

# **Analisi e comparazione dei sistemi DeFI che usano AMM e Liquidity Pools**

Cocco Mattia 65336 e Lepuri Tomas 65358

Università degli Studi di Cagliari

Corso di Laurea Magistrale in Informatica

Blockchain and Smart Contracts

Anno Accademico 2024-2025

# Indice

<i>Decentralized Finance</i> .....	3
<i>Automated Market Makers</i> .....	4
Esempio implementativo .....	5
<i>Liquidity pools</i> .....	8
Esempio implementativo .....	8
<i>Attori Principali</i> .....	11
Decentralized Exchanges .....	11
Liquidity providers e Yield Aggregators .....	12
Decentralized Autonomous Organizations.....	14
Oracoli, Wallet e API .....	15
Sicurezza.....	16
<i>Valutazione delle strategie di profitto</i> .....	17
Trading Fees .....	17
Liquidity Mining.....	18
Lending su LP Tokens.....	20
Servizi infrastrutturali .....	20
<i>Uniswap v4</i> .....	21
Hooks .....	21
Singleton e risparmio del gas .....	24
Funzioni elementari .....	25
<i>Scenario d'uso completo</i> .....	27
Creazione dei token .....	27
Automated Market Maker .....	28
Test .....	32
<i>Bibliografia</i> .....	37
<i>Licenza</i> .....	37

## Decentralized Finance

La DeFI, o **Finanza Decentralizzata**, consiste in un insieme di protocolli e applicazioni, generalmente basate su tecnologie blockchain, con lo scopo di offrire servizi finanziari senza la necessità di intermediari centralizzati, come banche, broker o assicurazioni, di cui bisogna necessariamente fidarsi.

Nella finanza tradizionale infatti (centralizzata), ogni transazione e servizio in generale sono gestiti da un ente centrale, rigidamente regolamentato da requisiti e autorizzazioni, con una trasparenza limitata e velocità delle operazioni strettamente legata ai ritmi lavorativi del settore.

Al contrario, la DeFI sfrutta gli **smart contracts**, programmi autonomi e immutabili, in continua esecuzione su blockchain pubbliche, come Ethereum, che permettono di automatizzare operazioni come scambi di valute, prestiti, gestione di assets e rendimenti, garantendo più accessibilità, trasparenza, rapidità e disponibilità dei servizi.

Possiamo infatti riassumere alcuni principi cardine:

- **Decentralizzazione o disintermediazione:** elimina l'esigenza di fiducia verso banche o altre istituzioni.
- **Trasparenza:** il codice è open-source e le transazioni sono registrate su blockchain pubblicamente consultabili.
- **Accessibilità:** è possibile accedere a questi servizi con meno controlli bancari o territoriali, dato che molti provider di servizi hanno comunque bisogno di dotarsi di misure di controllo e KYC per poter essere regolamentati nell'UE o simili.
- **Composability:** i protocolli DeFI possono essere integrati tra loro, creando applicazioni più complesse.

La nascita della Finanza Decentralizzata può essere ricondotta alla diffusione di Ethereum, che ha introdotto la possibilità di scrivere questi contratti intelligenti. Tra i primi protocolli, vale la pena citarne alcuni:

- **MakerDAO:** il protocollo che permise la prima stablecoin decentralizzata.
- **Compound:** un protocollo di tasso di interesse autonomo.
- **Uniswap:** uno swap di criptovalute decentralizzato, che verrà approfondito in seguito.

Con la sua evoluzione, questo ecosistema è arrivato a comprendere decine di miliardi di dollari in Total Value Locked.

## Automated Market Makers

Un AMM, o **Automated Market Maker**, è un tipo di algoritmo, impiegato principalmente negli Exchange Decentralizzati, che consente lo scambio di token senza la necessità di una controparte diretta umana (senza un order book).

Nei mercati tradizionali la domanda di beni o valute viene abbinata all'offerta, mentre gli AMM utilizzano una formula matematica per determinare il prezzo dei token, sulla base della disponibilità nei liquidity pools: sostanzialmente degli smart contracts che detengono riserve di due o più token, come verrà spiegato meglio al capitolo successivo.

Un noto esempio è Uniswap v2, che utilizza la seguente formula:

$$x \cdot y = k$$

Dove:

- $x$  e  $y$  sono le quantità dei due token nel pool.
- $k$  è una costante fissa.
- Ogni swap modifica  $x$  e  $y$ , mantenendo  $k$  costante.

Supponiamo di avere un pool ETH/DAI con:

- $x = 10$  ETH
- $y = 200$  DAI
- Quindi  $k = 10 \cdot 200 = 2000$

Se uno swap aggiungesse 1 ETH al pool, l'utente riceverebbe meno DAI in cambio di ETH dato che dovrebbe essere soddisfatta  $11 \cdot y = 2000$ .

Questo è solo un modello di AMM, chiamato **Constant Product**, dato che  $k$  è una costante invariabile, il prezzo si adatta dinamicamente al rapporto tra le quantità dei token e la curva dei prezzi è iperbolica.

Esistono altri modelli di AMM, tra cui:

- **Constant Mean**: che generalizza il modello a più di due token, permette di dare pesi diversi ai token che influiscono sulla determinazione dei prezzi, con formula:

$$\prod_{i=1}^n x_i^{w_i} = k$$

- **Logarithmic Market Scoring Rule:** utilizzato per i prediction markets, permette di calcolare il prezzo marginale di ogni outcome e ha formula del costo:

$$C(q) = b \cdot \log \left( \sum_i e^{qi/b} \right)$$

Dove  $q_i$  è la quantità acquistata dall'evento  $i$  e  $b$  è un parametro di liquidità.

Altri esempi di attori rilevanti nel campo degli AMM, oltre Uniswap, sono:

- **SushiSwap:** una fork di Uniswap che implementa delle differenze nel meccanismo di incentivazione per i fornitori di liquidità.
- **PancakeSwap:** AMM Constant Product che opera sulla Binance Smart Chain, con focus su token BEP-20.
- **Balancer:** AMM Constant Mean che consente la creazione di pool con più di due token e pesi personalizzabili associati.
- **Augur:** piattaforma di predizione del mercato che utilizza appunto un AMM LMSR.

## Esempio implementativo

L'applicazione dei concetti appena descritti può essere osservata perfettamente nel **core smart contract** di Uniswap v2.

La funzione di **swap** prende come parametri le quantità dei token e l'indirizzo che riceverà i token in uscita, in seguito controlla subito che almeno uno dei due amount sia positivo e la liquidità del pool sia sufficiente:

```
// this low-level function should be called from a contract which performs important safety checks
function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock {
    require(amount0Out > 0 || amount1Out > 0, 'UniswapV2: INSUFFICIENT_OUTPUT_AMOUNT');
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2: INSUFFICIENT_LIQUIDITY');
```

Il contratto invia all'utente i token richiesti e verifica quanto è stato depositato in cambio, confrontando i saldi attuali con le riserve, per poi calcolare gli importi in ingresso:

```
if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
if (data.length > 0) IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out, amount1Out, data);
balance0 = IERC20(_token0).balanceOf(address(this));
balance1 = IERC20(_token1).balanceOf(address(this));
}
uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
```

Viene applicata la **fee** di Uniswap dello 0.3%, che equivale a moltiplicare per 997/1000:

```
{ // scope for reserve{0,1}Adjusted, avoids stack too deep errors
uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
```

Viene effettuato il controllo fondamentale, che si assicura che nessuno possa ottenere più di quanto dovrebbe e che il valore del pool non diminuisca:

```
require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1000**2), 'UniswapV2: K');
```

Che equivale a controllare che:

$$(x + \Delta x \cdot 0.997) \cdot (y - \Delta y) \geq k$$

Infine vengono aggiornate le riserve e viene emesso l'**evento** di Swap, che verrà salvato nel log della blockchain:

```
_update(balance0, balance1, _reserve0, _reserve1);
emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
```

In sintesi:

- L'utente dice quanto vorrebbe ricevere.
- Il contratto calcola se sono stati ricevuti abbastanza token in cambio, tenendo conto anche della fee con la formula matematica.
- In caso sia tutto corretto invia i token richiesti, aggiornando le riserve, altrimenti la transazione fallisce.

## Liquidity pools

Un **Liquidity Pool** consiste in una riserva di due o più token che vengono bloccati tramite uno smart contract che consente agli utenti di scambiarli automaticamente, secondo regole predefinite, solitamente tramite un AMM.

Gli utenti che forniscono i fondi al pool sono detti **Liquidity Providers** e ricevono in cambio degli LP tokens, che rappresentano la loro quota nel pool, in questo modo con l'aumentare del valore del pool tramite le fees, i providers vedranno una plusvalenza della loro quota.

Naturalmente se il valore dei token (come ETH) diminuisce fortemente rispetto al momento del deposito, un Liquidity Provider può perdere denaro anche se il valore totale del pool è cresciuto tramite le fees, rischio chiamato **Impermanent Loss**.

Sostanzialmente, automated market makers e liquidity pools lavorano in simbiosi:

- Il pool contiene i due token.
- Gli utenti possono swapparli.
- I Liquidity Providers guadagnano sulle commissioni.
- I prezzi vengono determinati dinamicamente sulla base del rapporto tra token.

## Esempio implementativo

Anche in questo caso, nel core contract di Uniswap v2 sono presenti alcune funzioni che permettono di comprendere meglio i concetti teorici.

La prima funzione restituisce le riserve correnti dei token e il timestamp dell'ultima operazione, utile per calcolare i prezzi medi ponderati nel tempo (TWAP), le riserve sono variabili di stato che vengono aggiornate a ogni swap o cambiamento della liquidità:

```
function getReserves() public view returns (uint112 _reserve0, uint112 _reserve1, uint32 _blockTimestampLast) {  
    _reserve0 = reserve0;  
    _reserve1 = reserve1;  
    _blockTimestampLast = blockTimestampLast;  
}
```

La funzione **mint** invece viene chiamata ogni volta che un Liquidity Provider aggiunge liquidità al pool. Innanzitutto calcola il saldo dei token, leggendo le riserve attuali:

```
// this low-level function should be called from a contract which performs important safety checks
function mint(address to) external lock returns (uint liquidity) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    uint balance0 = IERC20(token0).balanceOf(address(this));
    uint balance1 = IERC20(token1).balanceOf(address(this));
    uint amount0 = balance0.sub(_reserve0);
    uint amount1 = balance1.sub(_reserve1);
```

In seguito, se un provider è il primo in assoluto, la liquidità viene calcolata come radice quadrata del prodotto delle quantità di token, altrimenti è calcolata sulla base dei token aggiunti alle riserve esistenti:

```
if (_totalSupply == 0) {
    liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
    _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
} else {
    liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply) / _reserve1);
}
```

Infine, se la liquidità è sufficiente, vengono emessi gli LP token, vengono aggiornate le riserve e viene emesso l'evento di **Mint**:

```
require(liquidity > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED');
_mint(to, liquidity);

_update(balance0, balance1, _reserve0, _reserve1);
if (feeOn) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-date
emit Mint(msg.sender, amount0, amount1);
```

La funzione **burn**, al contrario, permette ai Liquidity Providers di prelevare la loro quota, “bruciando” gli LP token. Per prima cosa legge le riserve e i saldi attuali, calcola quanti token vuole bruciare l’utente e quanti ne deve restituire all’indirizzo to:

```
// this low-level function should be called from a contract which performs important safety checks
function burn(address to) external lock returns (uint amount0, uint amount1) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    address _token0 = token0; // gas savings
    address _token1 = token1; // gas savings
    uint balance0 = IERC20(_token0).balanceOf(address(this));
    uint balance1 = IERC20(_token1).balanceOf(address(this));
    uint liquidity = balanceOf[address(this)];

    bool feeOn = _mintFee(_reserve0, _reserve1);
    uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
    amount0 = liquidity.mul(balance0) / _totalSupply; // using balances ensures pro-rata distribution
    amount1 = liquidity.mul(balance1) / _totalSupply; // using balances ensures pro-rata distribution
    require(amount0 > 0 && amount1 > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_BURNED');
```

In seguito, distrugge i token, aggiorna le riserve e genera l’evento **Burn**:

```
_burn(address(this), liquidity);
_safeTransfer(_token0, to, amount0);
_safeTransfer(_token1, to, amount1);
balance0 = IERC20(_token0).balanceOf(address(this));
balance1 = IERC20(_token1).balanceOf(address(this));

_update(balance0, balance1, _reserve0, _reserve1);
if (feeOn) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-date
emit Burn(msg.sender, amount0, amount1, to);
```

# Attori Principali

## Decentralized Exchanges

Sicuramente gli **Exchange Decentralizzati**, o **DEX**, hanno un ruolo fondamentale quando si parla di Finanza Decentralizzata che utilizza AMM e Liquidity Pools, dato che sono le piattaforme che effettivamente consentono lo scambio dei token tra utenti, senza intermediari.

Gli exchange tradizionali (centralizzati) custodiscono i fondi e gestiscono l'order book, mentre i DEX, tramite le tecnologie discusse nei capitoli precedenti, delegano tutta la logica di scambio a Smart Contracts su blockchain pubbliche.

I DEX più diffusi fanno uso di AMM per determinare i prezzi dinamicamente e facilitare le transazioni tramite Liquidity Pools.

Nello specifico, questo tipo di exchange svolge dei compiti precisi tra cui:

- **Gestione dei liquidity pools**

I DEX ospitano uno o più smart contracts che rappresentano dei pool di liquidità, formati da coppie di token.

Come si è già discusso, gli utenti possono ottenere una quota del pool, rappresentata dagli LP token, depositando entrambi i token nella proporzione richiesta.

- **Calcolo dei prezzi**

I prezzi, come anticipato, sono determinati tramite formule matematiche attraverso gli AMM, e non sulla base di domanda e offerta.

Ogni operazione di swap sposta l'equilibrio del pool, modificando i prezzi in tempo reale.

- **Permesso dello swap**

Gli utenti possono versare un token in cambio dell'altro, in cambio di una fee.

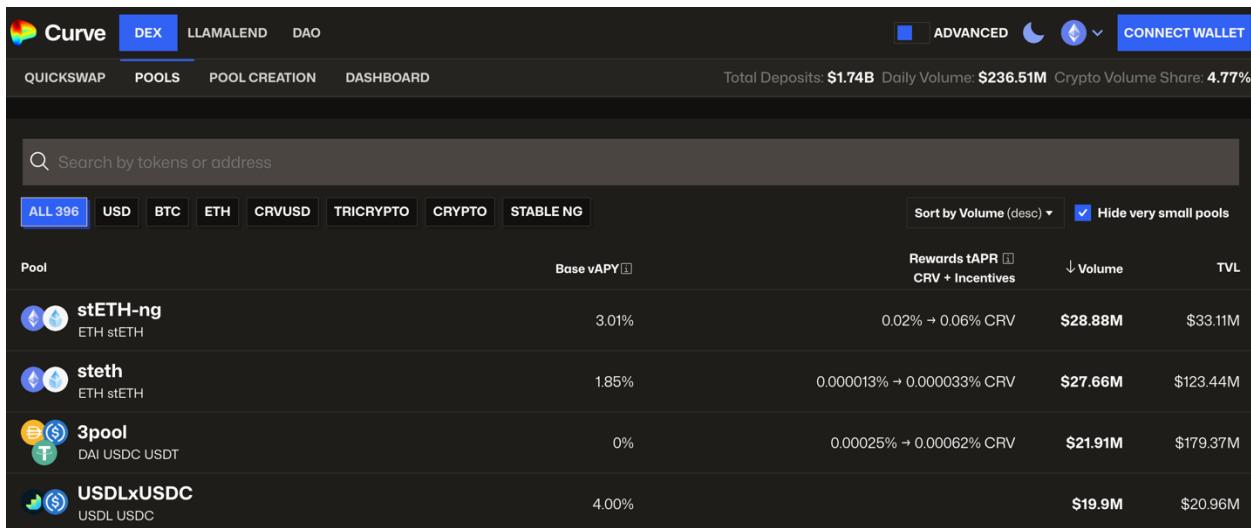
Alcune funzioni gestiscono gli output di questi swap, in base alle riserve a alla logica dell'AMM.

- **Distribuzione delle fees**

Le fees accumulate con gli scambi accrescono il valore del pool e vengono distribuite proporzionalmente ai liquidity providers, nel caso in cui decidano di ritirare la liquidità.

I maggiori esponenti sul mercato implementano principalmente i protocolli citati in termini di AMM:

- **Uniswap**
- **Balancer**
- **SushiSwap**
- **PancakeSwap**
- **Curve Finance**: DEX ottimizzato per gli stable coin.



The screenshot shows the Curve Finance DEX interface. At the top, there are tabs for QUICKSWAP, POOLS (which is selected), POOL CREATION, and DASHBOARD. The header also displays "Total Deposits: \$1.74B", "Daily Volume: \$236.51M", and "Crypto Volume Share: 4.77%". Below the header is a search bar with the placeholder "Search by tokens or address". Underneath the search bar is a filter section with buttons for ALL 396, USD, BTC, ETH, CRVUSD, TRICRYPTO, CRYPTO, and STABLE NG. To the right of this is a sorting option "Sort by Volume (desc)" with a checked checkbox for "Hide very small pools". The main table lists four liquidity pools:

Pool	Base vAPY	Rewards tAPR CRV + Incentives	↓ Volume	TVL
stETH-ng ETH stETH	3.01%	0.02% → 0.06% CRV	\$28.88M	\$33.11M
steth ETH stETH	1.85%	0.000013% → 0.000033% CRV	\$27.66M	\$123.44M
3pool DAI USDC USDT	0%	0.00025% → 0.00062% CRV	\$21.91M	\$179.37M
USDLxUSDC USDL USDC	4.00%		\$19.9M	\$20.96M

## Liquidity providers e Yield Aggregators

È stato già spiegato come la figura del liquidity provider abbia il ruolo di depositare i fondi in cambio di LP token, permettendo agli utenti di scambiare i token e ottenendo una quota che potrebbe aumentare di valore grazie alle fees.

Nel settore della DeFI, tuttavia, la maggior parte delle realtà si serve principalmente di liquidity providers istituzionali o aziendali, dato che la liquidità in larga scala riduce lo **slippage** e aumenta la profondità del mercato, rendendo i DEX più attraenti per gli utenti finali.

Molti DEX fungono anche da Liquidity Providers per le loro piattaforme, tuttavia esistono altre aziende specializzate nell'offrire liquidità per servizi finanziari di questo tipo, tra cui **Galaxy Digital**, **GSR Markets**, **Cumberland** e **Jump Crypto**.

Oltre al principale compito di fornire liquidità, per il funzionamento del sistema, le aziende che agiscono da liquidity providers possono avere anche altri ruoli:

- **Servizi da broker e custodia:**

Molti LP offrono accesso alla liquidità per effettuare operazioni a margine e infrastrutture per la custodia dei token, con la possibilità di prestiti. Per questo motivo c'è bisogno di sistemi robusti che implementino multi-sig, audit avanzati e siano orientati alla gestione del rischio per operare a livello istituzionale.

Tra questi vale la pena citare **Fireblocks** e **Alloy Capital**.

- **Ottimizzazione della liquidità:**

Tramite tecniche come la distribuzione della liquidità e la Yield Aggregation, i liquidity providers cercano di massimizzare i ricavi dalle fees e minimizzare il rischio di impermanent loss.

**Yearn Finance** e **Beefy** sono tra i principali Yield Optimizer.

The screenshot shows the Beefy platform's user interface. At the top, there's a navigation bar with links for 'Beefy', 'Vaults', 'Dashboard', 'DAO', 'Resources', and a 'Connect Wallet' button. Below the navigation, a dark header displays 'Portfolio' with summary statistics: DEPOSITED \$0, MONTHLY YIELD \$0, DAILY YIELD \$0, AVG. APY 0%, TVL \$279.80M, and VAULTS 1169. A 'Platform' section follows. Underneath, a search bar and various filters like 'All', 'Saved', 'My Positions', 'Boosts', and categories like 'Stablecoins', 'Blue Chips', 'Memes', 'Correlated', 'Single', 'LP', 'CLM', 'Vaults', and 'Pools' are visible. The main area lists two assets: 'rETH-USDC' and 'USDC (Varlamore)'. Each asset entry includes its logo, name, category, current APY, daily yield, and total value locked (TVL). For 'rETH-USDC', the APY is 35.04% (with a 10.35% margin), and TVL is \$49,304. For 'USDC (Varlamore)', the APY is 7.86% (with a 7.34% margin), and TVL is \$1.76M.

Inoltre è importante specificare che le istituzioni tendono a scegliere pool con assets stabili o pool protetti, con meno volatilità ma rendimenti più contenuti, in modo da ridurre il rischio di perdita.

## Decentralized Autonomous Organizations

Per gestire in maniera comunitaria l'evoluzione dei protocolli e del progetto generale, i DEX fanno uso dei corrispettivi DAO, come:

- **Uniswap DAO.**
- **Curve DAO.**
- **Balancer DAO.**

Tramite i DAO, è possibile gestire tutte le decisioni in maniera decentralizzata e automatizzata tramite smart contracts.

Alcuni DAO hanno il proprio token per consentire le votazioni, che principalmente riguardano i seguenti argomenti:

- **Definizione parametri:**  
Decisioni riguardo aggiornamenti di fees, incentivi e smart contracts.
- **Gestione treasury:**  
Impegno nella crescita sostenibile e cambiamenti riguardanti finanziamenti, marketing e sviluppo.
- **Situazioni critiche:**  
Alcuni DAO, come l'**Emergency DAO** di Curve incrementano la sicurezza con framework di emergenza.
- **Distribuzione potere politico:**  
Il bilanciamento tra poteri è gestito con modelli come **veCRV** che supportano il voto token-based.

## Oracoli, Wallet e API

Sono importanti anche tutte quelle infrastrutture che offrono strumenti per dati esterni, portafogli e interfacce utente.

Sicuramente hanno un ruolo fondamentale:

- **Oracoli:**

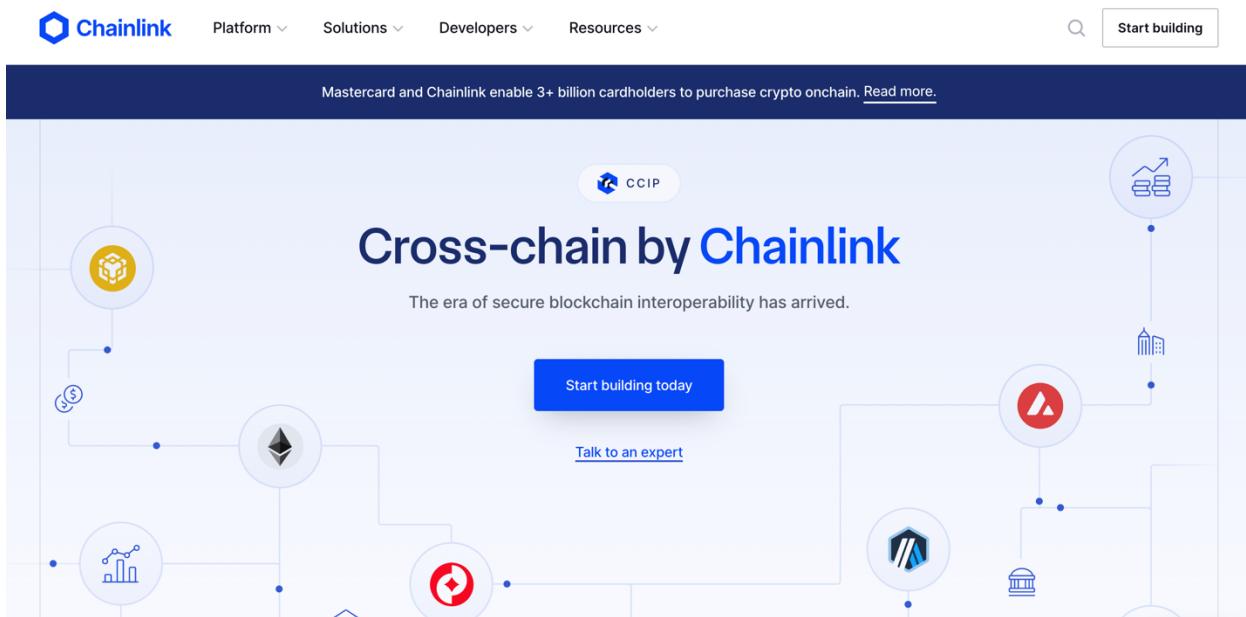
Sono importanti per la fornitura di prezzi esterni, in modo da evitare possibili manipolazioni. Tra queste si può citare **ChainLink**, che connette istituzioni finanziarie e blockchain da questo punto di vista.

- **Wallets:**

Offrono in genere interfacce per gestire i fondi e i token di cui si è in possesso, a volte possono essere utilizzati anche per gestire gli LP token e le quote di possesso dei pool. Sicuramente il più noto resta **Metamask**.

- **API:**

Permettono la connessione con le librerie Web3 e l'interazione con la blockchain.



## Sicurezza

Infine, nel campo della DeFI, dove un errore può causare perdite di milioni di dollari, è fondamentale verificare e proteggere il codice, data la sua natura non intermediata.

Per questo motivo, alcune aziende si sono specializzate nella revisione completa del codice degli smart contracts e vengono ingaggiate dai DEX prima e durante il deploy.

Queste aziende ricoprono diversi compiti e rientrano in diverse categorie:

- **Smart Contract Audits:**

Si occupano della revisione manuale del codice, simulazione di exploit, analisi formale della sicurezza e report dettagliati a riguardo.

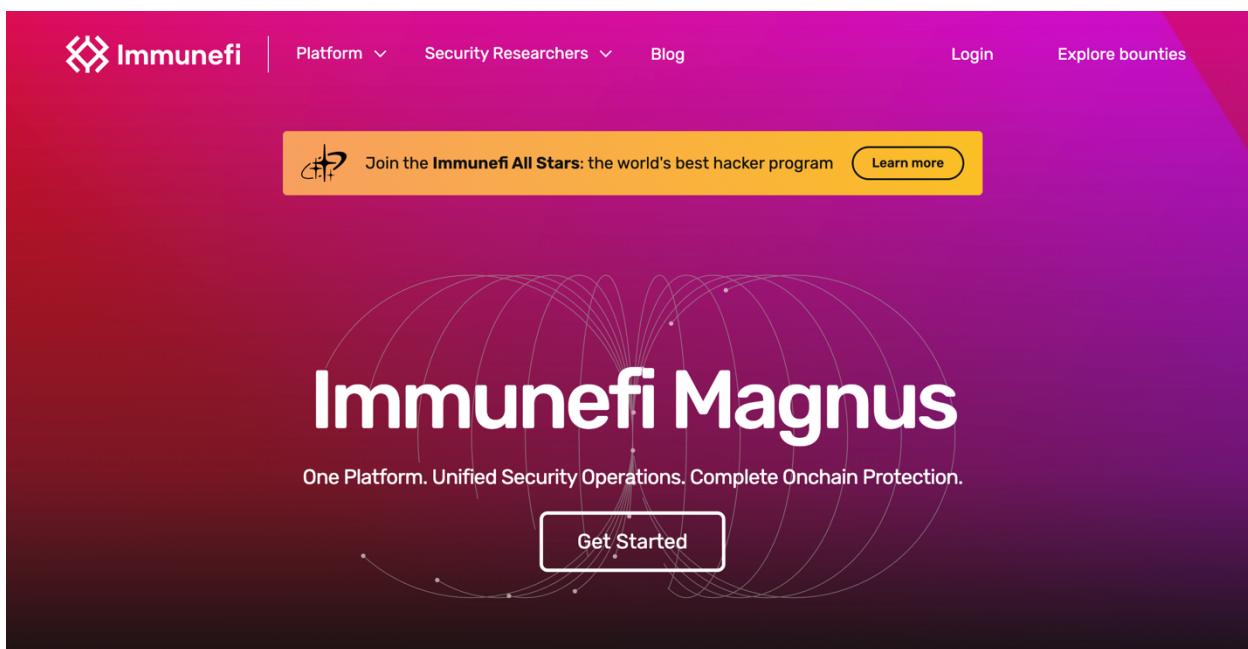
- **Bug Bounty:**

Molti protocolli di DeFI lavorano affiancandosi a delle piattaforme che offrono ricompense monetarie in cambio di segnalazioni di bug o falle di sicurezza, in genere stilando una classifica della gravità delle falle.

- **Strumenti di sicurezza:**

Altri tool garantiscono il monitoraggio, il deployment sicuro e l'automazione difensiva.

Sicuramente i leader del settore comprendono nomi come **OpenZeppelin** e **Immunefi**.



## Valutazione delle strategie di profitto

Gli utenti finali adoperando strumenti di DeFi in alcuni casi possono ottenere profitto, per esempio, tramite Staking, Trading e Arbitraggio (spesso illecito), tuttavia queste sono operazioni che espongono l'utente a un certo rischio, spesso in rapporto ad un guadagno minimo.

In questo settore, il vero profitto viene fatto dalle aziende che gestiscono l'infrastruttura, in seguito analizzeremo le principali strategie di profitto adoperate nel settore degli AMM.

### Trading Fees

Come è già stato accennato più volte, la fonte primaria di guadagno sono le fees guadagnate dagli scambi dei token, per questo motivo la maggior parte delle aziende che gestiscono DEX operano anche come Liquidity Providers.

In particolare, ogni scambio nel pool comporta una commissione che viene raccolta nel pool e redistribuita ai Liquidity Provider se ritirano i loro fondi, a volta parte di questa fee viene trattenuta dal protocollo o dalla tesoreria dell'azienda.

Nella maggior parte dei casi le fee sono comunque inferiori allo 0.5%, dato che comunque più le fee sono basse più un exchange è appetibile al pubblico, alcuni esempi sono:

- **Uniswap v2:** 0.3%
- **Uniswap v3/v4:** 0.01-0.4%
- **Curve:** 0.04%

Analizzando il report mensile di Uniswap, nel Giugno 2025 su un volume di **103.8 miliardi** di dollari di trading volume, sono state guadagnate **59.8 milioni** di dollari di fees, di cui 661.6 mila dollari sono stati trattenuti in tesoreria.

Financial statement	Jul 2025 Jul 1 - Jul 31	Jun 2025 Jun 1 - Jun 30
<b>GMV</b>		
Net deposits	\$7.5 B 6.2%	\$7.9 B 3.9%
Trading volume	\$1.9 B 98.2%	\$103.8 B 16.5%
<b>Income statement</b>		
Fees	\$1.2 M 98.0%	\$59.8 M 18.4%
(Supply-side fees)	\$1.2 M 98.0%	\$59.8 M 18.4%
Revenue	\$0.0 N/A ⓘ	\$0.0 N/A ⓘ
(Expenses)	\$277.7 K 96.7%	\$8.4 M 7.3%
(Cost of revenue)	\$11.8 K 96.5%	\$340.0 K 52.6%
(Token incentives)	\$265.9 K 96.7%	\$8.0 M 8.9%
Gross profit	\$-11.8 K 96.5%	\$-340.0 K 52.6%
Earnings	\$-277.7 K 96.7%	\$-8.4 M 7.3%
<b>Treasury</b>		
Treasury	\$2.5 B 5.6%	\$2.6 B 15.1%
Treasury (net)	\$12.2 K 98.2%	\$661.6 K 241.7%

Le fees corrispondono a circa lo 0,057% del volume di scambio, probabilmente dovuto al fatto che la commissione su Uniswap v3/v4 è configurabile a partire dallo 0.01%.

## Liquidity Mining

Alcune aziende che lanciano exchange decentralizzati, piattaforme di prestiti e altri servizi di DeFi possono incentivare gli utenti a fornire liquidità al protocollo in cambio di token, come avviene per le ICO.

Questi token non sono regalati casualmente ma seguono un piano (tokenomics):

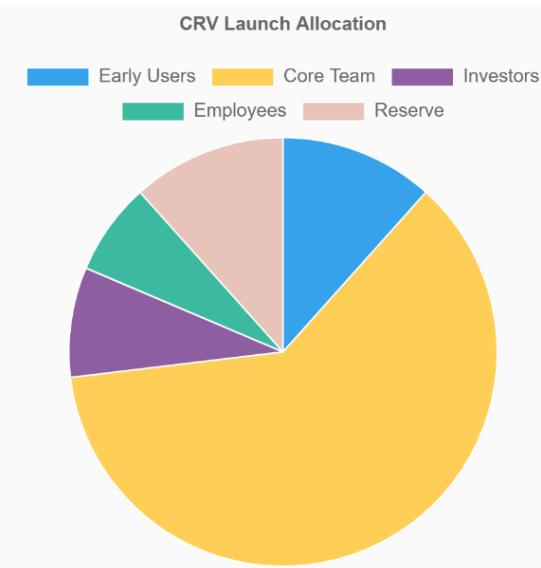
- Il team/protocollo trattiene una grossa porzione dei token.
- I token distribuiti fanno crescere il Total Value Locked.
- Vengono attirati più utenti che generano più fees.
- Inoltre il valore del token dovrebbe aumentare sul lungo termine, permettendo all'azienda di guadagnare anche attraverso la grossa parte di token trattenuta.

In sintesi, i token sono utilizzati come strumento di acquisizione utenti, aumentano di volume e le fees, possono permettere guadagni futuri e creano ecosistemi basati su staking, voti e reward.

Alcuni protocolli cercano di ottenere dei loop fornendo incentivi per mettere in staking e quindi bloccare delle quantità di token, incoraggiando l'utente a investire e ottenendo così più liquidità.

Un esempio esplicativo è CRV token, lanciato da Curve:

- I Liquidity Provider guadagnano dei CRV token.
- I CRV token possono essere rivenduti oppure possono essere bloccati (veCRV) per aumentare di 2.5 volte i futuri guadagni di CRV token, ottenere il permesso di votare nelle proposte della governance e guadagnare le fees sugli swaps.



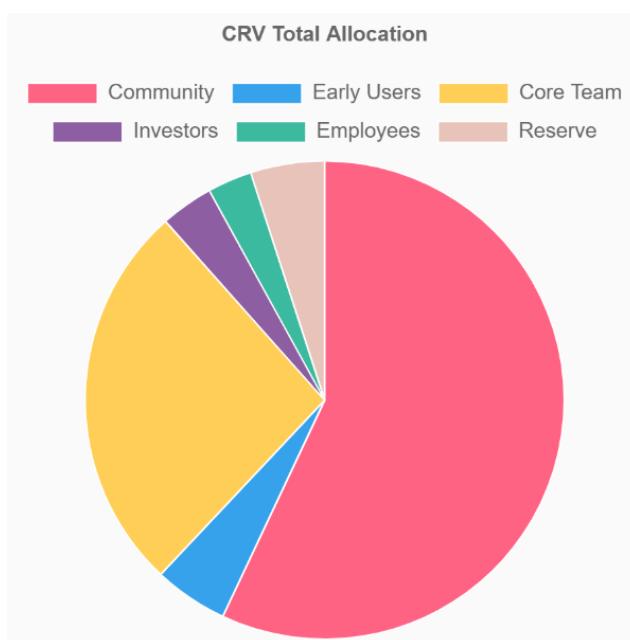
Al momento del lancio, 13 Agosto del 2020, il 61.47% dei token appartenevano appunto al Core Team e i token erano interamente bloccati in contratti chiamati Vesting Contracts predisposti per sbloccarli gradualmente a 1-4 anni dal lancio.

In questo modo Curve si è assicurato una liquidità di partenza attraverso i token bloccati e detenendone la maggior parte spera di poter guadagnare anche da lì in futuro.

Tuttavia per adesso il valore del token CRV è sceso drasticamente dal momento del lancio.



Anche la distribuzione dei token è cambiata:



## **Lending su LP Tokens**

Un altro metodo con cui le aziende riescono a generare profitto consiste nel massimizzare il rendimento di una posizione in un Liquidity Pool, prendendo prestiti in cambio di LP tokens usati come collaterali, per poi riciclare la liquidità attraverso strategie complesse.

Un esempio può essere quello di riciclare liquidità versandola in un protocollo in cambio di LP tokens, prendere stablecoins in prestito utilizzando gli LP tokens come collaterali e reinvestire anche gli stablecoins nel pool.

In questo modo le aziende e i Power Users possono ottenere un effetto leva, che comporta un aumento del profitto ma sicuramente espone a un rischio altrettanto maggiore.

Nella pratica, si può utilizzare Uniswap in combinazione con Aave per fornire liquidità ai pool e prendere prestiti con LP token come collaterali.

## **Servizi infrastrutturali**

Infine, molte piattaforme riescono a generare ulteriori guadagni integrando strumenti come esecuzione automatica di funzioni, servizi di sicurezza o personalizzazioni, guadagnando attraverso fees per chiamate on-chain e abbonamenti premium.

## Uniswap v4

Uniswap è uno dei leader nel campo degli Exchange Decentralizzati (non-custodial) basati su AMM.

La versione 4 è l'ultimo protocollo di casa Uniswap, lanciato in mainnet a Gennaio 2025, erede del v2 e del v3. In particolare si tratta di un protocollo di AMM con una peculiarità rispetto ai predecessori: permette la creazione di features personalizzate sul protocollo Uniswap, senza il bisogno di progettare e creare nuovi AMM da zero. Questa libertà di personalizzazione è resa possibile attraverso gli **“hooks”**, che permettono a sviluppatori e Liquidity Providers di eseguire azioni in momenti specifici della vita del pool.

Uniswap offre anche l'accesso a deep liquidity e altre features esistenti, aggiungendo la possibilità di livelli di fees illimitati, che permettono la personalizzazione dei pool e una maggiore efficienza dei costi.

I pool di liquidità vengono gestiti con un singolo smart contract, chiamato **singleton**, a scopo di ridurre drasticamente la spesa di gas nella creazione di un pool, che inoltre offre un sistema di accounting rapido, che consente solo il trasferimento dei saldi netti attraverso gli swap.

Uniswap v4 è retrocompatibile concettualmente con v3 ma presenta un'architettura molto diversa, in particolare:

- Il core del **pricing** può ancora utilizzare modelli constant product, ma è reso modulare.
- Mint, Burn e Swap esistono ancora, ma sono gestite attraverso un'interfaccia chiamata **PoolManager**.
- Ogni pool è gestito virtualmente dal **singleton** e non è un contratto autonomo con un suo storage per reserves, fees e liquidity come in v3.
- Per creare un **nuovo pool** non si crea un nuovo contratto ma si aggiungono nuove entry al singleton.
- **ETH nativo** è supportato nativamente, mentre nelle versioni precedenti veniva utilizzato WETH, come token ERC-20 per impacchettare ETH, necessario per le interfacce.
- La **custom logic** è possibile per ogni pool, integrabile tramite hooks.

## Hooks

Gli hooks rappresentano una delle principali novità introdotte in Uniswap v4 e consistono in degli smart contracts che agiscono sostanzialmente come plugin per modificare il comportamento dei pool, degli swap, delle fees e della liquidità, un po' come le estensioni aggiungono funzionalità al browser.

Questa personalizzazione viene resa possibile dall'esecuzione di azioni in momenti specifici, come prima di uno swap o dopo l'aggiunta di liquidità.

All'atto pratico il contratto hook viene definito durante la creazione del pool e viene chiamato dal PoolManager in corrispondenza degli eventi chiave a cui è stato associato, per esempio:  
**beforeInitialize, afterSwap, beforeSwap, beforeAddLiquidity...**

Un esempio semplice ed esplicativo di hook si può trovare nella repository di templates per creare hook (uniswapfoundation/v4-template).

L'hook **Counter** permette di contare quante volte vengono effettuate le operazioni di swap e di aggiunta e rimozione di liquidità su un pool. Per iniziare, il pool viene associato a 4 contatori e viene creato il costruttore che eredita da BaseHook:

```
contract Counter is BaseHook {
    using PoolIdLibrary for PoolKey;

    // NOTE: -----
    // state variables should typically be unique to a pool
    // a single hook contract should be able to service multiple pools
    // ----

    mapping(PoolId => uint256 count) public beforeSwapCount;
    mapping(PoolId => uint256 count) public afterSwapCount;

    mapping(PoolId => uint256 count) public beforeAddLiquidityCount;
    mapping(PoolId => uint256 count) public beforeRemoveLiquidityCount;

    constructor(IPoolManager _poolManager) BaseHook(_poolManager) {}
}
```

In seguito viene creata una funzione che restituisce una struttura di permessi, che specifica su quali eventi del pool vogliamo abilitare la callback; in questo caso solo **beforeSwap, AfterSwap, beforeAddLiquidity, beforeRemoveLiquidity**:

```
function getHookPermissions() public pure override returns (Hooks.Permissions memory) {
    return Hooks.Permissions({
        beforeInitialize: false,
        afterInitialize: false,
        beforeAddLiquidity: true,
        afterAddLiquidity: false,
        beforeRemoveLiquidity: true,
        afterRemoveLiquidity: false,
        beforeSwap: true,
        afterSwap: true,
        beforeDonate: false,
        afterDonate: false,
        beforeSwapReturnDelta: false,
        afterSwapReturnDelta: false,
        afterAddLiquidityReturnDelta: false,
        afterRemoveLiquidityReturnDelta: false
    });
}
```

Infine vengono definite le funzioni che verranno chiamate in corrispondenza degli eventi relativi al pool che, come si poteva intuire, incrementano i contatori, tra cui:

```
function _beforeSwap(address, PoolKey calldata key, SwapParams calldata, bytes calldata)
    internal
    override
    returns (bytes4, BeforeSwapDelta, uint24)
{
    beforeSwapCount[key.toId()]++;
    return (BaseHook.beforeSwap.selector, BeforeSwapDeltaLibrary.ZERO_DELTA, 0);
}

function _afterSwap(address, PoolKey calldata key, SwapParams calldata, BalanceDelta, bytes calldata)
    internal
    override
    returns (bytes4, int128)
{
    afterSwapCount[key.toId()]++;
    return (BaseHook.afterSwap.selector, 0);
}
```

La personalizzabilità permessa tramite gli hooks è sicuramente un enorme miglioramento rispetto a Uniswap v3 e rappresenta una delle novità introdotte da questo protocollo, tuttavia questi componenti introducono anche alcune **vulnerabilità**, legate al fatto che il codice dell'hook rimane critico, nonostante i controlli eseguiti dal PoolManager sui permessi.

Un hook malevolo o mal implementato può condurre alla perdita di fondi o stalli del pool e per questo motivo nella documentazione di Uniswap si ricorda spesso di utilizzare solo gli hooks di cui ci si fida:

**Note:** Hooks are developed by independent third-party developers who are not affiliated with Uniswap Labs. Exercise caution when adding hooks, as some may be malicious or cause unintended consequences. See more about hooks [here](#).

Alcune aziende si stanno già specializzando sull'esecuzione di audit su questi smart contracts. Possiamo citare **Hacken** e **CertiK**.

## Singleton e risparmio del gas

Come già anticipato, una differenza fondamentale rispetto alle versioni precedenti di Uniswap è il fatto che i pool vengano gestiti attraverso un unico contratto, detto singleton, che corrisponde al **PoolManager**.

Nel PoolManager si può osservare come tutti i pool vengano gestiti attraverso una singola mappa:

```
mapping(PoolId id => Pool.State) internal _pools;
```

Permettendo al singolo contratto di gestire tutte le logiche comuni, come swap, aggiunta e rimozione di liquidità ed evitando la duplicazione del codice relativo.

La funzione di **inizializzazione**, permette di creare un nuovo pool attraverso PoolManager:

- Vengono effettuati dei controlli per evitare **overflow**.
- Viene generato l'**id** univoco del pool.
- Le **variabili** del pool vengono salvate in un'unica mappa nel PoolManager basata sull'id.
- Viene emesso l'**evento** di inizializzazione.
- Viene restituito l'intervallo discreto del prezzo iniziale (**tick**).

Nella funzione **initialize** è anche possibile notare `beforeInitialize` e `afterInitialize`, che richiamerebbero l'attenzione di eventuali hook interessati a questi eventi:

```
function initialize(PoolKey memory key, uint160 sqrtPriceX96) external noDelegateCall returns (int24 tick) {
    // see TickBitmap.sol for overflow conditions that can arise from tick spacing being too large
    if (key.tickSpacing > MAX_TICK_SPACING) TickSpacingTooLarge.selector.revertWith(key.tickSpacing);
    if (key.tickSpacing < MIN_TICK_SPACING) TickSpacingTooSmall.selector.revertWith(key.tickSpacing);
    if (key.currency0 >= key.currency1) {
        CurrenciesOutOfOrderOrEqual.selector.revertWith(
            Currency.unwrap(key.currency0), Currency.unwrap(key.currency1)
        );
    }
    if (!key.hooks.isValidHookAddress(key.fee)) Hooks.HookAddressNotValid.selector.revertWith(address(key.hooks));

    uint24 lpFee = key.fee.getInitialLPFee();

    key.hooks.beforeInitialize(key, sqrtPriceX96);

    PoolId id = key.toId();

    tick = _pools[id].initialize(sqrtPriceX96, lpFee);

    // event is emitted before the afterInitialize call to ensure events are always emitted in order
    // emit all details of a pool key. poolkeys are not saved in storage and must always be provided by the caller
    // the key's fee may be a static fee or a sentinel to denote a dynamic fee.
    emit Initialize(id, key.currency0, key.currency1, key.fee, key.tickSpacing, key.hooks, sqrtPriceX96, tick);

    key.hooks.afterInitialize(key, sqrtPriceX96, tick);
}
```

La creazione di un pool come dati aggiuntivi di un contratto già esistente senza dubbio riduce di molto la spesa di gas legata al deploy, dato che nelle versioni precedenti per creare un pool occorreva effettuare la deploy di un nuovo contratto.

Tutti i pool vengono tracciati dallo stato globale del PoolManager, entro il quale ogni pool possiede 4 variabili:

- **sqrtPriceX96**: radice quadrata del prezzo iniziale tra due token.
- **fee**: percentuale di fees.
- **tickspacing**: ogni quanti tick si può posizionare la liquidità.
- **hook**: indirizzo del contratto associato al pool.

Il gas viene risparmiato in parte anche grazie al fatto che gli eventuali hooks vengono richiamati soltanto dagli eventi a cui sono interessati, attraverso i permessi.

## Funzioni elementari

Le funzioni principali analizzate in campo AMM e liquidity pools (swap, mint, burn...) sono disponibili anche in Uniswap v4, tuttavia come accennato, vengono chiamate attraverso l'interfaccia PoolManager, esempio di riutilizzo del codice.

Alcune differenze implementative possono essere osservate nella funzione di swap per esempio, dove vengono gestiti anche beforeSwap e afterSwap e dove vengono utilizzate le nuove variabili delta:

```
BeforeSwapDelta beforeSwapDelta;
{
    int256 amountToSwap;
    uint24 lpFeeOverride;
    (amountToSwap, beforeSwapDelta, lpFeeOverride) = key.hooks.beforeSwap(key, params, hookData);

    // execute swap, account protocol fees, and emit swap event
    // _swap is needed to avoid stack too deep error
    swapDelta = _swap(
        pool,
        id,
        Pool.SwapParams({
            tickSpacing: key.tickSpacing,
            zeroForOne: params.zeroForOne,
            amountSpecified: amountToSwap,
            sqrtPriceLimitX96: params.sqrtPriceLimitX96,
            lpFeeOverride: lpFeeOverride
        }),
        params.zeroForOne ? key.currency0 : key.currency1 // input token
    );
}

BalanceDelta hookDelta;
(swapDelta, hookDelta) = key.hooks.afterSwap(key, params, swapDelta, hookData, beforeSwapDelta);
```

Questi nuovi oggetti e variabili vengono utilizzati per gestire i cambiamenti di stato e le interazioni con i pool:

- BeforeSwapDelta **beforeSwapDelta**: dati per calcolare lo swap prima che avvenga.
- BalanceDelta **swapDelta**: rappresenta la differenza dei token coinvolti nello swap, secondo le regole dell'AMM.
- BalanceDelta **hookDelta**: il delta aggiuntivo causato dagli hook interessati a afterSwapReturnDelta.

## Scenario d'uso completo

Per comprendere appieno le funzioni fondamentali su cui si basano gli AMM, è stato implementato e provato in testnet un market maker capace di compiere le operazioni principali del caso.

## Creazione dei token

Per iniziare sono stati creati due contratti per definire i due token che il nostro AMM sarà capace di gestire. I token si chiamano **TokenA** e **TokenB**:

- Ereditano da **ERC20**.
- Chiamano il costruttore.
- Introducono il modifier **onlyowner** per evitare che vengano manipolati da qualcuno che non è il proprietario.
- Implementano la funzione **mint** per creare token e distribuirli a un address.

```
contract TokenA is ERC20{
    address owner;
    error notTheOwner(address _address);
    constructor() ERC20("TokenA", "A") {
        owner = msg.sender;
    }

    modifier onlyOwner(){
        if (msg.sender != owner) revert notTheOwner(msg.sender);
        _;
    }

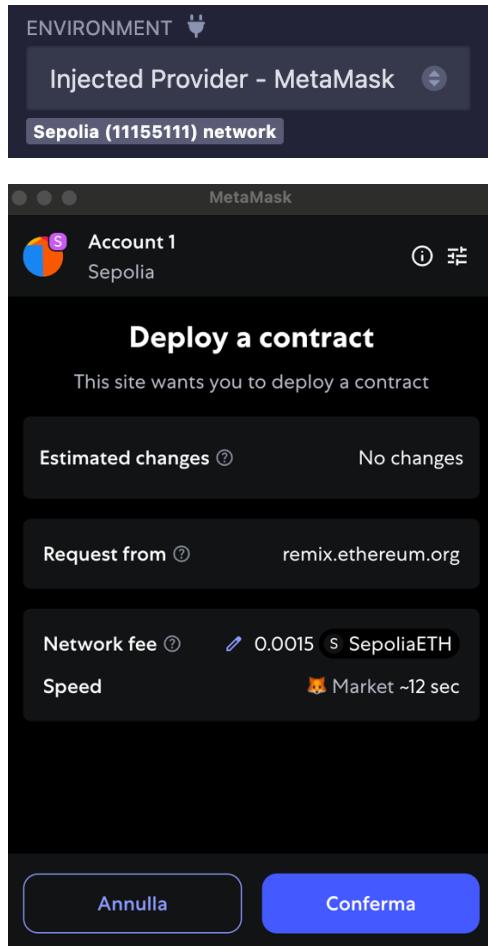
    function mint(address recipient, uint256 quantity) public onlyOwner{
        _mint(recipient,quantity);
    }
}
```

```
contract TokenB is ERC20{
    address owner;
    error notTheOwner(address _address);
    constructor() ERC20("TokenB", "B") {
        owner = msg.sender;
    }

    modifier onlyOwner(){
        if (msg.sender != owner) revert notTheOwner(msg.sender);
        _;
    }

    function mint(address recipient, uint256 quantity) public onlyOwner{
        _mint(recipient,quantity);
    }
}
```

È stato eseguito il deploy dei due contratti sulla testnet **Sepolia**, tramite **Remix IDE**, con un account su cui avevamo ottenuto dei **SepoliaETH** tramite dei faucets:



## Automated Market Maker

In seguito è stato creato un contratto chiamato **bcsc2025amm**, che implementa un vero e proprio AMM che:

- Gestisce un **liquidity pool** su TokenA e TokenB.
- Permette ai liquidity providers di versare TokenA e TokenB in cambio di quote del pool con la **mint**.
- Permette ai liquidity providers di prelevare TokenA e TokenB in cambio di quote con la **burn**.
- Permette agli altri utenti di effettuare **swap** tra i due token.

Il contratto utilizza gli indirizzi dei due token creati in precedenza, che infatti verranno inseriti al momento del deploy e definisce le riserve di token e la liquidità mappata come quote di ciascun utente:

```
contract bcsc2025amm {
    IERC20 public tokenA;
    IERC20 public tokenB;

    uint256 public reserveA;
    uint256 public reserveB;

    mapping(address => uint256) public liquidity;
    uint256 public totalLiquidity;

    constructor(address _tokenA, address _tokenB) {
        tokenA = IERC20(_tokenA);
        tokenB = IERC20(_tokenB);
    }
}
```

La prima funzione è la **mint**, che permette di versare TokenA e TokenB in cambio delle **quote** di liquidity del pool:

```
function mint(uint256 amountA, uint256 amountB) external returns (uint256 shares) {
    tokenA.transferFrom(msg.sender, address(this), amountA);
    tokenB.transferFrom(msg.sender, address(this), amountB);

    if (totalLiquidity == 0) {
        shares = sqrt(amountA * amountB);
    } else {
        shares = min(
            (amountA * totalLiquidity) / reserveA,
            (amountB * totalLiquidity) / reserveB
        );
    }

    require(shares > 0, "Liquidità mintata insufficiente");

    liquidity[msg.sender] += shares;
    totalLiquidity += shares;
    reserveA += amountA;
    reserveB += amountB;
}
```

In particolare la **mint**:

- Trasferisce i token all'indirizzo del contratto.
- Calcola le quote da restituire in maniera diversa se è la prima fornitura di liquidità.
- Si assicura che la liquidità minata sia sufficiente.
- Se è tutto apposto, incrementa la liquidità dell'utente e totale del pool e aggiorna i valori delle riserve di token.

La **burn** invece consente l'operazione opposta, restituendo TokenA e TokenB in cambio delle quote di liquidità:

```
function burn(uint256 shares) external returns (uint256 amountA, uint256 amountB) {  
    require(liquidity[msg.sender] >= shares, "Non hai abbastanza quote");  
  
    amountA = (shares * reserveA) / totalLiquidity;  
    amountB = (shares * reserveB) / totalLiquidity;  
  
    liquidity[msg.sender] -= shares;  
    totalLiquidity -= shares;  
    reserveA -= amountA;  
    reserveB -= amountB;  
  
    tokenA.transfer(msg.sender, amountA);  
    tokenB.transfer(msg.sender, amountB);  
}
```

In particolare:

- Si assicura che l'utente abbia le quote che vuole “bruciare”.
- Calcola le quantità di token in base alla liquidità totale del pool.
- Aggiorna le liquidità e le riserve.
- Trasferisce i token all'indirizzo dell'utente.

L'ultima funzione è la **swap**, che permette di scambiare TokenA e TokenB, eseguendo le seguenti operazioni:

- Controlla che il token che si sta cercando di scambiare sia TokenA o TokenB.
- Trasferisce i token all'indirizzo del contratto.
- Calcola l'output aggiungendo una fee dello 0.3%.
- Se l'output calcolato non è nullo, trasferisce i token risultanti all'utente.
- Aggiorna le riserve.

```

function swap(address tokenIn, uint256 amountIn) external returns (uint256 amountOut) {
    bool isAtoB = tokenIn == address(tokenA);
    require(isAtoB || tokenIn == address(tokenB), "INVALID_TOKEN");

    (IERC20 tokenFrom, IERC20 tokenTo, uint256 reserveFrom, uint256 reserveTo) =
        isAtoB ? (tokenA, tokenB, reserveA, reserveB) : (tokenB, tokenA, reserveB, reserveA);

    tokenFrom.transferFrom(msg.sender, address(this), amountIn);

    uint256 amountInWithFee = (amountIn * 997) / 1000;
    amountOut = (amountInWithFee * reserveTo) / (reserveFrom + amountInWithFee);

    require(amountOut > 0, "INSUFFICIENT_OUTPUT");

    tokenTo.transfer(msg.sender, amountOut);

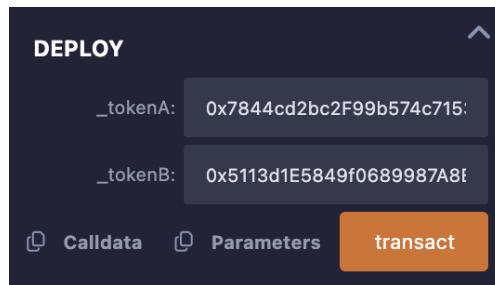
    if (isAtoB) {
        reserveA += amountIn;
        reserveB -= amountOut;
    } else {
        reserveB += amountIn;
        reserveA -= amountOut;
    }
}

```

Il modello è **constant product** e segue la formula:

$$\frac{amountInWithFee \cdot reserveB}{reserveA + amountInWithFee}$$

Anche di questo contratto è stata eseguita la deploy, inserendo come indirizzi dei token quelli ottenuti con le deploy dei rispettivi contratti:



## Test

Per prima cosa abbiamo mintato **13 TokenA** e **10 TokenB** nel nostro account (formato decimale a 18):

MINT

recipient: 0x515Ffd65894B3a3569468

quantity: 130000000000000000000000

Calldata Parameters transact

balanceOf 0x515Ffd65894B3a3569468

0: uint256: 130000000000000000000000

In seguito abbiamo chiamato la **approve** con l'indirizzo del contratto di AMM, per permettergli di gestire per conto nostro fino a 200 TokenA e 200 TokenB:

APPROVE

spender: 0xc5110d8c44a6a6d7b74e53

value: 200000000000000000000000

Calldata Parameters transact

totalLiquidity

0: uint256: 0

In questo momento la **liquidità** del pool è nulla, infatti non è stato versato ancora nessun token:

totalLiquidity

0: uint256: 0

Come prossimo passo abbiamo versato 10 TokenA e 10 TokenB tramite la **mint**:

In cambio di questi token, abbiamo ricevuto **10 quote** del pool (LP token), che ora risultano assegnate al nostro address, e che rappresentano anche la liquidità totale.

Come si può vedere dal codice, le quote del primo mint vengono calcolate come la radice quadrata del prodotto delle quantità dei due token.

Ora possiamo eseguire la **swap**, per esempio dei 3 TokenA che ci rimanevano.

Prima dello swap abbiamo 3 TokenA e 0 TokenB:

The image contains two screenshots of a blockchain wallet interface. Both screenshots show a call to the `balanceOf` function on the contract address `0x515Ffd65894B3a3f`.  
The top screenshot shows the result of the first call: `0: uint256: 30000000000000000000000000000000`.  
The bottom screenshot shows the result of the second call: `0: uint256: 0`.

Per eseguire lo swap dobbiamo specificare anche l'address del contratto di TokenA:

The image shows a "SWAP" transaction input screen. It has fields for `tokenIn:` set to `0x7844cd2bc2F99b574c715` and `amountIn:` set to `30000000000000000000000000000000`. Below the fields are buttons for `Calldata`, `Parameters`, and a prominent orange `transact` button.

In cambio dei **3 TokenA** otteniamo **2.302 TokenB**, calcolati con la formula descritta al paragrafo precedente:

The image shows a "Transaction request" confirmation screen. It displays "Estimated changes" with the following details:  
- You send: -3 `0x7844c...a0512` (in red)  
+ You receive: +2,302 `0x5113d...740A7` (in green)

Possiamo vedere che ora il nostro balance di TokenA è vuoto, mentre quello di TokenB contiene i TokenB appena ricevuti:

The image consists of two vertically stacked screenshots from a blockchain interface. Both screenshots show a call to the `balanceOf` function on the address `0x515Ffd65894B3a3f`.  
The top screenshot shows the result: `0: uint256: 0`.  
The bottom screenshot shows the result: `0: uint256: 2302363174505426833`.

È possibile anche notare che le **riserve** del pool ora contengono 13 TokenA e circa 7.6 TokenB, che in totale sono più dei 10+10 token iniziali, di conseguenza ora le quote hanno un valore superiore e se non fossimo stati noi a fare la swap, prelevando i fondi avremmo guadagnato una plusvalenza.

Per provare a prelevare liquidità, eseguiamo la **burn** con 5 delle nostre 10 quote e osserviamo come la liquidità diminuisce e i nostri bilanci dei token aumentano ancora una volta:

The image consists of three vertically stacked screenshots from a blockchain interface illustrating the **BURN** process.  
The top screenshot shows the **BURN** interface with the input `shares: 50000000000000000000000000000000` and buttons for `Calldata`, `Parameters`, and `transact`.  
The middle screenshot shows the **Transaction request** screen with the heading **Estimated changes**. It shows the user receiving `+ 6,5` (TokenA) and `+ 3,849` (TokenB).  
The bottom screenshot shows the `liquidity` balance for the address `0x515Ffd65894B3a3f`: `0: uint256: 50000000000000000000000000000000`.

Possiamo notare che per 5 quote abbiamo ottenuto 6.5 TokenA e 3.849 TokenB, mentre all'inizio 5 quote valevano leggermente meno: 5 TokenA e 5 TokenB (a parità di valore dei due token).

## Conclusioni

Con i contratti precedentemente illustrati è possibile implementare il cuore di un Automated Market Maker basato su un pool tra TokenA e TokenB.

Ricollegandosi alla teoria, tramite questo nucleo è possibile osservare molte dinamiche che avvengono continuamente in ambito DeFI basata su AMM e Liquidity Pools. In particolare:

- I **liquidity providers** versano **token** in cambio di **quote** del pool.
- Qualunque utente può eseguire lo **swap** tra token, sulla base della formula dell'AMM.
- I liquidity providers possono **bruciare** le quote e prelevare i token.
- Il pool aumenta il valore attraverso le **fees** e i liquidity providers possono trarre guadagno da questo processo, dato che le quote valgono sempre di più se tutto va per il verso giusto.

Tutto il materiale è presente nella repository <https://github.com/Matti02co/BCSC2025-CoccoLepuri>.

## Bibliografia

Andrea Cesaretti - Auto-Paired DeFi Pools: Illusory Liquidity and Price Manipulation. A Critical Review.

Yongge Wang UNC Charlotte - Automated Market Makers for Decentralized Finance (DeFi).

Hayden Adams, Noah Zinsmeister, Dan Robinson – Uniswap v2 Core.

Coinbase.com

<https://github.com/Uniswap/v2-core/tree/master>

<https://docs.uniswap.org/contracts/v2/overview>

<https://docs.balancer.fi/>

<https://resources.curve.finance/>

<https://www.horizen.io/academy/liquidity-in-defi/>

<https://www.ulam.io/blog/best-crypto-exchange-liquidity-providers#>

<https://www.alloy.capital/articles/institutional-defi-the-asset-management-service-provider-landscape>

<https://www.fireblocks.com/>

<https://docs.beefy.finance/>

<https://coinmarketcap.com/academy/article/a-deep-dive-into-how-the-top-daos-work>

<https://immunefi.com/bug-bounty-program/>

<https://chain.link/whitepaper>

<https://tokenterminal.com/explorer/projects/uniswap/financial-statement>

<https://coinmarketcap.com/currencies/curve-dao-token/>

<https://github.com/uniswapfoundation/v4-template/blob/main/src/Counter.sol>

<https://github.com/Uniswap/v4-core/blob/main/src/PoolManager.sol>

## Licenza

© 2025 Mattia Cocco & Tomas Lepuri – Licenza CC BY 4.0