Session 6

Navigator sessions

Date	Topics	
15/12/23	o1js review	
19/1/24	Development workflow, design approaches, techniques and useful patterns	
26/1/24	Recursion	
9/2/24	Application storage solutions	
1/3/24	Utilising decentralisation	
15/3/24	zkOracles and decentralised exchanges	
5/4/24	Ensuring security	
26/4/24	Review Session	

Today's topics

- What's new
- zkOracles
- Decentralised Exchanges

What's new

Road to upgrade

"The Devnet upgrade is the final milestone before Mina's mainnet upgrade. It provides the ecosystem one last opportunity to test the Berkeley features and upgrade process thoroughly."
Originally scheduled to start on March 14th and run for 1 week, this has been postponed.
Tentative dates are

Date	Time (UTC)	Downtime Timer	Task	Owner
Devnet Pre-Upgrade				
Thu Mar 14	19:00		Devnet upgrade GO/NOGO decision	Mina Foundation & o1Labs
Thu Mar 14			Share Devnet upgrade schedule in Discord & Twitter	Mina Foundation
Thu Mar 14			Start initial archive node migration	archive node operators
Mon Mar 18			Devnet release of a build with stop-slots features	o1Labs
Mon Mar 18			Community and Partners start upgrading their nodes	block producers and SNARK workers
Devnet State Finalization Period				
Thu Mar 21	9:00	0:00	**stop-transactions-slot triggered**	**Milestone**
Thu Mar 21	14:00	5:00	Last incremental archive node migration step	archive node operators
Thu Mar 21	14:00	5:00	**stop-network-slot triggered**	**Milestone**
Devnet Berkeley Network Preparation				
Thu Mar 21	14:00	5:00	Run state export	o1Labs
Thu Mar 21	15:00	6:00	Manual o1Labs archive node migration	o1Labs
Thu Mar 21	15:00	6:00	Prepare Devnet Berkeley build	o1Labs
Thu Mar 21	16:00	7:00	Start Devnet Berkeley infrastructure	o1Labs
Thu Mar 21	18:00	9:00	Start Devnet Berkeley seed nodes	o1Labs
Thu Mar 21	19:00	10:00	Publish Devnet Berkeley build	o1Labs
Thu Mar 21	20:00	11:00	**Package Published**	**Milestone**
Thu Mar 21	20:00	11:00	Community and partners upgrade their nodes	block producers and SNARK workers
Thu Mar 21	20:00	11:00	Community archive nodes and Rosetta API upgrade	archive node operators
Devnet Berkeley Network Launch				
Thu Mar 21	21:00	12:00	**1st Berkeley block created**	**Milestone**
Thu Mar 21			Monitor Devnet Berkeley network	Mina Foundation & o1Labs
Tue Mar 26			Check min-window density after 4.67 days of grace period	Mina Foundation & o1Labs

New Mainnet version released

See announcement

zkOracles

See video
See tutorial

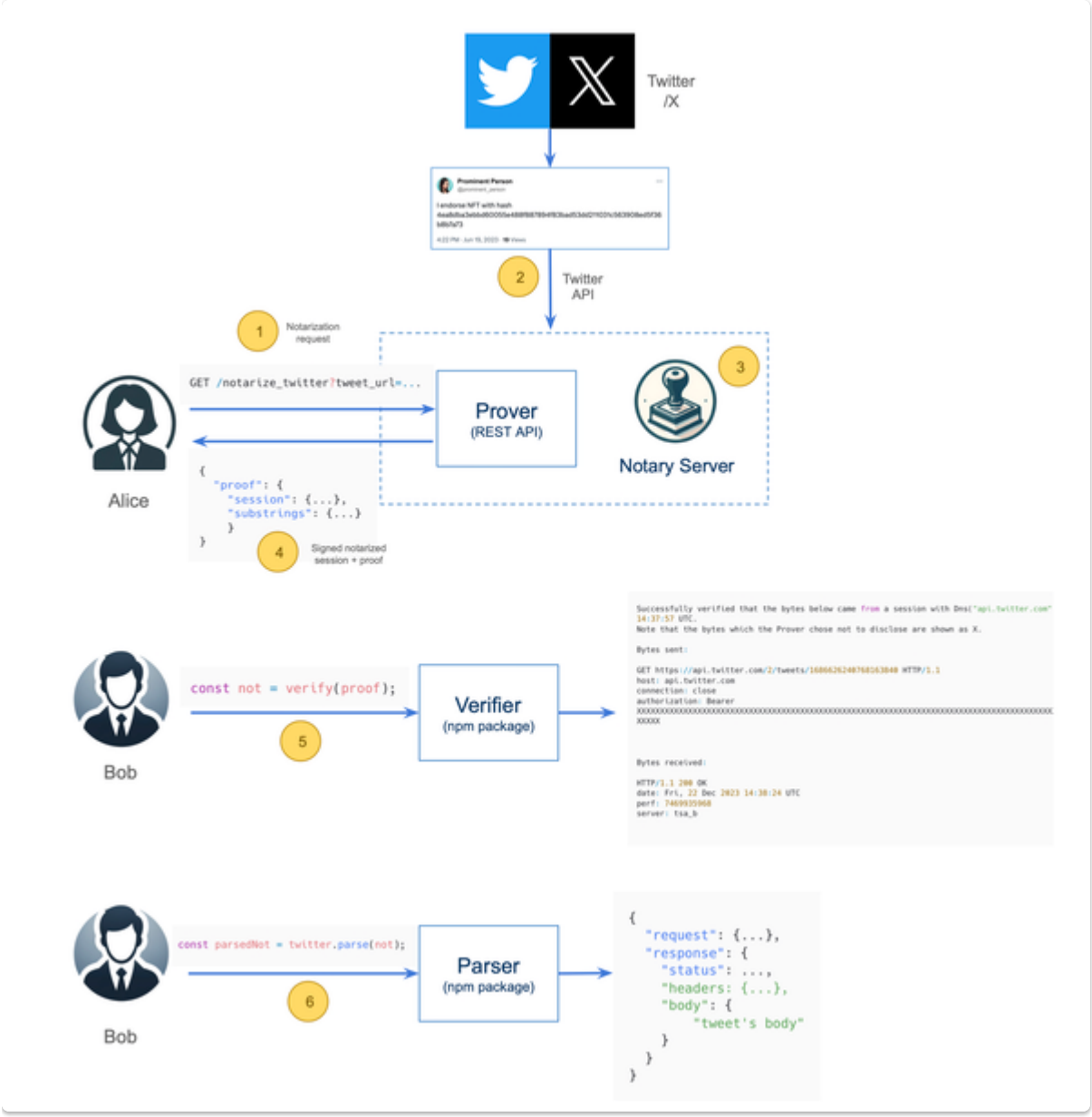
Mina has a strong use case with zkOracles

- there is the need for trustworthy data
- ideally the data source should not have to be changed to provide this data

The idea behind an oracle is to allow a zkApp to get data from any HTTPS data source,
but this has not been fully realised, see this thread

This is a similar idea to that of TLSNotary, but with Mina we can integrate this with zkApps
To do this in Mina, you need to have support for the signature schemes used by TLSNotary.
See this discussion.
There was a project from cohort 2 called zkNotary that tried to do this, see there comprehensive writeup

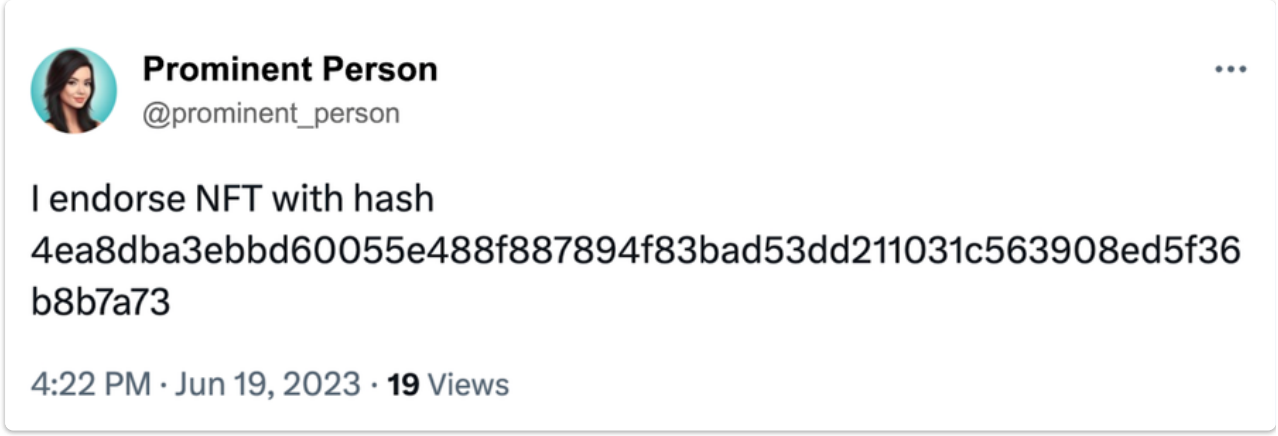
This is the description of the proposed flow in zkNotary



- Step 1: Alice calls the zkNotary Prover REST API with the tweet's URL as a querystring argument.

```
GET /notarize_twitter?
tweet_url=https://twitter.com/mathy782
/status/1670919907687505920
```

This is the tweet that is being notarized for this example:



- Step 2: The zkNotary Prover and the Notary Server use MPC to create a joint TLS session with the Twitter API and retrieve the tweet's data.
- Step 3: After closing the TLS session with Twitter, the Notary Server executes the notarization of the TLS session and signs it with its private key.
- Step 4: The Prover REST API replies to Alice with a JSON object containing the signed notarization of the TLS session with the Twitter server as well as the corresponding proof generated by the Notary Server.
- Step 5: Alice sends the signed notarized session and the proof to Bob. In order to verify it by himself, he uses the `verify()` function from the `zknotary-verifier` npm package. This function takes as input the proof and the Notary Server's public key

and returns a plain text file containing the transcript from the TLS session.

Note: Given that the verifier's output is a plain text file, in order for Bob to make sense of it and extract the important information (in our example, the tweet's text), he needs to parse it. But because the contents of the transcript vary substantially depending on their source (Twitter, Reddit, Discord, etc) we have decided to write a set of utility functions that correctly parse the verifier's output, given its source.

- Step 6: Bob uses the `twitter.parse()` function from the `zknotary-parsers` npm package to parse the plain text file and receive in return a structured JSON object from which he can easily extract the important information, i.e, the tweet's data:

```
{
  "request": { },
  "response":
    "status": "...",
    "headers": { },
    "body": {
```

```
          "data": {
            "edit_history_tweet_ids":
[
              "1686626240768163840"
            ],
            "id":
"1686626240768163840",
            "text": "I endorse this
NFT:
0f1a6f87599424233134e77a0214f9ebe2a2a7
da73ed3025801412ff34e3ae2a"
          }
        }
}
```

Process as shown in tutorial 7

See Tutorial

The tutorial creates an oracle that

- Fetches data from the desired source
- Signs it using a Mina-compatible private key
- Returns the data, signature, and public key associated with the private key
- Allows the signature to be verified by the zkApp

The complete code is available here
You can try out a version here

Data returned by the Oracle

- `data`: An object of the information you are interested in and can have any form.
- `signature`: A signature for the `data` created using the oracle operator's private key. Smart contracts use this signature to verify that data was provided by the expected source.
- `publicKey`: The public key of the oracle is the same for all requests to this oracle.

The data it returns is in JSON format
for example


```
{"data":
{"id":1,"creditScore":787},"signature":"7m
XGPCbSJUiYgZnGioezZm7GCy46CEUbgcCH9nrJYXQQ
iwwVrA5wemBX4T1XFHUw62oR2324QNnkUVXW6yYQLs
PsqxZ3nsYR","publicKey":"B62qoAE4rBRuTgC42
vqvEyUqCGhaZsW58SKVW4Ht8aYqP9UTvxFWBgy"}
```


The relevant parts are. :
Setting up a private key to sign the data

```
  let privateKey =
    process.env.PRIVATE_KEY ??

'EKF65JKw9Q1XWLDZyZNGysBbYG21QbJf3a4xnEoZP
Z28LKYGMw53';
```

The data we are providing

```
  const knownCreditScore = (userId:
number) => (userId === 1 ? 787 : 536);
```

creating a signature on the data

```
const signature = client.signFields(
  [BigInt(userId), BigInt(creditScore)],
  privateKey
);
```

zkApp

Turning to the contract the will be verifying and using the data

We keep the public key of the oracle, as we need it to verify the signature

```
@state(PublicKey) oraclePublicKey =
State<PublicKey>();
```

We need a method to verify the data
It has these arguments :

- `id`: The id of the user whose credit score is requested is required to prevent bad actors from querying somebody else's data and claiming it as their own.
- `creditScore`: The credit score of the user that is a number between 350 and 800 (this tutorial uses mock credit scores).

- `signature`: A cryptographic signature of `id` and `creditScore`. This is what the smart contract uses to verify that the data was provided by the expected source.

within the method we can verify the signature as follows

```
const validSignature =
signature.verify(oraclePublicKey, [id,
creditScore]);
```

and we need to assert that this verified

```
validSignature.assertTrue();
```

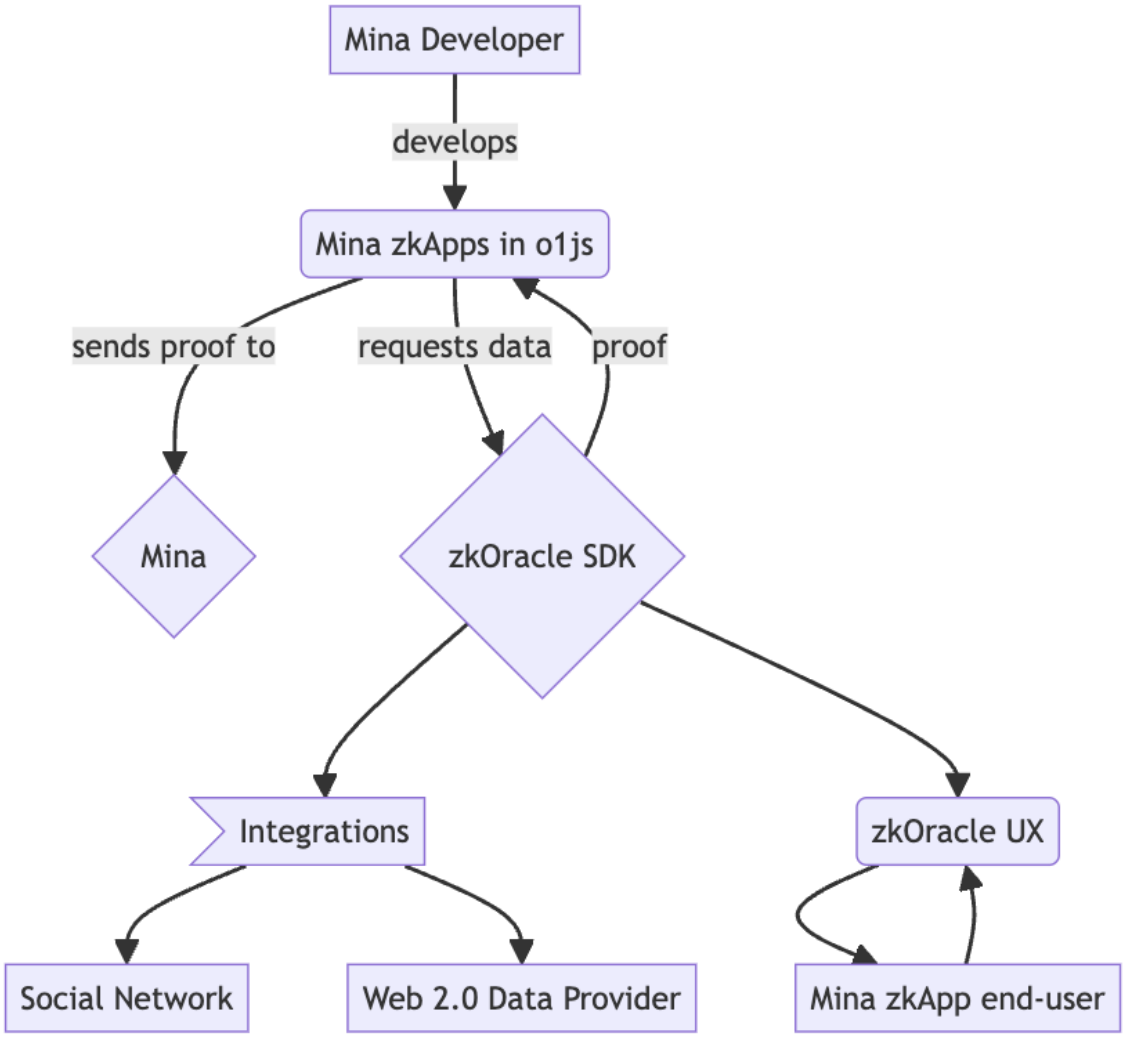
You can then use the data and add further assertions, in this case we could check that the credit score is greater than a certain amount.

```
creditScore.assertGreaterThanOrEqual(Field
(700));
```

In the tutorial events are then emitted to indicate that a sufficient credit score has been received.

RFP for zkOracle integration

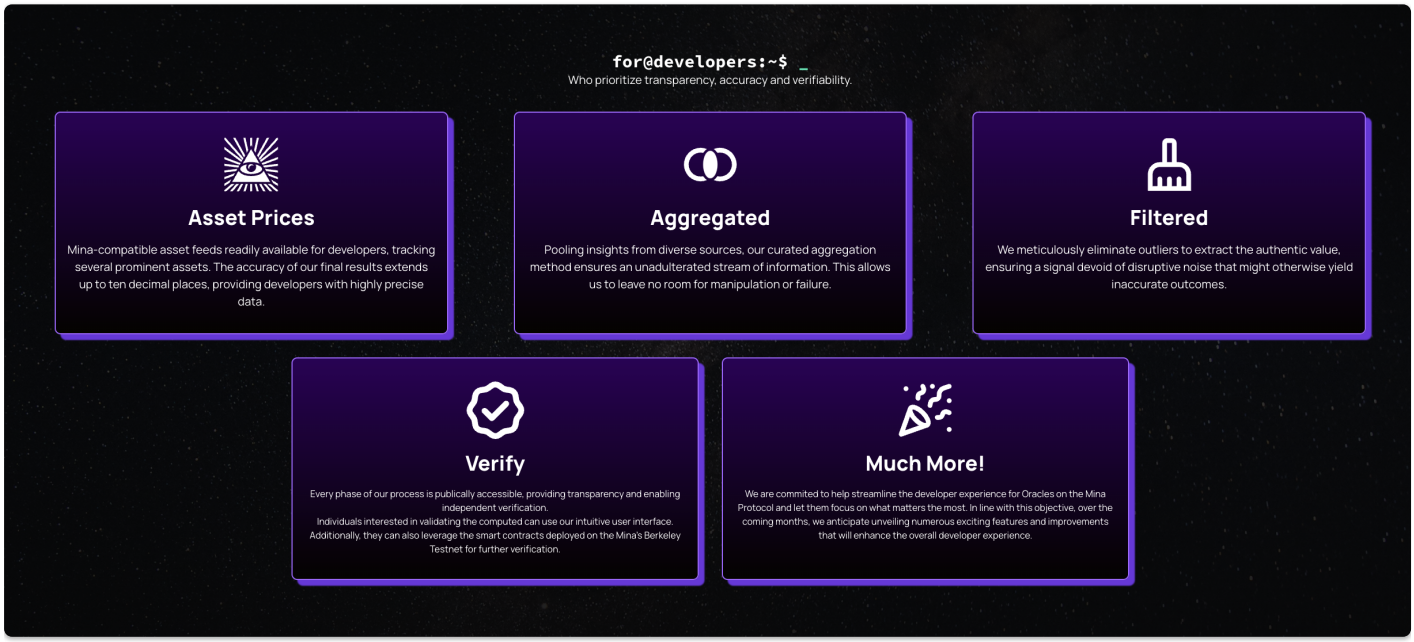
see ## Ecosystem Advancement (RFP): zkOracle Integration for o1js



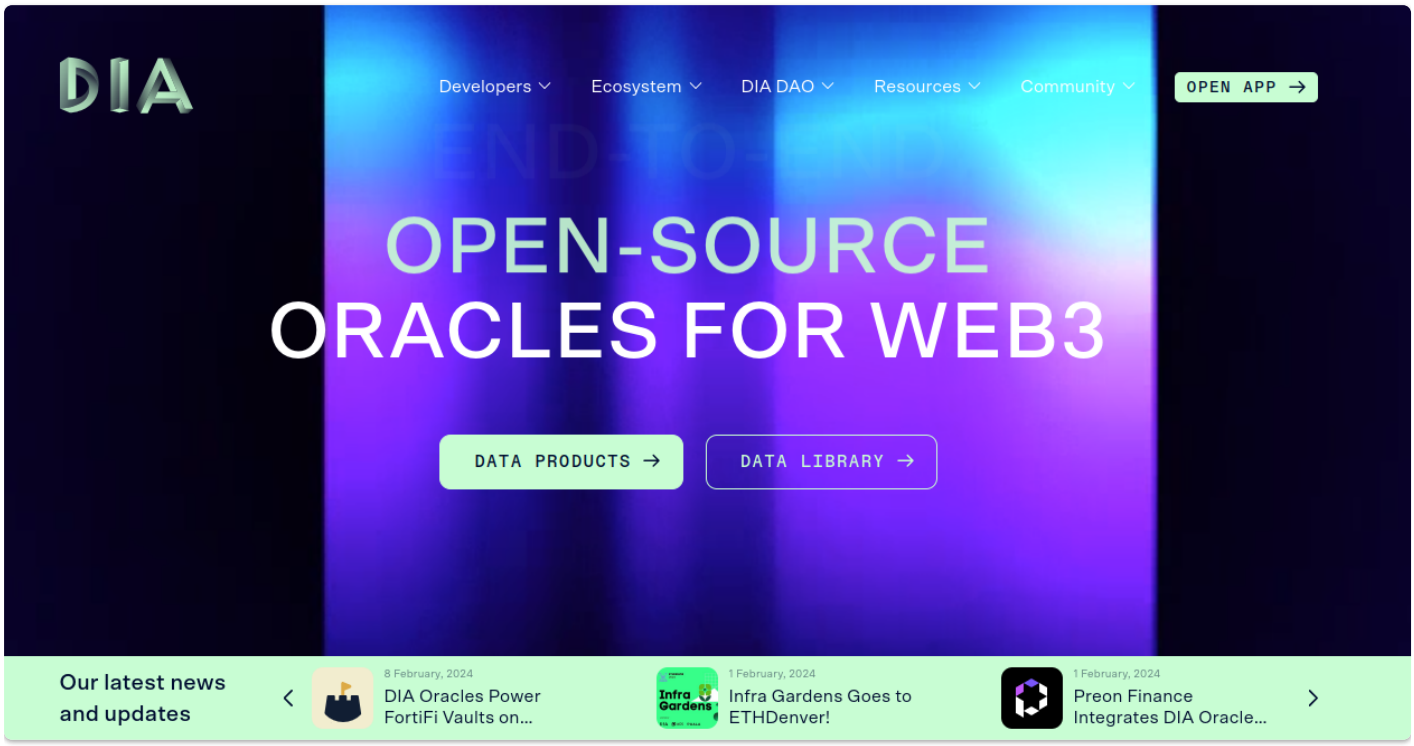
Doot Foundation

Another project from a previous cohort is from the Doot Foundation

See Docs



DIA Data



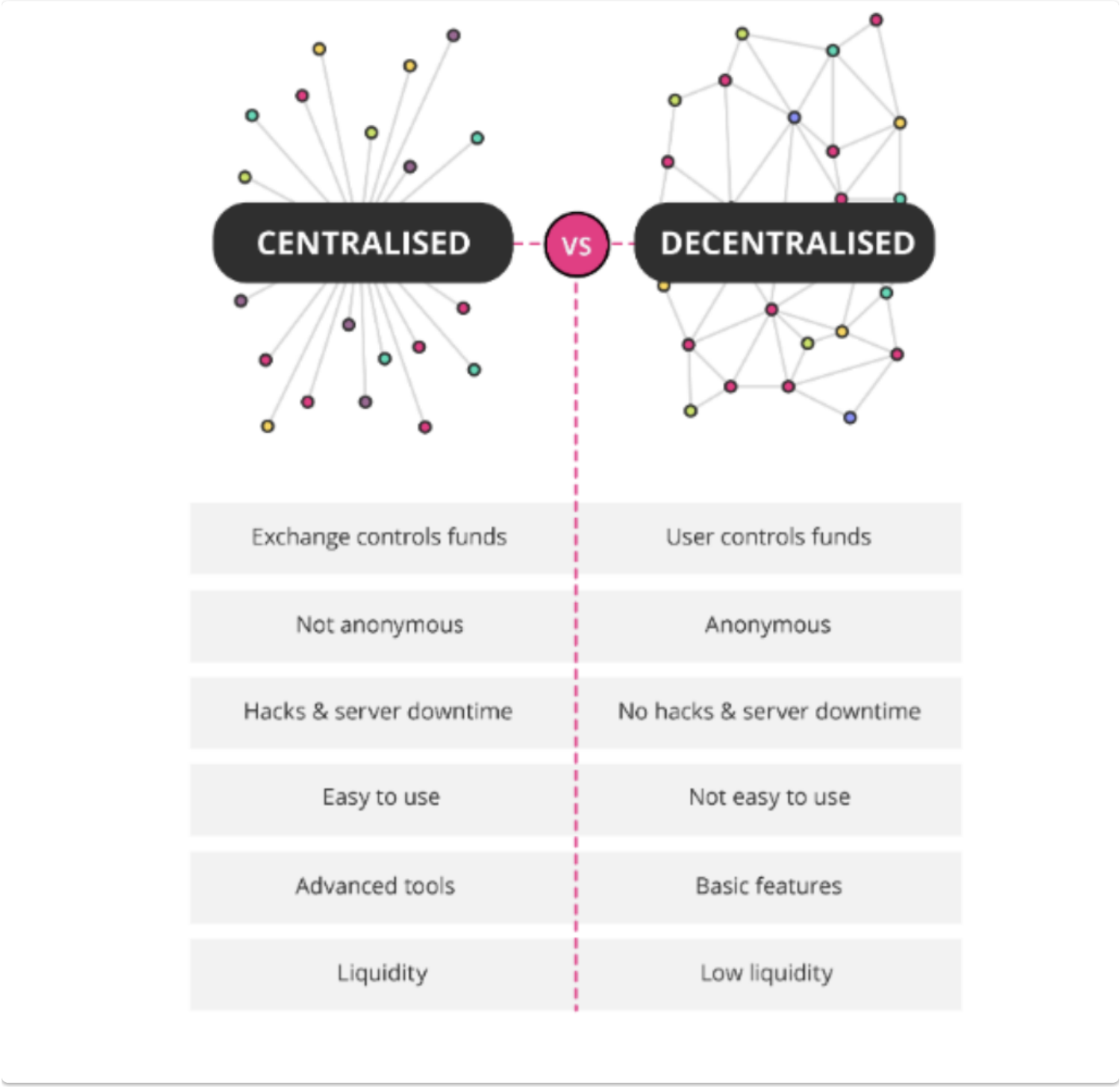
See Blog post

- POC using zk circuits
  This concentrates on proving that the computations done to enrich data has been done correctly.
  It is also able through the analysis of metadata to give an indication of the reliability of the data source.

Decentralised Exchanges

Introduction

Decentralised Exchanges are a protocol to provide asset exchange without the platform holding the users assets



Decentralised Exchanges allow for more 'democratisation' since token creators do not

> need their tokens to be listed by the exchange (costly permissioning)

Automatic Market Makers

Incentivising Users

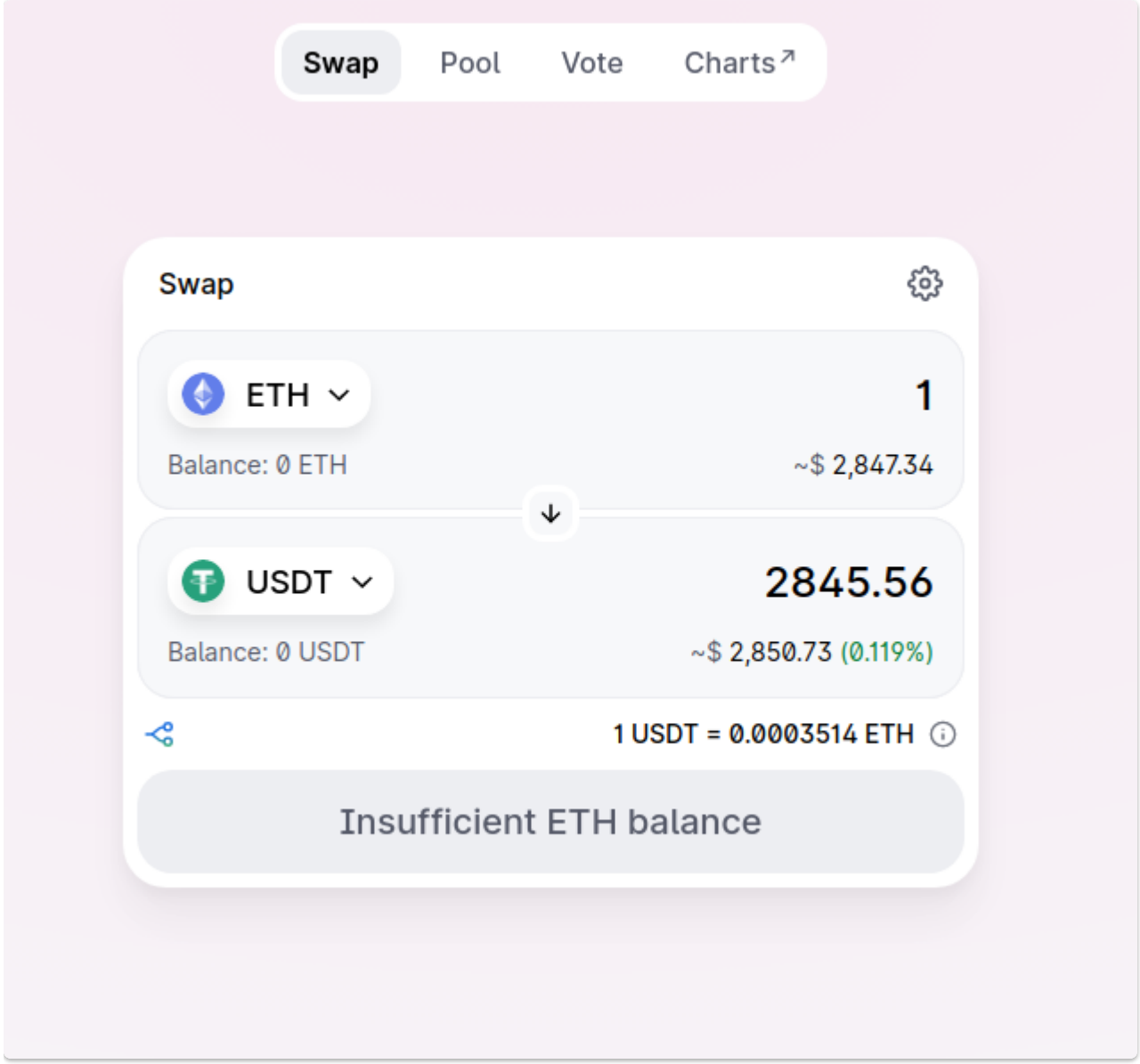
- Users deposit funds into a liquidity pool, for example ETH and USDT
- This pool (a token pair) allows users to exchange (or maybe lend or borrow) tokens
- Interacting with the exchange incurs fees
- These fees are paid to the liquidity providers

They are characterised as constant function market makers.

From Constant Function Market Makers

The term “constant function” refers to the fact that any trade must change the reserves in such a way that the product of those reserves remains unchanged (i.e. equal to a constant).

The technique and pricing for this is often referred to as XYK , since we are looking to maintain the ration of 2 tokens (X,Y) so this should always be equal to a constant K


Swap
Pool
Vote
Charts
Swap
ETH
1
Balance: 0 ETH
~$ 2,847.34
USDT
2845.56
Balance: 0 USDT
~$ 2,850.73 (0.119%)
1 USDT = 0.0003514 ETH
Insufficient ETH balance

Add Liquidity

Clear All UNI ETH

Select Pair

ETH UNI

0.3% fee tier

1% select

Edit

Deposit Amounts

ETH 1

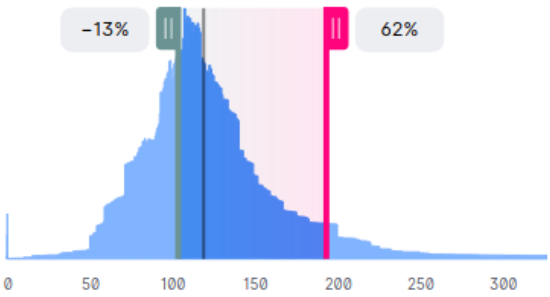
Balance: 0 ETH (Max) ~$ 2,847.22

UNI 36.1374

Balance: 0 UNI (Max) ~$ 862.921

Set Price Range

Current Price: 118.94 UNI per ETH



Min Price
− 103.94 +
UNI per ETH

Max Price
− 192.82 +
UNI per ETH

Full Range

Insufficient UNI balance

LP Tokens

Typically the liquidity provider receives LP tokens when they add liquidity, say ETH and USDT

Later they can take liquidity by providing LP tokens to the contract and will receive back ETH and USDT.

Ideally they will make a profit

Note that on Uniswap v3 LP positions are given by NFTs as opposed to the ERC-20 tokens used in Uniswap V1 and V2.

See Intro to AMMs

Price Impact / Slippage / Impermanent Loss

Price Impact

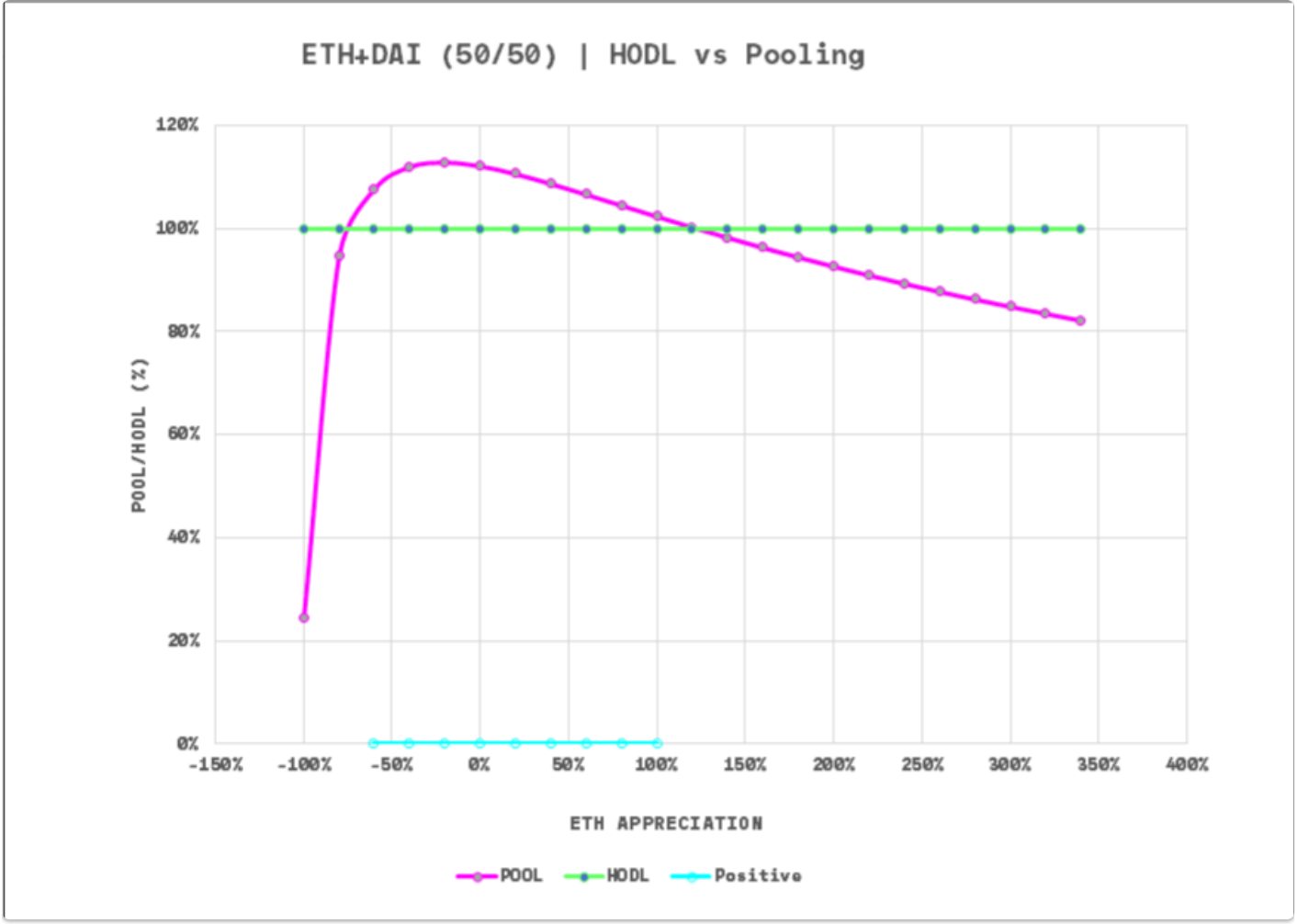
Each swap changes the price of the underlying tokens, in terms of each other, a large transaction can substantially impact a token's price.

Slippage

Slippage is the aggregated price impact of other traders from the time you submit a trade to the time your trade executes on the blockchain. Similar to price impact, deep liquidity pools help mitigate slippage.

Impermanent Loss

Impermanent loss occurs when the **value of your percentage-based share of a liquidity pool** is *less* at the time of withdraw than the **value of the amount of tokens you deposited**, at the time of withdraw, had you never deposited them and held them instead.



While liquidity providers can use stablecoins, yields, and rewards to help lessen the impact of impermanent loss they can also reduce this by using liquidity pools that use ratios other than 50/50.

Other types of DEXs

Coming from TradFi we also have Decentralised Central Limit Order Book (dCLOB) exchanges. See Intro

The dCLOB uses a time sequenced order book, and a matching engine which can match orders coming into the system in real time.
Although similar in design to centralised CLOBS, dCLOBS typically have lower liquidity and poorer performance.

Composability

A multitude of DeFi applications (“Money LEGOs”) can be connected to create new financial products.
See Monolith article

The applications on the Ethereum network can run interchangeably, and they all support ETH and other ERC-20 tokens. They can be used in endless combinations, with no third party intermediary controlling any element of the network activity. Composability is a core basis of DeFi, and it’s what’s helped the ecosystem grow so quickly.

Risks associated with composability

There are the usual risks associated with a protocol or smart contracts, but in addition there is a risk when combining products.
The contracts for one application may be secure until combined with those of another application. For example, on 12th March 2020, a crash in the price of ETH wreaked havoc for DeFi protocols as holders rushed to exit their positions. A gas price spike caused a lag across

> price oracles, leading to widespread liquidations in protocols like Maker.

DEX problems

Liquidity

Early DEXs suffered from lack of liquidity, though this was solved to some extent with the introduction of AMMs.

Coordination

Associated with lack of liquidity, early DEXs needed to participants to interact within a short time span

Non standard tokens

The use of the ERC20 standard has helped reduce development costs for exchange and wallet builders.
There are conventions around the number of decimals a token will have, and deviation from this can be frustrating.

Privacy

Typically a Dex will only be as private as the chain it is built on, hence Mina has real potential here.
Other efforts in this area include Aztec, and Renegade who use collaborative SNARKS and MPC,

Building a DEX

Typical interactions

Users typically interact with a DEX with

- A swap, providing one token and receiving another from a token pool.
- Adding liquidity - adding amounts of both tokens to a pool and receiving LP tokens as a reward.
- Removing liquidity - exchanging LP tokens for pool tokens.

More sophisticated exchanges will allow multiple routes and multiple hops, optimising interaction between other exchanges and multiple pools to get the best price for a swap.

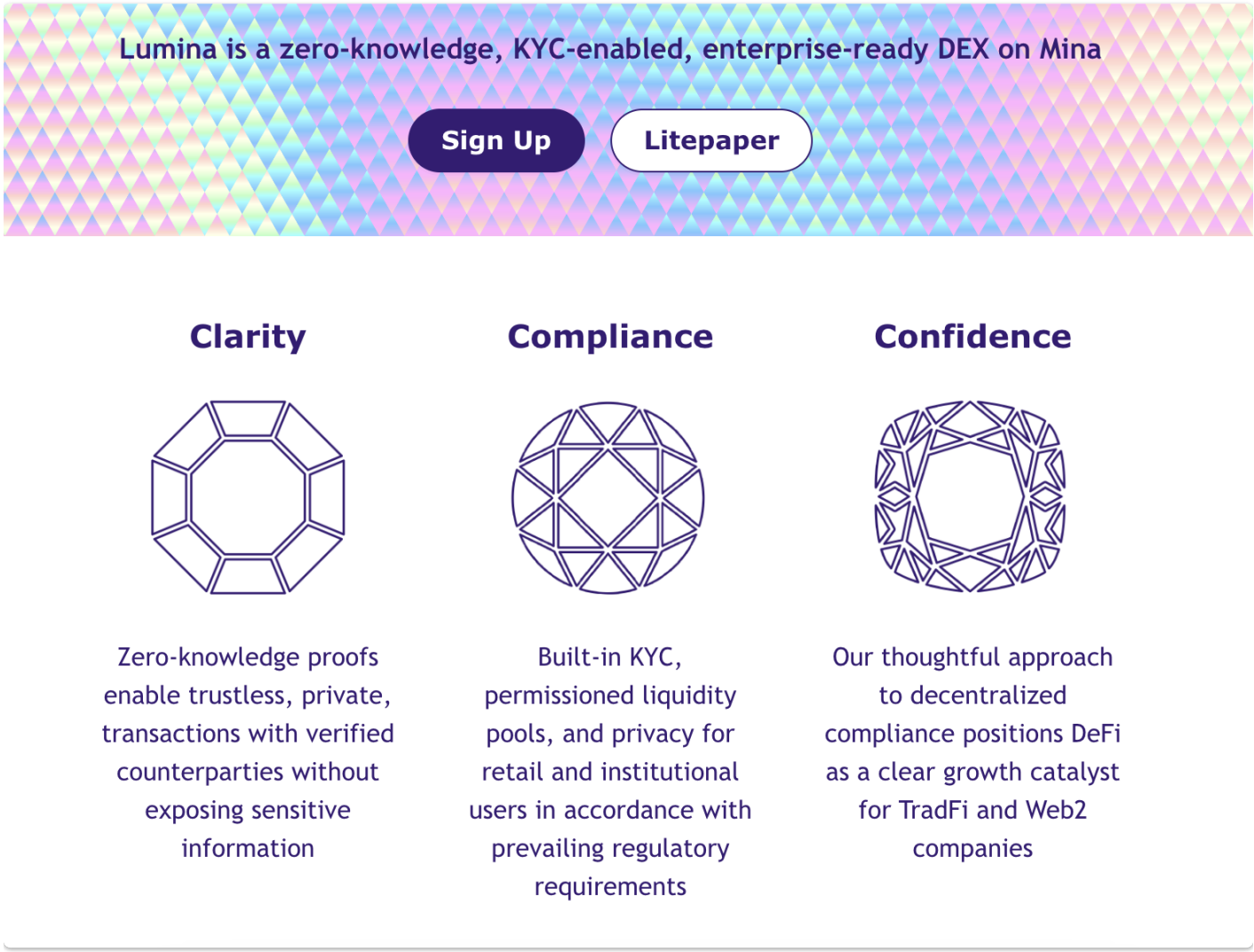
Typical Architecture

- Token Pools
  - often 2 tokens could can be more.
  - The pool needs an identity, usually created from sorting the symbols and hashing the result.
- Router contract

        - The interface that the user sees, it will coordinate the user interaction.
        - Finds the correct pool and initiates swaps etc.
- On ZKP based systems a UTXO approach, as opposed to an account approach is often used.

Example DEXs on Mina

Lumina



See Site

See Litepaper

Lumina DEX will be a rollup.

There will be 4 UTXO operations

- Deposit
- Swap
- Send
- Withdraw

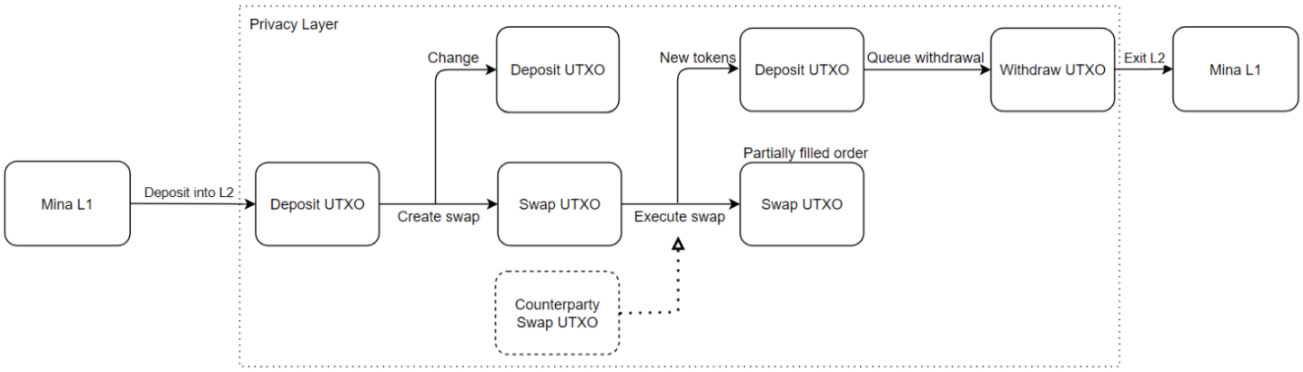


Figure 1: DEX flow for a single user

We would normally see the user creating proofs as they prepare the transaction. Lumina also suggests that relayers can be used to outsource the proof process.
Doing this securely adds extra complexity to the process.

Pool operators

Pool operators can create a permissioned liquidity pool for swapping and sending, and set or update KYC requirements via governance.

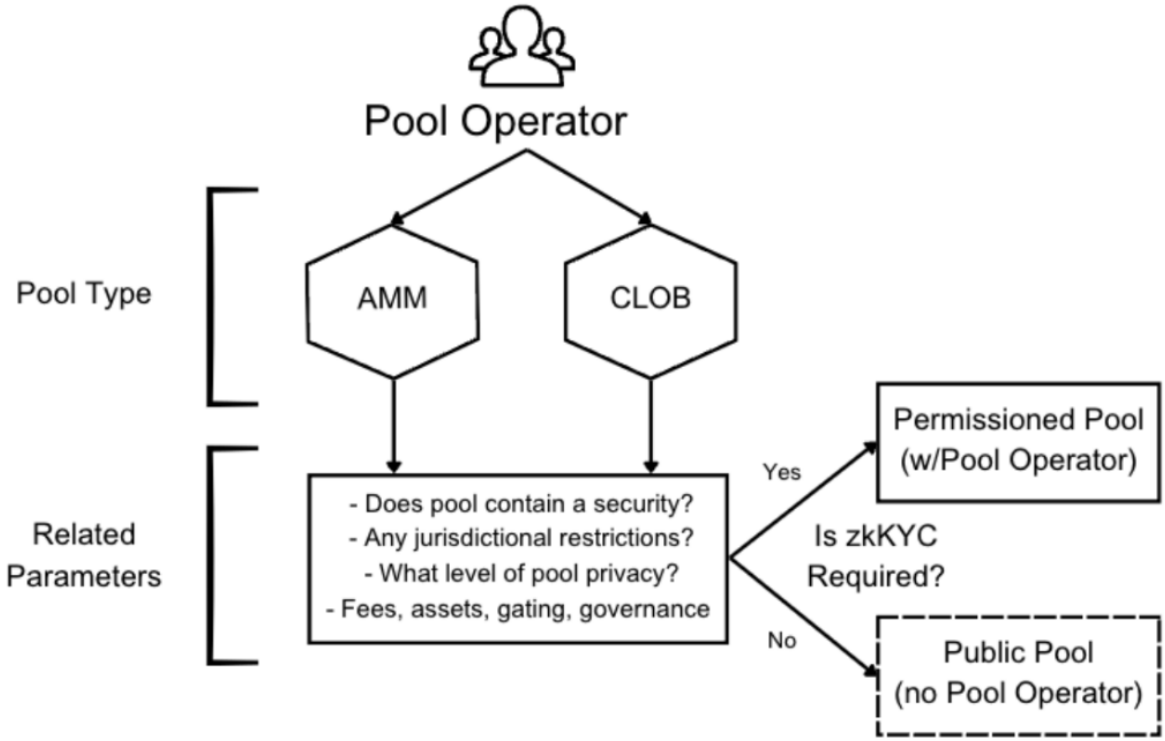


Figure 3: Pool Operator permission flow

```
class CreateSwap {
    function execute(Field, swap_commitment: Swap): L2TxResult {
        // 1) Check swap content to make sure it's correct
        // 2) Create new commitment for change utxo if required
        return {
            new_commits_or_nullifiers: [
                deposit_nullifier,
                swap_commitment,
                change_commitment
            ],
            add_to_treasury: null, // no cost
        };
```

```
// Note: this needs to be executed collaboratively
class ExecuteSwap {
    function execute(
        bob_swap: Swap,
        alice_swap: Swap,
        bob_relayer: null | Address,
        alice_relayer: null | Address,
    ): L2TxResult {
        // 1) Check if swaps partially overlap
        // 2) If relayers used, check that they're part of the KYC set
        // 3) Execute swap
        // 4) Calculate treasury fee and relayer fees
        // 5) Create nullifiers for both swaps
        // 6) Create new commiments for
        //        received tokens from the swap
        //        any liquidity left a swap after executing
        //        relayer fees
        return {
            new_commits_or_nullifiers: [combine (5) and (6)],
            add_to_treasury: treasury fee from (4),
        };
    }
```

Also Lumina and Zeko joined forces

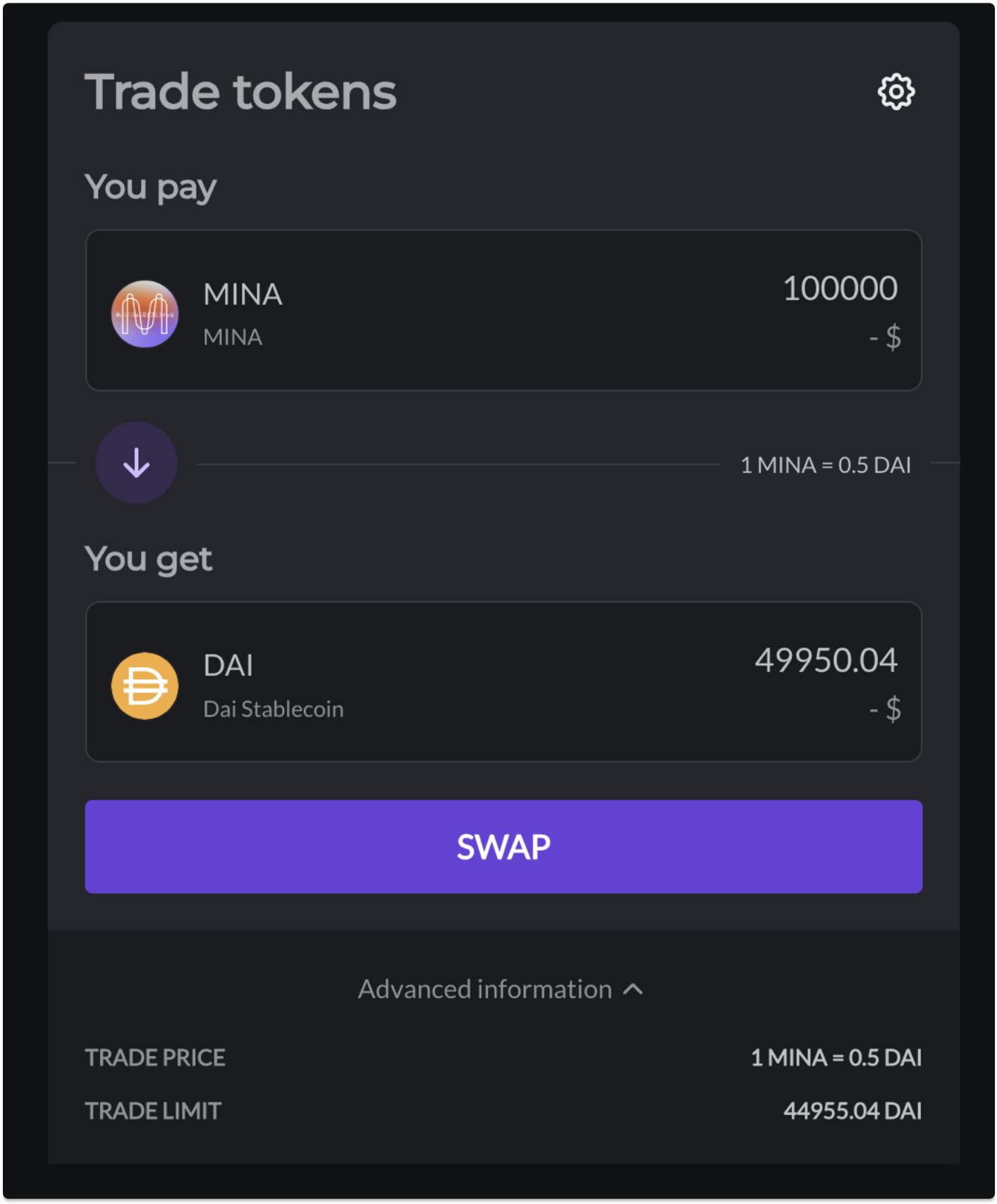
Zeko details

Kaupang Dex

Open source DEX built using Protokit as the L2

[Kaupang Demo]

[Repo]



They are part of Cohort 3 for further development, to produce an AMM and an orderbook.

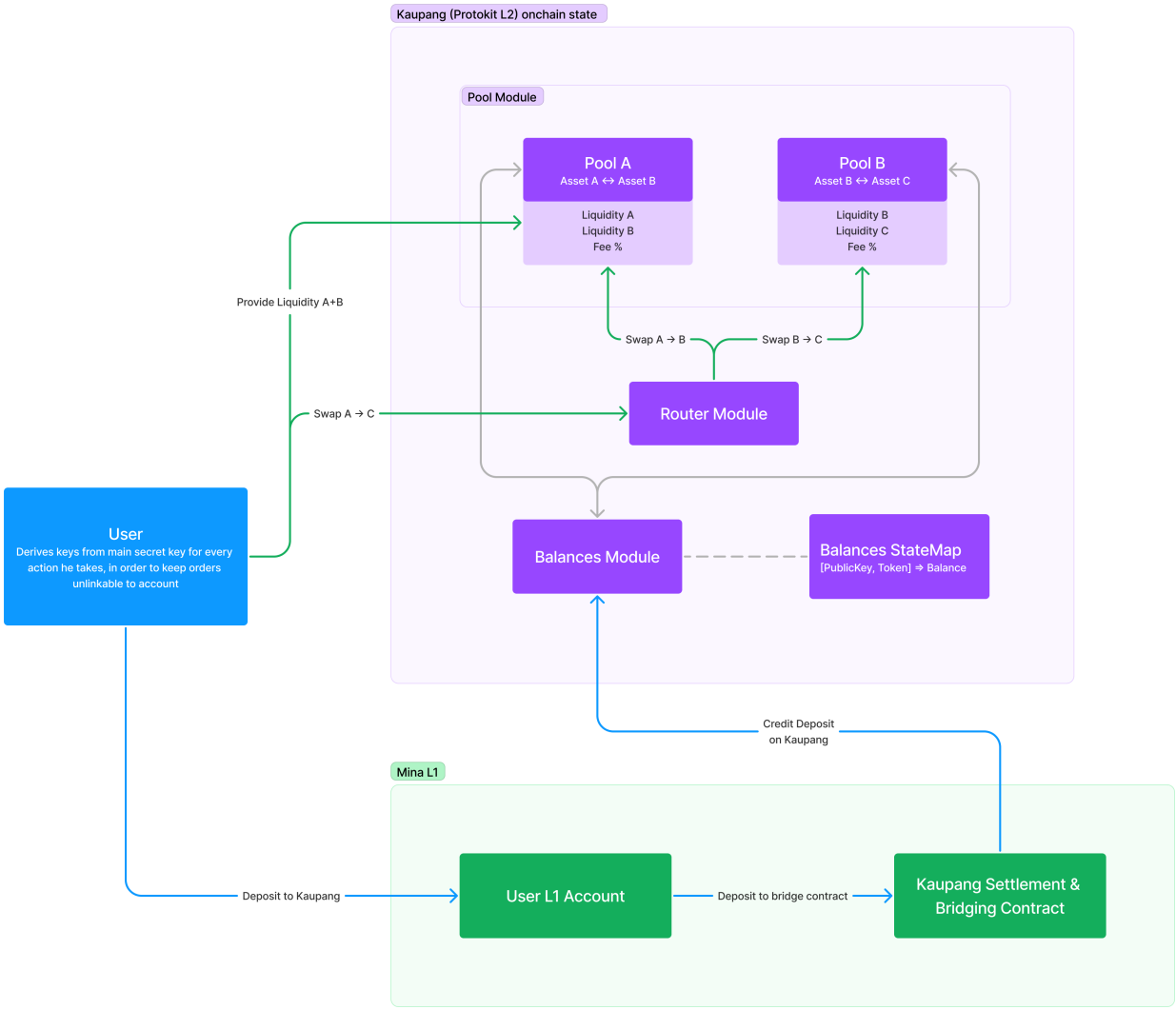
Architecture details taken from their proposal

- Built as an L2 using protokit
- Bridging of tokens between L1 and L2 using the protokit settlement & bridge contracts
- Recursive zk-proof architecture built as Protokit runtime and protocol modules
- Privacy enabled client side execution thanks to Protokit's hybrid execution model. Using ZkPrograms on the client side and sending proofs as arguments to runtime modules to be processed on-chain (sequentially).
- Private UTXO based architecture for the private order book, utilizing a combination of append-only merkle trees, in-circuit encryption, state proofs and zero-knowledge proofs.
- On-chain routing API to ensure atomicity of routed trades
- Shared security with the MINA L1, thanks to settling the L2 state to the L1 settlement

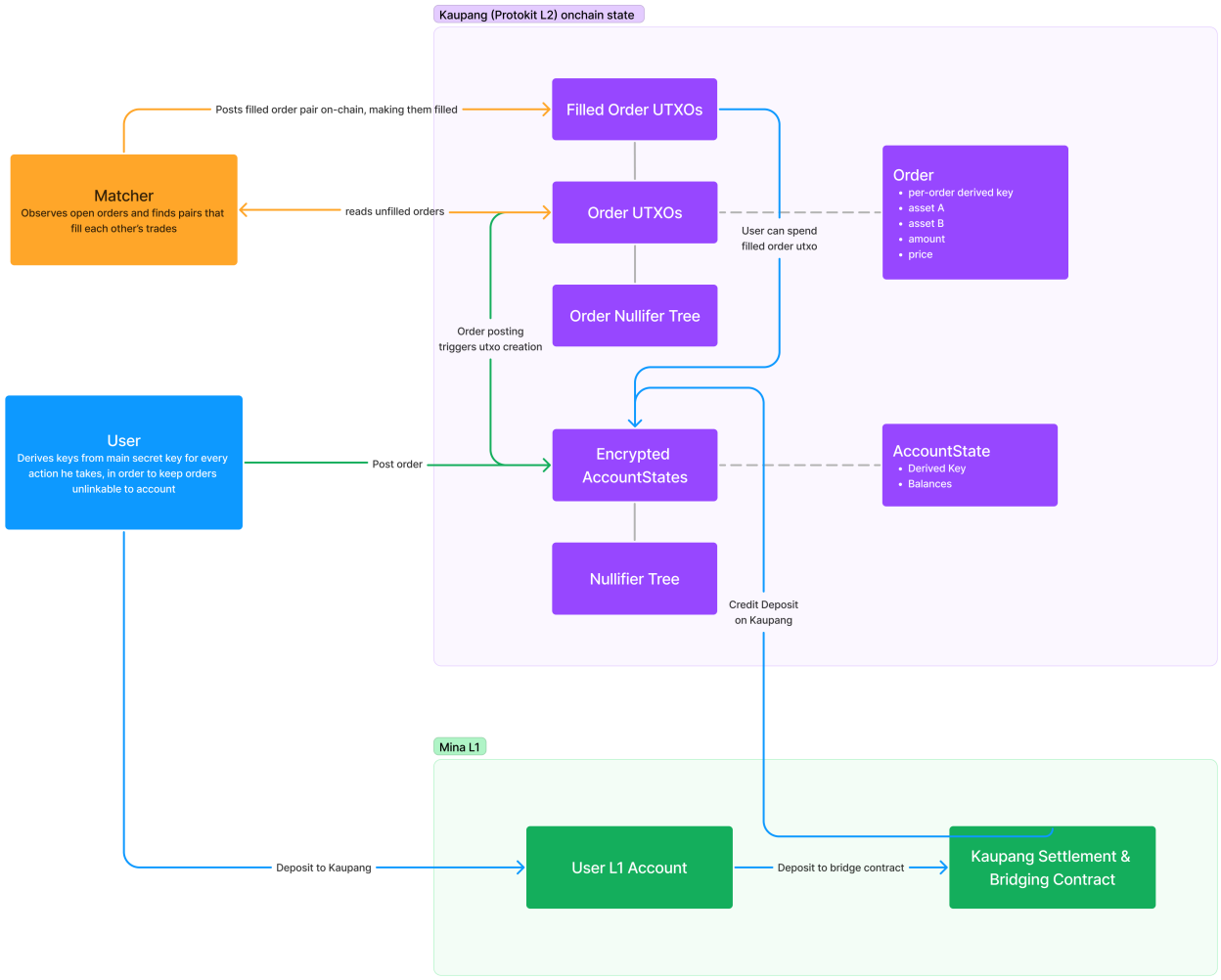
smart contract
- Background indexing and data processing of the L2 blocks to offer a data-rich UI

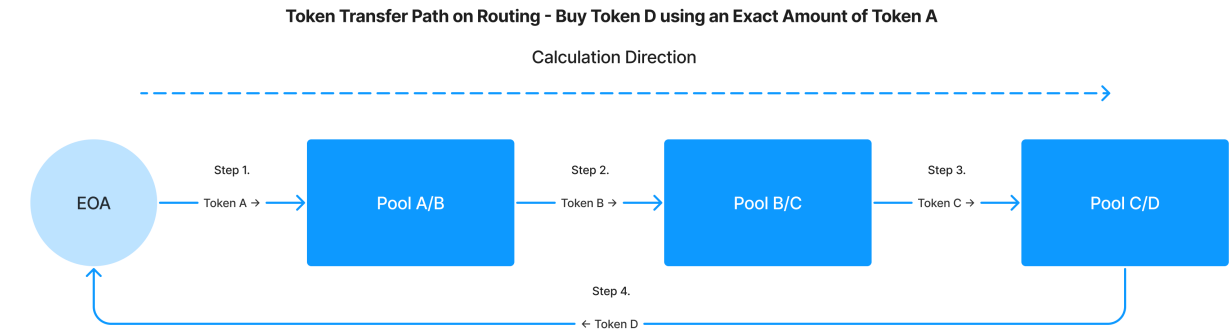
Routing API represents an essential component of the Kaupang XYK architecture. Routing enables users to have smoother experience in performing their swaps, as they are able to navigate through a multitude of pools in a single transaction.
While the infrastructure that enables utilization of routing is present in the runtime itself (also visible in the diagram at the top of the page), routed path is provided by the API which ensures highest output amount possible for the user.

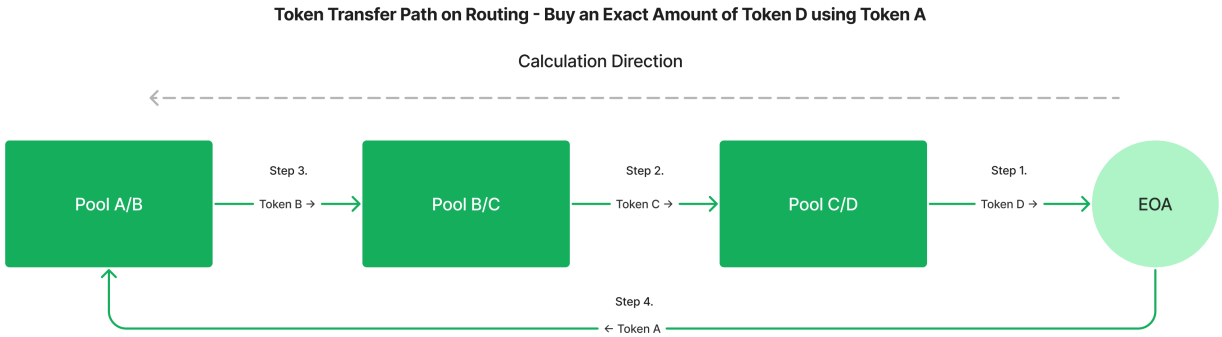
Kaupang XYK Architecture



Kaupang Orderbook Architecture







Example Kaupang Code

```
 public calculateTokenOutAmount(
    tokenInId: TokenId,
    tokenOutId: TokenId,
    tokenInAmount: Balance
  ) {
    const pool =
PoolKey.fromTokenIdPair(tokenInId,
tokenOutId);

    const tokenInReserve =
this.balances.getBalance(tokenInId, pool);
    const tokenOutReserve =
this.balances.getBalance(tokenOutId,
pool);
```

```
    return
this.calculateTokenOutAmountFromReserves(
      tokenInReserve,
      tokenOutReserve,
      tokenInAmount
    );
  }
```

```
@runtimeMethod()
  public buy(
    tokenInId: TokenId,
    tokenOutId: TokenId,
    tokenOutAmount: Balance,
    maxTokenInAmount: Balance
  ) {
    this.assertPoolExists(tokenInId,
tokenOutId);
    const pool =
PoolKey.fromTokenIdPair(tokenInId,
tokenOutId);

    const tokenInAmount =
this.calculateTokenInAmount(
      tokenInId,
      tokenOutId,
      tokenOutAmount
```

```
    );

    const isMaxTokenInAmountSufficient =

tokenInAmount.lessThanOrEqual(maxTokenInAm
ount);

    assert(isMaxTokenInAmountSufficient,
errors.tokenInAmountTooHigh());

    this.balances.transfer(
      tokenOutId,
      pool,
      this.transaction.sender,
      tokenOutAmount
    );

    this.balances.transfer(
      tokenInId,
      this.transaction.sender,
      pool,
      tokenInAmount
    );
  }
```

Kaupang and Staking

Kaupang DEX proposes to delegate L1 tokens on behalf of its users and letting a DAO vote on

to whom to delegate.