



UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Strategie di automatizzazione di liquidità nella finanza decentralizzata

Relatore: Prof. Alberto Leporati

Correlatore: Ing. Paolo Antonio Rossi

Relazione della prova finale di:

Christian Kobril

Matricola 856448

Anno Accademico 2021-2022

Sommario

Qui scrivo il sommario

Indice

Capitolo 1

Introduzione

Il tema centrale di questa Tesi di Laurea è l'utilizzo della blockchain nell'ambito della **finanza decentralizzata**.

In particolare, si approfondirà un prodotto software concretamente sviluppato durante il mio *Project Work*, svolto presso l'azienda *Five Elements Labs Srl*.

Inoltre, verranno trattati i concetti di *pool di liquidità*, *automated market maker* (AMM) e *strategie multiple* su Uniswap V3.

1.1 Cos'è la blockchain

Alla base di tutti i protocolli e prodotti software analizzati in questa Tesi, vi è un minimo comune denominatore che prende il nome di **blockchain**^{[?] 1}.

La blockchain altro non è che un database distribuito (o in termini contabili, *libro mastro*) **condiviso** tra più nodi di una rete, **validati** dalla rete stessa e strutturato come una catena di blocchi contenenti delle **transazioni**.

Le transazioni sono degli scambi di valore effettuati sulla blockchain tra un mittente (sender) e un destinatario (receiver). Quando una transazione è inviata sulla blockchain, i dettagli di essa quali criptovalute scambiate, asset (beni digitali), mittente e destinatario vengono validate da tutti i nodi della rete e salvate in modo permanente e trasparente sulla blockchain.

1.1.1 Caratteristiche della blockchain

È possibile riassumere le funzionalità di questa rete di blocchi in 3 fondamentali concetti:

- **tracciabilità** - tutti i partecipanti alla rete e le transazioni registrate devono poter essere rintracciabili da chiunque
- **immutabilità** - una volta che una transazione viene salvata sulla blockchain, non può essere in alcun modo alterata
- **sicurezza** - la *crittografia asimmetrica* utilizzata nella rete garantisce invio e ricezione di transazioni in modo sicuro, senza bisogno di intermediari esterni

L'utilizzo della blockchain è oggi in **piena crescita**: vi sono numerose aziende e sviluppatori indipendenti che costruiscono prodotti innovativi sfruttando le peculiarità di questa tecnologia ad alto potenziale.

1.1.2 Le dimensioni del mercato attuale

Nel 2021, secondo un report di **Electric Capital**^[?]] mostrato nella figura ?? è stata raggiunto un nuovo record di **18.000 sviluppatori attivi al mese** nello sviluppo di software basati su blockchain.

Si stima inoltre che più del **65%** degli sviluppatori blockchain abbia iniziato a lavorare nel 2021 in questo nuovo settore, con un aumento di **34.000** persone, il più alto nella storia.

12

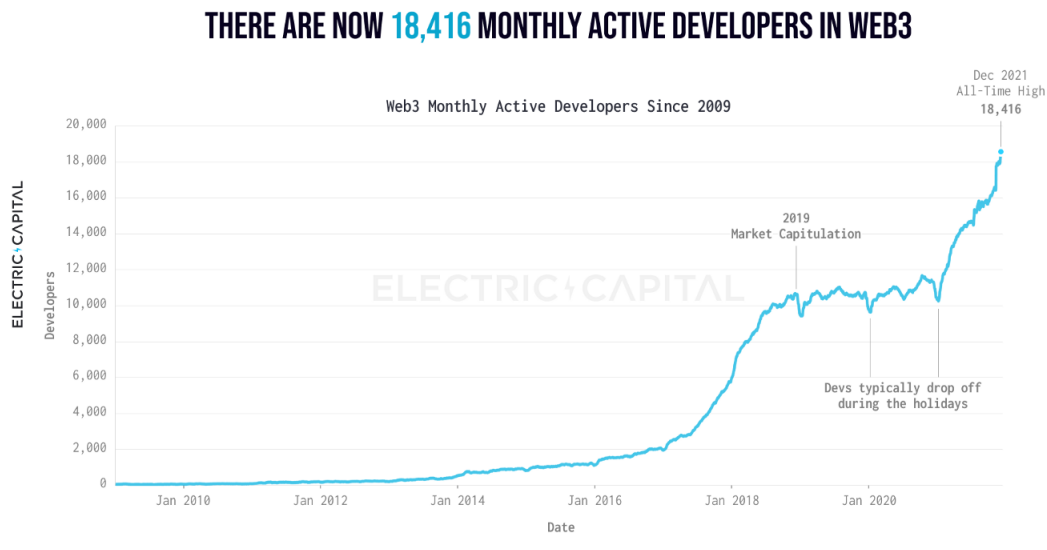


Figura 1.1: Crescita del numero di sviluppatori nel settore Web3 secondo Electric Capital

Osservando gli avvenimenti accaduti in questo ambito, credo che questo forte aumento di sviluppatori sia dovuto allo **scoppio** della tecnologia **NFT** e al rapido crescere del valore delle principali cripto valute, **Bitcoin** e **Eth**, che nel 2021 hanno raggiunto il massimo storico, attirando l'attenzione sul mercato e mettendo in luce il potenziale di questa innovativa tecnologia.

1.2 Cos'è la Finanza Decentralizzata

Il termine **finanza decentralizzata** viene usato per classificare tutti quei servizi finanziari che avvengono direttamente tra due entità su una blockchain.

1.2.1 Differenze tra Finanza Decentralizzata e Tradizionale

Per comprendere il concetto su cui si basa la finanza decentralizzata (da ora in poi *DeFi*), è bene dare un rapido sguardo alla sua controparte: la finanza tradizionale (o centralizzata).

Nella finanza centralizzata, ogni singola operazione finanziaria tra due persone (bonifici, prestiti, scambio di risorse, mutui) richiede l'interazione con soggetti di terze parti (tipicamente banche o altri enti).

Ciò incrementa le già prolisse tempistiche burocratiche, oltre ad aggiungere i costi dovuti al servizio fornito dagli enti che permettono l'operazione.

Diversamente, la finanza decentralizzata permette l'interazione tra due soggetti senza l'intermediazione di un sistema centralizzato (come una banca), bensì mediante un applicativo software costruito sopra la tecnologia blockchain, rendendo le operazioni rapide, pubbliche e sicure.

1.3 Applicazioni decentralizzate

Gli applicativi software utilizzati in DeFi vengono chiamati *dApps* (*decentralized applications*), ovvero particolari prodotti software che interagiscono con le blockchain. Di tali reti noi considereremo solo quella di **Ethereum**^{[?] 1} e le sue estensioni, note per la loro scalabilità ed accessibilità.

1.3.1 Blockchain Ethereum

La blockchain principale di Ethereum (conosciuta come **Mainnet**) è una rete sicura, scalabile, decentralizzata e soprattutto **programmabile**; ossia è possibile costruirci sopra e distribuire dApps, rendendo la rete una sorta di gigantesco marketplace in cui trovare servizi finanziari, videogiochi, social network e diverse altre applicazioni. Tuttavia, il largo utilizzo della rete da parte degli utenti, ha causato una periodica congestione del traffico delle transazioni, con un conseguente aumento dei costi per l'invio di transazioni e diminuendo la velocità, da parte della rete, di elaborarle. Per tale ragione, sono state realizzate delle blockchain separate da Ethereum, ma che la estendono per aumentarne la scalabilità: tali reti sono conosciute come reti di Livello 2 (**Layer 2**).

Le reti Layer 2 (per esempio, Polygon^{[?] 1} e Arbitrum^{[?] 1}) puntano ad aumentare la velocità delle transazioni e il volume di esse (ossia, più transazioni al secondo), in modo tale da rendere la rete accessibile a chiunque, abbattendone i costi.

1.3.2 Smart Contracts

Sulle blockchain di Ethereum (Layer 2 compreso), risiedono dei particolari programmi denominati *Smart Contracts*^{[?] 1}, i quali sono costruiti come un insieme di dati e funzioni, che risiedono ad uno specifico indirizzo sulla rete.

I contratti sono come un insieme di regole, che se soddisfatte, eseguono una determinata funzione: ciò permette alle dApps di essere regolamentate automaticamente, senza la

necessità di un mediatore. Inoltre possono contenere dei fondi (criptovalute), proprio come se fossero dei wallet digitali.

Il codice di backend delle dApps è contenuto negli Smart Contracts, distribuiti su una rete decentralizzata; vengono utilizzati per la logica applicativa delle dApps, mentre il salvataggio dei dati è effettuato sulle blockchain di Ethereum dove risiedono gli Smart Contracts.

1.3.3 Accesso alle dApps

Il concetto di login ideato nel web tradizionale (noto anche come Web2), tipicamente caratterizzato dall'inserimento di un'email e una password, viene sorpassato da una nuova autenticazione del **Web3**^{[?] 1}: attraverso il proprio *portafoglio digitale* (da ora in poi, wallet^{[?] 1}) è possibile accedere al proprio account e gestire i propri assets digitali, eventualmente mettendoli a disposizione della dApp a cui si è connessi.

I wallet presentano il vantaggio di non dover fornire nomi, indirizzi fisici, email o altre informazioni personali, garantendo la riservatezza dei propri dati; basta creare un wallet per avere immediato accesso alle piattaforme, senza registrazioni.

1.4 Operazioni in DeFi

Ogni operazione nella DeFi viene detta **transazione**. Una transazione è permanentemente salvata sui registri della blockchain, rendendo ogni singola operazione, associata ad un identificativo, consultabile in qualsiasi momento da qualsiasi persona. Tale trasparenza viene difficilmente concessa dalle banche, ponendo la DeFi come un sistema aperto e rintracciabile.

1.5 Flessibilità

L'ultima caratteristica della DeFi che ritengo importante citare è ciò che più la contraddistingue dalla finanza tradizionale: la sua flessibilità.

In qualsiasi momento un utente può trasferire i propri assets digitali, senza dover chiedere il permesso a soggetti di terze parti, evitando costose commissioni e con un'attesa che va da pochi secondi a pochi minuti.

queste
due
sezioni
possono
diven-
tare
una
sola o
essere
tol-
te/sostituite

non
sono
opera-
zioni
SO-
LO in
DeFi,
si può
spie-
gare
meglio

Capitolo 2

Uniswap, piattaforma di Liquidity Providing

Avendo approfondito cosa è la DeFi, quali sono le sue caratteristiche e i principali vantaggi, ritengo necessario concentrarsi su quali sono i prodotti "dApps" che hanno messo le basi per il lavoro svolto durante il mio Project Work.

intro
da rivedere

2.1 Uniswap

Innanzitutto, è bene distinguere la piattaforma Uniswap^{[?] 1} dall'omonimo protocollo.

La dApp di Uniswap (conosciuta come Uniswap Interface) è una piattaforma che permette agli utenti l'interazione con il protocollo Uniswap.

intro
su uniswap
da rivedere

Quest'ultimo è una suite di Smart Contracts, realizzata con lo scopo di formare ciò che nella finanza decentralizzata è definito come **Automated Market Maker** (da ora in poi AMM^{[?] 1}).

Il protocollo Uniswap implementa un sistema progettato per scambiare criptovalute **ERC-20** sulla blockchain di Ethereum.

2.2 Scambi nei mercati tradizionali

La maggior parte dei mercati tradizionali ad accesso pubblico utilizza ciò che viene definito *Order Book*^{[?] 1}, ossia un elenco degli ordini di acquisto e vendita attualmente aperti per un asset, organizzati per prezzo.

Sostanzialmente, un *sistema di corrispondenza*^{[?] 1} si occupa di abbinare gli ordini di acquisto con quelli di vendita, usando l'order book per eseguire le operazioni tra i partecipanti dello scambio.

non
spiego
cosa
sia un
asset

2.3 Automated Market Maker

La rivoluzione introdotta dalla blockchain sta nella possibilità di creare nuovi tipi di scambi che abbinano algoritmicamente ordini di acquisto e vendita utilizzando gli smart

contracts.

Tali scambi vengono detti *Scambi Decentralizzati (DEX)*.

Un AMM è un protocollo DEX che si basa su un algoritmo di valutazione per prezzare gli asset mediante una formula matematica.

Il citato order book viene rimpiazzato con una pool di liquidità^{[?] 1}, contenente due asset, entrambi valutati l'uno rispetto all'altro.

Quando un asset viene scambiato per un altro, i prezzi relativi dei due asset cambiano, e viene determinato un nuovo tasso di mercato per entrambi. In questo modo acquirenti e venditori interagiscono direttamente con la pool (e di conseguenza gli smart contracts), senza dover interagire tra di loro in modo diretto.

2.3.1 Esempio pratico di AMM

Un esempio pratico che mi ha aiutato a comprendere il funzionamento degli AMM è quello dei contadini di mele e patate.

Immaginiamo di essere un contadino e di avere solo patate, senza la possibilità di coltivare, e di conseguenza mangiare, nient'altro.

Un giorno ci viene proposto di effettuare degli scambi con un venditore di mele attraverso un messaggero, il quale decide di custodire mele e patate in un *contenitore magico*, in modo tale che rimangano a disposizione per gli scambi.

La regola fondamentale per questo scambio è una sola: *il contenitore magico dovrà sempre contenere una determinata quantità di mele e patate tale per cui il loro prodotto sia uguale ad una costante k* .

Tale regola è in realtà la formula alla base dell'AMM di Uniswap, conosciuta come **Constant Product Formula** ?? (rappresentata graficamente nella figura ??):

$$x * y = k \tag{2.1}$$

dove:

x = numero di mele nel contenitore
 y = numero di patate nel contenitore
 k = valore costante

In genere la costante k è detta **liquidità** del contenitore.

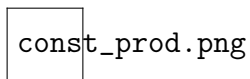


Figura 2.1: Grafico della Constant Product Formula applicata all'esempio dei contadini

Inizialmente il contenitore sarà perfettamente bilanciato, per esempio con 500 mele e 500 patate corrispondente ad una liquidità (k) pari a 250.000.

2.3.2 Scambi nel contenitore

Immaginiamo ora che un contadino voglia scambiare le sue patate per delle mele grazie al contenitore (*scambio 1*). Poniamo che il numero di patate offerte dal contadino sia pari a 70. Ricordando che la costante k deve sempre corrispondere a 250.000, il calcolo per la quantità di mele che dovrà essere presente nel contenitore è:

$$M_n = \frac{k}{P_0 + P_1} \quad (2.2)$$

dove:

M_n = nuova quantità di mele nel contenitore

P_0 = quantità iniziale di patate nel contenitore

P_1 = quantità di patate aggiunte al contenitore

Questo perché per bilanciare il contenitore mantenendo il valore k costante, il numero di mele deve diminuire. Dunque, il nuovo numero di patate nel contenitore corrisponde a 570, mentre il numero di mele, calcolato mediante la **formula ??** è uguale a 438: di conseguenza il contadino riceverà $500 - 438 = 62$ mele.

A questo punto, possiamo facilmente dimostrare che se un contadino volesse scambiare le proprie mele, riceverebbe più patate rispetto alle mele che vuole inviare. Poniamo che il contadino decida di inviare 70 mele (*scambio 2*): ripercorrendo i passaggi sopra presentati, la quantità di patate ricevute in cambio sarà uguale a 78. Ciò è dovuto al fatto la quantità di mele nel contenitore prima dello scambio 2 era minore rispetto alla quantità di patate.

Nella realtà dei fatti, questo contenitore magico è conosciuto come **pool di liquidità**.

2.4 Pool di liquidità

È possibile vedere una pool di liquidità come uno spazio in cui i contadini (trader) possono mettere a disposizione la propria liquidità di mele e patate (criptovalute, nello specifico token ERC-20¹⁷); tali utenti vengono definiti fornitori di liquidità (*liquidity providers*, da ora in poi LP).

Come ricompensa per la liquidità fornita, gli LP ricevono commissioni sulle operazioni che avvengono nella pool a cui partecipano. Tali commissioni si applicano sulle singole transazioni effettuate con la liquidità fornita da un LP, e possono variare di percentuale dal 0.01% fino all'1%.

Nel caso di Uniswap, gli LP depositano un valore equivalente di due token; per esempio, 50% ETH e 50% USDC (una stablecoin che replica il dollaro USA) nella pool ETH/USDC.

2.5 Protocollo Uniswap V3

Uniswap V3 è l'ultima versione del protocollo rilasciata da Uniswap nel maggio 2021^{[?] 1}. Tale protocollo definisce le funzionalità della suite di smart contracts con cui gli utenti interagiscono, introducendo importanti novità rispetto al suo predecessore, V2.

2.5.1 Posizioni

Utilizzando l'interfaccia Uniswap, gli utenti possono connettere il loro wallet personale per mettere a disposizione un certo ammontare di liquidità all'interno di una pool. Tale liquidità, come spiegato in precedenza, dovrà mantenere un'equa proporzione tra i due asset messi a disposizione: tale operazione viene definita come *apertura di una posizione* (o in inglese, position minting).

equa
propor-
zione,
come
mai?

Su Uniswap V3, le posizioni vengono rappresentate mediante NFT (ERC-721^{[?] 1}), i quali certificano un determinato wallet, in questo caso chi effettua il minting, come proprietario della posizione.

2.5.2 Complicazioni di Uniswap V2

In precedenza, nella V2 di Uniswap, i LP mettevano a disposizione i propri asset per scambi **a qualsiasi intervallo di prezzo**.

Consideriamo che io, trader che utilizza Uniswap V2, voglia mettere a disposizione due miei asset chiamati token *A* e token *B*. Ricordando che l'intervallo di prezzo è sempre il prezzo di *A* rispetto al prezzo di *B* (*A su B*), in questa versione del protocollo la liquidità viene uniformemente distribuita su tutti i prezzi possibili.

In questo modo non vi è alcuna perdita di liquidità, portando però un importante svantaggio: la maggior parte della liquidità non viene mai utilizzata negli scambi.

Provando a considerare una pool contenente una coppia di due stable coins, ossia token il cui prezzo rimane relativamente costante nel tempo (ad esempio, USDC vale quanto il dollaro americano), possiamo essere certi che la liquidità al di fuori del tipico intervallo di prezzo dei suddetti stable coins non verrebbe mai utilizzata.

Per esempio in Uniswap V2 la coppia DAI/USDC (**entrambi stable coins dal valore di circa \$1**) utilizza circa il 0.50% del capitale totale disponibile per gli scambi all'interno del range tra \$0.99 e \$1.01^{[?] 1}: una quantità estremamente minima rispetto al capitale.

Il resto della liquidità è distribuito nella restante fascia di prezzo tra 0 e ∞ (escluso il range sopra citato), rendendo quel capitale inutilizzabile (e dunque, non consentendo agli LP di guadagnare commissioni).

Ciò è dovuto al fatto che la liquidità sia uniformemente distribuita in un range di prezzo da 0 a ∞ , e non *concentrata* nel giusto intervallo: per tale ragione è stato introdotto **Uniswap V3**.

2.5.3 Liquidità concentrata

Ciò che rende Uniswap V3 un protocollo davvero valido è l'idea della *Liquidità Concentrata*^{[?] 1}. Tale liquidità viene distribuita in un intervallo di prezzo personalizzabile, a scelta dell'utente.

Riprendendo l'esempio sopra citato, un trader potrebbe decidere di investire nella pool DAI/USDC, scegliendo come range il più proficuo, ossia quello compreso tra \$0.99 e \$1.01. In tal modo, la liquidità concentrata garantirà un guadagno superiore di commissioni (da ora in poi fees) sfruttando il capitale messo a disposizione dai LPs.

2.5.4 Tick di prezzo

Per rendere la liquidità concentrata funzionale, lo spazio continuo del prezzo è stato partizionato in **tick**.

I tick sono i limiti di aree discrete nello spazio del prezzo. Tali limiti sono posizionati in modo tale che il diminuire o aumentare di 1 tick rappresenti l'aumento o la diminuzione percentuale del 0.01% del prezzo in ogni punto dello spazio.

Dunque quando una posizione viene creata, un LP non può scegliere qualsiasi valore per il range di prezzo: è necessario che il limite inferiore (**lower tick**) e il limite superiore (**upper tick**) corrispondano a dei tick di prezzo validi.

2.5.5 Swap e Fees

Il modo più utilizzato per interagire con Uniswap V3 è tramite gli scambi (da ora in poi **swap**). Uno swap è relativamente semplice: un utente seleziona un token ERC-20 del quale è proprietario e un token che vorrebbe scambiare per esso. Uniswap venderà il token attualmente in possesso dell'utente, restituendo una quantità proporzionale del token desiderato, sottraendo una **swap fee**, ossia quella percentuale riconosciuta ai LPs per aver messo a disposizione la loro liquidità con la quale è avvenuto lo scambio.

Tuttavia, la transazione potrebbe richiedere alcuni minuti, a seconda della rete su cui avviene, rilevando un fenomeno conosciuto come **slippage**.

Lo slippage è l'alterazione di prezzo di un token che avviene mentre la transazione è in attesa di essere completata.

Tale alterazione ha una soglia di tolleranza dell'1%; superata tale soglia l'operazione viene rifiutata e lo swap annullato, onde evitare grosse perdite per l'utente.

2.6 Complicazioni di Uniswap V3

Qualora il prezzo di un token dovesse muoversi verso una direzione (di discesa o di salita), il proprietario della posizione si ritroverebbe con un ammontare superiore di uno dei due token rispetto all'altro, in quanto il prezzo dell'uno sull'altro cambierà, fino a quando l'intera liquidità sarà relativa a solo uno dei due asset.

Per esempio, se in una pool ETH/USDC il prezzo di ETH dovesse diminuire, la percentuale di liquidità relativa a ETH aumenterebbe, per bilanciare il valore immesso all'interno della posizione stessa. Allo stesso modo, se ETH dovesse aumentare di valore, la percentuale di USDC aumenterebbe a sua volta.

Con l'aumentare o il diminuire del prezzo di un asset nella pool, tale prezzo potrebbe uscire dall'intervallo che un LP ha impostato per una certa posizione. Nel momento in cui una posizione si dovesse trovare fuori dall'intervallo scelto (**Out Of Range position**) la liquidità diventerebbe inattiva (**idle liquidity**) e l'utente proprietario di tale liquidità non guadagnerebbe più fees.

Tuttavia, nel momento in cui il valore del primo token sul secondo dovesse tornare nell'intervallo di prezzo iniziale, il LP tornerebbe a guadagnare fee.

È proprio dal problema della liquidità inattiva che nasce **Orbit DeFi**, il prodotto sviluppato durante il mio project work.

Capitolo 3

Orbit, piattaforma per automatizzare strategie DeFi

Ho avuto l'opportunità di svolgere il mio Project Work per Five Elements Labs, azienda specializzata nella produzione di software nel mondo blockchain; in particolare nei settori DeFi e NFT.

Durante tale esperienza, mi sono unito allo sviluppo del loro principale prodotto: **Orbit**^[?] .

3.1 Introduzione ad Orbit

Orbit è una piattaforma di gestione di liquidità nel settore DeFi: ossia un **layer** che permetta ai propri utenti di automatizzare strategie e di ottimizzare posizioni di liquidità su Uniswap V3.

Può essere interpretato come **un'estensione** del proprio wallet, in grado di gestire gli assets degli utenti per fornire la possibilità di ottenere un ritorno aggiuntivo dalla liquidità concentrata.

3.2 Perché nasce Orbit

Con il rapido crescere del settore DeFi e del corrispondente ecosistema di protocolli, ognuno con le proprie caratteristiche e logiche, risulta sempre più complesso gestire delle strategie di liquidità al passo con i tempi.

3.2.1 Gas fee

Ogni transazione avvenuta sulla blockchain ha un "*costo*" chiamato **gas fee**^[?] .

È possibile affermare che il gas sta alla blockchain come la **benzina** sta alla macchina; è necessario per far funzionare i nodi che compongono la rete.

Sostanzialmente è l'unità di misura dello *sforzo computazionale* fatto dai nodi per sostenere una transazione, la quale può prevedere diverse complesse operazioni al suo interno.

Tale benzina deve essere in qualche modo pagata, per questo esistono commissioni sul gas (gas fee).

Diverse blockchain con basse gas fee (come **Polygon**^{[?] 1}) stanno diventando sempre più popolari, spostando l'attenzione degli sviluppatori e degli utenti verso la possibilità di costruire piattaforme e strumenti veloci ed efficaci.

Orbit dunque cavalca quest'onda di innovazione e di creatività, portando la liquidità concentrata verso un nuovo livello, costruendo uno strumento efficiente e facile da utilizzare per professionisti e novizi del mondo DeFi.

3.3 Vantaggi per gli utenti

Automatizzare strategie riguardo posizioni su Uniswap V3 ha un impatto diretto sui **ritorni generati** da queste ultime.

La maggior parte dei protocolli presenti sul mercato forniscono ai trader **strategie attive**, sulle quali è necessario compiere delle scelte complesse conosciute e comprese perlopiù da utenti professionisti.

Le funzionalità fornite da Orbit permettono agli utenti di creare **strategie passive**, senza il necessario bisogno di rimanere aggiornati sui protocolli, bensì lasciando alla piattaforma il compito di gestire la propria liquidità.

Inoltre, nella prima versione del protocollo, sarà Orbit ad occuparsi dei costi di gas dovuti alle transazioni rivolte agli smart contract dell'applicazione.

3.4 Modelli di gestione di liquidità

Nello stato attuale della DeFi, vi sono concretamente due modelli ben distinti di gestione di liquidità: **Aggregatori**^{[?] 1} e **Smart Vaults**^{[?] 1}.

3.4.1 Aggregatori

Gli Aggregatori sono delle particolari piattaforme DeFi, le quali permettono ai propri utenti di effettuare transazioni verso diverse dApps **in un unico posto**. Ciò permette di risparmiare tempo e aumentare l'efficienza delle transazioni.

Tali Aggregatori permettono ai trader di **replicare** strategie di utenti esperti e di applicarle al proprio portfolio. Per esempio, possono confrontare i prezzi degli assets su diverse piattaforme, proponendo lo scambio più conveniente all'utente.

Tuttavia, protocolli utilizzatori di Aggregatori come *Yearn Finance*^{[?] 1}, *Beefy*^{[?] 1} o *Idle*^{[?] 1} consentono all'utente l'utilizzo di strategie singole, tipicamente con un modello "*Black Box*", ovvero senza verificarne l'effettivo funzionamento interno.

3.4.2 Smarts vaults

Contrariamente agli Aggregatori, gli **Smart Vaults** garantiscono un'alta **personalizzazione** delle strategie scelta direttamente dagli utenti.

Il funzionamento è relativamente semplice: un utente diventa *proprietario* di un particolare Smart Contract, effettuando una transazione che ne crea un'istanza contenente l'indirizzo del suo creatore¹.

Successivamente, il contratto creato viene utilizzato come *un'estensione* del proprio wallet per gestire assets e per interagire con altri protocolli per consentire allocazioni automatiche di liquidità: tale Smart Contract è chiamato Smart Vault.

Il modello a Smart Vault è stato poco utilizzato in passato, principalmente a causa delle alte gas fee richieste dalle reti per attivare strategie multiple.

Tuttavia, con l'avvento di blockchain sempre meno costose in termini di gas, è ora possibile utilizzare gli Smart Vaults per integrare facilmente nuovi protocolli e fornire all'utente la possibilità di avere un totale controllo sulle interazioni con essi.

Per tali ragioni, il modello a Smart Vault è stato scelto per la realizzazione di Orbit.

3.5 Moduli di Orbit

La prima versione di Orbit rilasciata ad Agosto 2022 presenta 3 caratteristiche principali chiamate **moduli**:

- **Autocompound**
- **Autorebalance**
- **Idle liquidity**

I moduli sono delle *opzioni* attivabili su una determinata posizione di Uniswap V3 depositata nel proprio Smart Vault su Orbit.

Ciascuno di tali moduli è rappresentato da uno smart contract, il quale innescherà delle specifiche **azioni** sulla base di determinati parametri relativi allo stato della posizione su cui agiscono.

3.5.1 Autocompound module

Il modulo di autocompound è sostanzialmente la traslazione verso il Web3 del *compounding* nella finanza tradizionale. Il compounding si effettua quando un investitore decide di aggiungere gli interessi maturati da un investimento al capitale inizialmente investito, rendendoli a loro volta idonei per generare ulteriore interesse.

Nel mondo DeFi gli interessi generati sono le fee generate da una posizione, mentre il capitale investito è la liquidità messa a disposizione in una determinata pool.

¹Approfondimenti tecnici riguardanti la creazione dello Smart Vault e dei relativi Smart Contracts verranno trattati nel capitolo 5

Ricordiamo che i LP guadagnano fee su Uniswap V3 mentre i token delle posizioni da loro aperte si trovano in un determinato range di prezzo. Il modulo di autocompound reinveste periodicamente le fee generate; tale operazione viene effettuata da bot esterni chiamati **keepers**.

Gli utenti più esperti possono scegliere delle soglie, superate le quali debba essere azionato il modulo di Autocompound, provocando il reinvestimento delle fee. Per esempio, un utente può scegliere di innescare il modulo solo quando il valore delle fee generate supera il **5%** del totale della liquidità investita nella posizione.

Nella tabella sottostante si può notare di quanto possa aumentare il **ritorno generato** con il modulo di Autocompound attivo (i dati sono presi da uno studio effettuato da Five Elements Labs):

Ritorno in percentuale		
Autocompound attivato	Autocompound disattivato	Differenza
5,00%	4,89%	0,11%
10,00%	9,57%	0,43%
15,00%	14,06%	0,94%
20,00%	18,37%	1,63%
30,00%	26,50%	3,50%
40,00%	34,10%	5,90%
60,00%	47,93%	12,07%

Il modulo di autocompound presenta dunque un vantaggio per l'utente **incrementale**: più la posizione genera fee, più l'uso del modulo diventerà redditizio.

3.5.2 Autorebalance module

I successivi due moduli vertono su una delle maggiori complicazioni di Uniswap V3: la **liquidità inattiva**.

Nel momento in cui questa Tesi viene scritta, il **TVL** (*Total Value Locked*, ossia capitale in DeFi) è pari a circa **\$10 miliardi**, rendendo la liquidità inattiva un problema non indifferente.

Il modulo di **autorebalance** permette di spostare il range di prezzo di una posizione che si trova fuori dal range di guadagno effettivo, centrandolo sul prezzo attuale del rapporto della coppia dei due Token presenti nella pool.

Nella figura ??, si può notare come il prezzo attuale, rappresentato dallo spillo viola, si trovi attualmente all'interno del range verde. In questo caso, il proprietario della posizione sta guadagnando delle fee.

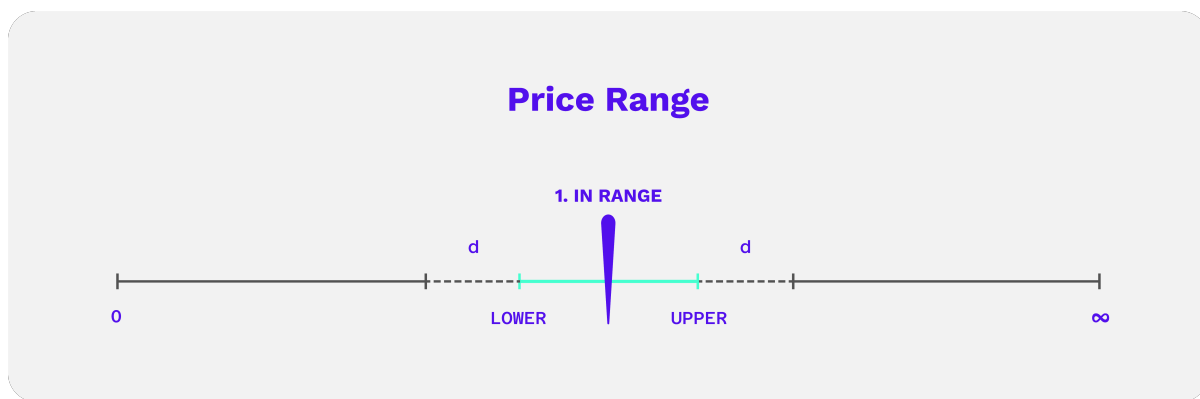


Figura 3.1: Esempio di posizione in range - Orbit Documentation

Possiamo invece trovarci nella situazione della figura ??, dove il prezzo attuale del rapporto della coppia dei Token della pool si trova al di fuori del range scelto inizialmente dall'utente.

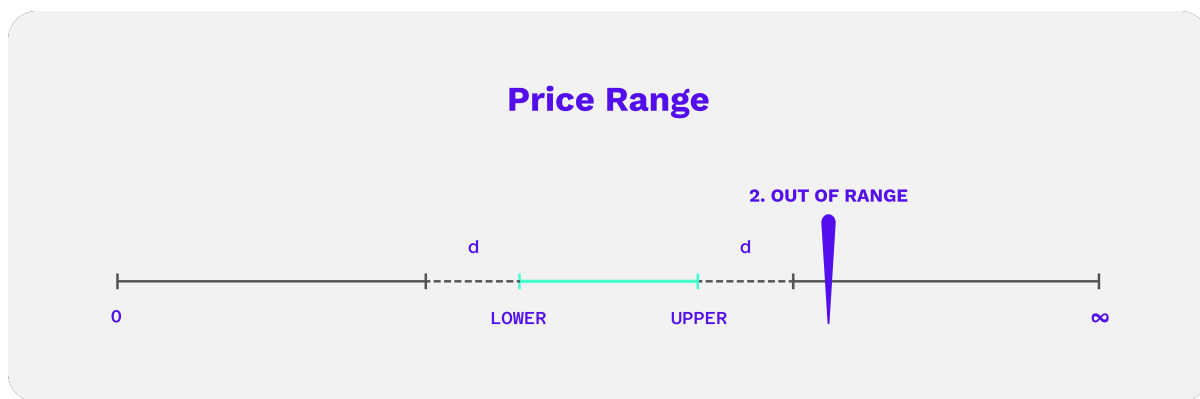


Figura 3.2: Esempio di posizione fuori range - Orbit Documentation

In questo caso, una volta superata la soglia delimitata dai due segmenti d , estensioni del **lower e upper bound** dell'intervallo di prezzo, il modulo di autorebalance viene innescato, impostando un nuovo range di prezzo **centrato sul prezzo attuale** (figura ??). In questo modo gli utenti non smettono mai di guadagnare fee, nonostante il prezzo possa variare molto rispetto al range scelto alla creazione della posizione.

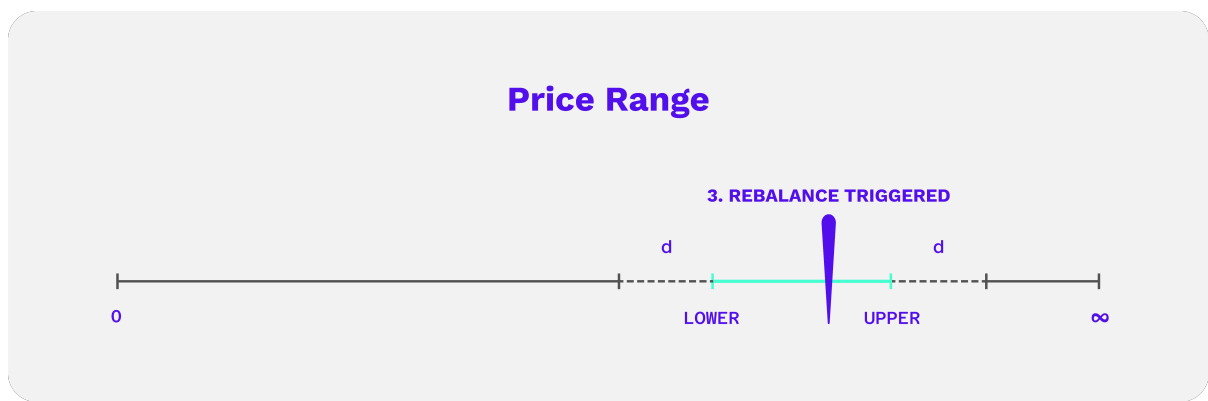


Figura 3.3: Nuovo range dopo l'attivazione del modulo di autorebalance - Orbit Documentation

3.5.3 Idle liquidity module

L'ultimo modulo di Orbit, **Idle liquidity module**, affronta sempre il problema della liquidità inattiva, ma adottando una diversa strategia rispetto all'autorebalance. Il suo innesco avviene comunque quando il prezzo attuale del rapporto dei due token in una pool si trova al di fuori del range scelto per una determinata posizione. Quando ciò accade, la liquidità della posizione viene trasferita su altri protocolli (al di fuori di Uniswap V3), come per esempio **Aave Protocol**, attualmente implementato su Orbit (figura ??).

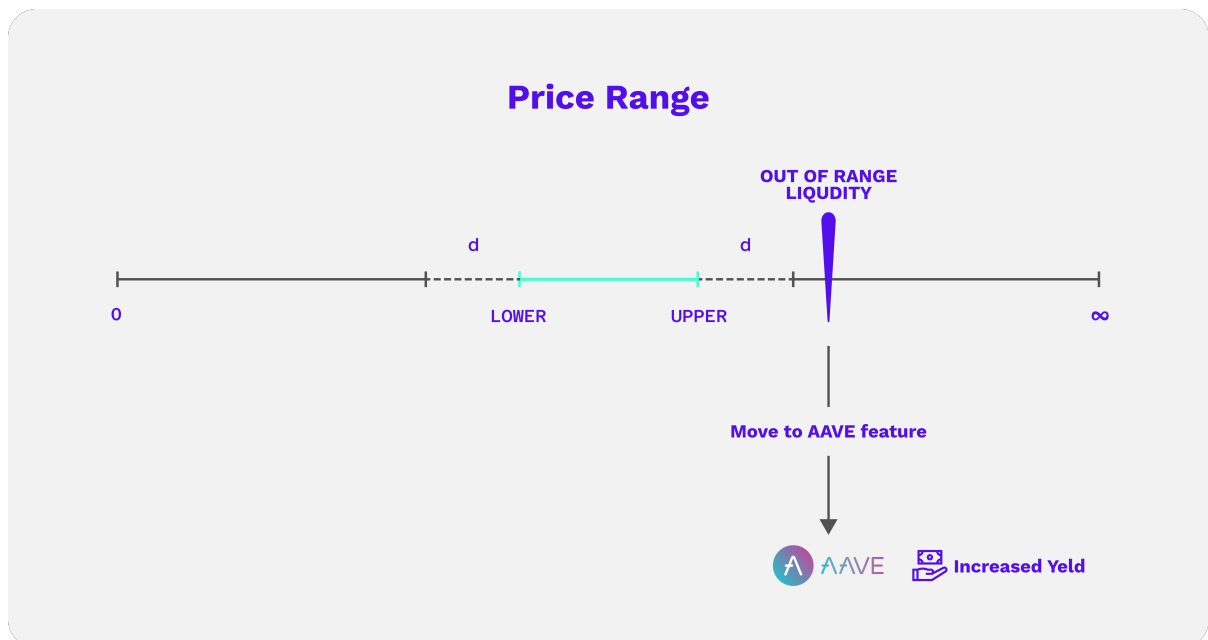


Figura 3.4: Liquidità inattiva spostata su Aave - Orbit Documentation

Tali protocolli permettono all'utente di usufruire di modi alternativi di guadagnare fee sulla liquidità fornita, così da rendere proficua la posizione nonostante si trovi al

di fuori dell'intervallo prestabilito. Una volta che il prezzo torna nell'intervallo scelto dall'utente, la posizione sugli altri protocolli viene chiusa, e nuovamente aperta su Uniswap V3, come mostrato in figura ??.

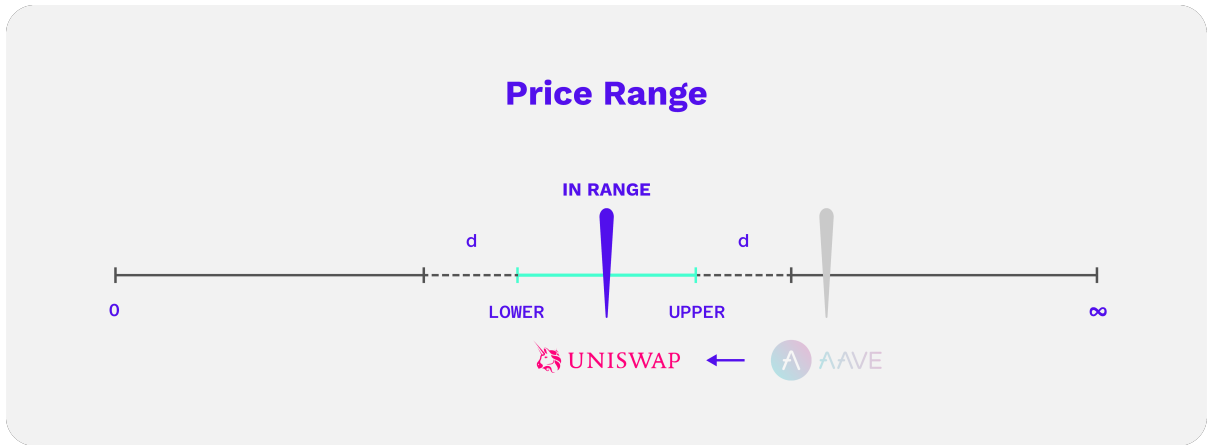


Figura 3.5: Posizione chiusa su Aave ma riaperta su Uniswap V3 - Orbit Documentation

Per ovvie ragioni logistiche, il modulo di autorebalance e quello di idle liquidity sono *mutualmente esclusivi*, dunque non possono essere contemporaneamente attivi su una stessa posizione.

Capitolo 4

Tecnologie utilizzate nello sviluppo di Orbit

Orbit è una piattaforma che prende forma nel mondo del Web3, pertanto per la sua realizzazione sono state richieste tecnologie specifiche di questo emergente settore.

Possiamo logicamente suddividere Orbit in due macro parti: il **frontend**, ossia l'interfaccia della dApp^{[?] 1} con la quale l'utente interagisce direttamente, ed il **backend**, composto da una suite di contratti che racchiudono le logiche e i meccanismi su cui Orbit si basa.

4.1 Tecnologie Frontend e librerie utilizzate

La scelta del linguaggio utilizzato per la dApp Orbit, trattandosi di un'applicazione web, è ricaduta sul linguaggio **Javascript**^{[?] 1}.

Le caratteristiche del linguaggio, quali *versatilità, leggerezza e facilità d'apprendimento* hanno condizionato questa scelta.

Inoltre, a seguito di uno studio di mercato riguardante le librerie utilizzate dalle moderne dApps, incrociato con le competenze degli sviluppatori del team di Five Elements Labs, sono state selezionate una serie di librerie coerenti con il linguaggio scelto.

Lo sviluppo del frontend di Orbit e la relativa interazione con wallet e smart contracts è stato il principale compito affidatomi durante il mio Project Work.

4.1.1 Elementi dell'interfaccia

Per la creazione degli elementi che formano l'interfaccia di Orbit, è stata scelta **React-JS**^{[?] 1}, una libreria Javascript per la sviluppo di **Single Page Application** (SPA^{[?] 1}), che permette la creazione di componenti flessibili e riutilizzabili, nonché la più utilizzata nel 2021 tra le librerie web secondo *StackOverflow*^{[?] 1} (figura ??).

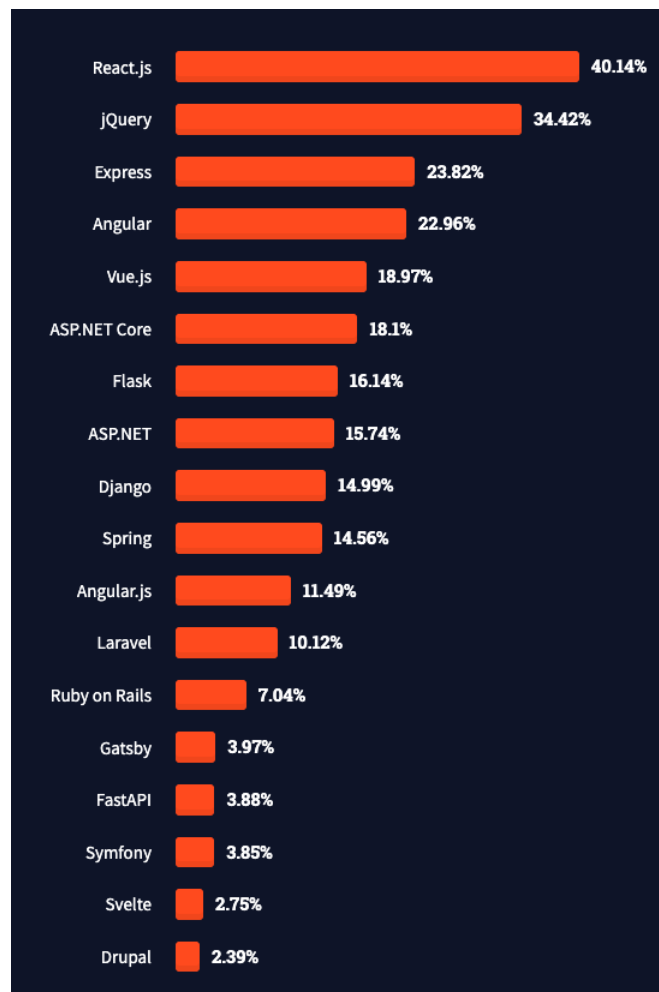


Figura 4.1: Librerie web più utilizzate del 2021 secondo un report di Stack Overflow

Per quanto riguarda invece lo stile dei componenti è stata utilizzata **Tailwind CSS**^{[?] 1}, un framework CSS che permette di utilizzare classi per il layout, colori, tipografia e altre caratteristiche stilistiche dell'interfaccia grafica.

4.1.2 Interazione con gli Smart Contracts

Ciò che più differenzia un'applicazione Web2 da una dApp in Web3 è l'interazione con gli Smart Contracts.

La libreria scelta per tale interazione è stata **ethers.js**^{[?] 1}: come suggerisce il nome, si tratta di una libreria Javascript che mira a facilitare l'interazione con la Blockchain di Ethereum (nel nostro caso, Polygon).

Tra le sue molteplici funzionalità, vi sono la *connessione di un wallet* alla dApp, la *creazione di istanze di oggetti che rappresentano Smart Contracts* utilizzabili direttamente nel Frontend ed infine la *firma dei messaggi relativi alle transazioni*.

4.2 Connessione wallet to wallet

Analogamente all'interazione "*Client-Server*" del web tradizionale, vi è un'interazione *Wallet-to-Wallet* nel Web3. Ogni utente può avere uno o più wallet digitali, ciascuno dei quali viene identificato da un indirizzo (**Ethereum Address**) a **42 caratteri alfanumerici**. Di questi, 40 sono in formato numerico esadecimale e i restanti 2 compongono il prefisso **0x**.

A questo punto, è lecito chiedersi come possa un wallet interagire con gli Smart Contracts di Orbit.

Nello stesso modo in cui i wallet possono comunicare tra loro, ossia attraverso l'invio e ricezione di transazioni, anche uno Smart Contract può ricevere delle transazioni, essendo identificato sulla blockchain con un proprio indirizzo esadecimale.

Per fare un esempio, lo Smart Contract del token **WETH** presente sulla rete Polygon, si trova all'indirizzo:

`0x7ceb23fd6bc0add59e62ac25578270cff1b9f619`

Ciò significa che se un wallet volesse interagire con il contratto, magari per controllare l'ammontare di WETH che possiede, dovrebbe inviare una transazione a tale indirizzo. Dunque, a seconda dei parametri impostati, verrà eseguita una specifica funzione presente nello Smart Contract, che potrebbe utilizzare con i dati presenti in esso, modificandoli o inviandoli al mittente della transazione.

4.2.1 Metamask e Wallet Connect

Per dare la possibilità ad un utente di connettere il proprio wallet ad Orbit ed effettuare le operazioni appena discusse, è stato scelto di implementare due sistemi: il primo, forse più noto nel Web3, è il wallet per criptovalute **Metamask**^{[?] 1}, mentre il secondo è il protocollo **WalletConnect**^{[?] 1}.

Metamask non è solo un wallet digitale, ma un vero e proprio *gateway* per interagire con applicazioni decentralizzate. Come si può vedere nella figura ??, si presenta come un'estensione del browser, dalla quale si possono gestire più wallet personali presenti su blockchain diverse, visualizzare lo storico delle transazioni effettuate, l'ammontare dei token ERC-20 posseduti dall'utente ed effettuare transazioni.

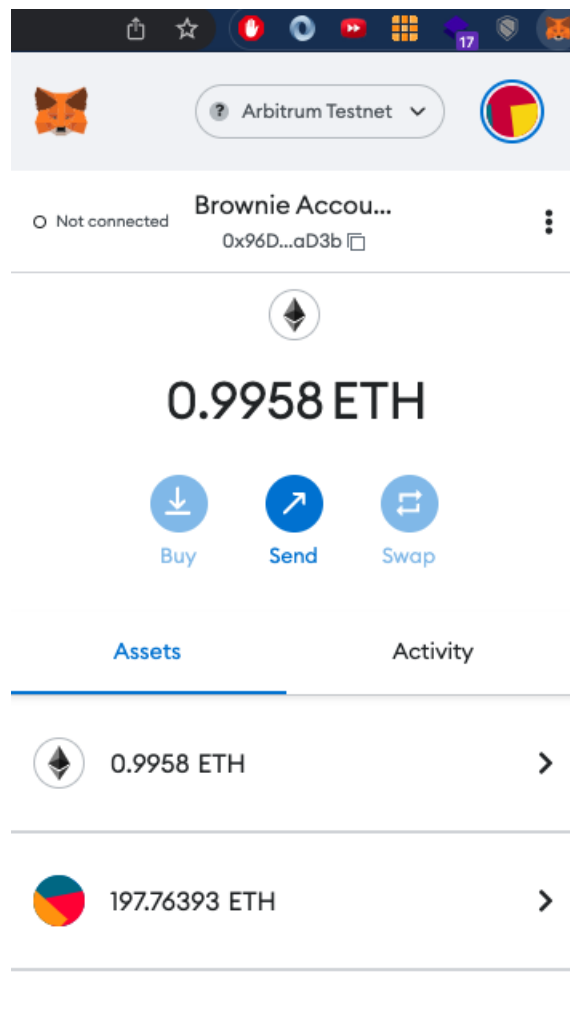


Figura 4.2: Screenshot di uno dei miei personali wallet su Metamask, estensione browser

WalletConnect è invece un protocollo che permette di integrare all'interno della propria dApp più tipologie di wallet (tra i quali anche Metamask).

Le principali caratteristiche di WalletConnect sono **adattabilità** e **facilità di integrazione** con un qualsiasi portafoglio digitale: basterà che l'utente selezioni la propria wallet app o che inquadri un QR Code per effettuare immediatamente la connessione alla dApp di Orbit.

Inoltre, è una valida soluzione per facilitare la connessione ad una dApp da dispositivi mobili, che allo stato attuale risulta ostica per la maggior parte dei prodotti Web3.

4.2.2 Usabilità del sistema

Nei mesi precedenti al lancio del prodotto, è stato svolto un importante lavoro di analisi di usabilità per rendere Orbit una piattaforma quanto più accessibile per i propri utenti, svolgendo delle sessioni di **Beta Testing**, attraverso le quali sono sorti dubbi e feedback che hanno estremamente semplificato il flusso utente del prodotto finale.

In quest'ultima parte, sono riuscito a dare il mio contributo attraverso nozioni imparate nel mio percorso di studi, in particolare in *Interazione Uomo-Macchina*, quali principi di design dell'usabilità, scelta di palette cromatiche inclusive per persone con anomalie visive quali il daltonismo, ed euristiche per facilitare l'esperienza utente.

4.3 Tecnologie Backend

Il Backend è ciò che l'utente non vede concretamente, ma che contiene la logica applicativa di Orbit, costruita sopra una suite di Smart Contracts che ricoprono le principali funzionalità della piattaforma.

4.3.1 Solidity

Per lo sviluppo degli Smart Contracts di Orbit è stato scelto **Solidity**^[?] , il linguaggio di programmazione che viene utilizzato per sviluppare la maggior parte delle tecnologie blockchain presenti sul mercato. Nonostante la sua iniziale difficoltà di apprendimento, l'ecosistema di Solidity permette di avere una vasta scelta di librerie e contratti consolidati su cui basare i propri progetti.

Si tratta di un linguaggio *orientato agli oggetti*, di alto livello e che supporta *complessi tipi di oggetti* definiti dallo sviluppatore, ideale per ospitare ogni tipo di strutture dati.

Tuttavia, come già accennato nel capitolo precedente, le blockchain **Ethereum-based** hanno un costo computazionale rilevante, misurato in gas. Per questa ragione bisogna prestare particolare attenzione quando si utilizzano linguaggi come Solidity, in quanto una semplice **implementazione di una funzione** mal realizzata potrebbe richiedere un costo extra che, a seconda della rete in cui ci si trova, potrebbe risultare in un'importante spesa per il mittente della transazione.

4.3.2 Hardhat

Data la delicatezza e fragilità dello sviluppo di una suite di Smart Contracts, è necessario avere uno strumento per testarne le principali funzioni, possibilmente prima che essi vengano rilasciati sulla blockchain. Proprio per questo è stato scelto di utilizzare **Hardhat**^[?] , un ambiente di sviluppo per costruire software sulla blockchain di Ethereum, il quale comprende diverse librerie che sinergizzano alla perfezione con lo sviluppo di Smart Contracts.

Hardhat Runner è una tra queste: viene utilizzato per costruire dei **task**, ossia azioni eseguibili tramite linea di comando. Per esempio, sono state realizzate specifiche azioni per compilare il codice degli Smart Contracts, eseguire dei test su di essi e distribuirli (*deploy*) sulla rete locale di Hardhat.

Infatti, uno dei tanti vantaggi di Hardhat è quello di testare i propri contratti su una rete locale, contenente 20 indirizzi utilizzabili per effettuare transazioni verso i propri Smart Contracts, al fine di avere un ambiente che preceda quello di produzione.

Inoltre, sono state realizzate decine di test per ogni funzione dei contratti di Orbit, realizzati nel linguaggio **Typescript** che, a differenza di Javascript, ha un forte controllo

sulla tipizzazione delle variabili, sulla costruzione di interfacce e fornisce una sicurezza maggiore nello sviluppo.

Per la costruzione delle Test Suite sono state utili delle nozioni imparate durante il corso di *Sicurezza e Affidabilità*, che mi ha permesso di dare un contributo nel lavoro svolto dai miei colleghi nello sviluppo dei contratti.

Capitolo 5

Architettura di Orbit

Durante i mesi di sviluppo di Orbit, sono state effettuate numerose ricerche da parte della squadra di sviluppo di Five Elements Labs, in modo tale da scegliere un'architettura e dei **design patterns** quanto più sicuri e aggiornati con le ultime tecnologie del mondo DeFi e Smart Contracts.

5.1 Principi fondamentali

L'architettura su cui si basa Orbit segue due principi fondamentali: **composability** (componibilità) e **upgradability** (aggiornabilità).

Il primo principio afferma che i componenti che formano un sistema software debbano essere realizzati in modo tale da poter essere riutilizzabili, per costruire nuove applicazioni. Un buon esempio di questo principio è quello dei mattoncini *Lego*: gli Smart Contracts di Orbit sono dei singoli mattoncini che combinati tra loro possono costruire strutture sempre più complesse, senza la necessità di essere riscritti da zero.

Il secondo principio invece, riguarda l'aggiornabilità di alcune funzionalità degli Smart Contracts. Gli Smart Contracts su **EVM**^{[?] 1} eseguono solo il codice per cui sono programmati al momento del loro *deploy* sulla blockchain.

Tuttavia, attraverso il Design Pattern adottato da Orbit, è possibile aggiornare alcune loro funzionalità tramite contratti esterni, abbattendo i costi di distribuzione sulla blockchain (approfondimento nella sezione ??).

5.2 Diamond Pattern

L'architettura presenta numerosi Smart Contracts, ognuno di essi adibito ad una specifica funzione all'interno della suite.

Il design pattern utilizzato è chiamato **Diamond Pattern**^{[?] 1} (figura ??) o standard EIP-2535. Si tratta di un approccio che consiste nella costruzione di un sistema di Smart Contract **modulari**, **estendibili** ed **aggiornabili**, contenuti in un spazio di memoria *virtualmente infinito*.

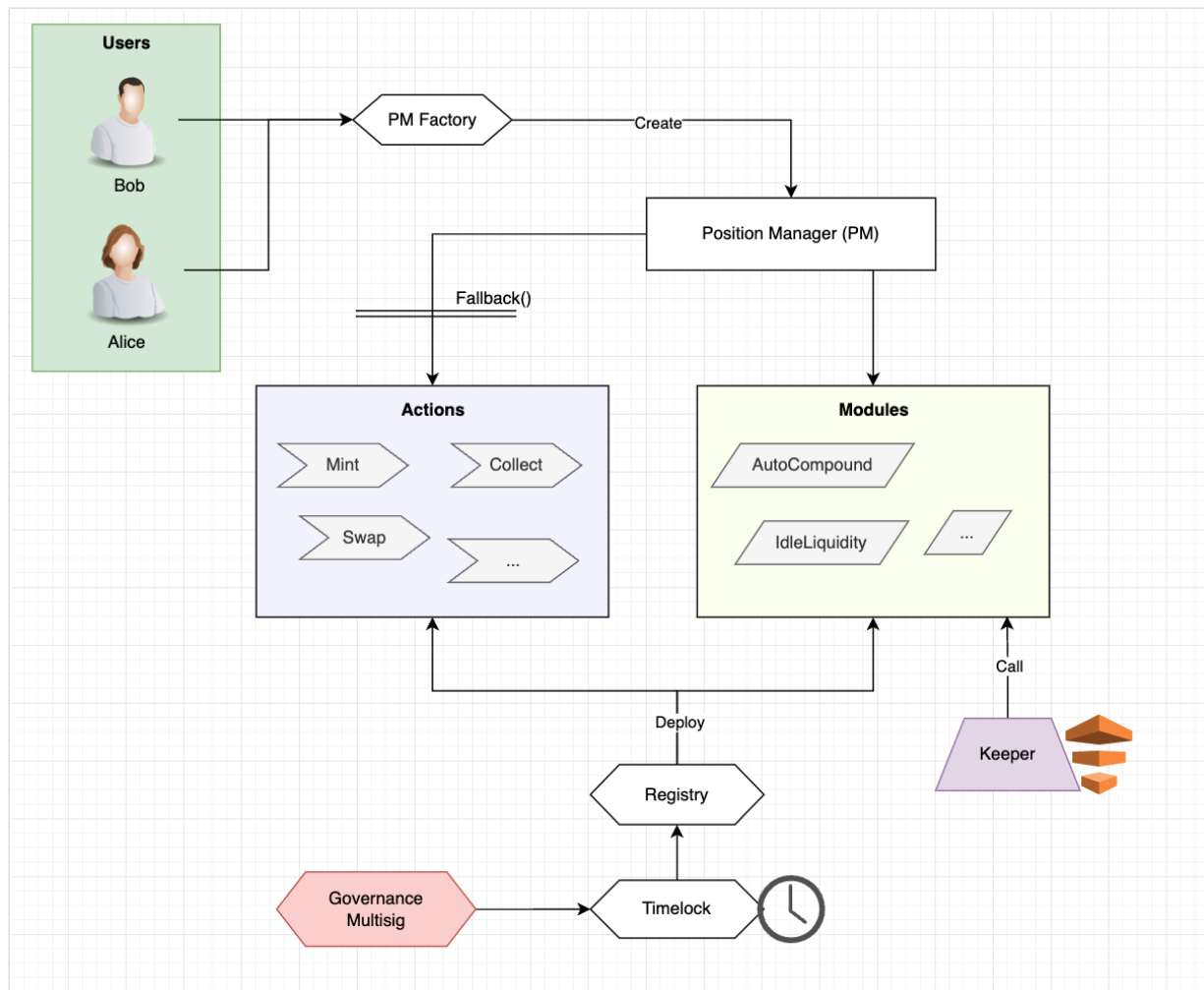


Figura 5.1: Architettura degli Smart Contracts di Orbit

5.2.1 Vantaggi del Diamond

Il Diamond Pattern è uno standard recente, che ha trovato largo uso nel mondo Web3. Il vantaggio principale sta nello sfruttare un **singolo smart contract** (chiamato Diamond), dunque un solo indirizzo ethereum, per contenere tutte le funzionalità alle quali possono accedere i contratti presenti nell'architettura.

Seppur tale contratto sia soggetto alla canonica limitazione di dimensione di **24KB**, può fare affidamento a codice presente su contratti esterni, rendendo lo spazio a sua disposizione **virtualmente infinito**.

Non meno importante è la sua **mutabilità**: tutte le funzioni di tipo *"external"*, ossia chiamabili da altri contratti, possono essere modificate, eliminate o aggiunte in un qualsiasi momento senza dover distribuire nuovamente il contratto sulla blockchain, evitando costi aggiuntivi e l'utilizzo di un nuovo indirizzo.

5.2.2 Creazione, aggiornamento e eliminazione di funzioni

Il **Diamond Pattern** possiede una forte **modularità**.

Vi sono funzioni "*external*", messe a disposizione da ulteriori contratti stateless chiamati **Facet**.

In questo caso, vi è un chiaro parallelismo tra il mondo reale e le nomenclature utilizzate nell'architettura: nel settore industriale i diamanti vengono tagliati creando delle sfaccettature (Facets), proprio come nel Diamond Pattern sono "tagliati" aggiungendo, modificando e eliminando funzioni dai Facets.

I contratti Facet vanno visti come entità separate, indipendenti tra loro ma che condividono l'accesso alle stesse funzioni, librerie e variabili di stato.

Quando un Facet viene distribuito sulla blockchain, deve essere aggiunto al Diamond tramite una transazione che ne specifica i dettagli (ad esempio, le firme di una collezione di funzioni chiamabili su quel Facet). In questo modo, il Diamond potrà eseguire il codice presente nelle funzioni del Facet, utilizzando il suo stesso storage.

5.2.3 Funzioni di fallback

Quando viene chiamata una funzione dal Diamond, è di conseguenza innescata una particolare funzione detta di **fallback**. Tale funzione determina quale Facet verrà chiamato, in base ai primi 4 *bytes di dati*, detti *function selector* (figura ??), per poi eseguire la funzione richiesta tramite il metodo **delegatecall**.

La funzione di fallback e il metodo `delegatecall` permettono ad un Diamond di eseguire una funzione di un Facet come se fosse implementata nel Diamond stesso.

Inoltre è possibile chiamare differenti Facet in una singola transazione, risparmiando notevoli quantità di Gas, raggruppando dati e funzionalità nel singolo Diamond.

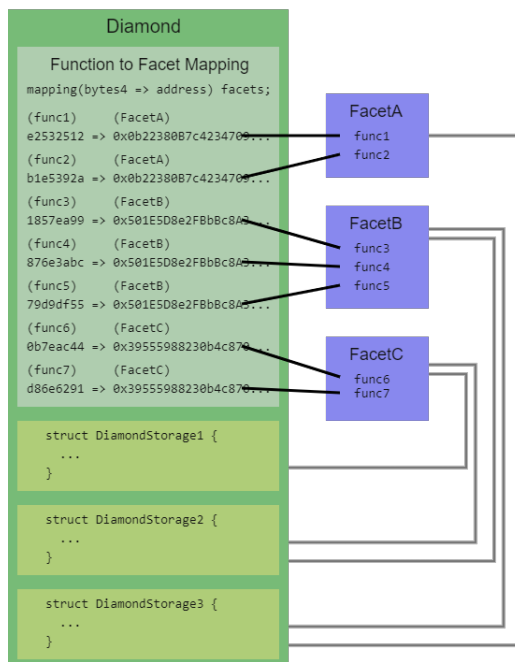


Figura 5.2: Mapping tra function selector e funzioni dei contratti Facet - EIP-2535^[7]

5.2.4 Storage nel Diamond Pattern

Il sistema che gestisce lo spazio di archiviazione (da ora in poi **storage**) può portare a problemi di collisione tra slot di memoria, a causa del metodo `delegatecall`.

Tale metodo crea un **sotto-contesto** in cui viene rimpiazzato il contratto Diamond con quello di un Facet, viene eseguita la funzione scritta sul Facet ed infine viene salvato il risultato per poi tornare al contesto principale.

Durante questo cambio di contesto, viene sostituito solo il **bytecode** del Diamond e non lo storage. Per tale ragione, se il metodo `delegatecall` provasse ad accedere un determinato slot di memoria di un Facet, esso potrebbe essere già occupato da ciò che si trova in quel determinato slot del Diamond, creando un problema di collisione in cui ci si aspetta una variabile che è effettivamente sostituita da un'altra.

Dunque, è necessario implementare uno speciale storage ad accesso casuale (slot arbitrario) associato al Diamond, appositamente creato per interagire con i Facet.

Quest'ultimi possono condividere l'accesso alle variabili di stato dello storage, usando le stesse strutture dati, che si devono trovare nelle stesse posizioni delle celle di memoria per essere accedute. In questo modo i Facet agiscono come entità separate ma con una memoria condivisa, che risiede sempre sul Diamond.

5.2.5 Position Manager e Position Manager Factory

Il contratto *Diamond* nell'architettura di Orbit è chiamato **Position Manager**; tale contratto rappresenta lo Smart Vault dell'utente.

Il suo compito principale è di gestire le posizioni di Uniswap V3 appartenenti all'utente, salvando all'interno di una struttura dati quali moduli (*autocompound*, *autorebalance*, *idle liquidity*) sono stati attivati per una relativa posizione.

Per creare un'istanza di un Position Manager è stato definito un ulteriore contratto chiamato "*PositionManagerFactory*". Ricevuta una specifica transazione, il PositionManagerFactory istanzia lo Smart Vault, creando un'associazione tra quest'ultimo e l'indirizzo del mittente della transazione. Tale associazione è salvata in una struttura dati conosciuta come **mapping**, all'interno del contratto stesso (figura ??).

```
contract PositionManagerFactory is IPositionManagerFactory {
    //array of position manager addresses created by this factory
    address[] public positionManagers;


    //mapping from user address to Position Manager address
    mapping(address => address) public override userToPositionManager;
```

Figura 5.3: Snippet di codice di una porzione del contratto PositionManagerFactory

5.2.6 Action

Gli **Action** sono Smart Contract che contengono una sola funzione; sono visti come dei singoli "mattoncini" che costituiscono le strategie proposte all'utente.

Sono estensioni del contratto **BaseAction** il quale presenta un'unica funzione **doAction**: è stato scelto un array di byte per uniformare l'input ricevuto e l'output emesso dal metodo, così da impostare uno standard funzionale per i blocchi Action, a prescindere dal loro contenuto (figura ??).



```
base_action.png
```

Figura 5.4: Snippet di codice di un esempio di contratto BaseAction

L'input viene decodificato e salvato in un'opportuna struttura dati, per poi svolgere la funzione della Action e codificare di nuovo in byte l'output ottenuto.

Utilizzati per comporre moduli più grandi, gli Action costituiscono le strategie proposte all'utente con contratti piuttosto basilari, così da poter essere riutilizzabili in diversi contesti.

Tuttavia, non tutti i contratti devono essere di tipo Action, in quanto non è consigliato (o voluto) che l'utente personalizzi troppi passaggi della strategia, ottenendo come risultato un'alta complessità d'uso.

Per esempio, poniamo di voler permettere ad un utente di aprire una posizione di Uniswap V3 senza richiedere a tale utente di approvare i Token con cui aprire la posizione. Questo perché vorremmo che il processo di approvazione dei Token, necessario per essere messi a disposizione, sia automatizzato, senza che l'utente debba approvarli da sé. Ne risulta che l'apertura della posizione (**Mint**) sarà un contratto Action, mentre l'approvazione dei Token sarà una funzione gestita diversamente (flusso mostrato in figura ??).

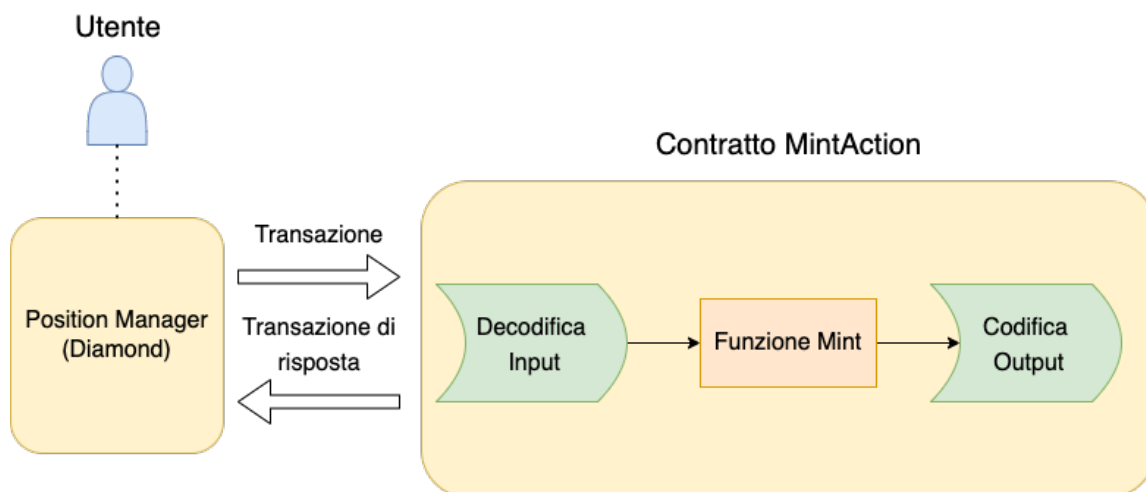


Figura 5.5: Flusso tra utente e contratto Mint Action - apertura di una posizione

5.2.7 Moduli

I moduli spiegati nella sezione ?? rappresentano il cervello di Orbit: controllano quali strategie ha attivato l'utente e comunicano al Position Manager come dovrebbe comportarsi per eseguirle.

I contratti dei moduli possono contenere solo due tipi di funzioni:

1. **pure functions**: sono funzioni che non leggono o modificano lo stato del contratto
2. **view functions**: sono funzioni che leggono solamente lo stato, senza modificarne il contenuto

Quando l'utente richiede l'attivazione di una strategia, i moduli inoltrano al Position Manager una lista di Action da chiamare, identificati mediante l'indirizzo dei contratti o un indice, e i relativi parametri in input. Inoltre, proprio come i contratti Action, i moduli utilizzano funzioni per decodificare e codificare input e output, durante l'interazione con il Position Manager (modulo di esempio mostrato in figura ??).

Tuttavia, i contratti presenti nell'architettura non sono stati pensati per controllare in che stato si trovino le posizioni di Uniswap V3 dell'utente in un determinato momento.

Per questa ragione sono stati creati degli script **off-chain** (non distribuiti sulla blockchain) chiamati **Keeper**, i quali vengono eseguiti periodicamente da delle **funzioni lambda**^{[?] [?]} caricate su **AWS**^{[?] [?]}, per controllare se è necessario eseguire delle determinate funzioni presenti sui moduli.

Per esempio, il keeper del modulo *Autocompound* controlla ogni x ore (dove x è un numero impostato nelle lambda) se l'utente ha superato una certa percentuale di fee accumulate sulla posizione, così da poter chiamare il contratto Autocompound per reinvestire tali fee.

```
contract ExampleModule {
  function swapAndDeposit () {
    ...
    //module execute callAction via PositionManager passing the swap action address and its input
    (amount0, amount1) = PositionManager.callAction(swapAction, parameters);

    //calculate parameters to pass to the next action in the sequence
    newParameters = mintParameters(amount0, amount1);

    //module execute callAction via PositionManager passing the mint action address and its input
    tokenId = PositionManager.callAction(mintAction, newParameters);
    ...
  }
}
```

Figura 5.6: Snippet di un contratto di esempio di un Modulo

5.2.8 Helpers

Gli **Helpers** sono particolari librerie o funzioni utilizzabili più volte nel codice. L'implementazione di tali librerie rispetta il principio **DRY** (Don't Repeat Yourself), evitando di scrivere codice duplicato all'interno degli Smart Contracts.

Invece, è possibile utilizzare le funzioni degli Helpers importando solo ciò di cui abbiamo bisogno negli opportuni contratti, senza dover riscrivere codice già esistente nella codebase.

5.2.9 Zapper

Il termine **Zapper** indica un contratto che permetta ad un utente di aprire o chiudere una posizione, mettendo a disposizione un solo token. Sebbene Uniswap V3 necessiti una coppia di token per aprire una posizione in una pool di liquidità, il contratto Zapper implementa la logica applicativa in grado di convertire il singolo token, depositato o prelevato dall'utente, nella coppia di token presente nella pool.

Il contratto Zapper presenta due funzioni principali: **zapIn** e **zapOut** (firma delle funzioni in figura ??).

La funzione di **zapIn** si occupa di effettuare degli *swap*, ossia degli scambi, per ottenere la giusta quantità dei due token da depositare nella pool, avendo in input un singolo token scelto dall'utente tra i due disponibili. Dopodiché, si occupa di aprire la posizione usando i due token ottenuti nel passaggio precedente.

La funzione di **zapOut** rappresenta il polo opposto: effettua il **burn** (chiusura) di una posizione, converte i due token presenti nella posizione in un singolo token a scelta dell'utente (swap) ed infine torna all'utente la corretta quantità del token selezionato.

Similmente alle Action, il contratto Zapper viene chiamato dal Position Manager per conto dell'utente, mediante l'interfaccia della dApp di Orbit.

```

//it takes a selected token, amount to mint, two pool token addresses, lower and upper
bound of the position and the fee of the pool. It returns the tokenId of the minted NF
T of the position
function zapIn(
    address tokenIn,
    uint256 amountIn,
    address token0,
    address token1,
    int24 tickLower,
    int24 tickUpper,
    uint24 fee
) public returns (uint256 tokenId);

//it takes tokenId of the position NFT to be burnt, a selected token to withdraw and it
returns the withdrawn amount
function zapOut(uint256 tokenId, address tokenOut) public returns (uint256 amount);

```

Figura 5.7: Firma delle funzioni zapIn e zapOut presenti nel contratto Zapper

5.3 Sicurezza

5.3.1 Registry Pattern

Uno dei principi cardini dell'architettura di Orbit è la *mutabilità* dei contratti.

L'obiettivo del **Registry Pattern** è avere un modo per aggiornare gli Smart Contracts ed accedere solamente alla versione più recente di essi.

Il contratto *Registry* si occupa di mantenere un **mapping** per tenere traccia di tutti i contratti dei moduli, controlla se e quali moduli sono attivi ed infine permette di disabilitare o modificarli qualora venisse riscontrato un bug o una nuova versione di tali contratti venisse distribuita sulla blockchain.

Un esempio di mapping all'interno di un contratto Registry è mostrato in figura ??:

```

struct Entry {
    address contractAddress;
    bool activated;
}

//mapping from module id to module info
mapping(bytes32 => Entry) public modules;

```

Figura 5.8: Esempio di mapping tra indirizzi nel contratto Registry

Una possibile rappresentazione visuale di tale contratto è mostrata in figura ??:

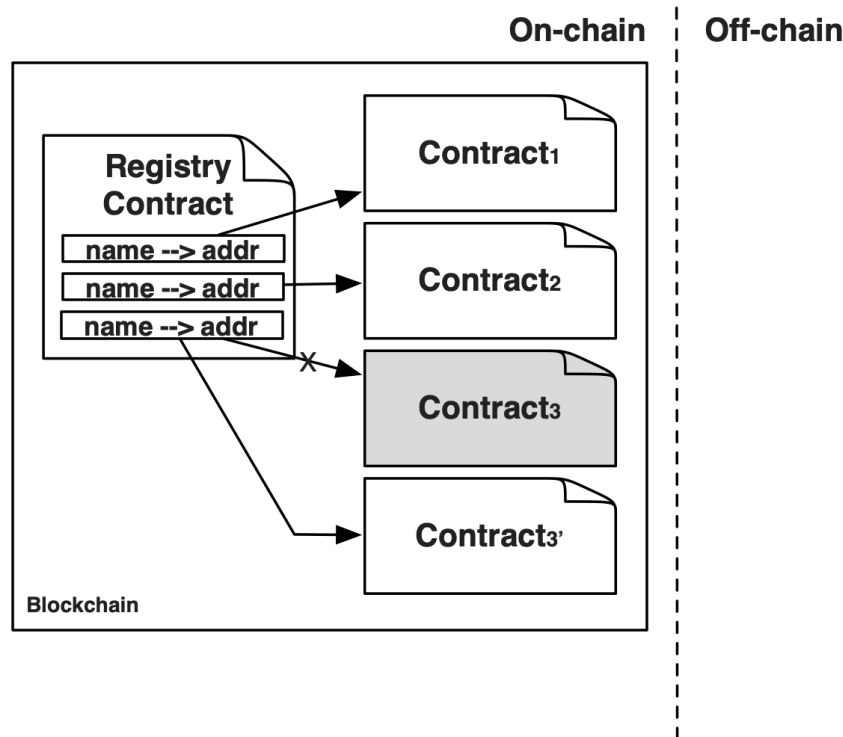


Figura 5.9: Registry Pattern, proposto da CSIRO^{[?] 1}

All'interno del contratto vi sono inoltre funzioni per aggiungere nuovi moduli o azioni e per modificarne lo stato, introducendo nuovi elementi all'interno del mapping. Quando il Diamond proverà ad accedere ad un contratto, il Registry si occuperà di *reindirizzarlo* all'ultima versione disponibile di tale contratto.

5.3.2 Timelock

Un **Timelock** è una porzione di codice all'interno di uno Smart Contract che può *bloc-care temporaneamente* alcune funzionalità del contratto stesso.

L'utilizzo di un timelock fornisce un ulteriore strato di sicurezza tra gli utenti di Orbit e il prodotto stesso.

Immaginiamo che vi possa essere una falla di sicurezza all'interno dell'architettura di questo tipo:

1. un malintenzionato aggiunge un Action e il relativo indirizzo all'interno delle Action disponibili
2. il malintenzionato può ora chiamare tale azione tramite il Position Manager di un utente
3. il codice malevolo nella Action ha pieno accesso ai fondi del contratto Position Manager e può potenzialmente prelevare tutta la liquidità investita in posizioni da parte dell'utente

Con un timelock, il primo passaggio (1) dell'attacco non è possibile: l'attaccante può solo inoltrare una richiesta di aggiunta della Action, ma tale cambiamento verrà propagato solo dopo un certo periodo di tempo (per esempio, una settimana). L'utente interessato e il team di sviluppo possono avere dunque il tempo di controllare il cambiamento proposto e spostare i fondi dell'utente in un posto sicuro, prima che venga approvato.

5.4 Audit

La dApp di Orbit è stata costruita dando priorità elevata alla sicurezza dei propri utenti. Per questa ragione è stata svolta un processo di **Audit**^{[?] [1]} da parte di *ByteRocket*. Un Audit è un esame metodico che comprende l'analisi del codice degli Smart Contracts di un prodotto.

Tale processo ha come obiettivo la scoperta di errori, problematiche, falle alla sicurezza e potenziali vulnerabilità nel codice. A fine analisi, viene rilasciato un report in cui vengono suggeriti miglioramenti e modi per risolvere le vulnerabilità e i bug, suddivisi per severità, come mostrato in figura ??.

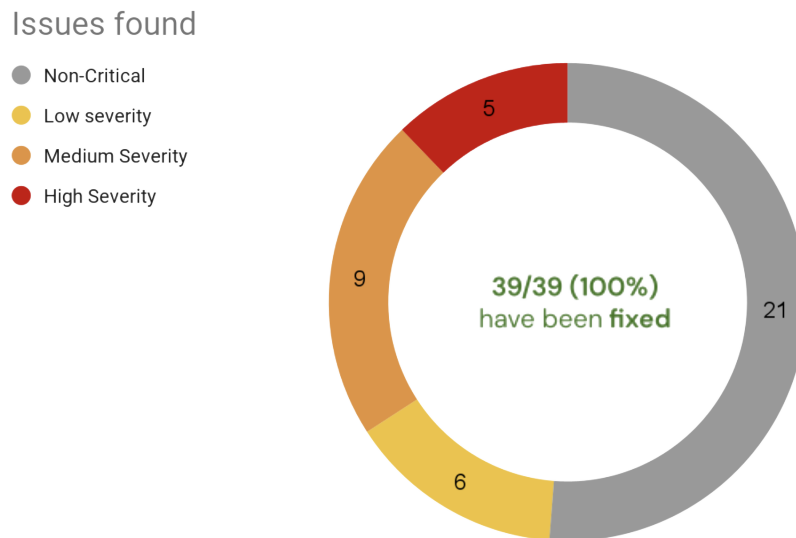


Figura 5.10: Problemi rilevati da ByteRocket nella prima Audit di Orbit

Come team di sviluppo, abbiamo risolto i 39 problemi trovati da ByteRocket, ottenendo una certificazione di completamento dell'Audit, al fine di garantire agli utenti di Orbit un elevato grado di sicurezza ed affidabilità.

Capitolo 6

Future implementazione all'interno di Orbit

aggiungo
riferi-
menti a
libri di
ts, js,
solidity,
maste-
ring
ethe-
reum,
block-
chain,
react