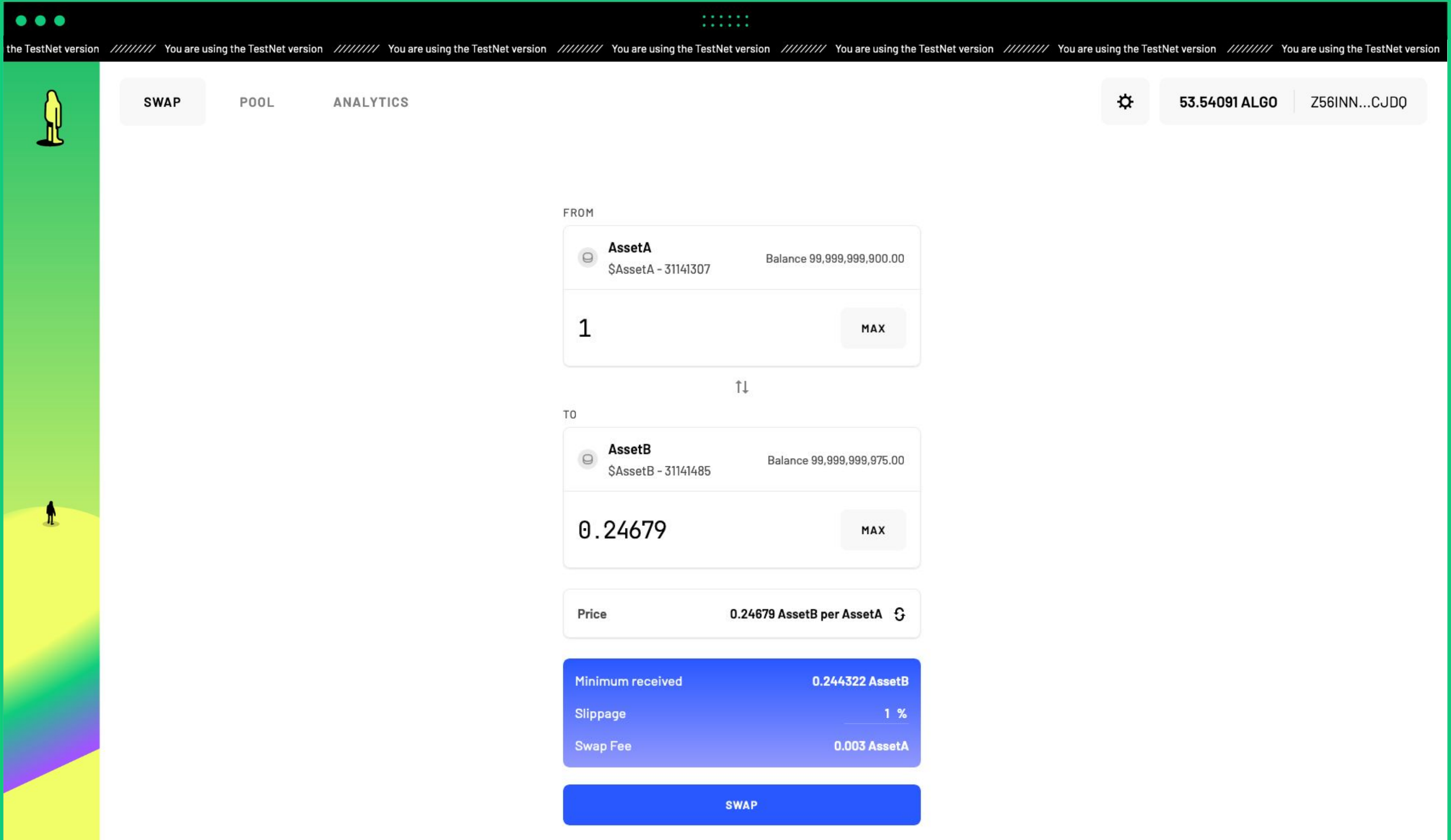
tinyman

Tinyman

Algorand Developer Office Hours
October 5th 2021

Intro

Tinyman is a permissionless decentralized trading protocol on Algorand



the TestNet version
You are using the TestNet version
You are using the TestNet version
You are using the TestNet version
You are using the TestNet version
You are using the TestNet version
You are using the TestNet version
SWAP
POOL
ANALYTICS
53.54091 ALGO
Z56INN...CJDQ
FROM
AssetA
$AssetA - 31141307
Balance 99,999,999,900.00
1
MAX
TO
AssetB
$AssetB - 31141485
Balance 99,999,999,975.00
0.24679
MAX
Price
0.24679 AssetB per AssetA
Minimum received
0.244322 AssetB
Slippage
1 %
Swap Fee
0.003 AssetA
SWAP

Screenshot: Signing with wallet

Screenshot: Add Liquidity

Screenshot: Manage Liquidity

Screenshot: Analytics

» Why Algorand?

Fast

Cheap

Secure

Reliable

Greeen

» Web App - Under the hood

JS AlgoSDK

Tinyman JS SDK

React

MyAlgoConnect

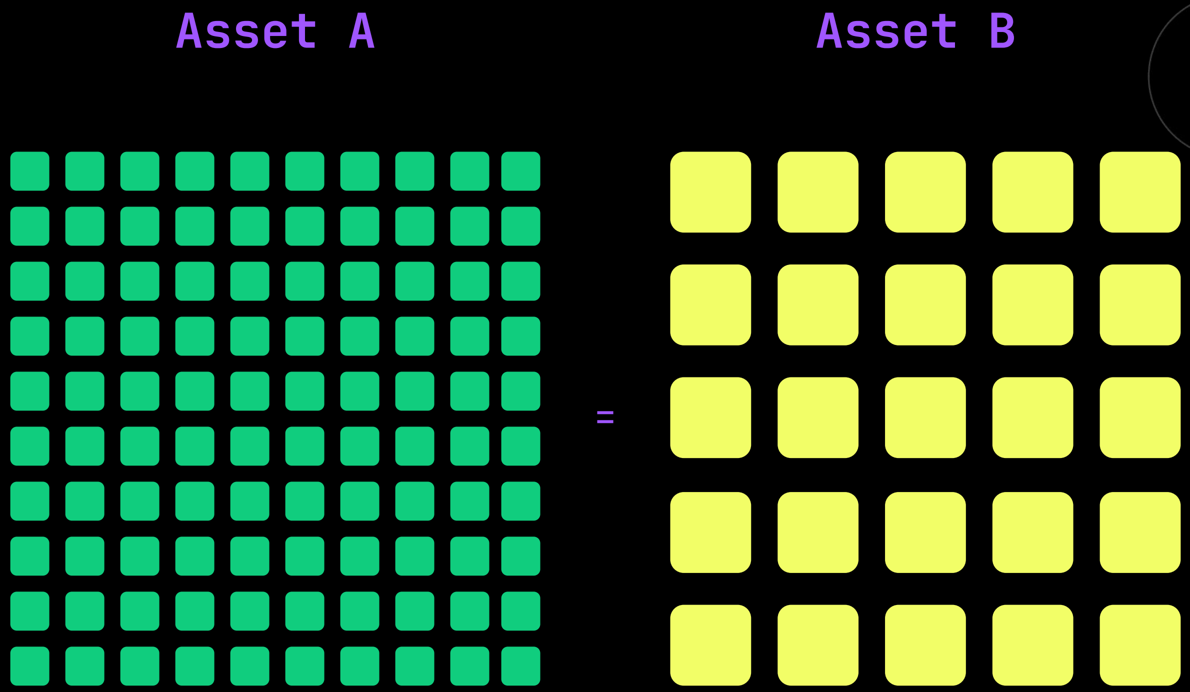
AlgoSigner

WalletConnect

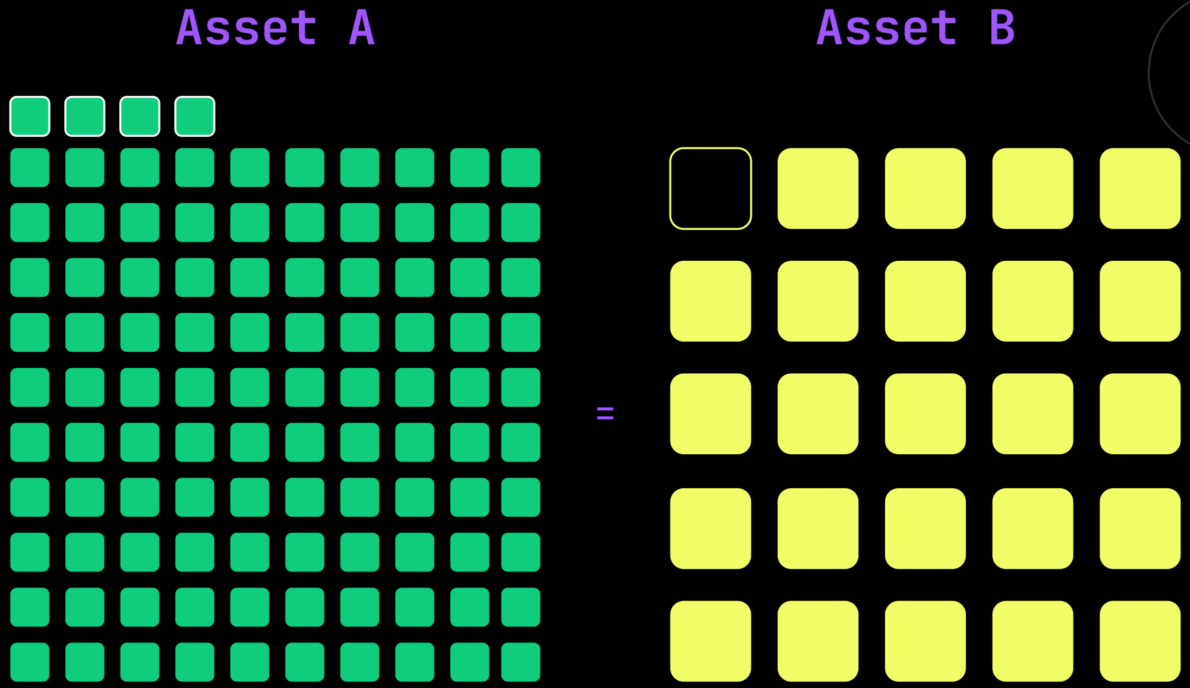
Analytics API

AMM Basics

» Pool



» Pool



» AMM

Automated Market Maker

Constant Product Market Maker

$$x * y = k$$

```
(asset_1_reserves * asset_2_reserves) == (new_asset_1_reserves * new_asset_2_reserves)
```

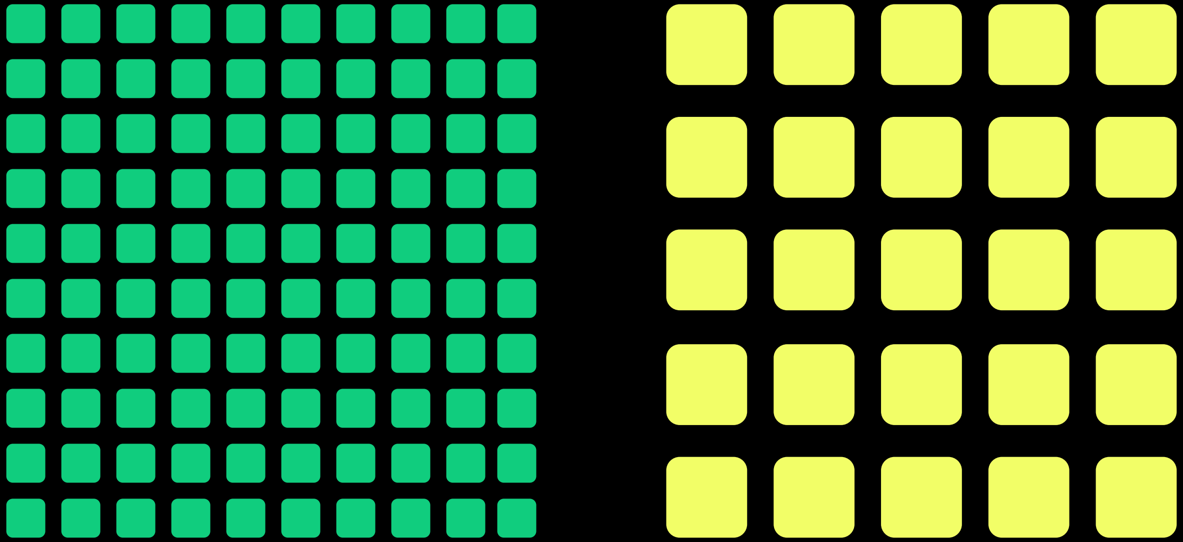
The product of the reserves of asset 1 and asset2 must remain constant

Swaps must not cause the pool to lose value

Pool

1A : 0.25B

Asset A | Asset B



100 | 25

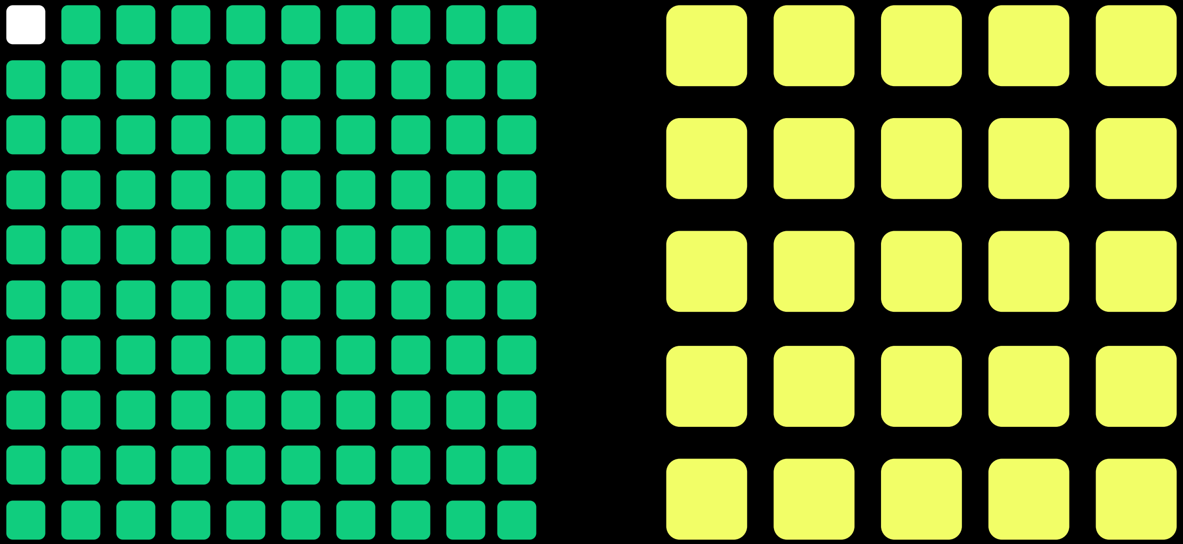
Product = 100 x 25 = 2500

» Pool

Buy 1A
Sell ?B

Asset A

Asset B



99

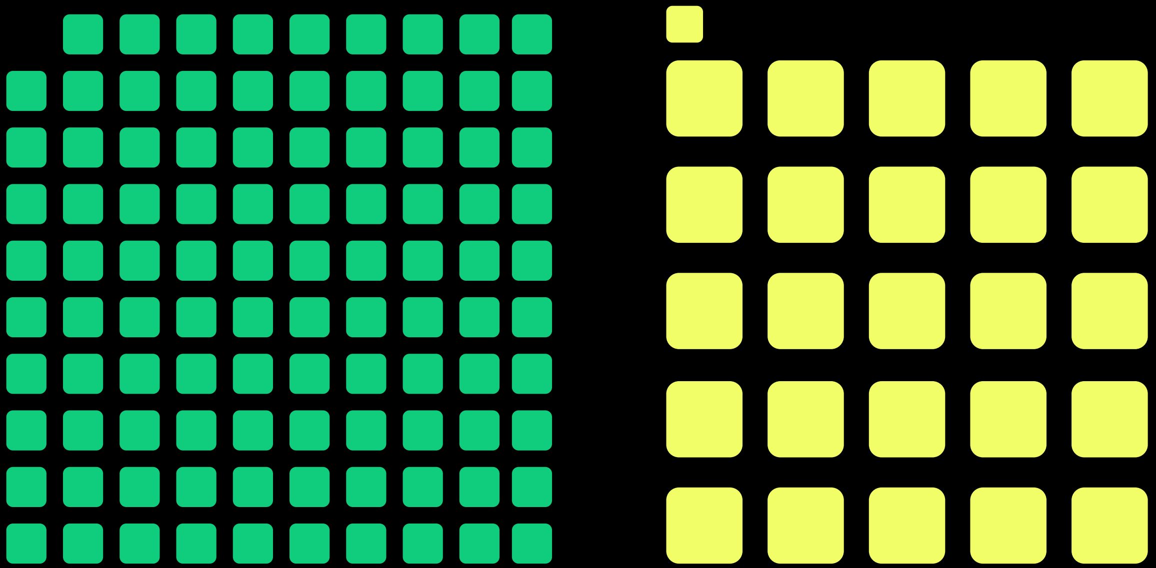
25

Product = 99 x 25 = 2475

» Pool

Buy 1A
Sell 0.25B
Rate: 1A:0.25B

Asset A　　Asset B

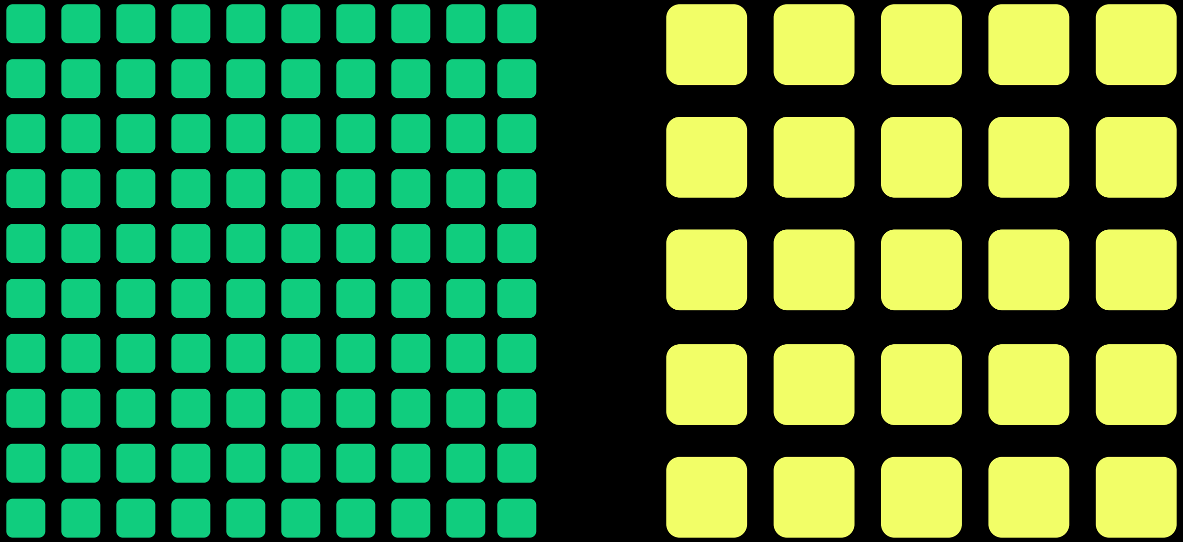


99　　25.25

Product = 99 x 25.25 = 2500

» Pool

Asset A **Asset B**



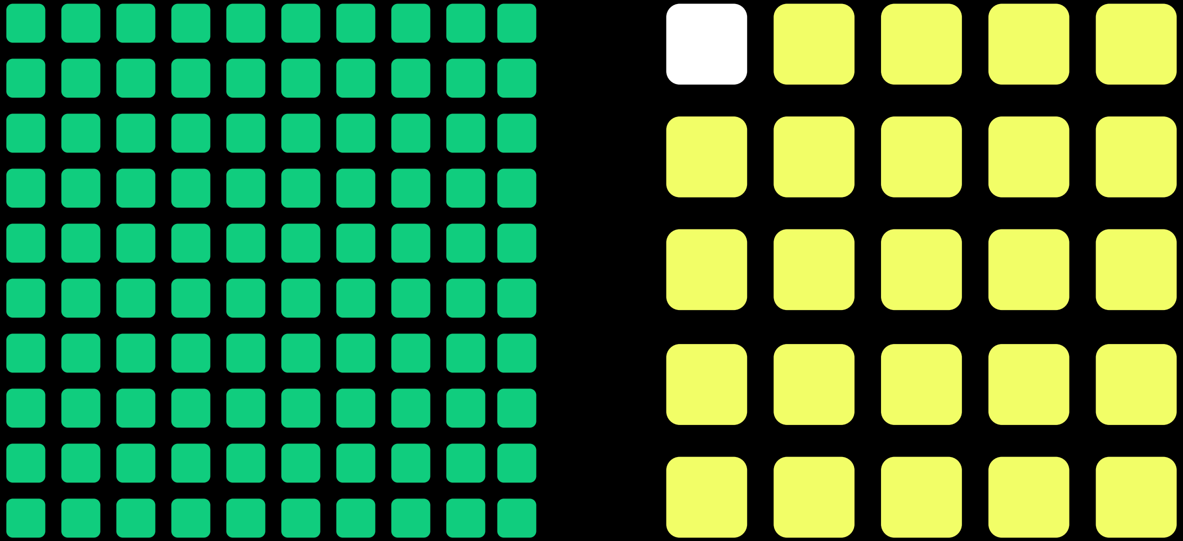
100 25

Product = 100 x 25 = 2500

» Pool

Buy 1B
Sell ?A

Asset A **Asset B**

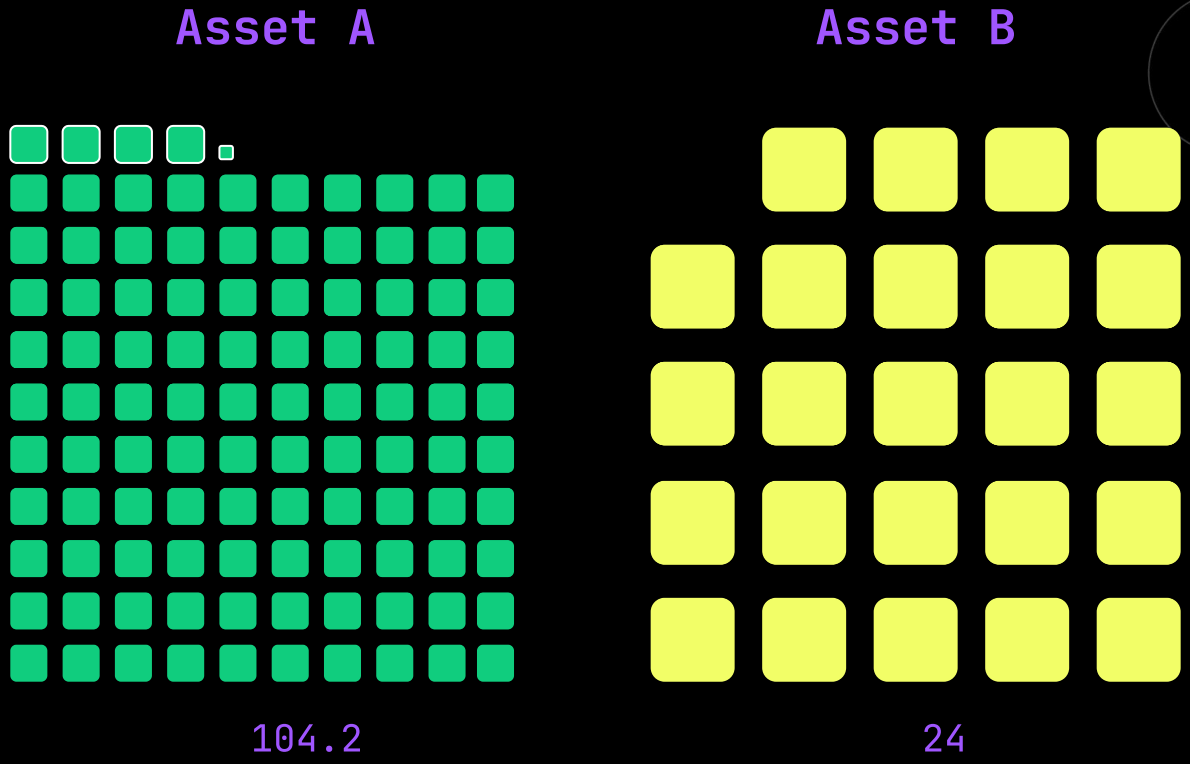


100 24

Product = 100 x 24 = 2400

» Pool

Buy 1B
Sell 4.2A
Rate: 1:0.23B

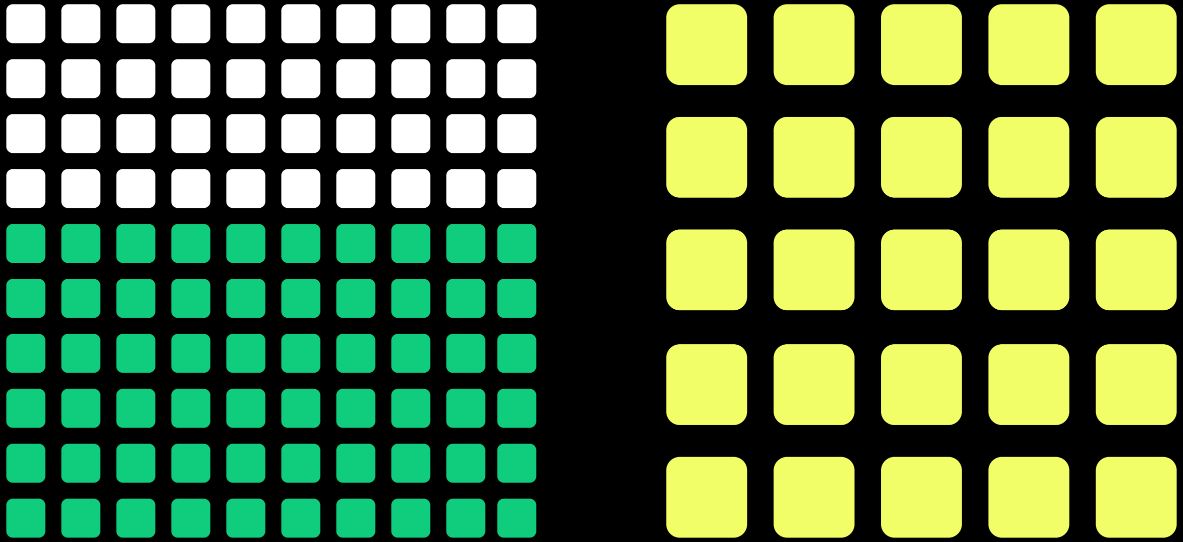


Product = 104.2 x 24 = 2500

» Pool

Asset A | Asset B

Buy 40A
Sell ?B



60 | 25

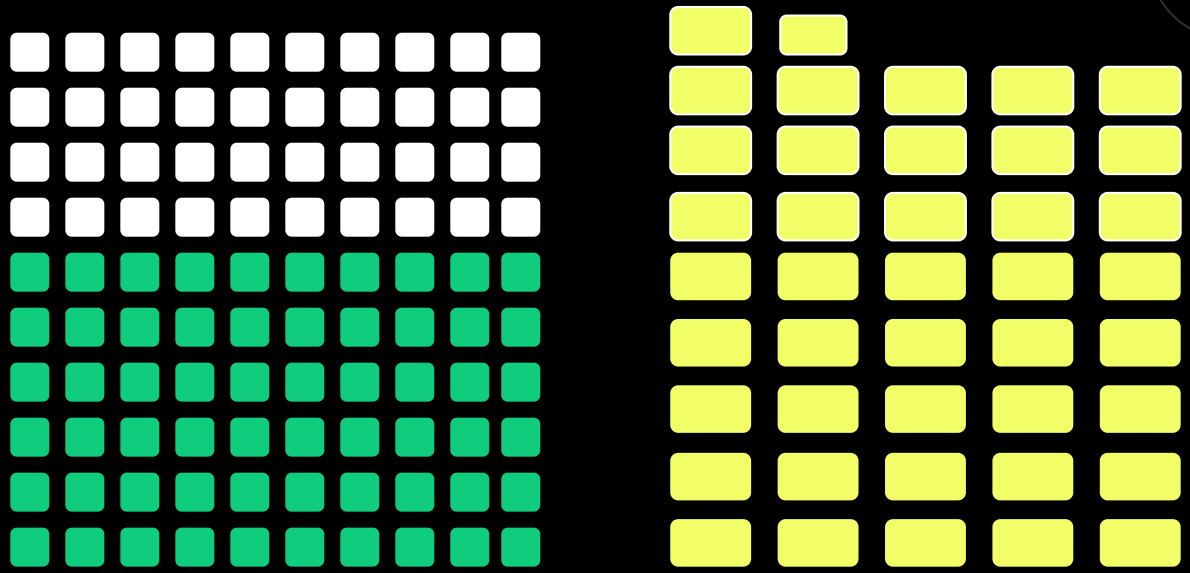
Product = 60 x 25 = 1500

Pool

Asset A

Asset B

Buy 40A
Sell 16.66B
Rate: 1B:2.5A



60

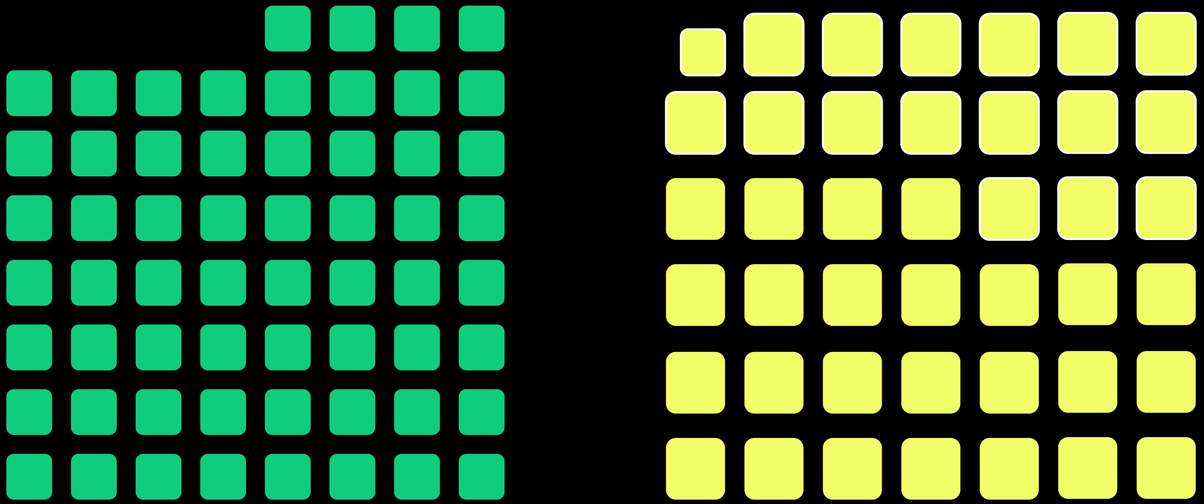
41.66

Product = 60 x 41.66 = 2500

Pool

Asset A | Asset B

Buy 40A
Sell 16.66B
Rate: 1B:2.5A

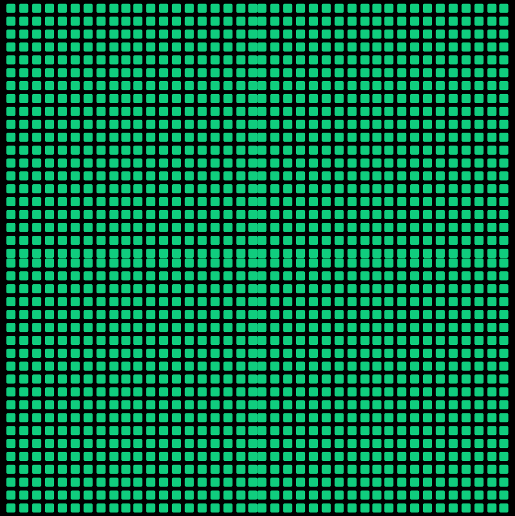


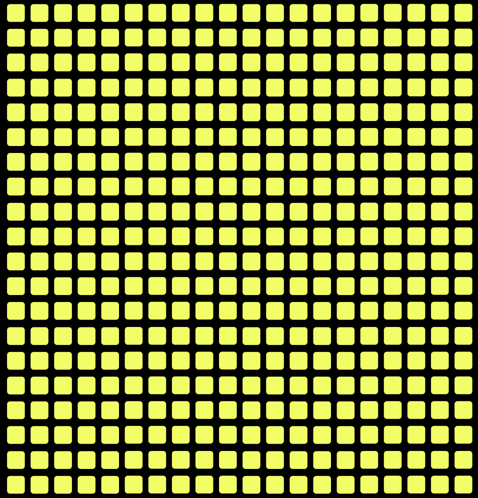
60 | 41.66

Product = 60 x 41.66 = 2500

» Pool

Asset A | **Asset B**

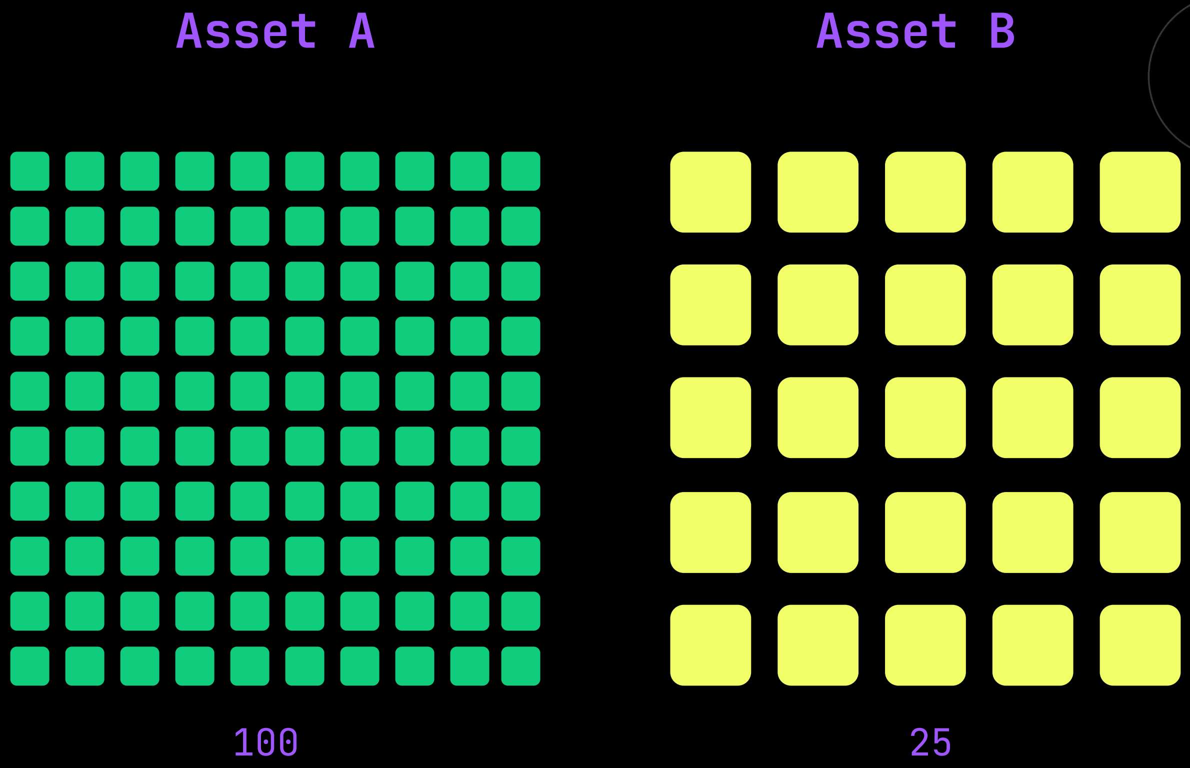


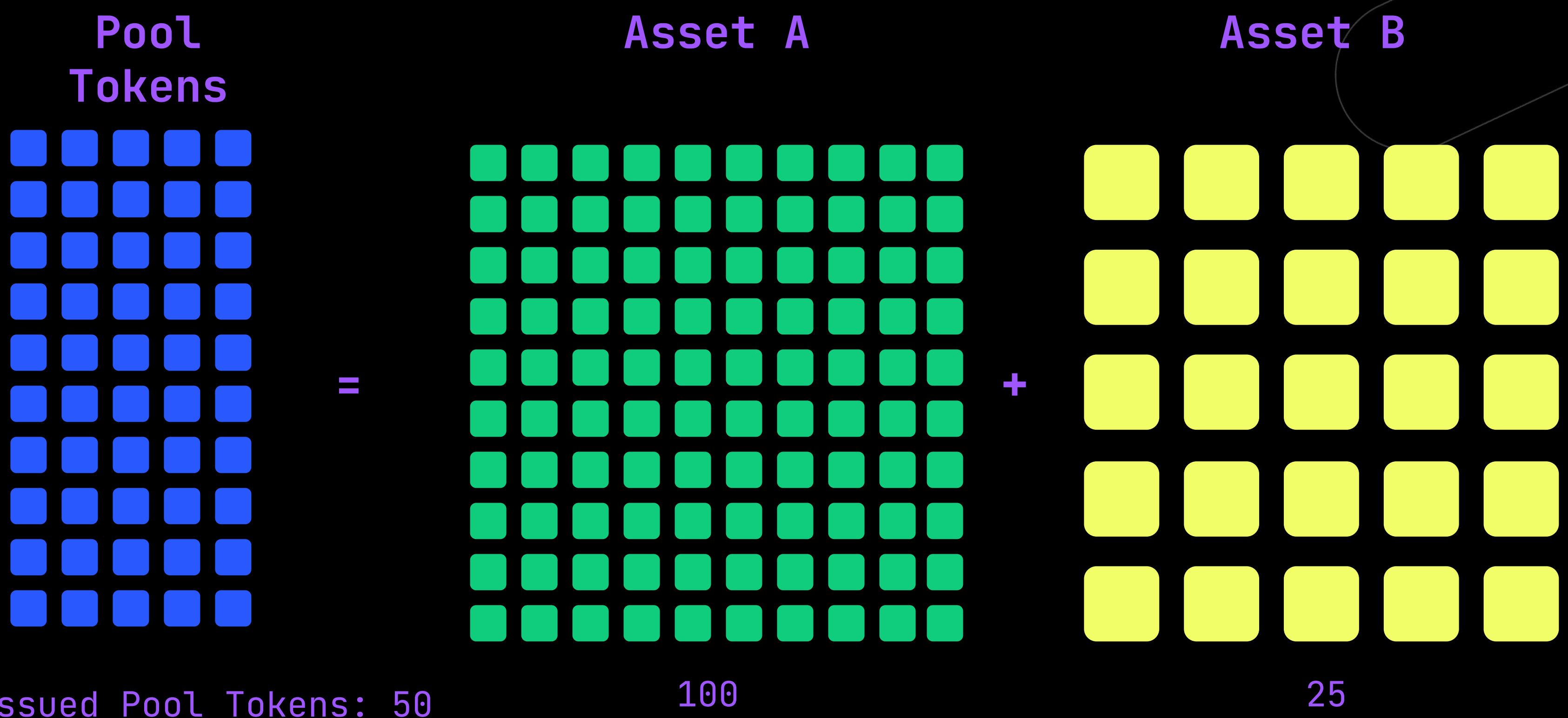


10000 | 2500

Product = 10000 x 2500 = 25000000

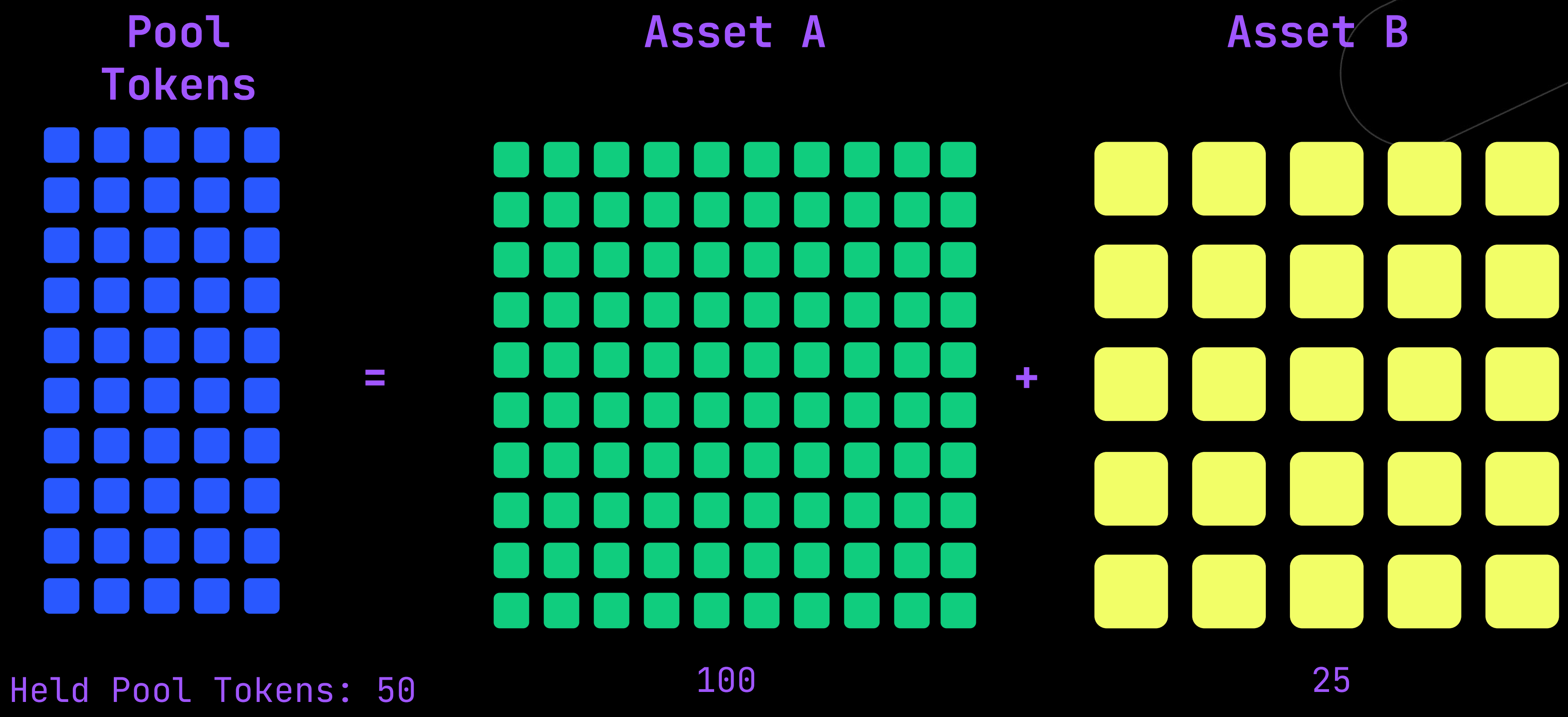
Poolers

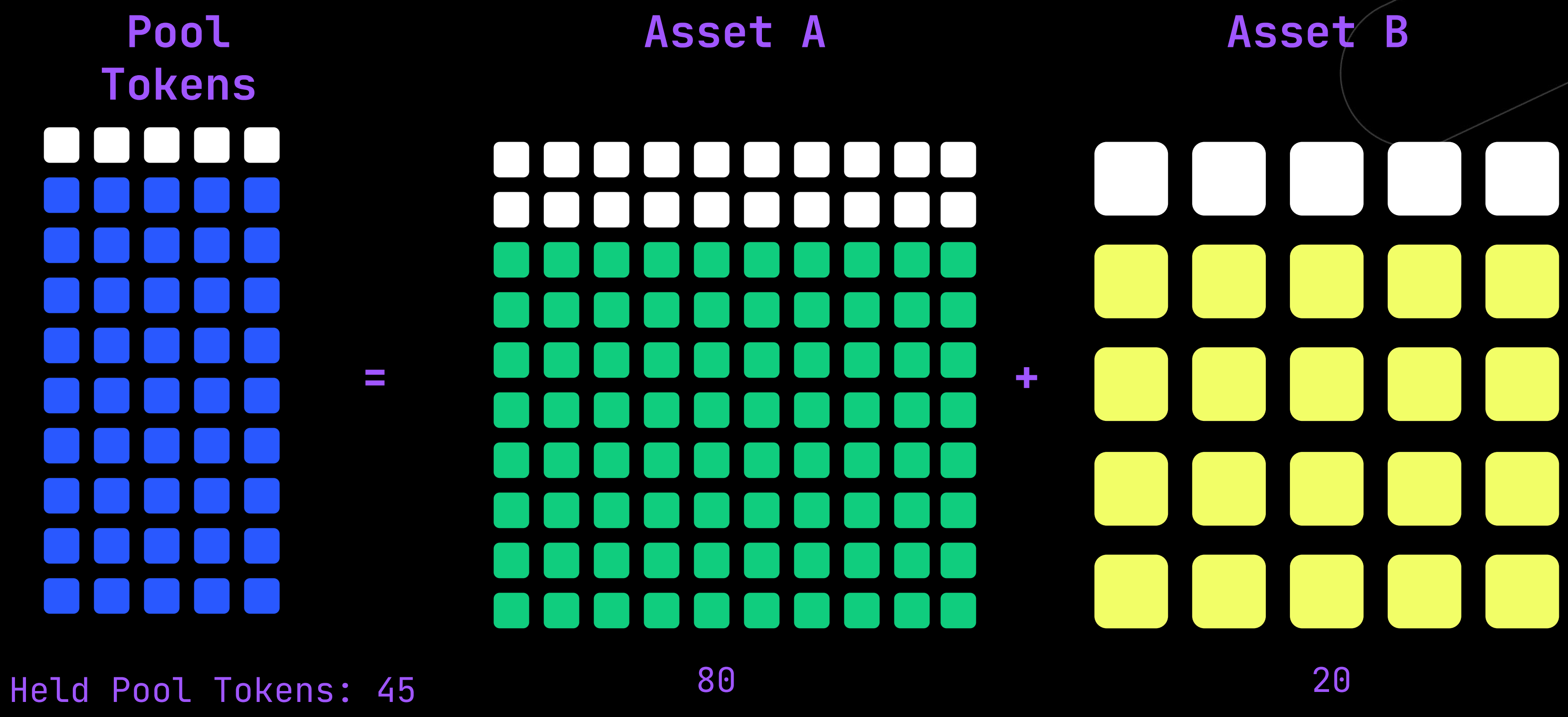

Asset A
Asset B
100
25



Issued Pool Tokens: 50
Held Pool Tokens: 50
Share of Pool: 100%

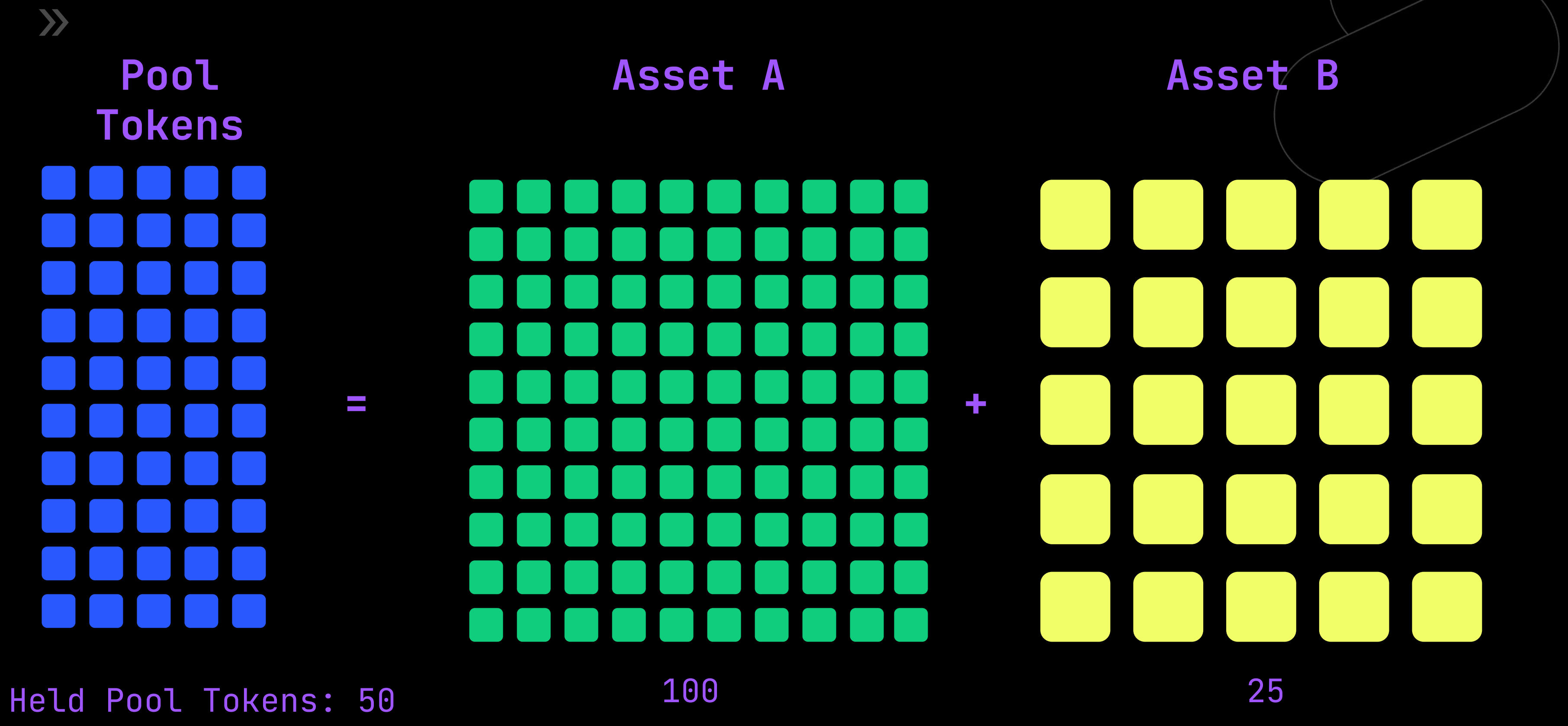
Issued Pool Tokens: 800
Held Pool Tokens: 50
Share of Pool: 6.25%

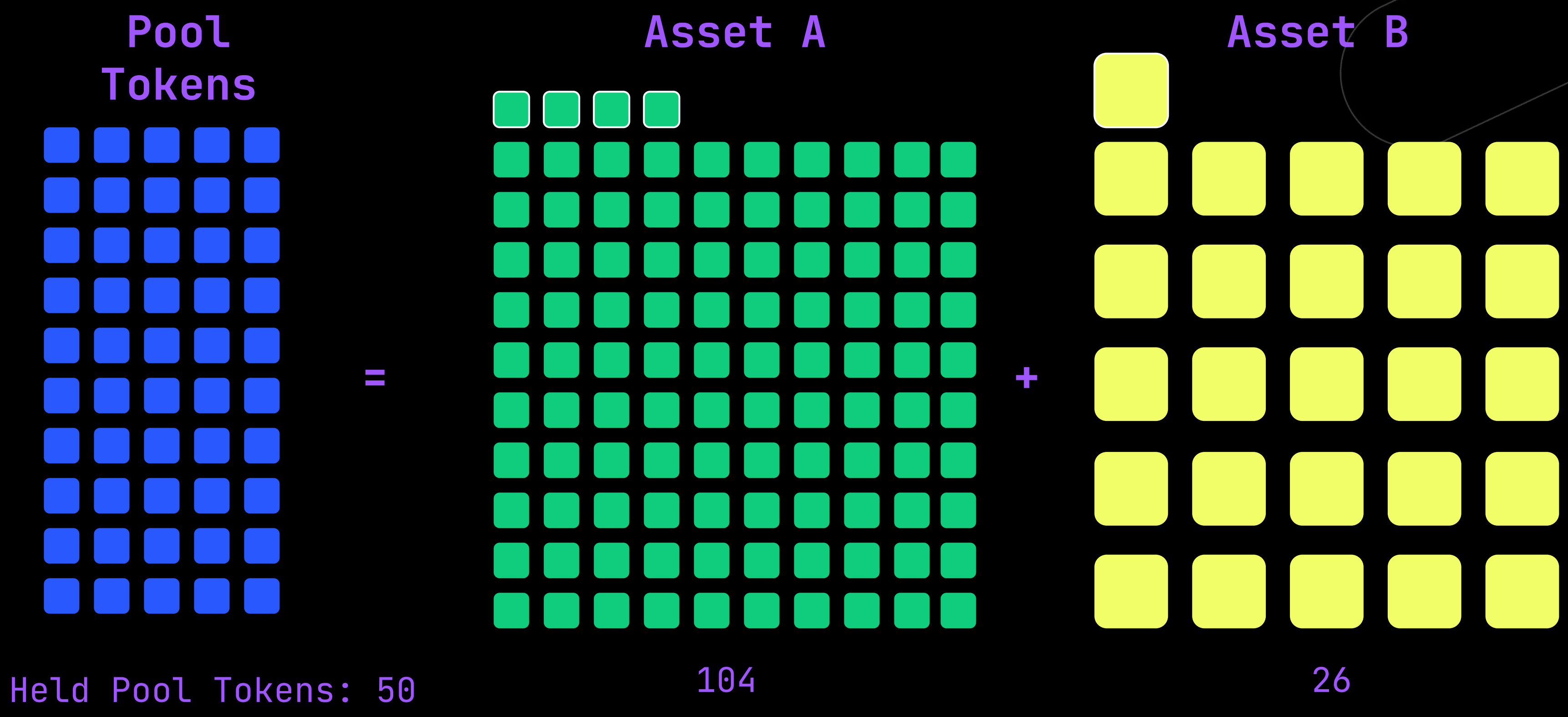

Pool Tokens
Asset A
Asset B
=
+
Held Pool Tokens: 50
100
25


Pool
Tokens
Asset A
Asset B
=
+
80
20
Held Pool Tokens: 45

Swap Fees

"By adding liquidity you'll earn 0.25% of all trades on this pair proportional to your share of the pool. Fees are added to the pool, accrue in real time and can be claimed by withdrawing your liquidity."

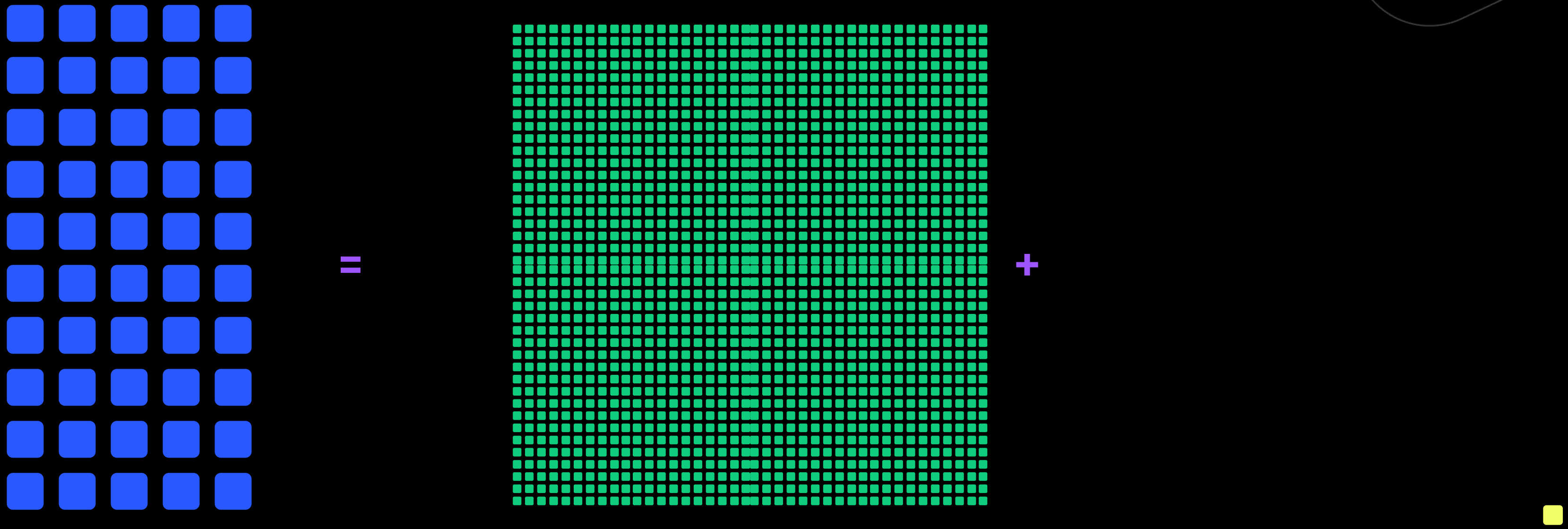

Pool Tokens
Asset A
Asset B
=
+
Held Pool Tokens: 50
100
25


Pool Tokens
Asset A
Asset B
=
+
Held Pool Tokens: 50
104
26

Pool Tokens

Asset A

Asset B



Held Pool Tokens: 50

10000

0.1

AMM on Algorand

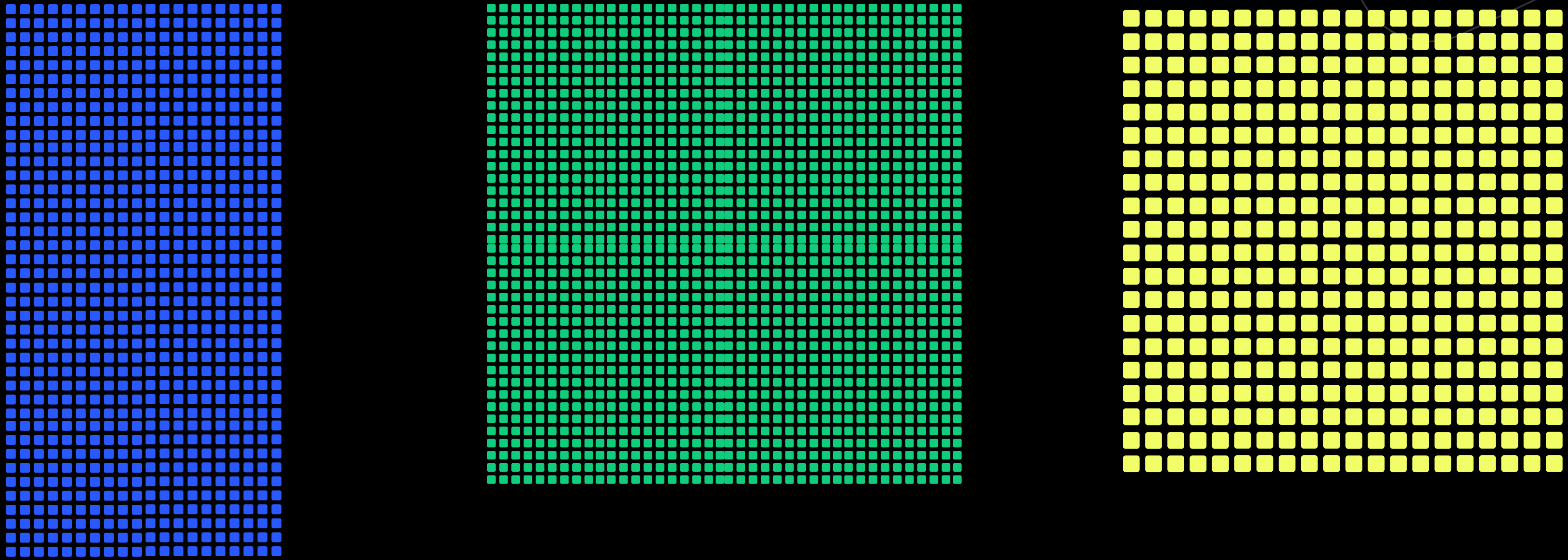
AMM on Algorand

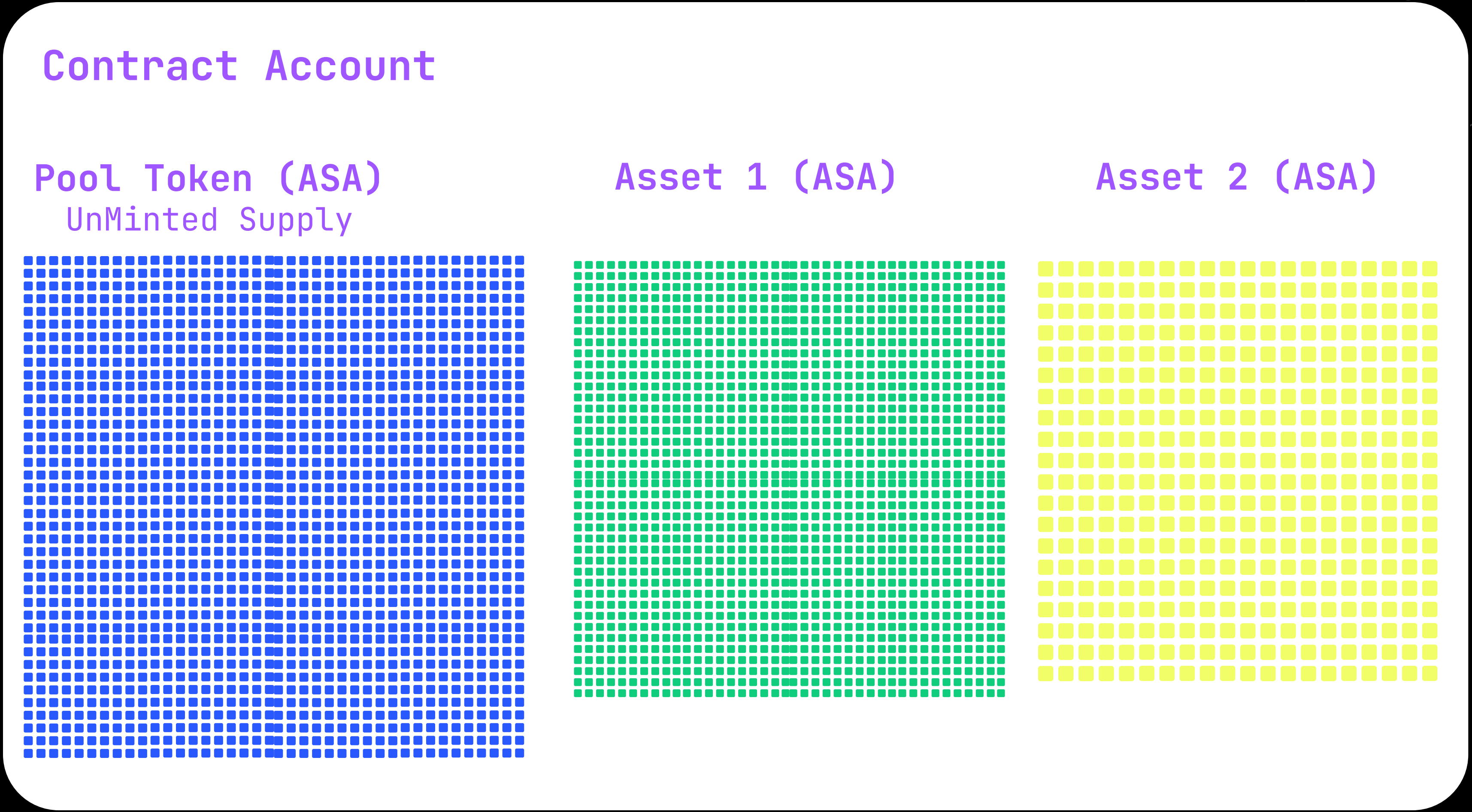
(April 2021, TEAL3)

Pool Tokens

Asset A

Asset B




Contract Account
Pool Token (ASA)
UnMinted Supply
Asset 1 (ASA)
Asset 2 (ASA)

» Contracts - Swap

	Transaction Type	Signer (Sender)	Receiver
0	Asset Transfer (Asset 1)	Swapper	Pool
1	Asset Transfer (Asset 2)	Pool (LogicSig)	Swapper

» Contracts - Swap

	Transaction Type	Signer (Sender)	Receiver
0	Asset Transfer (Asset 1)	Swapper	Pool
1	Asset Transfer (Asset 2)	Pool (LogicSig)	Swapper

Atomic Group

» Contract Responsibilities

- Enforce $x * y = k$
- Only allow minting of Pool Token in exchange for correct amount of assets 1 & 2
- Only allow withdrawal of Assets 1 & 2 in exchange for correct amount of Pool Token

» Pool Logic Signature

```
// enforce x * y = k

asset1_reserves = asset_holding_get(1, AssetBalance)

asset2_reserves = asset_holding_get(2, AssetBalance)

k = asset1_reserves * asset2_reserves

new_k = (asset1_reserves + gtxn[0].Amount) * (asset2_reserves - gtxn[1].Amount)

assert(new_k == k)
```

Pseudocode!

» Pool Logic Signature

Stateless contract! Cannot check asset balances

```
// enforce x * y = k

asset1_reserves = asset_holding_get(1, AssetBalance)

asset2_reserves = asset_holding_get(2, AssetBalance)

k = asset1_reserves * asset2_reserves

new_k = (asset1_reserves + gtxn[0].Amount) * (asset2_reserves - gtxn[1].Amount)

assert(new_k == k)
```

Pseudocode!

» Contracts - Swap

	Transaction Type	Signer (Sender)	Receiver
0	Asset Transfer (Asset 1)	Swapper	Pool
1	Asset Transfer (Asset 2)	Pool (LogicSig)	Swapper

Stateless! Cannot check asset balances

» Contracts - Swap

	Transaction Type	Signer (Sender)	Receiver
0	Application Call (“swap”)	Pool (LogicSig)	
1	Asset Transfer (Asset 1)	Swapper	Pool
2	Asset Transfer (Asset 2)	Pool (LogicSig)	Swapper

Stateful app

» Contracts - Swap

	Transaction Type	Signer (Sender)	Receiver	Transaction Fee
0	Application Call (“swap”)	Pool (LogicSig)		0.001
1	Asset Transfer (Asset 1)	Swapper	Pool	0.001
2	Asset Transfer (Asset 2)	Pool (LogicSig)	Swapper	0.001

» Contracts - Swap

	Transaction Type	Signer (Sender)	Receiver	Transaction Fee
0	Application Call ("swap")	Pool (LogicSig)		0.001
1	Asset Transfer (Asset 1)	Swapper	Pool	0.001
2	Asset Transfer (Asset 2)	Pool (LogicSig)	Swapper	0.001

Who Pays?

» Contracts - Swap

	Transaction Type	Signer (Sender)	Receiver
0	Payment	Swapper	Pool
1	Application Call (“swap”)	Pool (LogicSig)	
2	Asset Transfer (Asset 1)	Swapper	Pool
3	Asset Transfer (Asset 2)	Pool (LogicSig)	Swapper

» Pool Logic Signature

```
// ensure gtxn 1 is ApplicationCall to Validator App

assert(gtxn[1].Sender == txn.Sender)

assert(gtxn[1].TypeEnum == ApplicationCall)

assert(gtxn[1].ApplicationID == VALIDATOR_APP_ID)


// ensure gtxn 0 amount covers all fees

// ensure Pool is not paying the fee

assert(gtxn 0 Sender != txn Sender)

// ensure Pool is receiving the fee

assert(gtxn[0].Receiver == txn Sender)

// ensure fee amount is sufficient

assert(gtxn[0].Amount >= fee_total)
```

Pseudocode!

» Contracts - Mint (Add Liquidity)

	Transaction Type	Signer (Sender)	Receiver
0	Payment	Pooler	Pool
1	Application Call (“mint”)	Pool (LogicSig)	
2	Asset Transfer (Asset 1)	Pooler	Pool
3	Asset Transfer (Asset 2)	Pooler	Pool
4	Asset Transfer (Pool Token)	Pool (LogicSig)	Pooler

» Contracts - Bootstrap

	Transaction Type	Signer (Sender)	Receiver
0	Payment	User	Pool
1	Application Call (“bootstrap”)	Pool (LogicSig)	
2	Asset Creation	Pool (LogicSig)	
3	Asset Optin	Pool (LogicSig)	
4	Asset Optin	Pool (LogicSig)	

» Pool Logic Signature

```
if (gtxna[1].ApplicationArgs[0] == “bootstrap”):

      bootstrap:

      // ensure correct asset ids are included as args to the bootstrap call

      assert(gtxna[1].ApplicationArgs[1] == TMPL_ASSET_ID_1)

      assert(gtxna[1].ApplicationArgs[2] == TMPL_ASSET_ID_2)
```

TMPL_* variables are replaced in bytecode dynamically by clients when generating LogicSig for specific asset pair. This makes each LogicSig contract deterministically unique.

Unique contract → unique contract account address

Pseudocode!

» Contracts - Burn (Remove Liquidity)

	Transaction Type	Sender (Signer)	Receiver
0	Payment	Pooler	Pool
1	Application Call (“burn”)	Pool (LogicSig)	
2	Asset Transfer (Asset 1)	Pool (LogicSig)	Pooler
3	Asset Transfer (Asset 2)	Pool (LogicSig)	Pooler
4	Asset Transfer (Pool Token)	Pooler	Pool

» Contracts - Common Structure

	Transaction Type	Sender (Signer)	Receiver
0	Payment	*User*	Pool
1	Application Call (“*operation*”)	Pool (LogicSig)	
2			
3			
4			

» Swapping

```
// so simple :)

k = asset1_reserves * asset2_reserves

new_k = (asset1_reserves + sell_amount) * (asset2_reserves - buy_amount)

assert(new_k == k)
```

Pseudocode!

This would work perfectly

This would work perfectly

... in demos on Testnet!

This would cause huge frustration on Mainnet
with concurrent users!

This would cause huge frustration on Mainnet

with concurrent users!

Why?

Slippage

» Swapping - Slippage

1. Client looks up pool reserves from account and generates quote
2. User accepts quote and signs transactions
3. Client submits transactions
4. Transactions get executed by Algorand Node
5. User receives confirmation

» Swapping - Slippage

1. Client looks up pool reserves from account and generates quote
2. User accepts quote and signs transactions
3. Client submits transactions
4. Transactions get executed by Algorand Node
5. User receives confirmation

Between steps 1 & 4 multiple rounds may pass and the pool reserves may change due to other swaps/mints/burns, thus invalidating the quote.

Even within the same round multiple users may try to swap with the same pool.

» Swapping - Slippage

1. Client looks up pool reserves from account and generates quote
2. User accepts quote and signs transactions
3. Client submits transactions
4. Transactions get executed by Algorand Node
5. User receives confirmation

Between steps 1 & 4 multiple rounds may pass and the pool reserves may change due to other swaps/mints/burns, thus invalidating the quote.

Even within the same round multiple users may try to swap with the same pool.

The exchange rate has now **slipped** but the system has no tolerance for slippage because the user has signed transactions expecting a precise amount in return for their input.

A contract can only approve or deny transactions, not create or modify transactions!

Slippage Tolerance

» Swapping - Slippage Tolerance

Swapper: “I want to buy *100A* for *25B* but I’m will to accept *99A* at minimum. But I’d really prefer *100A*.”

i.e 1% slippage tolerance

1. Swapper signs transactions for the minimum amount they are willing to receive.
2. The contract calculates the the expected output amount at the time of execution.
3. The contract stores the difference as *excess* in local state (i.e. change/IOU).
4. The user *redeems* their *excess* from the pool after the swap completes.

» Swapping with Tolerance

```
k = asset1_reserves * asset2_reserves

asset_in_amount = gtxn[2].Amount

asset_out_amount = gtxn[3].Amount

calculated_amount_out = asset2_reserves - (k / (asset1_reserves + asset_in_amount))

// Calculate excess amount

excess = calculated_amount_out - asset_out_amount

excess_asset_2_amount += excess_asset_out

// Store excess amount in swapper's local state

app_local_put(1, "excess_2", excess_asset_2_amount)

put outstanding_asset_out_amount += excess_asset_out

// Store outstanding amount in pool's local state

app_local_put(0, "outstanding_2", excess_asset_2_amount)
```

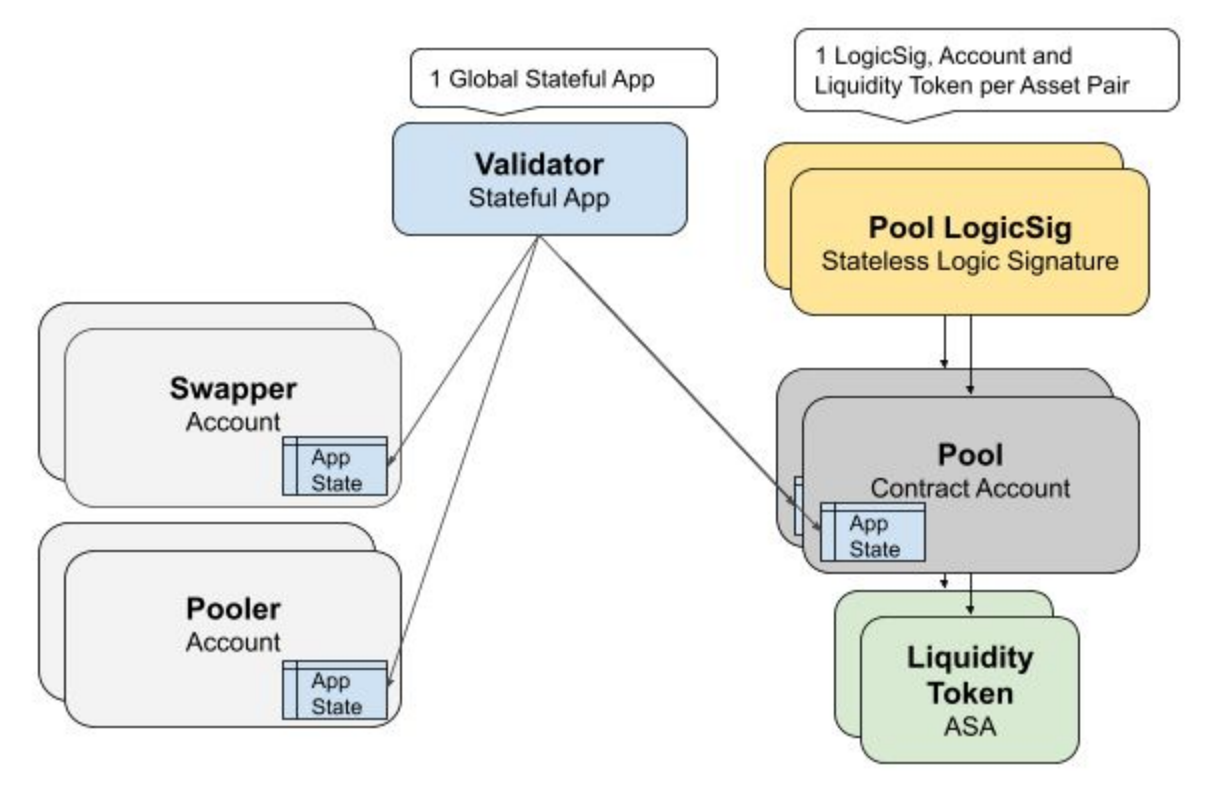
Pseudocode!

» Contracts - Operations

Redeem

	Transaction Type	Sender (Signer)	Receiver
0	Payment	Swapper	Pool
1	Application Call (“redeem”)	Pool (LogicSig)	
2	Asset Transfer	Pool (LogicSig)	Swapper

» Contracts - Architecture



» Immutable Code

No updates

No deletion

```
// Deny Update, Delete, CloseOut

txn OnCompletion
int UpdateApplication
==

txn OnCompletion
int DeleteApplication
==

||

txn OnCompletion
int CloseOut
==

||

bnz fail

fail:
  int 0
  return
```

TEAL

» Permissionless

Pool Token:

No clawback

No freeze

```
// ensure no asset freeze address is set
gtxn 2 ConfigAssetFreeze
global ZeroAddress
==
assert

// ensure no asset clawback address is set
gtxn 2 ConfigAssetClawback
global ZeroAddress
==
assert
```

TEAL

https://runtimeverification.com/blog/runtime-verification-audits-tinyman

Audit

Runtime Verification Audits Tinyman

Posted on September 22, 2021 by Silvia Barredo
Posted in Audits

Runtime Verification is thrilled to announce Tinyman's audit completion. Tinyman is a new project under development that focuses on bringing a decentralized trading protocol to the Algorand ecosystem and its community.

Tinyman's Audit Scope

Before Tinyman's launch, their team has decided to audit the project's code with Runtime Verification to identify any issues that could cause the system to malfunction or be exploited.

Tinyman's protocol is built with Algorand's Layer-1 smart contract language TEAL and aims to serve as a decentralized trading platform, similar to the Uniswap protocol, for the Algorand community. The protocol is built using two core smart contracts:

- The Pool Logic Signature Template (a stateless TEAL smart contract template), from which contract accounts representing liquidity pools are created.
- The Validator Application (a stateful TEAL smart contract), which implements the protocol's logic and maintains the state of the system on the blockchain. The contract was created by the core team and is immutable.

Runtime Verification performed an audit on the two contracts just mentioned above and the system's high-level documentation. The focus was reviewing the high-level business logic (protocol design) of Tinyman's system based on the provided documentation and reviewing the low-level implementation of the system in TEAL. In addition, the audit highlighted some informative findings that could improve the performance and efficiency of the implementation and optimize its code size.

Methodology

Runtime Verification team lead Musab Alturki conducted Tinyman's audit and published a detailed report on August 4th, 2021.

The first step consisted of rigorously reasoning about the business logic of the contract and validating security-critical properties to ensure the absence of loopholes in the logic. We also reviewed past public audit reports of Uniswap v1 and v2 and checked if the list of known issues could be applied to Tinyman, and in case they did, to check the code to make sure there is no space for any vulnerability. This step was crucial to conduct as Tinyman's system is based on Uniswap's AMM (Automated Market Maker) design.

The second step consisted of reviewing the contract source code to detect any unexpected (and possibly exploitable) behaviors. We applied an approach that consisted of constructing different high-level representations of the TEAL codebase to systematically check consistency between the logic and the low-level TEAL implementation.

The third step consisted of reviewing the TEAL guidelines published by Algorand to check for known issues and reviewing a list of known Ethereum security vulnerabilities and attack vectors to check whether they apply to TEAL smart contracts and, if they do, to check whether the code is vulnerable to them.

Results

The audit identified and highlighted some critical issues along with a number of informative findings (see the report for details). The Tinyman team properly addressed all the issues and concerns raised during the audit, and incorporated all the necessary changes in the smart contracts.

Finally, in a concluding phase of the audit, we further reviewed the security impact of the changes made based on the issues raised in the first phase and investigated their effects on other parts of the contracts to ensure that no new issues or vulnerabilities were introduced in the process.

We enjoyed working with the Tinyman team and wish them the best of luck with their project.

About Tinyman

Tinyman is a re-imagined decentralized trading protocol which utilizes the fast and secure framework of the Algorand blockchain, creating an open and safe marketplace for traders, liquidity providers, and developers.

About Runtime Verification

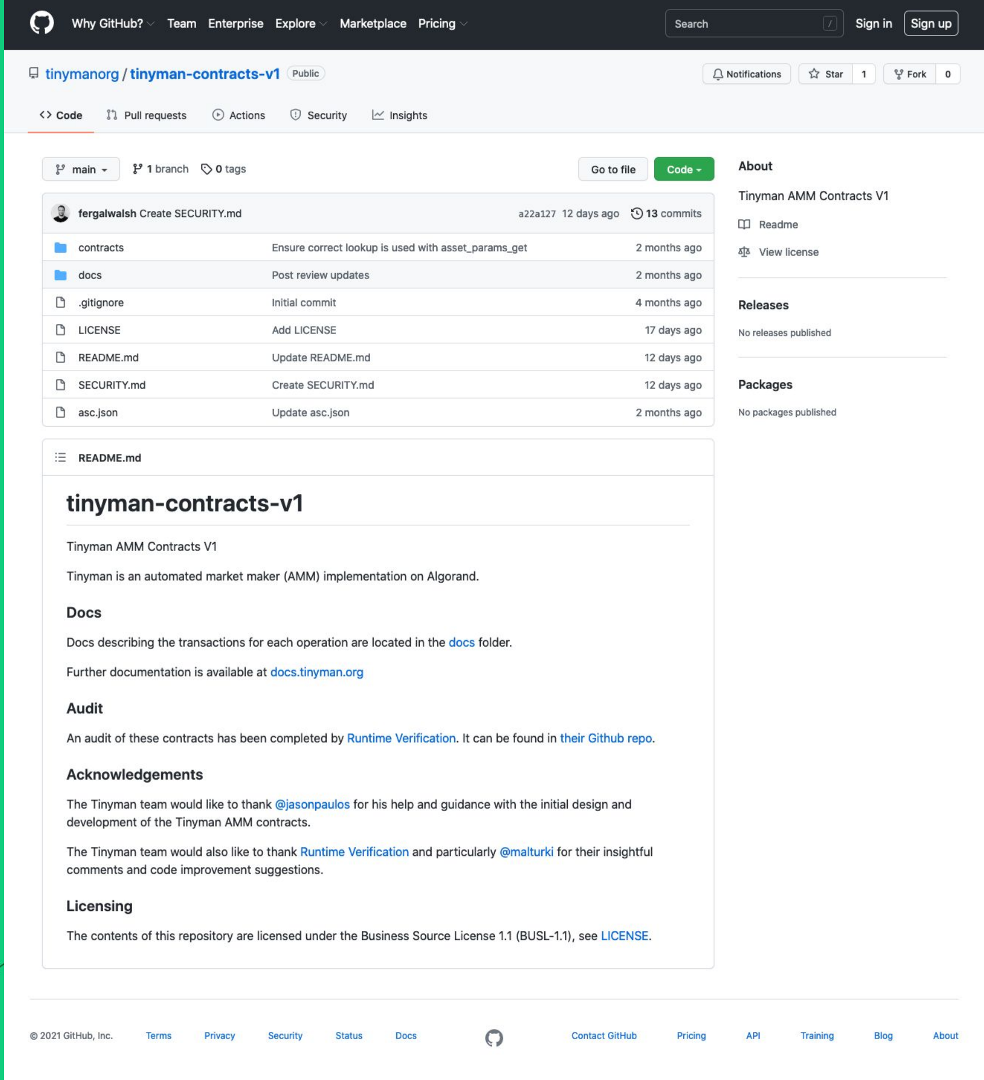
Runtime Verification is a technology startup based in Champaign-Urbana, Illinois. The company uses formal methods to perform security audits on virtual machines and smart contracts on public blockchains. It also provides software testing, verification services and products to improve the safety, reliability, and correctness of software systems in the blockchain field.

← From 0 to K Tutorial

Fully commented code is public on Github:

https://github.com/tinymanorg/tinyman-contracts-v1

Open Source Contracts



» Contracts - Testing

Integration Testing with Sandbox & Mocha

Go Tests

» Contracts - AVM 1.0+ / TEAL5+

Future design possibilities:

- Stateful Contract Account
- Inline slippage tolerance support with inner transactions
- Code reuse with functions

» Analytics - Web UI

» Analytics - Backend

Architecture

» Analytics - Backend

Block Processing

» Analytics - Backend

Asset Price Calculations

» SDKs

Official SDKs created by the development team:

Python - https://github.com/tinymanorg/tinyman-py-sdk

Javascript/Typescript - In use by web app. Will be published in coming months.

Community created SDKs:

.NET - https://github.com/geoffodonnell/dotnet-tinyman-sdk.
Thanks @geoffodonnell!

» Tinyman Oracle

Spot Price

TWAP - Time Weighted Average Price

Available from all pools. Can be read by other contracts to get asset price data on chain.

Roadmap

Public Testnet ✓

Mainnet Launch

Analytics Improvements

Web UI improvements

AMM V2

Gradual continuous decentralisation

Gradual continuous open-sourcing

» Thanks

We’d like to thank all those who supported Tinyman on Testnet and provided valuable feedback.

Thanks to the community members for asking questions.

Thanks to those community members who help out others by answering questions.

tinyman

tinyman.org

docs.tinyman.org

Twitter: @tinymanorg

Telegram: tinymanofficial

Github: github.com/tinymanorg/