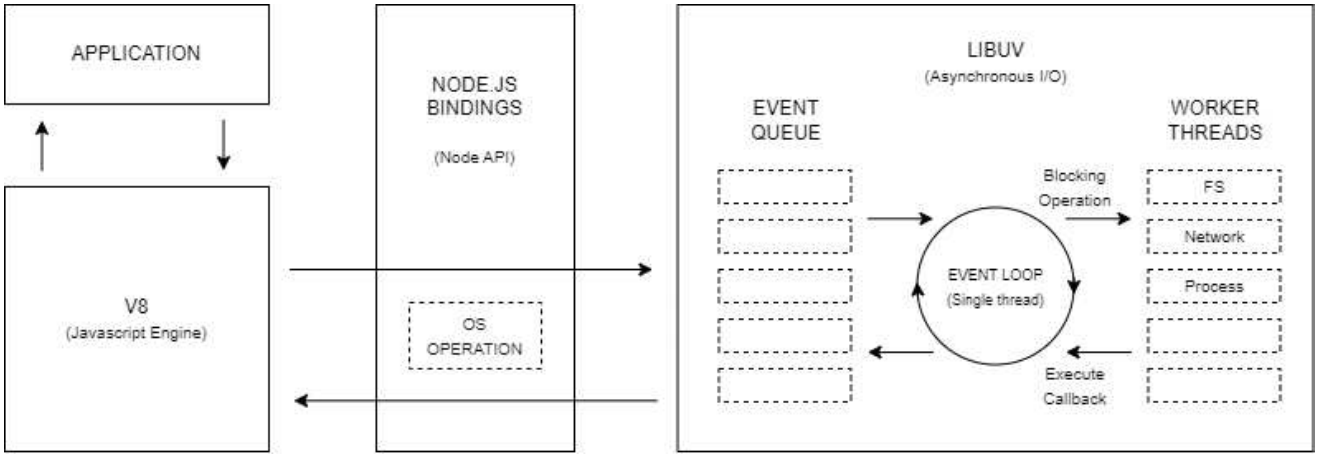

Figura 3.2: Architettura Node.js

Single-threaded Event Loop

Il cuore dell'architettura di *Node.js* è il **single-threaded event loop**, un meccanismo che consente di gestire operazioni asincrone in modo efficiente. A differenza dei server tradizionali che creano un nuovo thread per ogni richiesta, utilizza un **singolo thread** per gestire tutte le richieste, che esegue un ciclo continuo (loop) e gestisce eventi e richieste in modo asincrono.

Una delle caratteristiche distintive è il suo modello di **I/O non-bloccante**, in cui le operazioni di input/output, come la lettura di file o la gestione di richieste di rete, non bloccano l'esecuzione del codice. Per ovviare il problema dell'esecuzione del codice post I/O, vengono utilizzate le **callback**, ovvero delle funzioni passate come argomenti ad altre funzioni e chiamate quando l'operazione asincrona è terminata. Alternative introdotte nel corso del tempo a causa della ridotta leggibilità e manutenibilità del codice comportata dall'utilizzo di queste ultime, sono le **promises** (ovvero degli oggetti che rappresentano l'eventuale completamento, o fallimento, dell'operazione asincrona) e gli *statement* **async** e **await** (che forniscono una sintassi più semplice rispetto le callback).

Libuv

Libuv è una libreria multipiattaforma che fornisce una astrazione dell'*event loop* e per le operazioni asincrone, in modo da gestire il sistema di I/O non bloccante e da interfacciarsi coerentemente con le diverse piattaforme.

V8 JavaScript Engine

Node.js utilizza il motore **V8** di **Google** per eseguire codice JavaScript: si tratta di un *engine* noto per la sua velocità e per la capacità di compilare codice JavaScript in codice macchina nativo, migliorando notevolmente le prestazioni.

Binding

L'*engine* di *Node.js* è scritto in **C** e **C++**, con **binding** che permettono di collegare il codice JavaScript con le API di basso livello del sistema operativo. Questo gli consente di eseguire operazioni a livello

di sistema in modo efficiente e di fornire funzionalità avanzate come l'accesso al file system, la gestione delle reti e altro ancora.

3.3.2 Moduli Node.js

I moduli in *Node.js* possono essere visti come delle librerie: si tratta di insiemi di file di script progettati per offrire funzionalità ad alto livello agli sviluppatori. Esistono i moduli **core**, predefiniti e già ottimizzati per i compiti comuni (come la manipolazione dei file, etc.) e i moduli sviluppati da terzi, che possono essere installati attraverso **npm** (*Node Package Manager*), un *tool* fornito con *Node.js* che si connette ad una **repository** pubblica centrale dove programmatori possono pubblicare i proprio moduli.

3.4 HTML, CSS & Bootstrap

HTML (*HyperText Markup Language*) è il linguaggio base utilizzato per creare e strutturare il contenuto delle pagine web, definendo gli elementi e le sezioni dei documenti (come titoli, paragrafi, immagini, etc.).

CSS invece, acronimo di *Cascading Style Sheets*, è un linguaggio di stile usato per descrivere la presentazione di un documento scritto in HTML, che permette di controllare il layout, i colori, i font e altri aspetti estetici della pagina.

Dato che l'utilizzo di *HTML* e *CSS* in modo nativo risulta essere, al giorno d'oggi, macchinoso e poco efficiente, è nato **Bootstrap**, un framework front-end open-source che combina tali tecnologie per facilitare lo sviluppo di siti web reattivi e compatibili con diversi dispositivi, semplificando il processo di design e implementazione, riducendo il tempo necessario alla creazione di Web app. Comprende una vasta gamma di componenti predefiniti che rispecchiano i moderni stili di design standardizzati, in modo da creare siti web che oltre a risultare più solidi, appaiono anche coerenti e accattivanti su vari dispositivi e risoluzioni.

4

Implementazione soluzione

In questo capitolo è descritto il processo di concezione, progettazione, sviluppo e test del sistema di gestione di biglietti menzionato inizialmente, basato sulle tecnologie e nozioni descritte nei due capitoli precedenti.

Il risultato finale è costituito da una piattaforma sicura, scalabile e decentralizzata, capace di gestire e validare transazioni in modo trasparente ed efficiente, sfruttando la potenza del **cloud computing** con l'affidabilità delle **blockchain**.

4.1 Analisi dei requisiti e obbiettivi

4.1.1 Identificazione degli obbiettivi del Sistema

L'obbiettivo primario è la creazione di una piattaforma per la collezione di NFT legati a biglietti di eventi, in particolare concerti, sulla piattaforma blockchain e per l'ottenimento di benefici usufruibili sull'acquisto di nuovi biglietti o sulla partecipazione a nuovi eventi.

Il sistema inoltre deve prevedere un meccanismo di scambio degli NFT dei biglietti tra utenti in modalità "uno a uno".

4.1.2 Raccolta requisiti utente

Requisiti degli Utenti (Acquirenti di Biglietti)

Ogni utente potrà:

1. Richiedere l'ottenimento del NFT associato al biglietto acquistato sulla piattaforma principale;
2. Utilizzare gli NFT dei biglietti posseduti per ottenere degli NFT che rappresentano dei benefici;
3. Iniziare una richiesta di scambio (uno a uno) di NFT dei biglietti con un altro utente della piattaforma;
4. Accettare una richiesta di scambio ricevuta da un altro utente.

Requisiti degli Admin

Ogni admin potrà:

1. Aggiungere nuovi template per benefici ottenibili dagli utenti (*punto 2* dei casi d'uso dell'**utente**).

Diagramma dei casi d'uso

Il seguente diagramma dei casi d'uso riassume i requisiti utente degli **Utenti** e degli **Admin**:

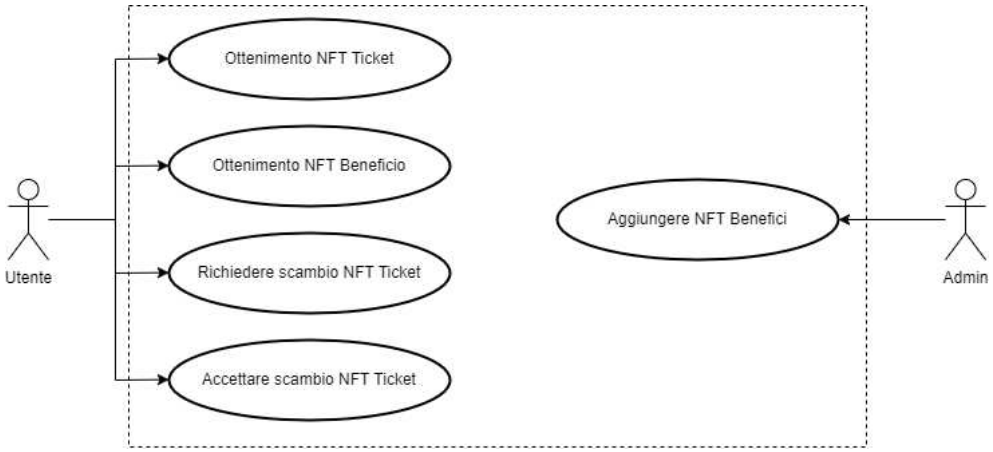

Figura 4.1: Diagramma casi d'uso

4.1.3 Analisi dei Requisiti

Requisiti Funzionali

I requisiti funzionali, che definiscono i compiti del sistema descrivendo le funzionalità e i comportamenti desiderati, sono riassunti come segue:

RF1, RF2	Emissione e Gestione degli NFT	Il sistema deve garantire l'ottenimento degli NFT legati ai biglietti dei concerti e all'ottenimento dei benefici
RF3, RF4	Scambio e Trasferimento degli NFT	Gli utenti devono poter scambiare i loro NFT tramite la piattaforma, mentre il sistema deve gestire in modo sicuro le transazioni di scambio (senza intermediari)
RF5	Verifica dell'Autenticità degli NFT	Gli utenti devono poter verificare l'autenticità degli NFT tramite una funzione di verifica sulla piattaforma

Tabella 4.1: Requisiti funzionali

Requisiti Non Funzionali

I requisiti non funzionali invece, aiutano a definire come il sistema deve comportarsi in diversi scenari, includendo caratteristiche di qualità come prestazioni, sicurezza, scalabilità usabilità e affidabilità:

RNF1	Sicurezza	Tutte le transazioni sulla piattaforma devono essere sicure e protette contro attacchi e frodi
RNF2	Scalabilità	La piattaforma deve essere in grado di gestire un alto numero di transazioni simultanee, soprattutto durante eventi con un'alta affluenza di pubblico (deve essere prevista la possibilità di scalare l'infrastruttura in base alla domanda)
RNF3	Usabilità	L'interfaccia utente deve essere intuitiva e facile da utilizzare per tutti gli utenti, indipendentemente dal loro livello di competenza tecnologica
RNF4	Performance	Il sistema deve garantire tempi di risposta rapidi (in termini di una decina di secondi) per tutte le operazioni principali, come l'emissione di NFT, lo scambio e la verifica dell'autenticità
RNF5	Interoperabilità	Il sistema deve essere compatibile con altre piattaforme e servizi utilizzati dagli organizzatori di eventi e dai rivenditori di biglietti. Inoltre, deve essere possibile integrare facilmente nuove funzionalità e servizi in futuro

Tabella 4.2: Requisiti non funzionali

4.2 Progettazione dell'architettura del sistema

La raccolta di tutti i requisiti (sia utente che di sistema, funzionali e non) nella fase iniziale è necessaria per una progettazione accurata del sistema, poiché permette di definire chiaramente tutti gli attori e le funzionalità fin dal principio, permettendo di eseguire la fase di progettazione in modo più mirato.

4.2.1 Architettura del sistema

La scelta architetturale di questo sistema prevede l'utilizzo di tre componenti principali: **web server**, che ospita le interfacce per permettere ai due attori di poter interagire all'interno della piattaforma secondo i rispettivi casi d'uso, i servizi **AWS**, che fungono da intermediari tra gli utenti e blockchain in modo da rendere il sistema più scalabile e robusto e infine la **blockchain**, che attraverso l'utilizzo di *Smart Contract* gestisce la logica in modo trasparente e affidabile.

4.2.2 Ganache e EC2

Per simulare la blockchain in ambiente di sviluppo e test, è stato scelto di installare il software **Ganache** su un'istanza **EC2**, configurata con un sistema operativo **Linux**. Per garantire l'accesso remoto al sistema, è stato eseguito un *mapping* tra la porta del processo e la porta esposta dell'istanza, in modo da sfruttare l'indirizzo IP già esistente. I passaggi chiave per configurare Ganache su EC2 includono:

- **Creazione** di un'istanza EC2 con una distribuzione Linux (ad esempio, Ubuntu);

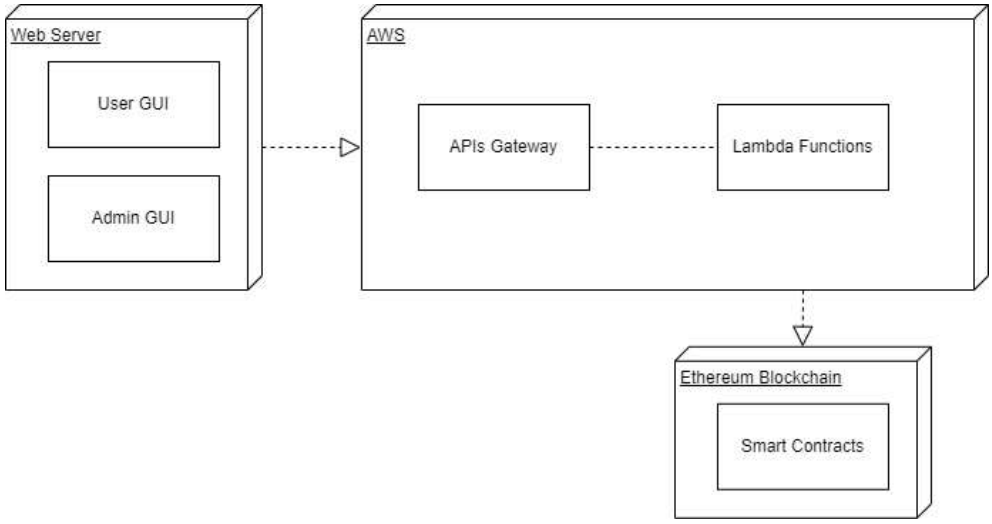

Figura 4.2: Diagramma del sistema

- **Configurazione** della sicurezza dell'istanza, includendo l'impostazione delle regole del firewall per permettere solo il traffico necessario;
- **Installazione** di Ganache e dei relativi strumenti di sviluppo;
- **Configurazione** di Ganache per l'accesso remoto e avvio del servizio.

4.2.3 Smart Contract

Remix IDE è il software utilizzato per la stesura e il testing degli *Smart Contract*. Questo ambiente di sviluppo integrato, accessibile via web, offre strumenti avanzati per scrivere, compilare e testare codice in Solidity, diventando particolarmente utile per il debugging e il testing rapido permettendo l'iterazione veloce durante il processo di sviluppo.

Standard ERC

Come anticipato nella sez. *Fungible Token & Non Fungible Token 2.1.3*, gli standard **ERC** definiscono le specifiche per i contratti sulla rete *Ethereum*. Quello utilizzato in questo progetto è **ERC-1155**, uno standard multi-token che supporta sia token fungibili che non fungibili.

Adottando questo standard, gli *Smart Contract* espongono diversi metodi base che ne definiscono una interfaccia:

- `balanceOf`: Restituisce il saldo di un tipo di token specifico per un indirizzo;
- `balanceOfBatch`: Restituisce i saldi di più token per più indirizzi;
- `safeTransferFrom`: Trasferisce una quantità specifica di un tipo di token da un indirizzo a un altro;
- `safeBatchTransferFrom`: Trasferisce più tipi di token da un indirizzo a un altro in una singola transazione;

- `setApprovalForAll`: Approva o revoca l'approvazione per un operatore a gestire tutti i tipi di token del proprietario;

- `isApprovedForAll`: Verifica se un operatore è approvato per gestire tutti i token di un proprietario.

Questi standard sono facilmente implementabili grazie alla libreria **OpenZeppelin** che, oltre che agli standard, mette a disposizione anche una serie di funzionalità per la sicurezza e utility (come gestione degli indirizzi, funzioni matematiche, etc.).

4.2.4 Ticket NFT

Il contratto `Ticket` è progettato per gestire biglietti digitali e i loro benefici associati, permettendo la creazione di biglietti come token unici e non fungibili (*NFT*), che possono essere trasferiti tra utenti in modo sicuro. Ad ogni biglietto possono essere associati dei benefici (vedi sez. *Benefit NFT 4.2.5*) , i quali possono essere richiesti dai possessori dei biglietti. Inoltre, nel contratto viene implementato il meccanismo di scambio attraverso un sistema di richiesta e approvazione.

Le varie specifiche, le definizioni di metodi e le variabili globali sono descritte nelle seguenti tabelle.

Importazioni

Inizialmente, il contratto include diversi moduli necessari:

ERC1155	Standard per token fungibili e non fungibili che permette di gestire diverse tipologie di token sotto un unico contratto.
Ownable	Modulo che permette di gestire le autorizzazioni in modo tale che solo il proprietario del contratto possa eseguire certe operazioni.
ERC1155Burnable	Estende ERC1155 per permettere di distruggere token, eliminandoli permanentemente dalla circolazione.
Benefit, IBenefit, ITicket	Contratti e interfacce che definiscono benefici e la struttura dei biglietti.

Tabella 4.3: Importazioni Ticket NFT

Variabili e costanti

Vengono definite alcune variabili chiave e costanti:

TOKEN NUM (costante)	Imposta un limite massimo di NFT che possono essere coniati.
MASTER WALLET (costante)	Indirizzo del portafoglio principale che gestisce alcune operazioni speciali come gli scambi di biglietti.
usedBenefit	Un'istanza del contratto UsedBenefit, che gestisce i benefici già utilizzati.
lastUsedTokenTicket lastusedTokenPendingTrade	Contatori per generare ID unici per i biglietti e per le richieste di scambio.

Tabella 4.4: Variabili e costanti Ticket NFT

Funzioni principali

Spiegazione dei metodi principali di Ticket NFT:

Costruttore	È eseguito una volta sola quando il contratto viene distribuito sulla blockchain. Inizializza l'URI (Uniform Resource Identifier) di base per i token, imposta il portafoglio principale e crea un'istanza del contratto UsedBenefit.
mintTicketNFT	Consente al proprietario del contratto di creare nuovi NFT di biglietti. Ogni volta che viene coniato un nuovo biglietto, viene generato un ID univoco e vengono aggiornate le mappe "ownerTicket" e "ticketOwner" per riflettere il nuovo possesso del biglietto.
safeTransferFrom	Sovrascrive la funzione di trasferimento sicuro dell'ERC1155, per assicurarsi che, quando un biglietto viene trasferito, anche le mappe dei proprietari vengano aggiornate correttamente.
tradeRequest	Inizia una richiesta di scambio di biglietti tra due utenti. Trasferisce temporaneamente il biglietto del richiedente al portafoglio principale e crea una struttura di scambio con i dettagli della richiesta.
tradeApprove	Permette al secondo partecipante dello scambio di approvare la richiesta. Una volta approvato, il biglietto del secondo partecipante viene anch'esso trasferito temporaneamente al portafoglio principale.
masterWalletExecTrade	Funzione chiamata dal portafoglio principale per eseguire effettivamente lo scambio. Trasferisce i biglietti tra i partecipanti e rimuove la richiesta di scambio completata.
mintBenefitNFT	Permette ai proprietari di biglietti di coniare nuovi NFT di benefici. Prima di coniare un beneficio, la funzione verifica che il chiamante possieda un numero sufficiente di biglietti idonei. (Vedi sez. *Benefit NFT 4.2.5*)

Tabella 4.5: Metodi principali Ticket NFT.

Richiesta di ottenimento Ticket NFT

1. Durante la fase di acquisto di un biglietto su un portale esterno, l'utente richiede di ricevere il proprio Ticket NFT all'interno del suo wallet (precedentemente configurato e inserito);

2. È richiamata automaticamente dal sistema la funzione `mintTicketNFT`, tramite il wallet *master* in modo da impedire la creazione di Ticket NFT senza autorizzazione;

3. Il TicketNFT appena richiesto compare nella wallet del cliente.

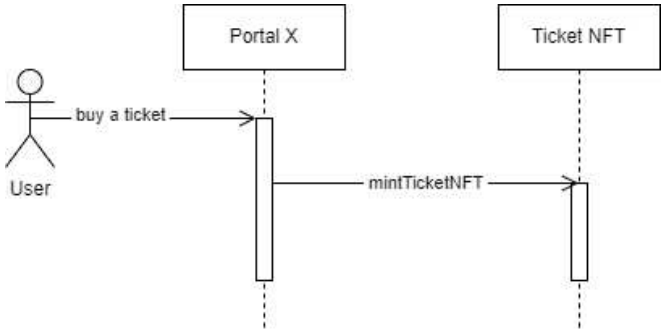

Figura 4.3: Ottenimento Ticket NFT

Meccanismo di scambio Ticket NFT

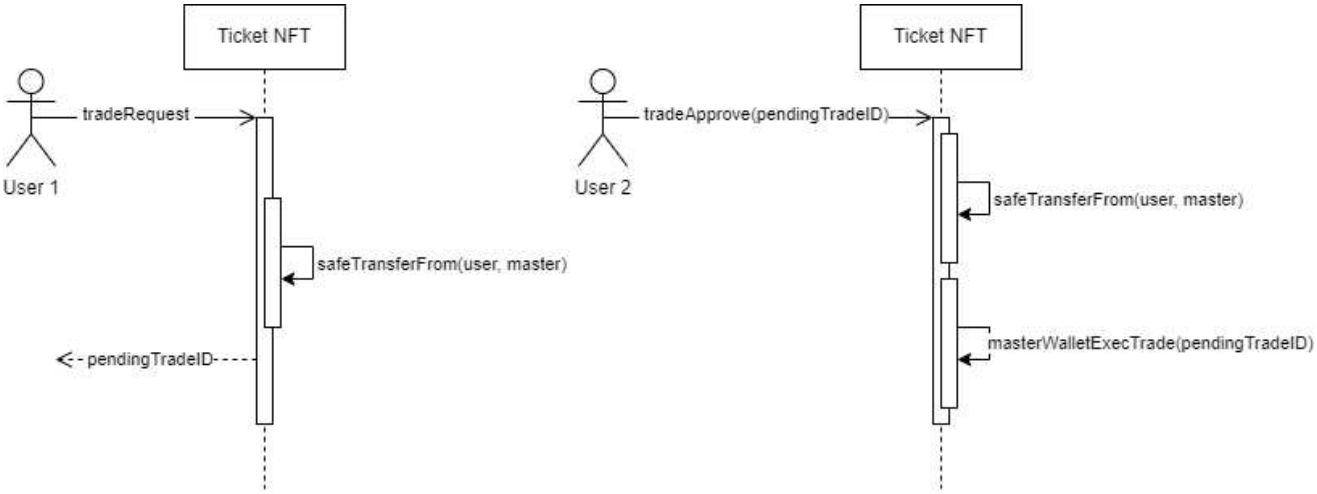

Figura 4.4: Trading Ticket NFT

1. Un utente, identificato come *User 1* nel diagramma in figura *4.4*, invia la richiesta di scambio al sistema, il quale crea una nuova transazione pendente trasferendo il possesso del NFT da scambiare nelle mani del wallet *master;*

2. *User 2*, conoscendo il `pendingTradeID` (condiviso in precedenza), manda la richiesta di approvazione della scambio, trasferendo anche il suo NFT nel portafoglio *master*. Automaticamente, il sistema controlla che i requisti siano rispettati e sposta nuovamente gli NFTs nei wallet degli utenti come definito nello scambio;

3. Lo scambio si conclude e ogni utente ha i nuovi Ticket NFT ottenuti grazie allo scambio.

4.2.5 Benefit NFT

Il contratto `Benefit` gestisce la parte legata ai benefici, consentendo la creazione di template per benefici e l'ottenimento di NFT (legati a questi template) per i diversi utenti. Ogni template definisce delle *regole* per l'ottenimento (come il numero di ticket minimi di un determinato evento) e la tipologia di beneficio offerta (sconto, ingresso omaggio, posto riservato, etc.).

Ogni Ticket NFT può essere utilizzato per ottenere un determinato Benefit NFT una e una sola volta: nel caso in cui si provi a richiedere due o più volte il beneficio con lo stesso ticket (o insieme di

ticket) la transazione fallirà. Inoltre, lo scambio dei ticket tra utenti non cancella lo storico di utilizzo dei Ticket NFT.

Anche per questo contratto, le varie specifiche, definizioni di metodi e variabili globali sono di seguito illustrate:

Importazioni

Include gli stessi standard e librerie di Ticket NFT.

Variabili e costanti

Le variabili e costanti utilizzate per la gestione dello stato corrispondono a:

TOKEN NUM (costante)	Indica il numero di token per ogni BenefitNFT creato.
lastUsedTokenBenefit	Contatore che tiene traccia dell'ultimo ID di BenefitNFT utilizzato.
lastUsedTokenTemplateBenefit	Contatore che tiene traccia dell'ultimo ID di BenefitTemplate utilizzato.
ownerBenefits	Mapping che associa gli indirizzi dei wallet a una lista di BenefitNFT posseduti.
benefitsTemplate	Mapping che associa gli ID dei BenefitTemplate ai dettagli del template stesso.

Tabella 4.6: Variabili e costanti Benefit NFT

Funzioni principali

Spiegazione dei metodi principali di Benefit NFT:

mint	Consente di creare un nuovo BenefitNFT e associarlo a un indirizzo di wallet specifico. Questo BenefitNFT viene poi aggiunto alla lista dei benefici posseduti dal wallet.
addNewBenefitTemplate	Permette al proprietario del contratto di aggiungere un nuovo template di beneficio, specificando i dettagli come l'ID dell'evento, la tipologia del beneficio, il numero minimo di biglietti richiesti, il tipo di TicketNFT e l'ID dell'artista.

Tabella 4.7: Metodi principali Benefit NFT

Processo di aggiunta template

1. Gli utenti amministratori del sistema, tramite il wallet *master*, richiamano la funzione `addNewBenefitTemplate` per creare dei nuovi template;

2. Ad ogni utente compaiono nel sistema tutti i template da cui ottenere i Benefit NFT, in base ai Ticket NFT in possesso.

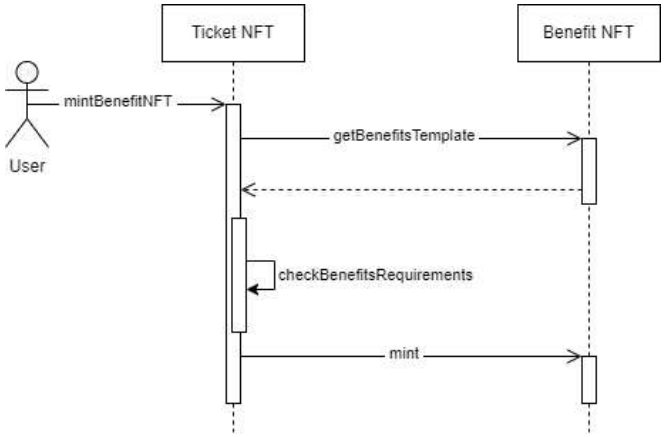

Figura 4.5: Ottenimento Benefit NFT

Ottenimento Benefit NFT

Il processo di ottenimento dei Benefit NFT inizia nello Smart Contract `Ticket`, poiché, prima del *minting*, devono essere controllati e soddisfatti i requisiti descritti nel template del beneficio.

1. Un utente richiama la funzione di `mintBenefitNFT` (esposta dal contratto `Ticket`), la quale richiede il template del beneficio che vuole ottenere;

2. È eseguita una funzione di controllo che verifica se il wallet soddisfi tutti i requisiti imposti dal template;

3. In caso affermativo, la funzione richiama il *minting* di un Benefit NFT associato al template nel wallet personale dell'utente. In caso contrario, la transazione si interrompe.

4.2.6 Backend serverless

Per collegare le funzionalità e la logica degli Smart Contract con l'interfaccia web in modo tale da permettere l'interazione con l'utente, è stato scelto di implementare un servizio cloud **serverless**, in combinazione con il tool **AWS SAM**.

Definizione dei servizi

Tramite la scrittura del template, sono stati definiti tutti i servizi e le lambda functions che andranno ad operare come tramite nel sistema. La versione del formato del template è `2010-09-09` con una trasformazione specifica per le risorse serverless (`AWS::Serverless-2016-10-31`).

- La sezione `Globals` stabilisce impostazioni globali per le funzioni lambda e l'API, come il timeout, il runtime Node.js 20.x, e le variabili d'ambiente che includono gli indirizzi e i parametri necessari per la connessione alla blockchain;

- Le funzioni sono raggruppate in due categorie principali: funzioni per il **signer** (ovvero eseguibili solamente dal proprietario del contratto) e funzioni per gli **user**, associate ad endpoint API specifico che permette l'interazione tramite chiamate *HTTP*:

  - Le funzioni per il **signer** comprendono operazioni come l'ottenimento del wallet da un ticket, il recupero di template dei benefici, l'eseguire il minting dei ticket, l'aggiunta di template dei benefici, la conferma di scambi di ticket e il recupero ticket e scambi in sospeso
  - Le funzioni per gli **user** includono invece operazioni per ottenere ticket, recuperare template di benefici, gestire scambi di ticket, effettuare il minting dei benefici e avviare le azioni di scambio e accettazione

- Le funzioni lambda utilizzano due layer: `NodejsLayer` e `CommonLayer`. Il primo contiene le dipendenze necessarie per l'esecuzione delle funzioni, mentre il secondo contiene codice comune utilizzato dalle varie funzioni.

Esempio di definizione di una lambda functions

Il codice necessario per definire una singola lambda functions è il seguente:

```
MintTicketFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: src/signer-handlers/
      Handler: mint-ticket.mintTicketHandler
      Layers:
        - !Ref NodejsLayer
        - !Ref CommonLayer
      Events:
        Testing:
          Type: Api
          Properties:
            Path: /signer/mint/
            Method: post
```

- **Tipo**: indica di che tipo di risorsa si tratta;
- **Codice della funzione**: corrisponde al *path* da seguire per trovare il codice da eseguire;
- **Handler**: definisce il punto di ingresso della funzione Lambda;
- **Layers**: indica i layer utilizzati e accessibili a quella funzione;
- **Eventi**: definiscono come e quando la funzione Lambda viene invocata. In questo caso, la funzione è invocata tramite una chiamata HTTP POST;

4.2.7 Interfaccia utente

Per dare una veste grafica al sistema, è stato sviluppata anche la demo di una Web app grafica che permette agli utenti di compiere le azioni sopra citate in modo semplice e interattivo.

Di seguito, sono riportate alcune schermate principali che illustrano una tipica interazione tra un utente e il sistema.

Home utente

Nella schermata principale, compare la lista dei Ticket NFT e Benefit NFT, divisi rispettivamente in due sezioni, posseduti dall'utente che ha eseguito il login nella applicazione.

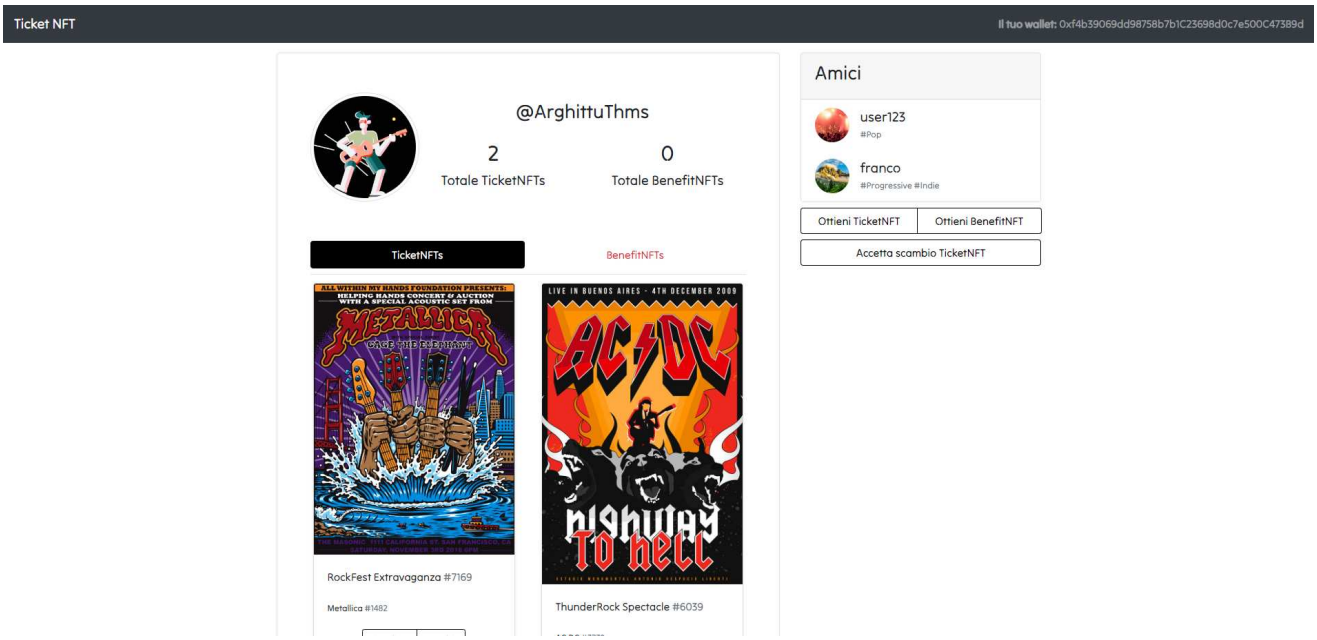

Figura 4.6: Home utente - Ticket NFTs

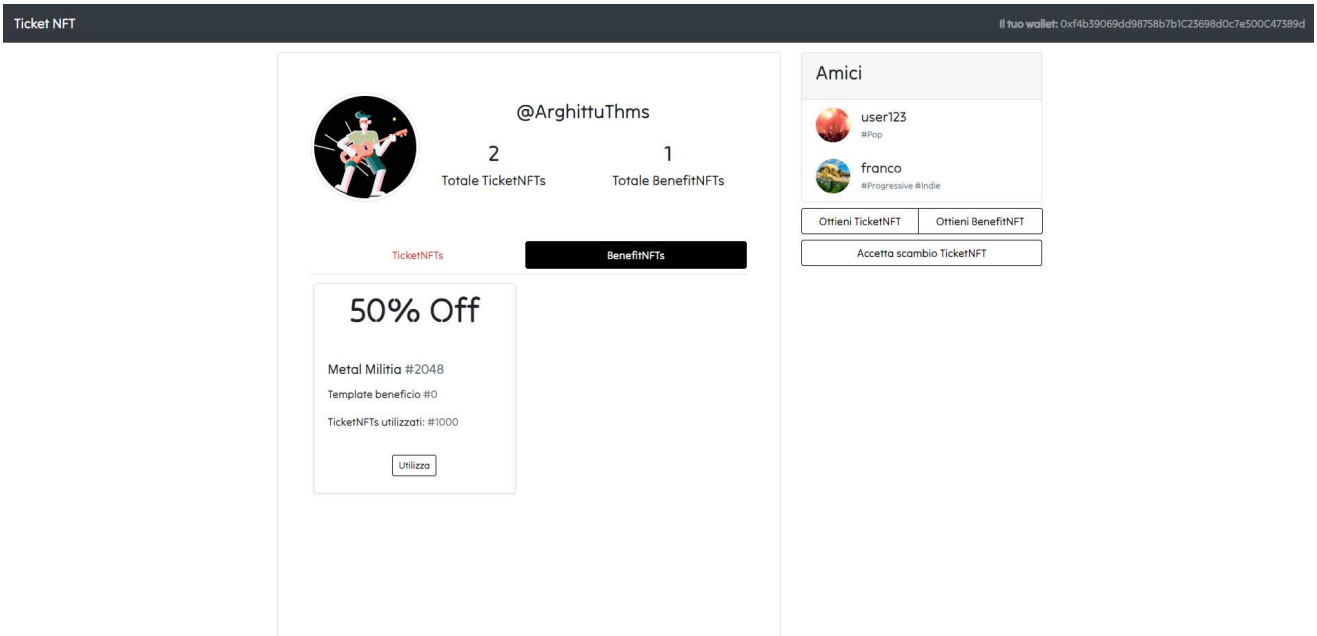

Figura 4.7: Home utente - Benefit NFTs

Vetrina benefici ottenibili

Accedendo alla vetrina dei benefici ottenibili, tramite il pulsante *Ottieni BenefitNFT* presente nella schermata principale, compare la lista di tutti i template dei benefici ottenibili (o quasi) dall'utente:

- Nel caso in cui la card compaia di colore acceso, il beneficio sarà pronto per essere ottenuto, in quanto il wallet possiede tutti i requisiti
- Nel caso in cui la card sia di colore grigiastro, viene indicato il fatto che l'utente non ha ancora soddisfatto il numero o la tipologia di biglietti richiesti

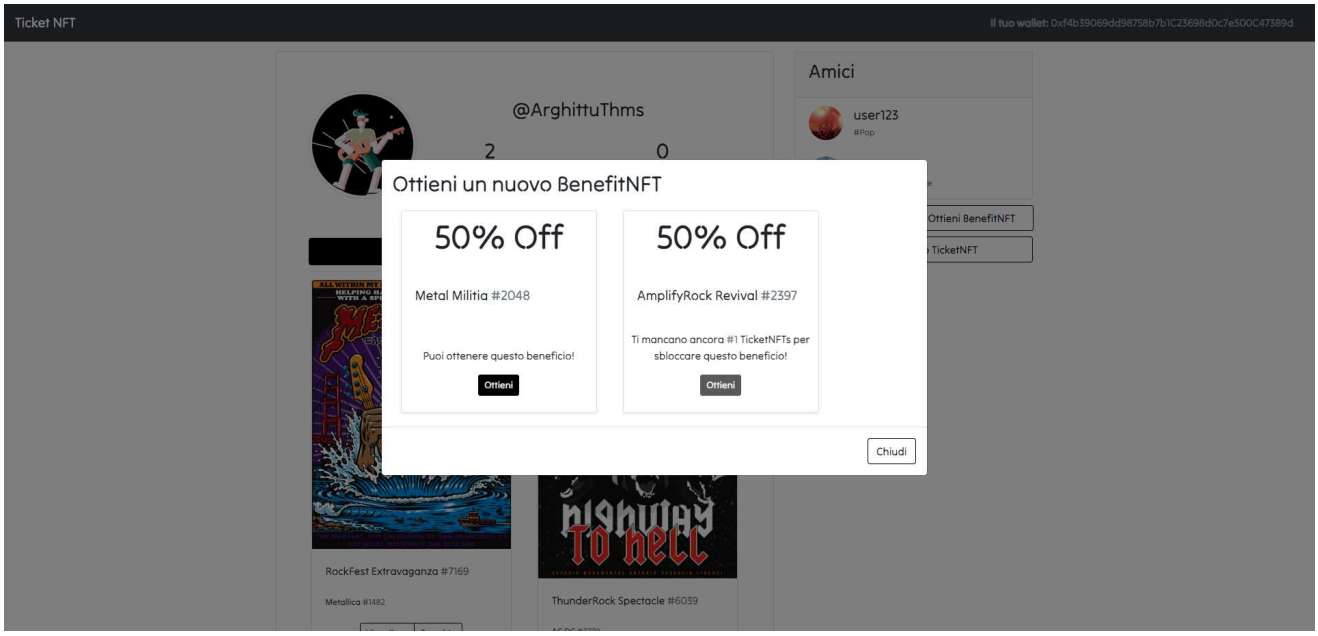

Figura 4.8: Vetrina benefici ottenibili

Conferma scambio Ticket NFT

In questa schermata sono visualizzate e riassunte le informazioni dello scambio prima della conferma.

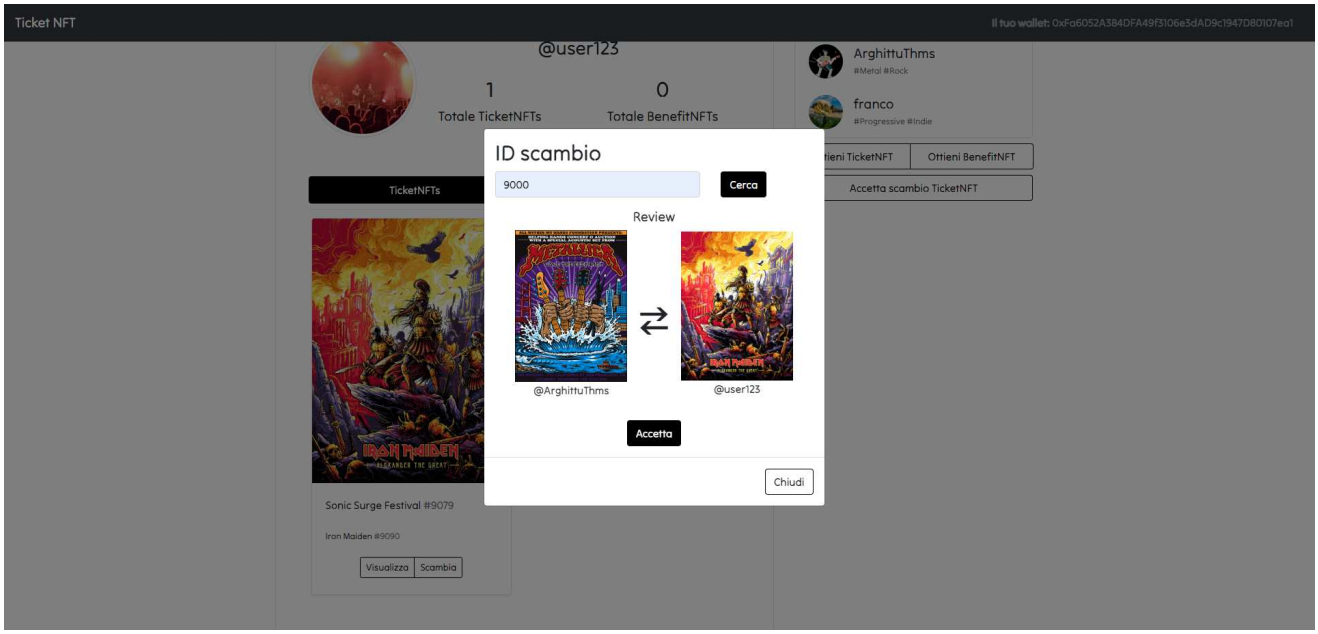

Figura 4.9: Conferma scambio Ticket NFT

5

Analisi e conclusioni

Per capire la sostenibilità del sistema e per valutare l'efficacia delle scelte implementative, in questo capitolo viene eseguita una breve analisi sul costo e sulle performance. In particolare, immaginando che un'alta percentuale dell'utenza si collega nel momento dell'uscita dei biglietti, si suppone che la piattaforma sia soggetta a picchi di connessioni di circa *50.000* utenti mensili.

5.1 Analisi dei costi

5.1.1 Ethereum blockchain

L'utilizzo della blockchain di Ethereum comporta costi legati alle **gas fees** per l'esecuzione dei contratti intelligenti (tariffe che possono variare in base alla congestione della rete e alla complessità delle operazioni).

Supponendo un costo medio di *50 gwei* per gas e una complessità media di transazione di *100.000 gas*, il costo per ogni transazione può essere stimato in circa *5-10 USD*. Per *50.000* transazioni mensili, i costi potrebbero oscillare tra *250.000* e *500.000* USD mensili.

5.1.2 Amazon Web Services

Lambda

I costi di lambda sono basati sul numero di richieste e sulla durata dell'esecuzione. Con *50.000* richieste mensili e un tempo di esecuzione medio di *200 ms* per funzione, il costo è stimato a circa *0.00001667 USD* per richiesta, per un totale di circa *0.8335 USD* mensili.

APIs Gateway

I costi sono di circa *3.50 USD* per milione di invocazioni e *0.09 USD* per *GB* di dati trasferiti. Considerando *50.000* richieste mensili, i costi si aggirano intorno ai *0.175 USD* mensili solo per le invocazioni.

5.2 Analisi delle performance

5.2.1 Ethereum blockchain

Le transazioni su blockchain possono avere una latenza variabile, tipicamente tra pochi secondi a diversi minuti, a seconda della congestione della rete. Inoltre, sebbene Ethereum offra una buona scalabilità, le limitazioni attuali della rete, di circa 15 transazioni al secondo, possono rappresentare un collo di bottiglia durante i picchi di richiesta.

5.2.2 Amazon Web Services

Lambda

Lambda garantisce scalabilità automatica, manutenibilità ridotta e alta disponibilità. Tuttavia, esiste una latenza iniziale (*cold start*) che può influenzare le performance durante i primi accessi.

APIs Gateway

Offre una gestione efficiente delle API e può scalare per gestire elevate quantità di traffico senza problemi di latenza significativi.

5.3 Considerazioni finali sull'efficacia del sistema

L'architettura proposta, che combina *Ethereum* e *AWS*, si è dimostrata efficace per soddisfare i requisiti del progetto. La scelta di una soluzione serverless con *AWS* ha permesso di mantenere contenuti i costi operativi e la complessità di gestione (sez. *5.1*), mentre l'utilizzo della blockchain di *Ethereum* ha garantito trasparenza e sicurezza nelle transazioni.

Tuttavia, i costi legati alle transazioni su Ethereum per una piattaforma di dimensioni medio/piccole potrebbero essere significativi per il mantenimento economico della stessa, mentre far ricadere l'intera spesa sugli utenti potrebbe risultare controproducente. Una possibile soluzione è rappresentata dall'adozione di una blockchain interna alla piattaforma basata su Ethereum e renderla aperta al pubblico, in modo che le transazioni siano gratuite e che la piattaforma stessa si occupi del mining dei nuovi blocchi. A tale scopo, risulta essere interessante un servizio di *AWS* ad-hoc che permette la gestione e l'implementazione di reti blockchain sia pubbliche che private: **Amazon Managed Blockchain**. Il servizio prevede un costo mensile basato sul numero di nodi e sulle risorse di calcolo utilizzate, con costi aggiuntivi per lo storage dei dati e per il traffico di rete. Considerando un nodo di rete con un consumo di risorse leggero e picchi di *50.000* transazioni mensili, il costo mensile stimato sarebbe di circa *30 USD* (stima approssimativa basata sui prezzi attuali disponibili nella piattaforma). Sebbene l'adozione di questa soluzione sia economicamente più vantaggiosa, in questo elaborato si è scelto di esplorare l'implementazione e la pubblicazione di *Smart Contract* su reti pubbliche come Ethereum.

Lo scopo iniziale del progetto è stato quindi raggiunto: è possibile sviluppare applicazioni basate sui principi del Web3, nonostante permangano molte limitazioni e sfide aperte da affrontare.

Bibliografia

[1] Vitalik Buterin. Ethereum: A next generation smart contract & decentralized application platform, 2014.

[2] Collin Connors e Dilip Sarkar. Benefits and limitations of web3, 01 2024.

[3] OpenJS Foundation. Node.js v20.2.0 documentation, 2024.

[4] Wensheng Gan, Zhenqiang Ye, Shicheng Wan, e Philip S Yu. Web 3.0: The future of internet. *ArXiv*, 04 2023.

[5] Haojun Liu, Xinbo Luo, Hongrui Liu, e Xubo Xia. Merkle tree: A fundamental component of blockchains, 09 2021.

[6] Ali Maetouq, Salwani Mohd, Noor Azurati, Nurazean Maarop, Nilam Nur, e Hafiza Abas. Comparison of hash function algorithms against attacks: a review, 01 2018.

[7] Bernhard K Meister e Henry C W Price. Gas fees on the ethereum blockchain: From foundations to derivatives valuations, 06 2024.

[8] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 10 2008.