Decentralized Exchange & Lending

Ali Yahya & Eddy Lazzarin

a16z

Important Disclosures

The views expressed here are those of the individual AH Capital Management, L.L.C. (“a16z”) personnel quoted and are not the views of a16z or its affiliates. Certain information contained in here has been obtained from third-party sources, including from portfolio companies of funds managed by a16z. While taken from sources believed to be reliable, a16z has not independently verified such information and makes no representations about the enduring accuracy of the information or its appropriateness for a given situation.

This content is provided for informational purposes only, and should not be relied upon as legal, business, investment, or tax advice. You should consult your own advisers as to those matters. References to any securities, digital assets, tokens, and/or cryptocurrencies are for illustrative purposes only and do not constitute a recommendation to invest in any such instrument nor do such references constitute an offer to provide investment advisory services. Furthermore, this content is not directed at nor intended for use by any investors or prospective investors, and may not under any circumstances be relied upon when making a decision to invest in any fund managed by a16z. (An offering to invest in an a16z fund will be made only by the private placement memorandum, subscription agreement, and other relevant documentation of any such fund and should be read in their entirety.) Any investments or portfolio companies mentioned, referred to, or described are not representative of all investments in vehicles managed by a16z, and there can be no assurance that the investments will be profitable or that other investments made in the future will have similar characteristics or results. A list of investments made by funds managed by Andreessen Horowitz (excluding investments for which the issuer has not provided permission for a16z to disclose publicly as well as unannounced investments in publicly traded digital assets) is available at https://a16z.com/investments/.

Charts and graphs provided within are for informational purposes solely and should not be relied upon when making any investment decision. Past performance is not indicative of future results. The content speaks only as of the date indicated. Any projections, estimates, forecasts, targets, prospects, and/or opinions expressed in these materials are subject to change without notice and may differ or be contrary to opinions expressed by others. Please see https://a16z.com/disclosures for additional important information.

Decentralized Exchanges (DEXs)

Ali Yahya

DEXs: Why do they matter?

- Enable Protocol Composability

<rant>
</rant>

- Non-Custodial
- Have Global Reach

Traditional Exchanges: Order Book Model

Order Book Based DEXs

Naive Approach:

- Implement an order book and matching engine on-chain.

Mostly unworkable with today's blockchains.

Order Book Based DEXs

Hybrid Approach — The Relayer Model

- Matching is done **off-chain** by a centralized "Relayer"
  - The relayer crafts a transaction off-chain that resembles an atomic-swap, then submits it to the blockchain
- Trade settlement is done on-chain

Many examples of DEXs that initially worked this way:

- 0x protocol
- EtherDelta
- Kyber
- Airswap

Order Book Based DEXs

Limitations of the Relayer Model

- Peer-to-peer —hard to bootstrap liquidity
- Market making is expensive
- Depends on the presence of a centralized party
- Less programmable/composable

Great resource:
Front-Running, Griefing, and the Perils of Virtual Settlement,
by Will Warren

Is there a simpler way to build a DEX?

Two most important problems

- Complexity
- Bootstrapping liquidity

Desired Characteristics

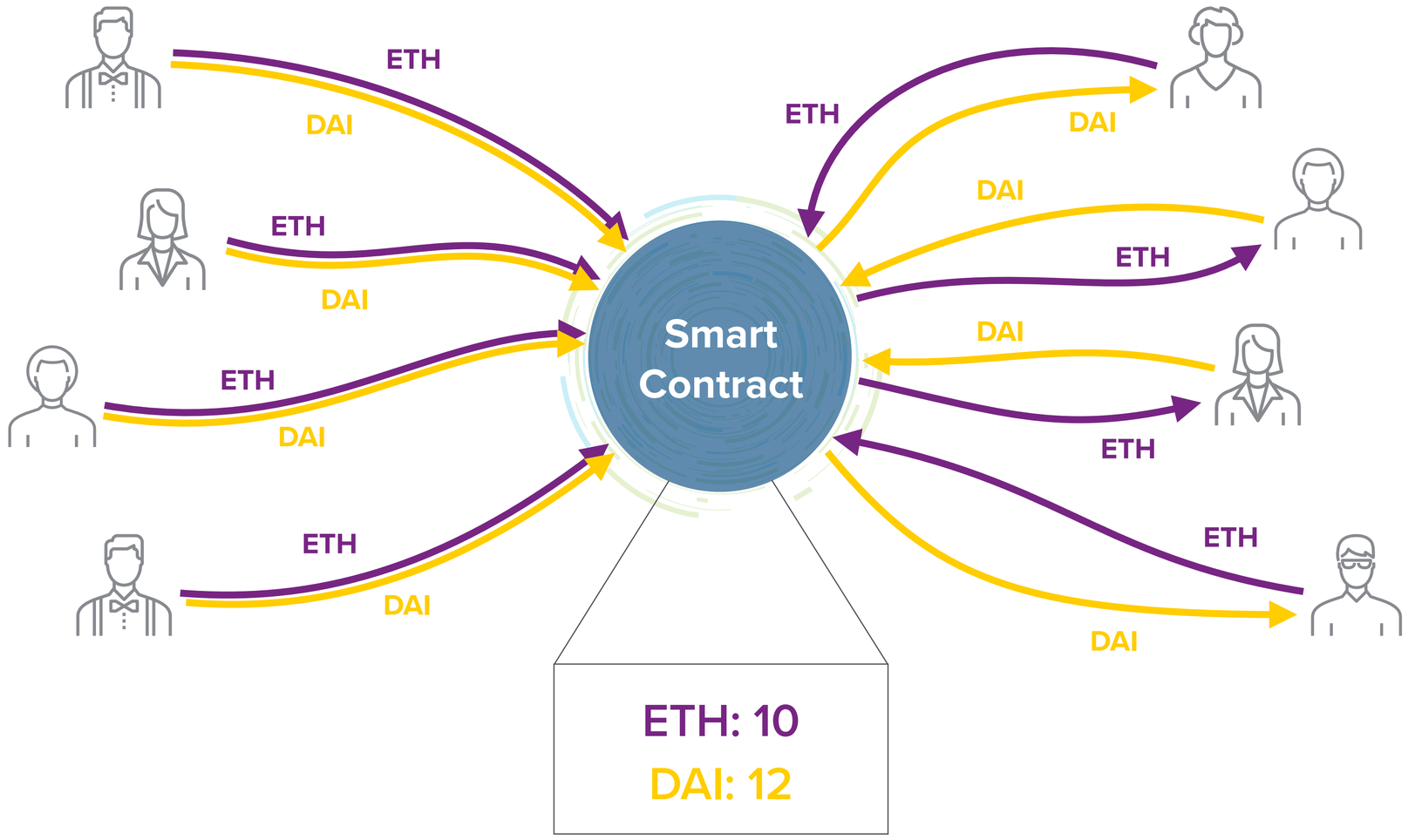
- Simple — buildable as a smart contract
- Automated liquidity — no dependence on active market-makers
- No single points of control — no dependence on centralized parties

A Bit of History: Automated Market Makers (AMMs)

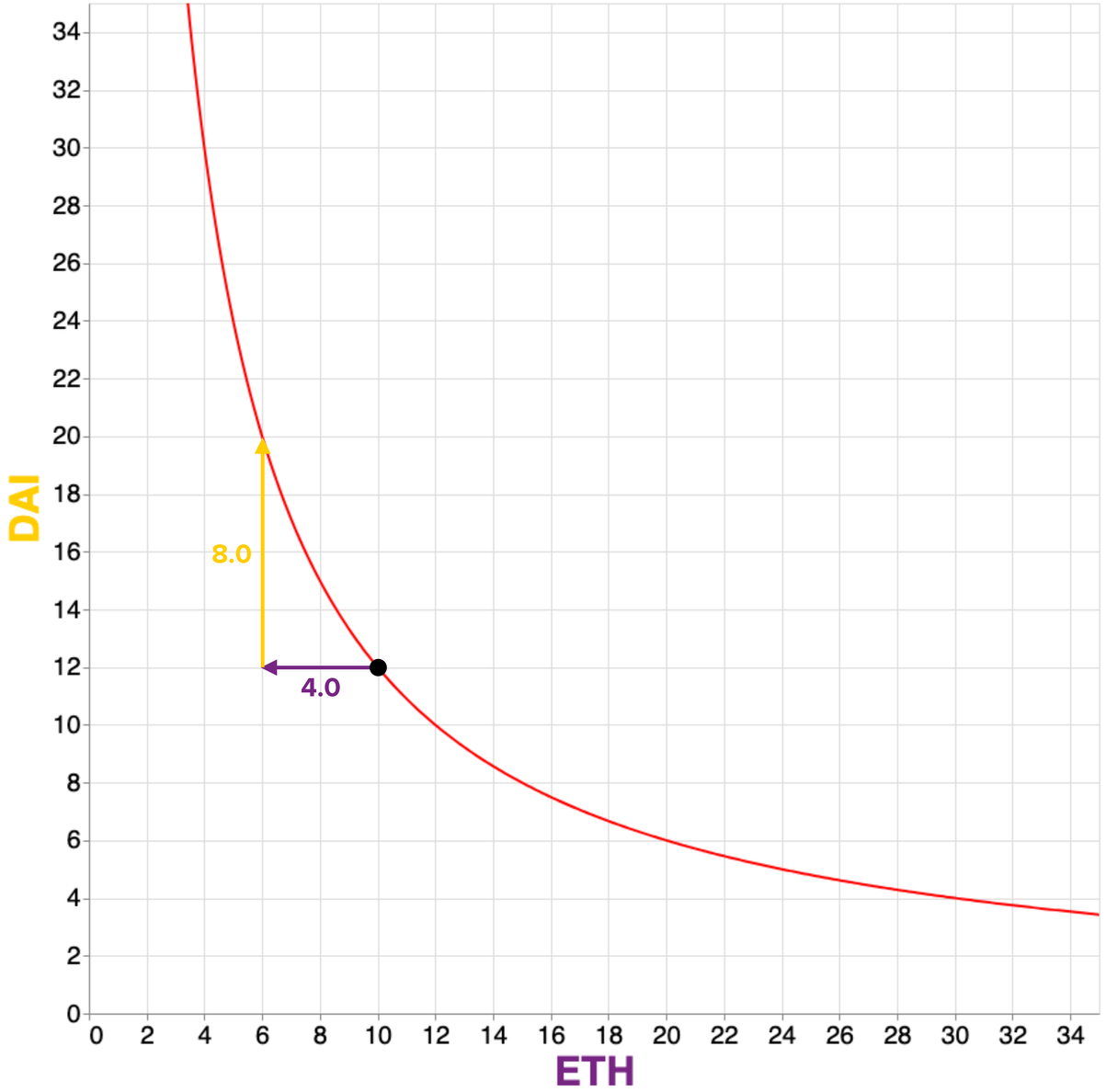
- Pricing shares in prediction markets — Hanson's Market Scoring Rules
  - Also used to price online ads
- Idea first explored in crypto in 2016 by:
  - Vitalik Buterin — reddit post
- Then generalized by Alan Lu and Martin Koppelman:
  - Blogpost: Building a Decentralized Exchange in Ethereum

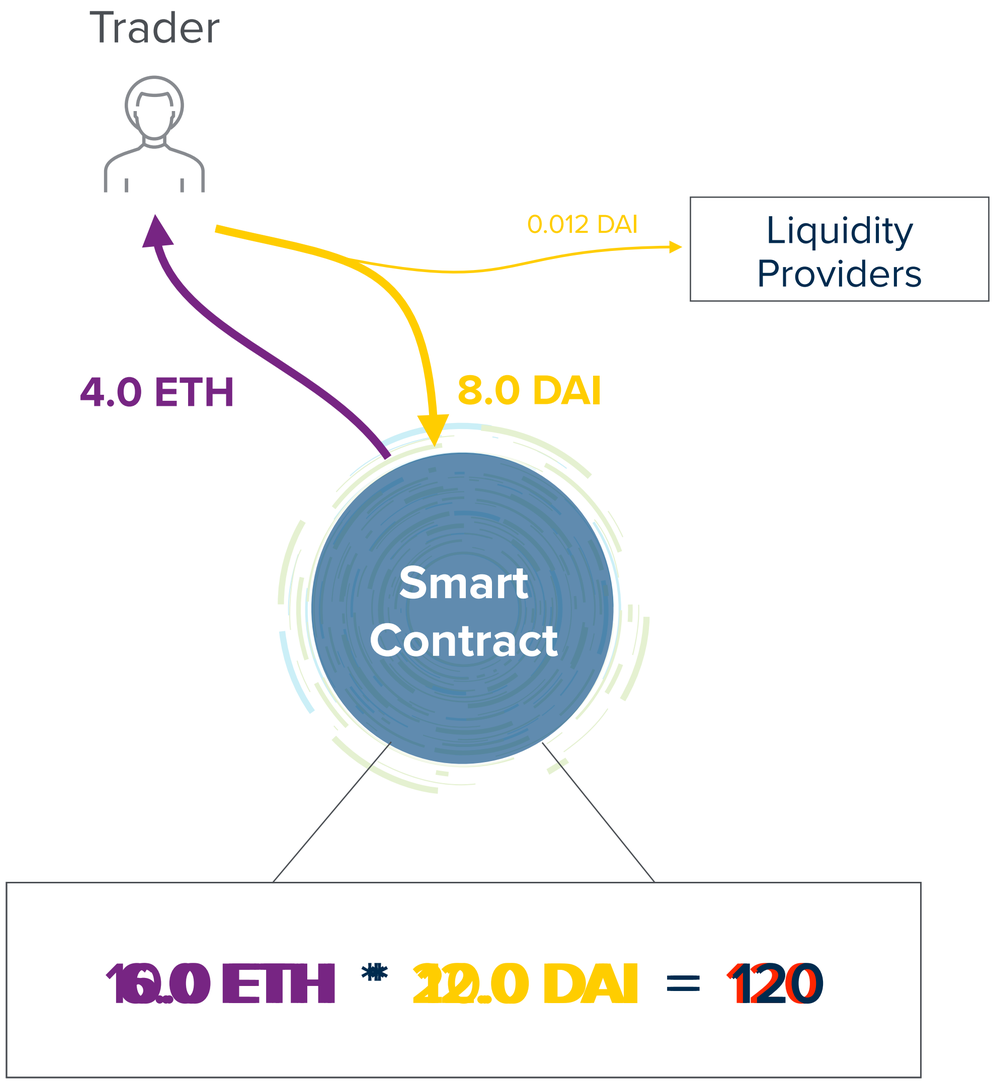
High Level Aspiration

Two-Sided Marketplace

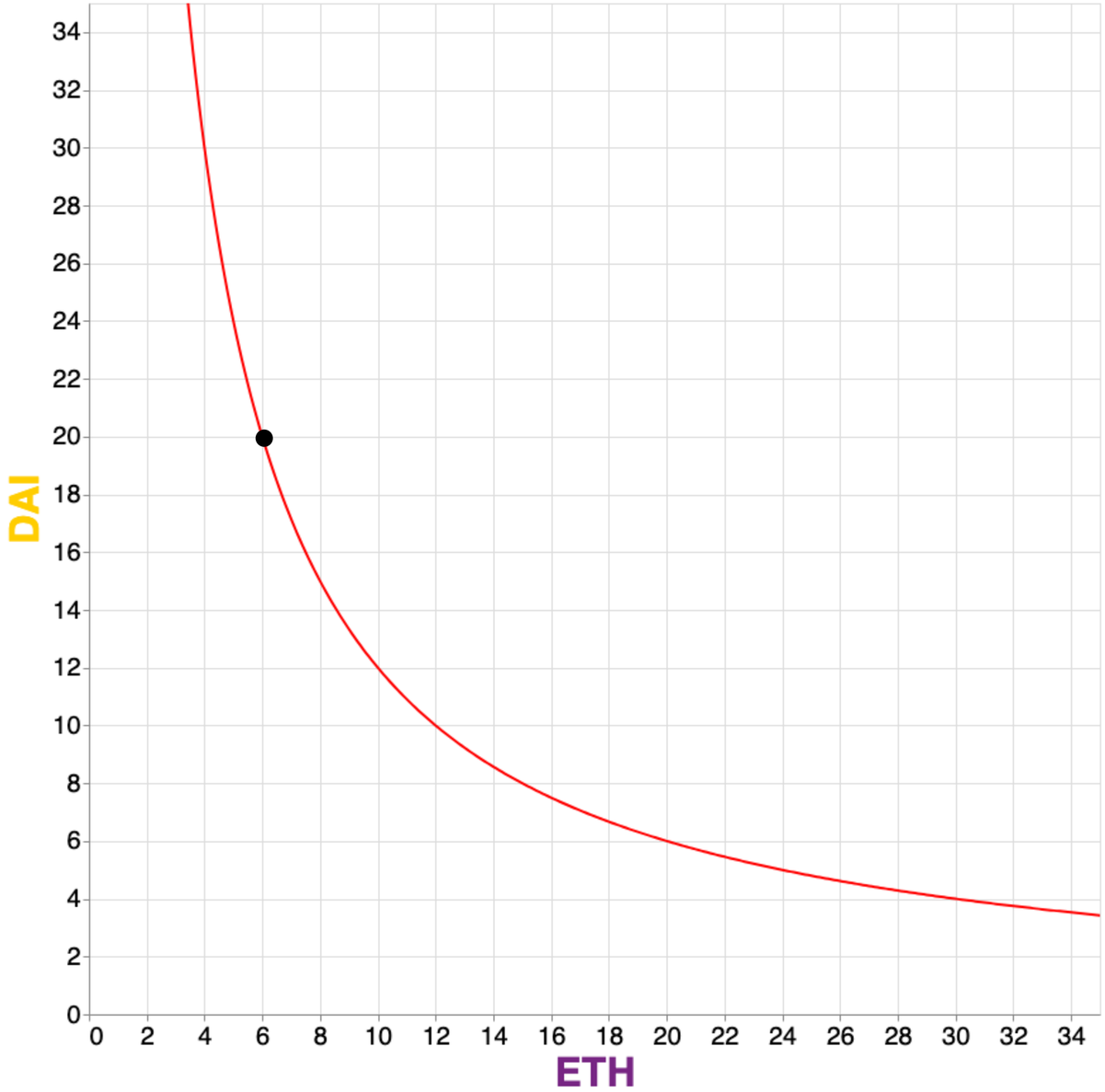


$$xy = k$$


DAI
ETH
8.0
4.0
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34


Trader
0.012 DAI
Liquidity Providers
4.0 ETH
8.0 DAI
Smart Contract
10.0 ETH * 12.0 DAI = 120

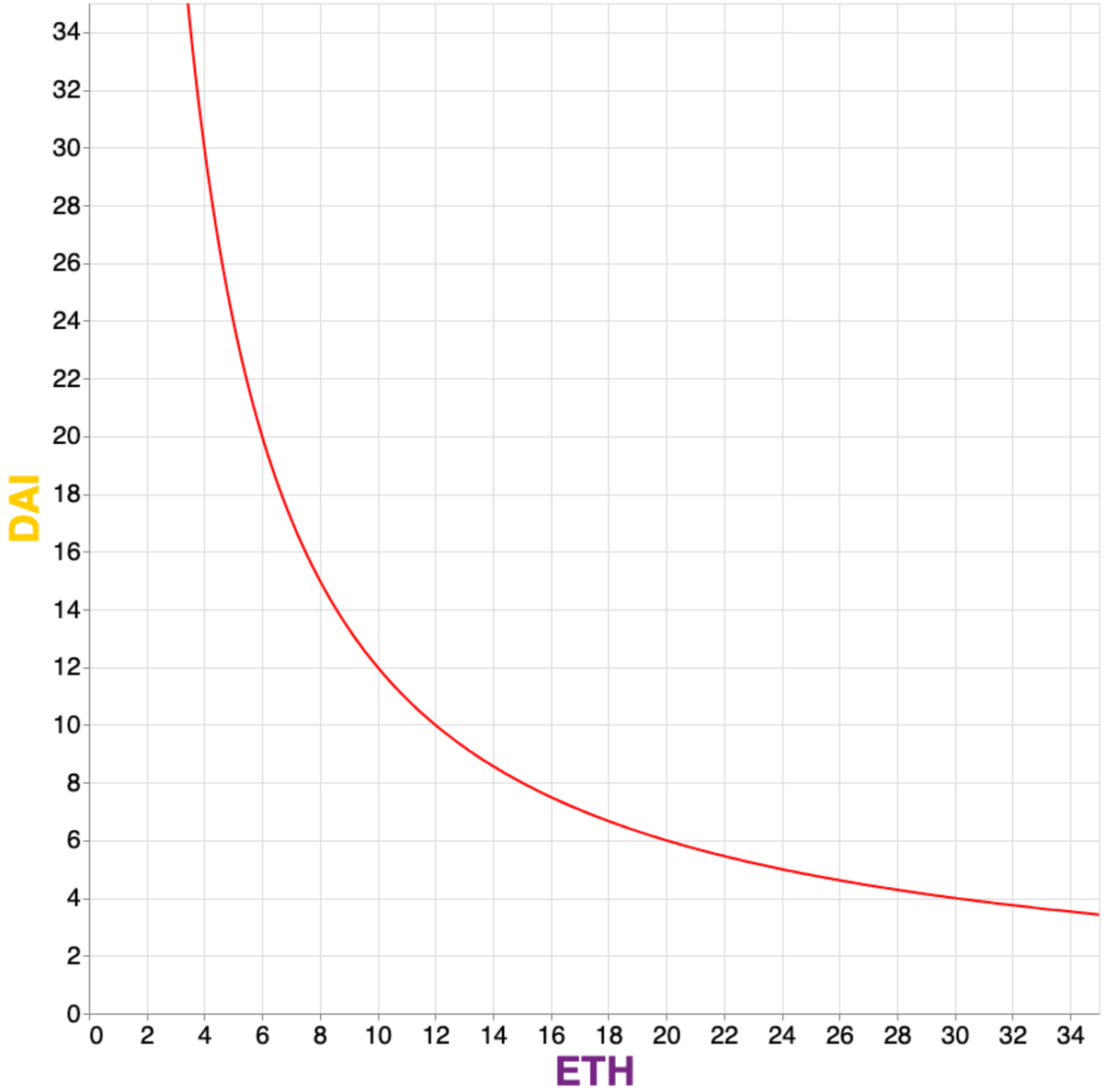
Invariant: k



Simple Pricing Rule

$$(x - \Delta x)(y + \Delta y) = k$$

$$xy = k$$

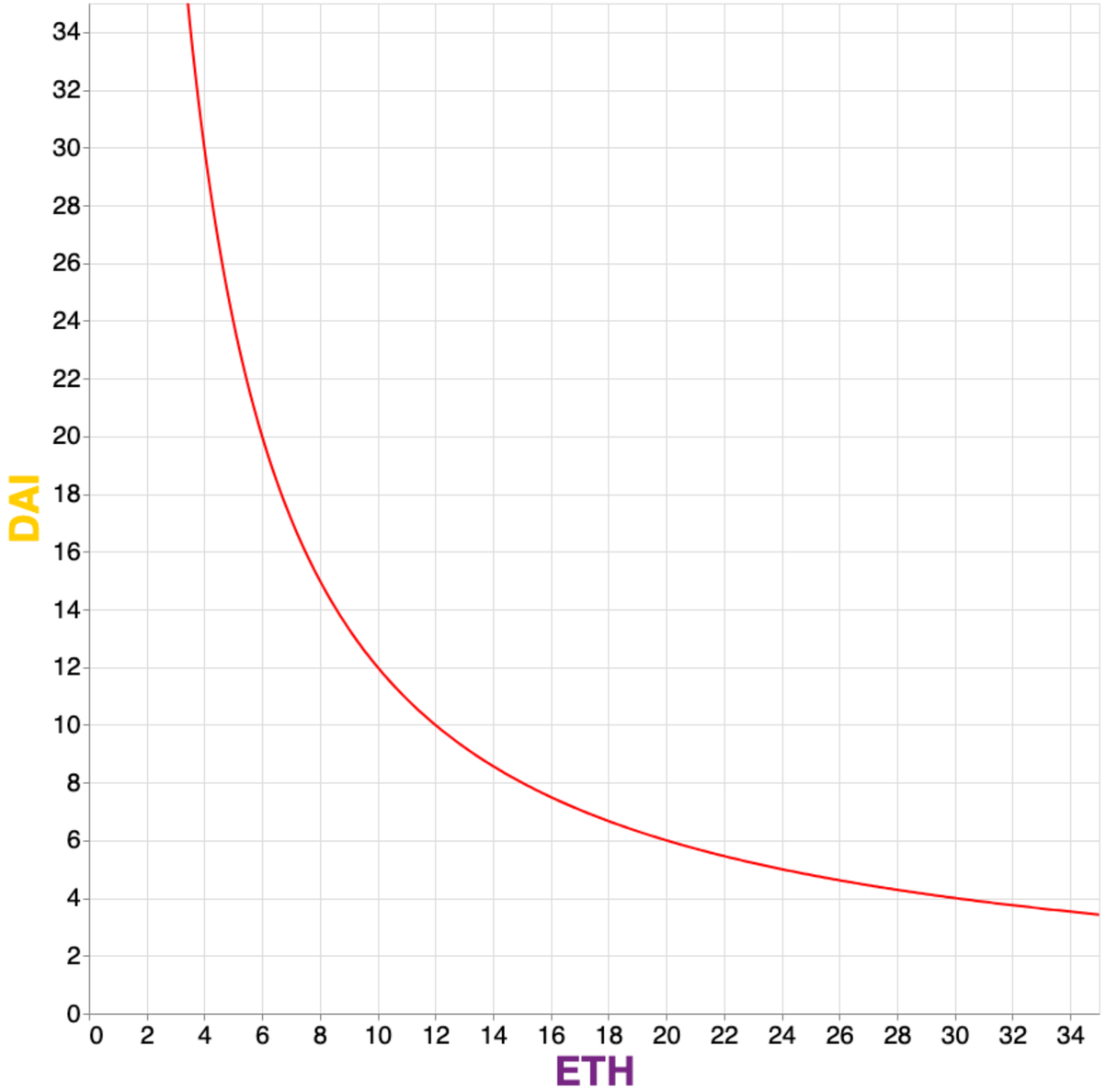


Simple Pricing Rule

$$(x - \Delta x)(y + \phi\Delta y) = k$$

where $(1 - \phi)$ is the percentage fee that is paid to liquidity providers, and where $\Delta x > 0$ and $\Delta y > 0$.

$$xy = k$$



Simple Pricing Rule

$$(x - \Delta x)(y + \phi\Delta y) = k$$

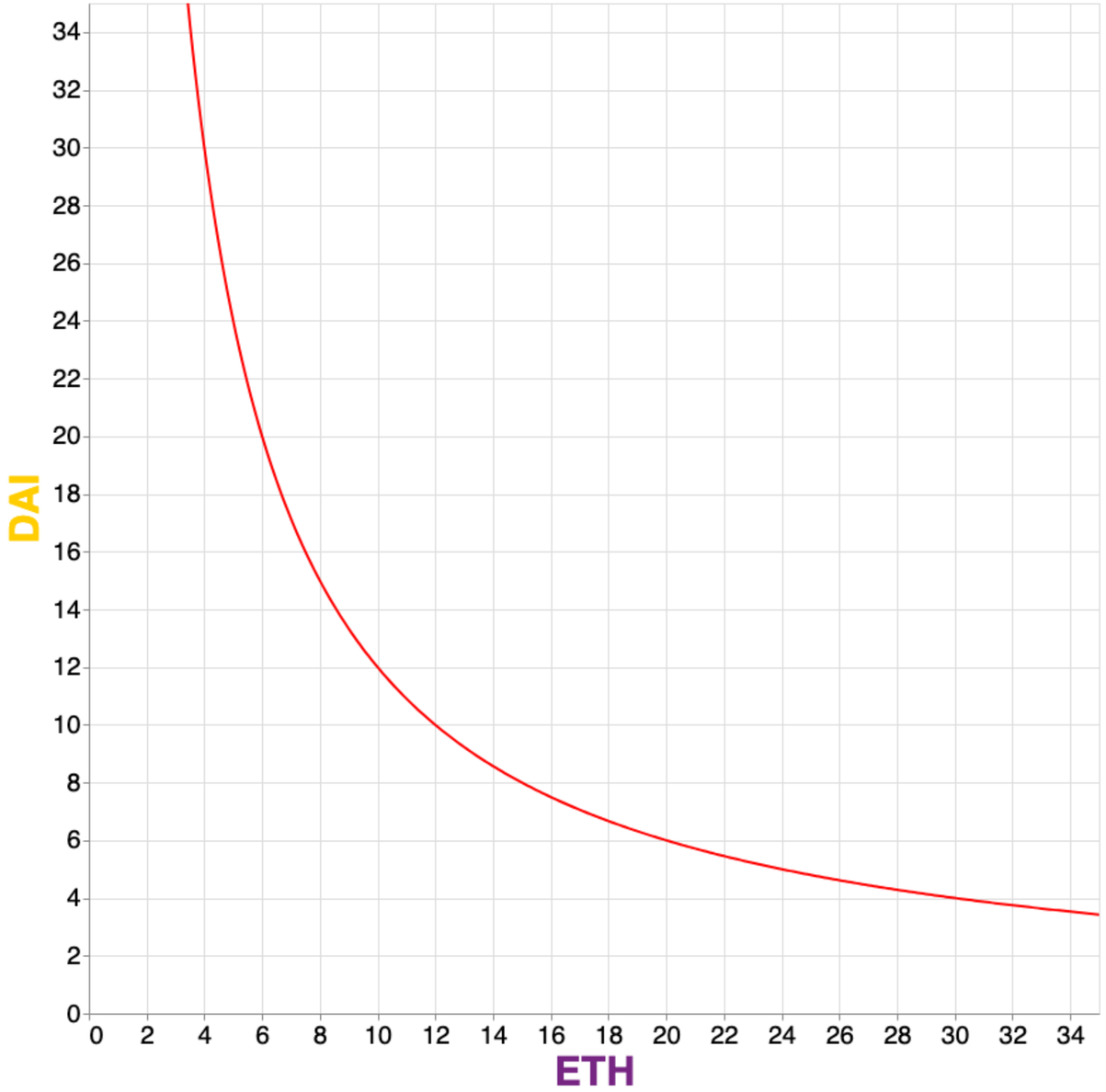
$$\phi\Delta y = \frac{xy}{x - \Delta x} - y$$

$$= \frac{xy - y(x - \Delta x)}{x - \Delta x}$$

$$= \frac{\cancel{xy} - \cancel{xy} - y\Delta x}{x - \Delta x}$$

$$\Delta y = \frac{1}{\phi} \cdot \frac{y\Delta x}{x - \Delta x}$$

$$xy = k$$



Simple Pricing Rule

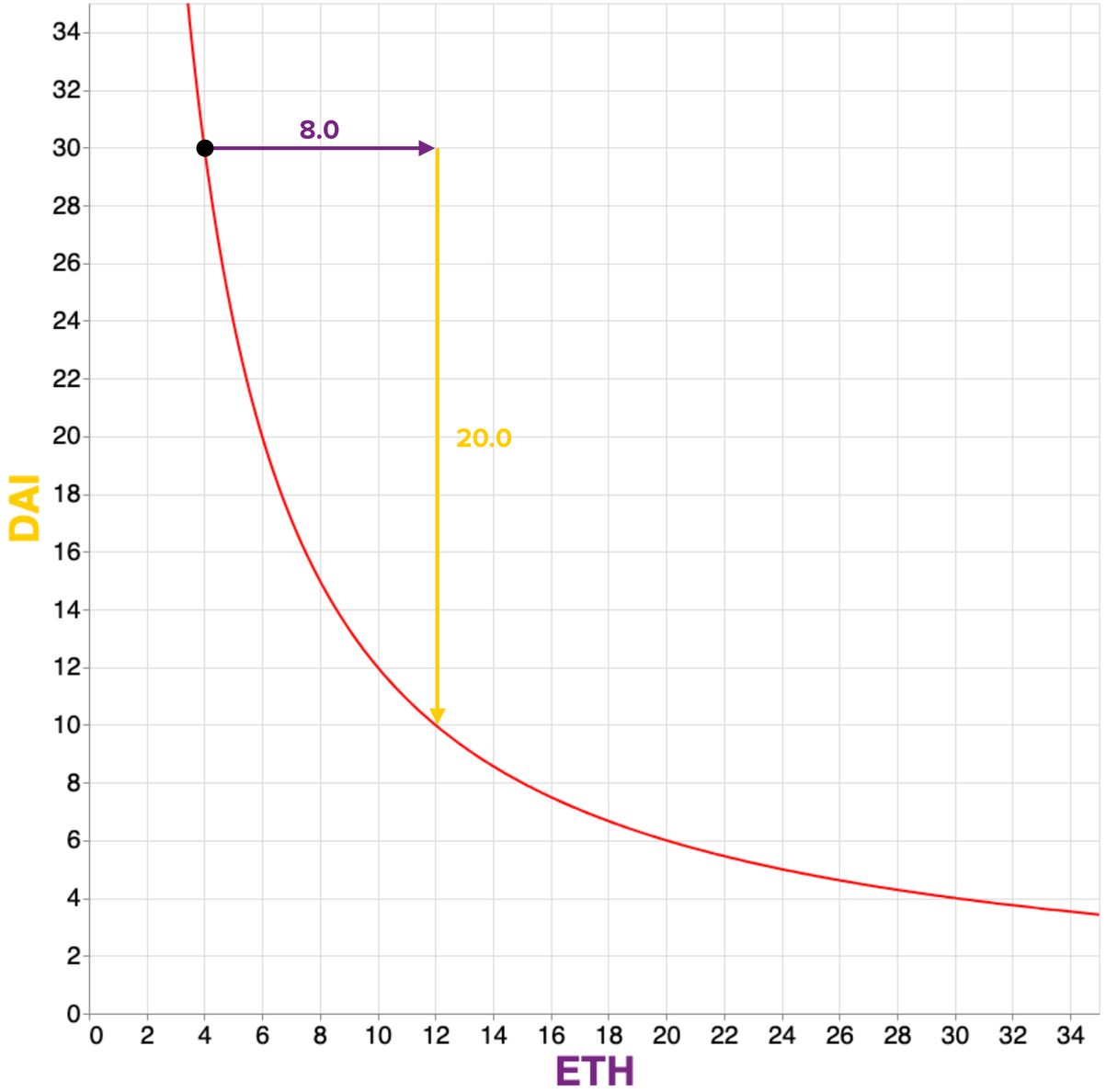
$$\Delta y = \frac{1}{\phi} \cdot \frac{y\Delta x}{x - \Delta x}$$

This rule specifies the price of *buying* Δx in terms of y.

A similar exercise (swapping xs and ys) produces a rule that specifies the price of *selling* Δx in terms of y:

$$\Delta y = \frac{y\phi\Delta x}{x + \phi\Delta x}$$

$$xy = k$$



Simple Pricing Rule

Example where the contract contains **4.0 ETH** and **30.0 DAI** and charges a fee for liquidity providers of 30 bps.

$$\Delta y = \frac{y\phi\Delta x}{x + \phi\Delta x}$$

$$\Delta y = \frac{30 * 0.997 * \Delta x}{4 + 0.997 * \Delta x}$$

Say a trader wants to *sell* **8.0 ETH** to the contract. How much **DAI** should she get in return?

$$\Delta y = \frac{30 * 0.997 * 8}{4 + 0.997 * 8} = 19.98$$

(The fee to liquidity providers is 0.02.)

In the Wild: Uniswap

Selling x for y

$$\Delta y = \frac{y\phi\Delta x}{x + \phi\Delta x}$$

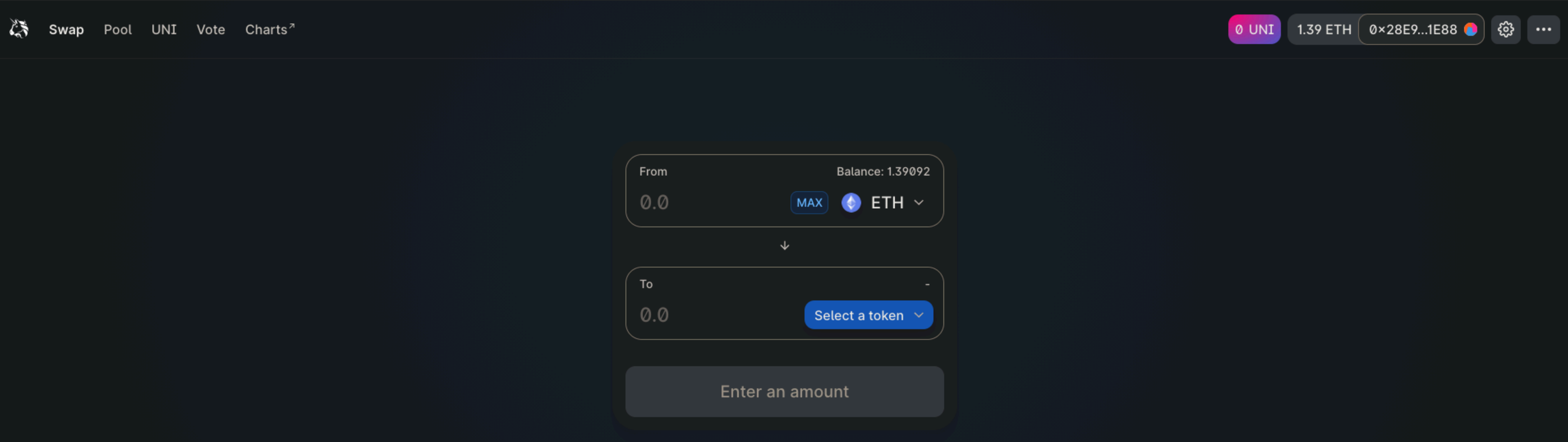
```
41
42      // given an input amount of an asset and pair reserves, returns the maximum output amount of the other asset
43      function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal pure returns (uint amountOut) {
44          require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
45          require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
46          uint amountInWithFee = amountIn.mul(997);
47          uint numerator = amountInWithFee.mul(reserveOut);
48          uint denominator = reserveIn.mul(1000).add(amountInWithFee);
49          amountOut = numerator / denominator;
50      }
51
```

Buying x for y

$$\Delta y = \frac{1}{\phi} \cdot \frac{y\Delta x}{x - \Delta x}$$

```
51
52      // given an output amount of an asset and pair reserves, returns a required input amount of the other asset
53      function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) internal pure returns (uint amountIn) {
54          require(amountOut > 0, 'UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUNT');
55          require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
56          uint numerator = reserveIn.mul(amountOut).mul(1000);
57          uint denominator = reserveOut.sub(amountOut).mul(997);
58          amountIn = (numerator / denominator).add(1);
59      }
60
```

UniswapV2Library.sol



Quick Demo: https://app.uniswap.org/

How to Think about an AMM's Price

Price is the ratio between assets (e.g. **DAI**) paid and assets (e.g. **ETH**) received.

If I pay **100 DAI** for **4 ETH**, then my price per ETH is 25 DAI.

In our notation, this is given by $\Delta y/\Delta x$.

Selling x for y

$$\Delta y = \frac{y\phi\Delta x}{x + \phi\Delta x}$$

Buying x for y

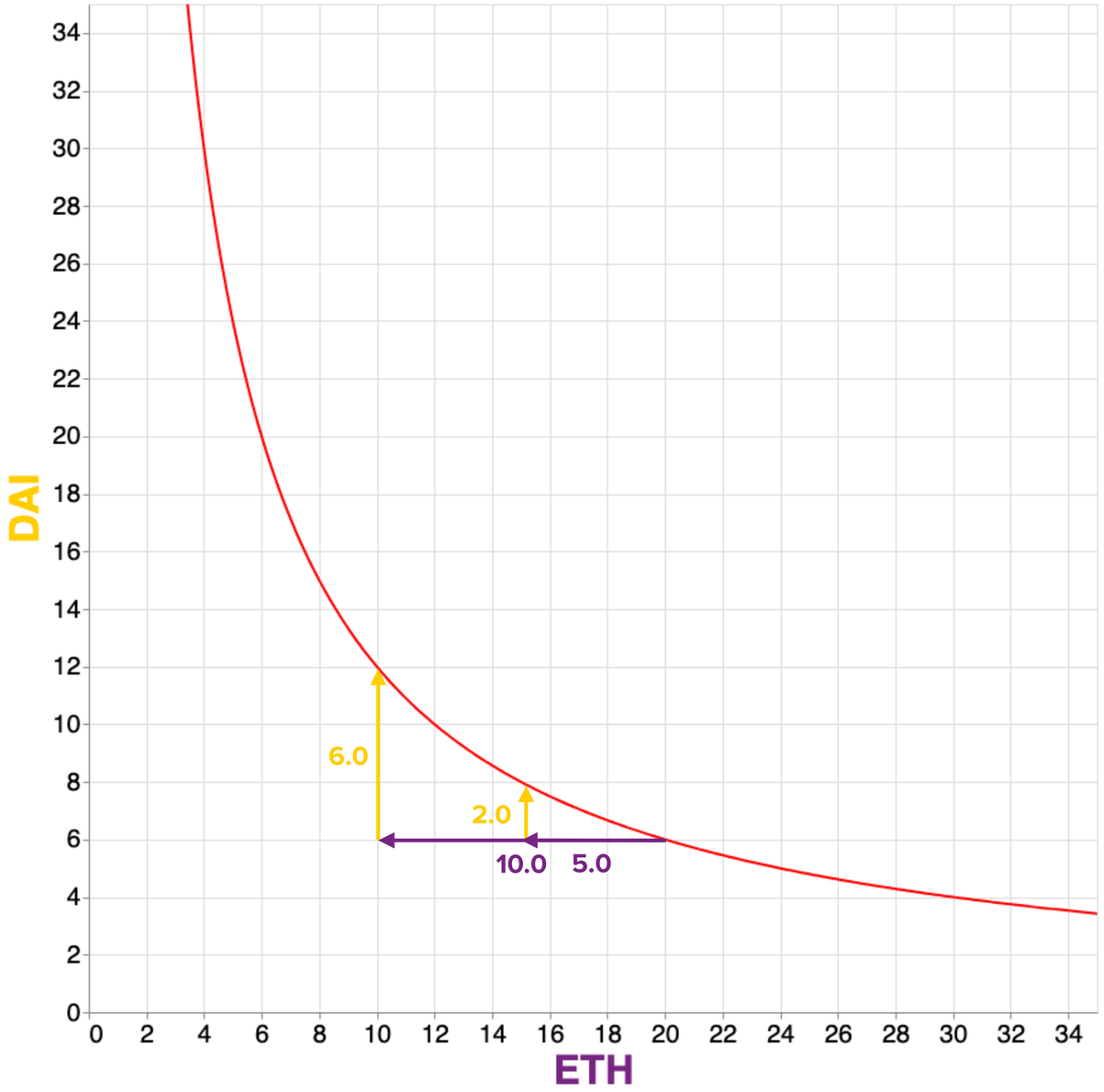
$$\Delta y = \frac{1}{\phi} \cdot \frac{y\Delta x}{x - \Delta x}$$

Divide both sides by Δx to get $\Delta y/\Delta x$.

$$\frac{\Delta y}{\Delta x} = \frac{y\phi}{x + \phi\Delta x}$$

$$\frac{\Delta y}{\Delta x} = \frac{1}{\phi} \cdot \frac{y}{x - \Delta x}$$

$$xy = k$$



Marginal Price & Slippage

Selling x for y

$$\frac{\Delta y}{\Delta x} = \frac{y\phi}{x + \phi\Delta x}$$

Buying x for y

$$\frac{\Delta y}{\Delta x} = \frac{1}{\phi} \cdot \frac{y}{x - \Delta x}$$

Observation #1

Pricing depends on the size of the trade, Δx.

For example with **20.0 ETH** * **6.0 DAI** = **120**,

Buying 10 ETH (i.e. $\Delta x = 10$) costs 6.02 DAI*

Or 0.602 DAI per ETH

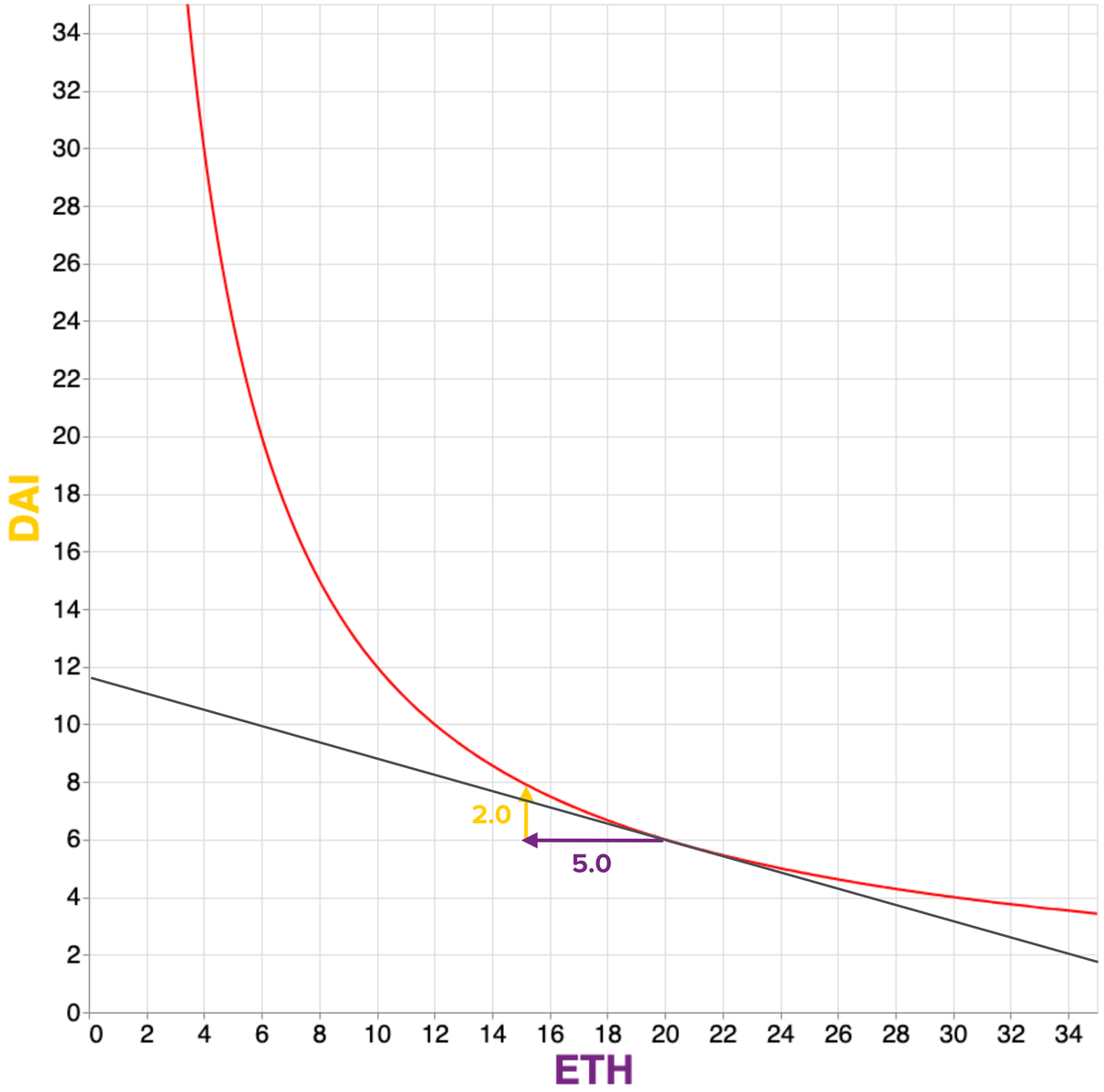
Whereas buying 5 ETH costs 2.006 DAI

Or 0.401 DAI per ETH

* assuming $\phi = 0.997$

a16z

$$xy = k$$



Marginal Price & Slippage

Selling x for y

$$\frac{\Delta y}{\Delta x} = \frac{y\phi}{x + \phi\Delta x}$$

Buying x for y

$$\frac{\Delta y}{\Delta x} = \frac{1}{\phi} \cdot \frac{y}{x - \Delta x}$$

In the limit, as Δx approaches 0:

$$\lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \phi \frac{y}{x}$$

$$\lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \frac{1}{\phi} \frac{y}{x}$$

And, if we set the fee to zero ($\phi = 1$), then:

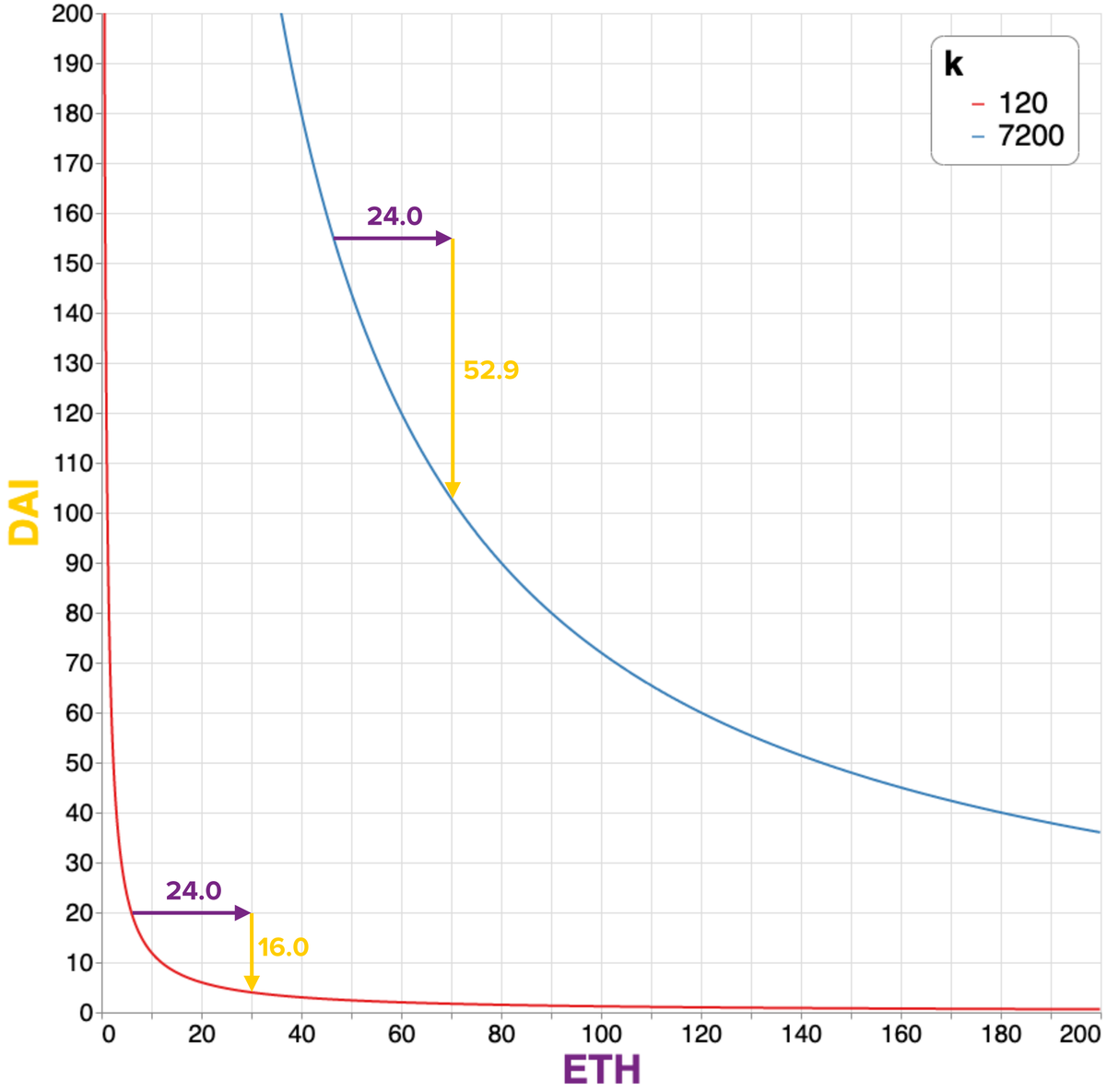
$$M_p = \frac{y}{x}$$

where M_p denotes *marginal price*

M_p is equal to the slope of the tangent line.

a16z

$$xy = k$$



Marginal Price & Slippage

Selling x for y

$$\frac{\Delta y}{\Delta x} = \frac{y\phi}{x + \phi\Delta x}$$

Buying x for y

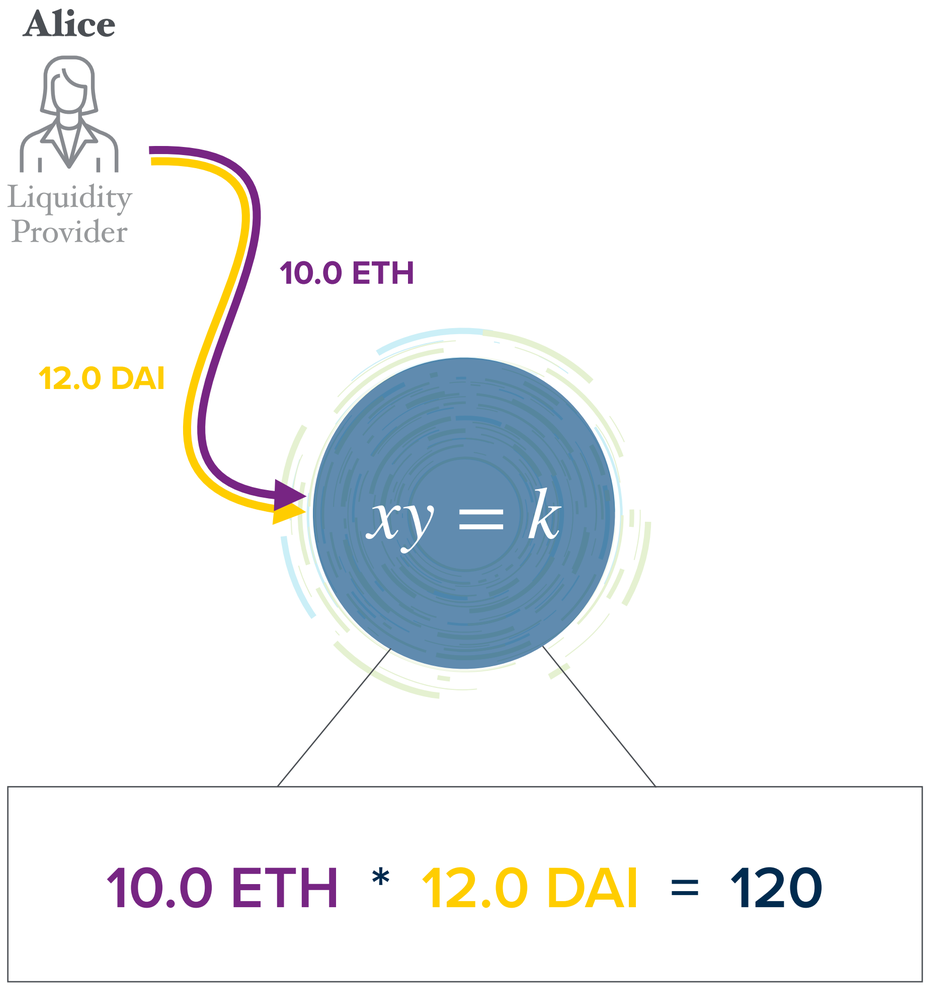
$$\frac{\Delta y}{\Delta x} = \frac{1}{\phi} \cdot \frac{y}{x - \Delta x}$$

Observation #2

Pricing depends on the size of x and y (i.e. k)

It's straightforward to see that, as k increases, the effective price of the AMM is less sensitive to Δx.

Incentives for Liquidity Providers



Alice deposits 10 ETH and 12 DAI of liquidity, which implies:

$$M_p = 1.2 \qquad \text{where } M_p \text{ denotes } \textit{marginal price}$$

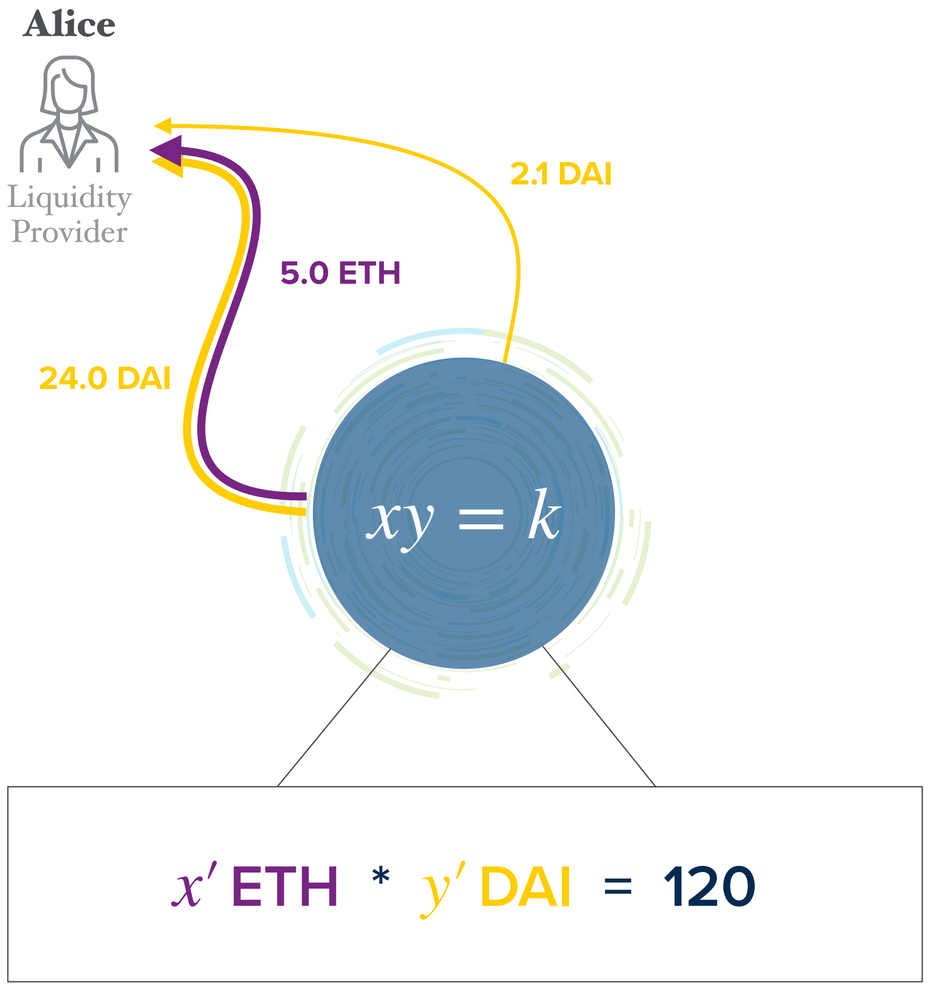
Alice waits for a month, during which traders drive $700 worth of volume through the AMM.

At the end of the month, Alice withdraws her ETH and DAI. By that time, the price of ETH has gone up 4x. The marginal price is now:

$$M_p' = 4.8$$

What is Alice's return?

Assume: $(1 - \phi) = 0.003$

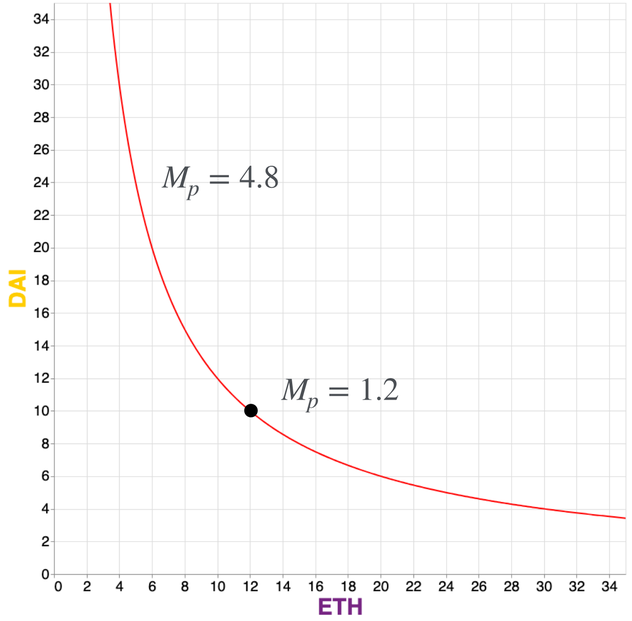


First, what does Alice earn from liquidity provider fees?

$$V(1-\phi) = 700 * 0.003 = \$2.1$$

where V denotes trading volume

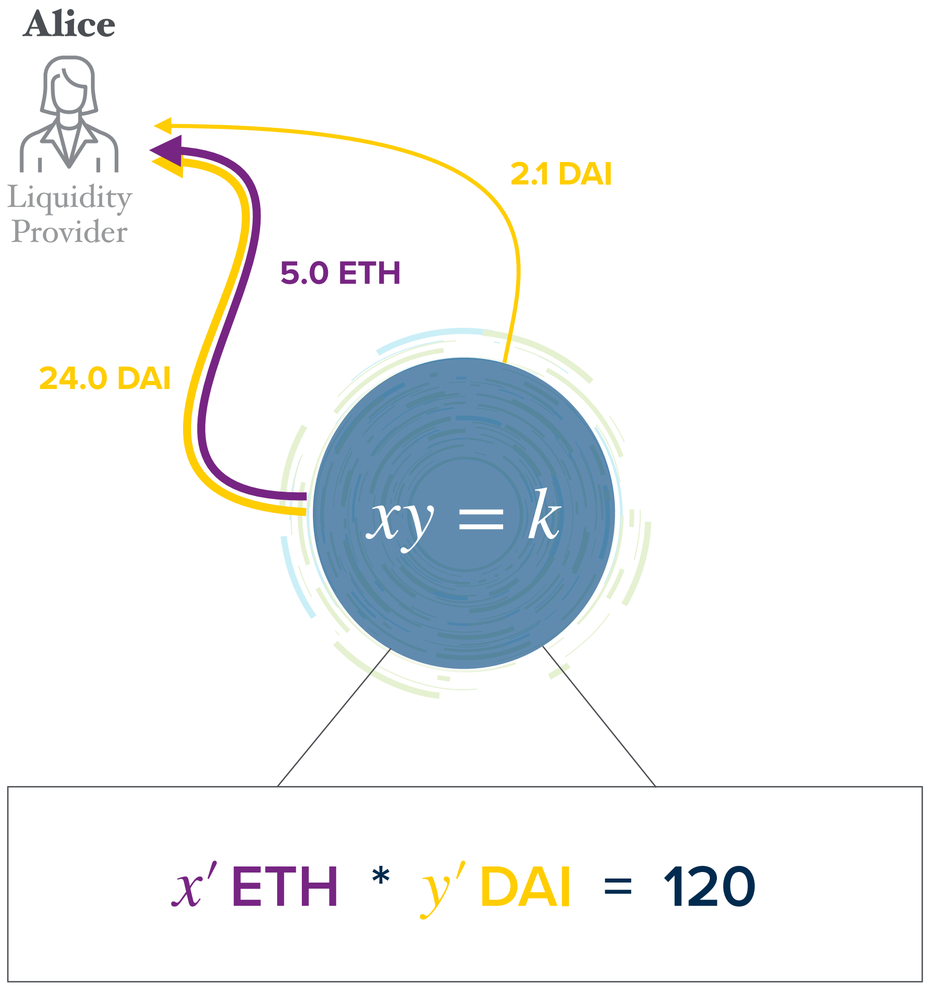
Second, how many ETH and DAI does Alice get back?



$x' = 5$ ETH

$y' = 24$ DAI

~~Impermanent~~ Divergence Loss



So, how did Alice do?

Measured in DAI, Alice now has:

$$R = 5\ \text{ETH} * \frac{4.8\ \text{ETH}}{\text{DAI}} + 24\ \text{DAI} + 2.1\ \text{DAI}$$

$$R = 50.1\ \text{DAI}$$

Not bad, but how would she have done if she had just held onto her 12 ETH and 10 DAI?

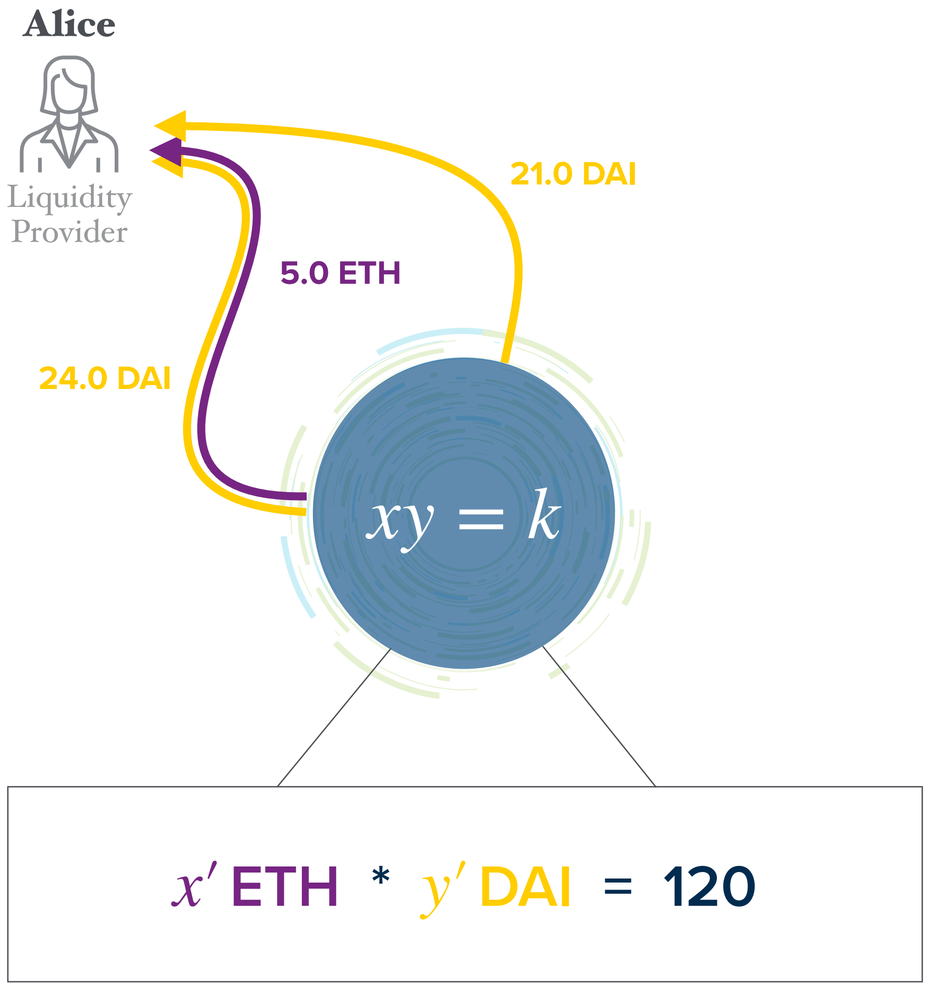
$$R_B = 12\ \text{ETH} * \frac{4.8\ \text{ETH}}{\text{DAI}} + 10\ \text{DAI}$$

$$R_B = 67.6\ \text{DAI}$$

This is called ~~impermanent~~ loss divergence

a16z

~~Impermanent~~ Divergence Loss



What if volume had been higher?

Say, volume had been $7,000 instead of $700:

$$V(1-\phi) = 7000 * 0.003 = \$21$$

Therefore,

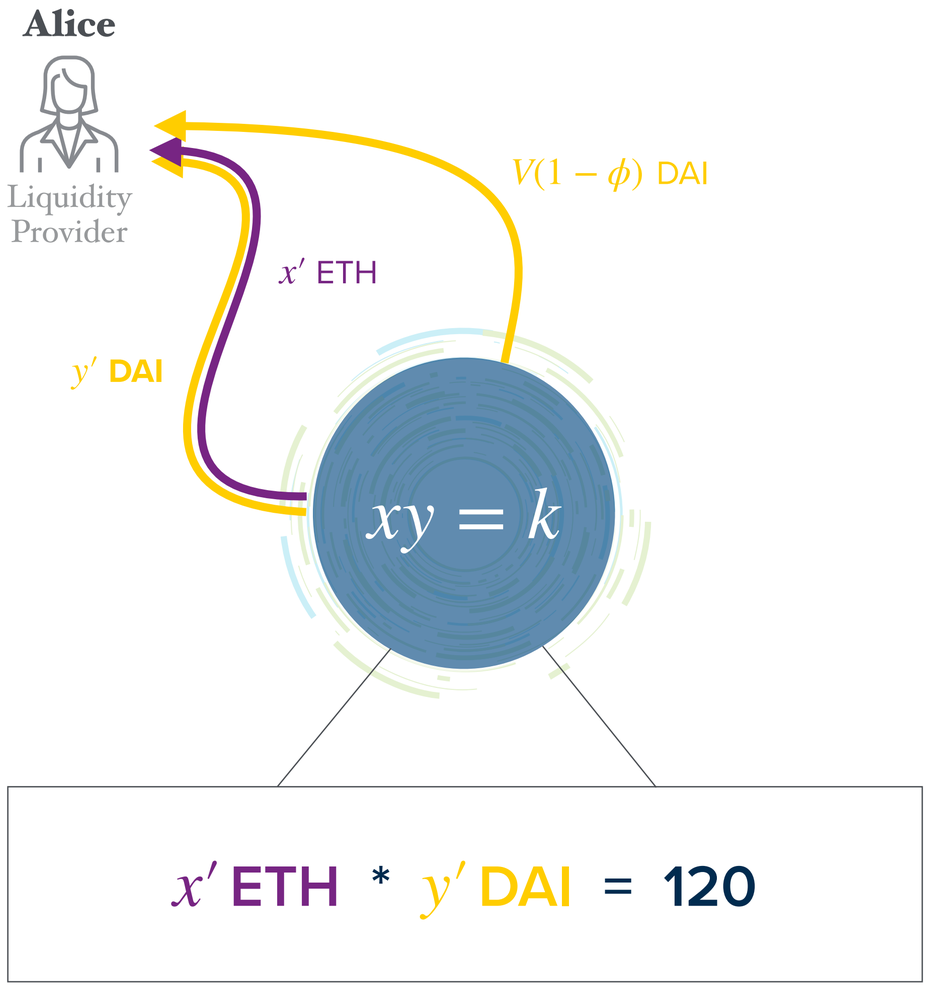
$$R = 5 \text{ ETH} * \frac{4.8 \text{ ETH}}{\text{DAI}} + 24 \text{ DAI} + 21 \text{ DAI}$$

$$R = 69.0 \text{ DAI}$$

This time, Alice's returns are greater than her baseline return R_B of 67.6 DAI. Her profit:

$$P_L = \frac{R}{R_B} - 1 = 2.1\,\%$$

~~Impermanent~~ Divergence Loss



More generally

Alice's return R is given by:

$$R = x'M_p' + y' + V(1-\phi)$$

Her baseline return R_B is given by:

$$R_B = xM_p + y$$

Her profit, in percentage terms is given by:

$$P_L = \frac{R}{R_B} - 1 = \frac{x'M_p' + y' + V(1-\phi)}{xM_p + y} - 1$$

Let's ignore the volume term for now, and simplify:

$$P_L = \frac{x'M_p' + y'}{xM_p + y} - 1 \quad \text{assuming } V = 0 \text{ for now}$$

Recall

$xy = k$ and $M_p = y/x$

Thus,

$$x = \sqrt{\frac{k}{M_p}} \quad \text{and} \quad y = \sqrt{kM_p}$$

Also,

$$x' = \sqrt{\frac{k}{M_p'}} \quad \text{and} \quad y' = \sqrt{kM_p'}$$

Finally, let:

$$M_p' = rM_p$$

~~Impermanent~~ Divergence Loss

Simplifying

$$P_L = \frac{x'M_p' + y'}{xM_p + y}$$ Step 1: let's express everything in terms of M_p and k.

$$P_L = \frac{\sqrt{\frac{k}{rM_p}}rM_p + \sqrt{krM_p}}{\sqrt{\frac{k}{M_p}}rM_p + \sqrt{kM_p}} - 1 = \frac{2\sqrt{r}\cancel{\sqrt{kM_p}}}{r\cancel{\sqrt{kM_p}} + \cancel{\sqrt{kM_p}}} - 1$$

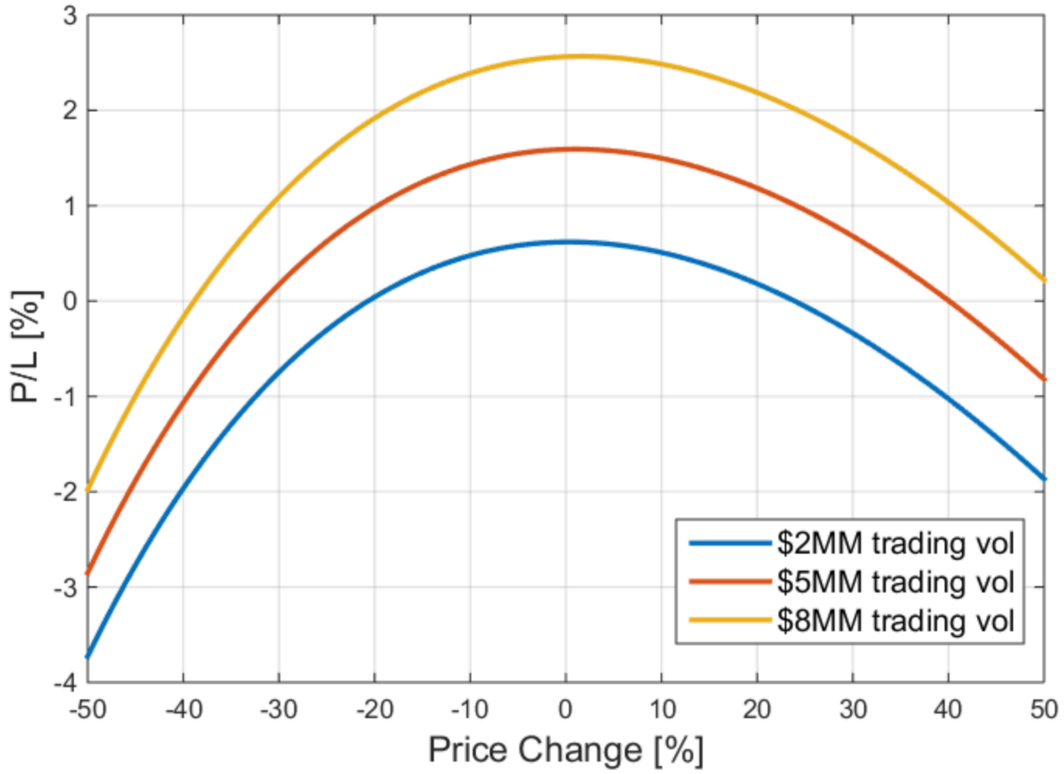
$$P_L = \frac{2\sqrt{r}}{r+1} - 1$$

Step 2: Reintroduce the volume term:

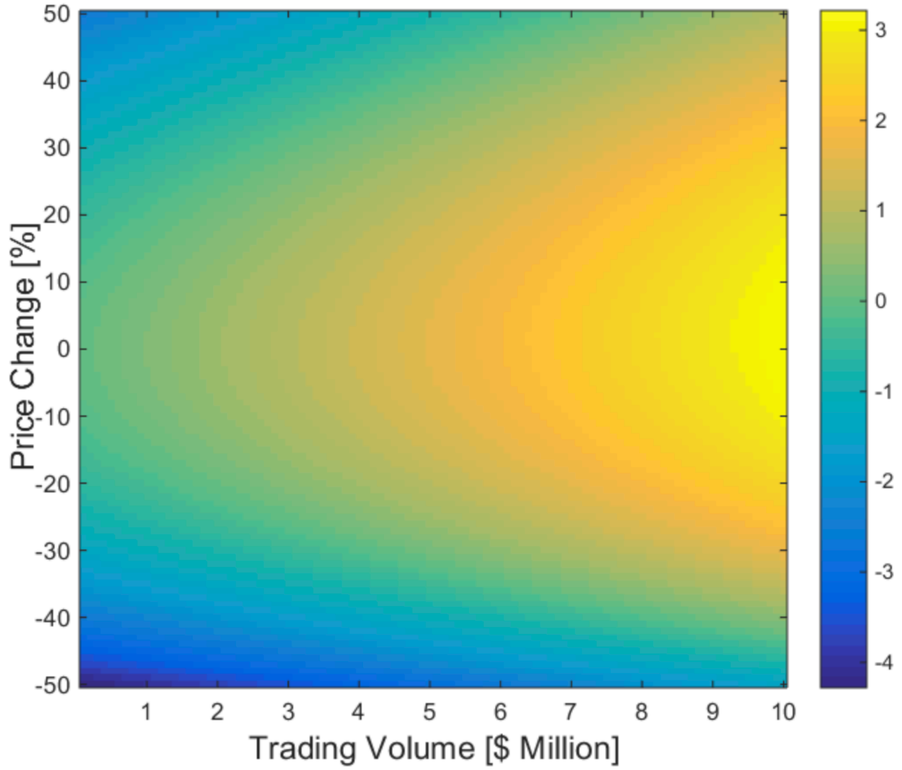
$$P_L = \frac{2\sqrt{r}}{r+1} + \frac{V(1-\phi)}{c} - 1$$

Step 3: Plot this equation

~~Impermanent~~ Divergence Loss



Optimal P/L occurs when the final price is equal to that at liquidity provisioning



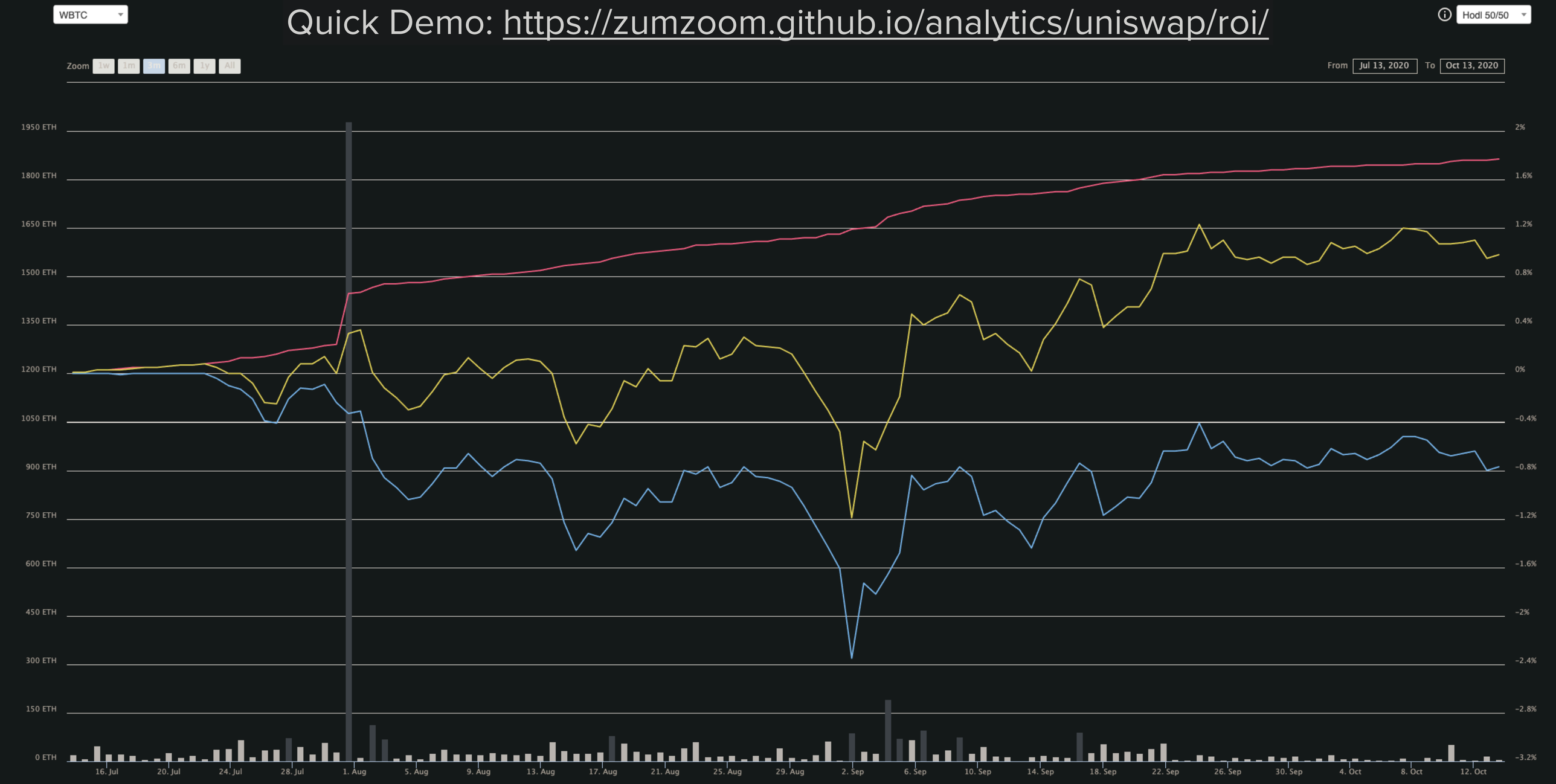
P/L percentage of liquidity provision on Uniswap for different scenarios of exchange trading volume and ETH price change

$$P_L = \frac{2\sqrt{r}}{r+1} + \frac{V(1-\phi)}{c} - 1$$

Image credit: https://www.tokendaily.co/blog/pnl-analysis-of-uniswap-market-making

Uniswap ROI By Token

Quick Demo: https://zumzoom.github.io/analytics/uniswap/roi/

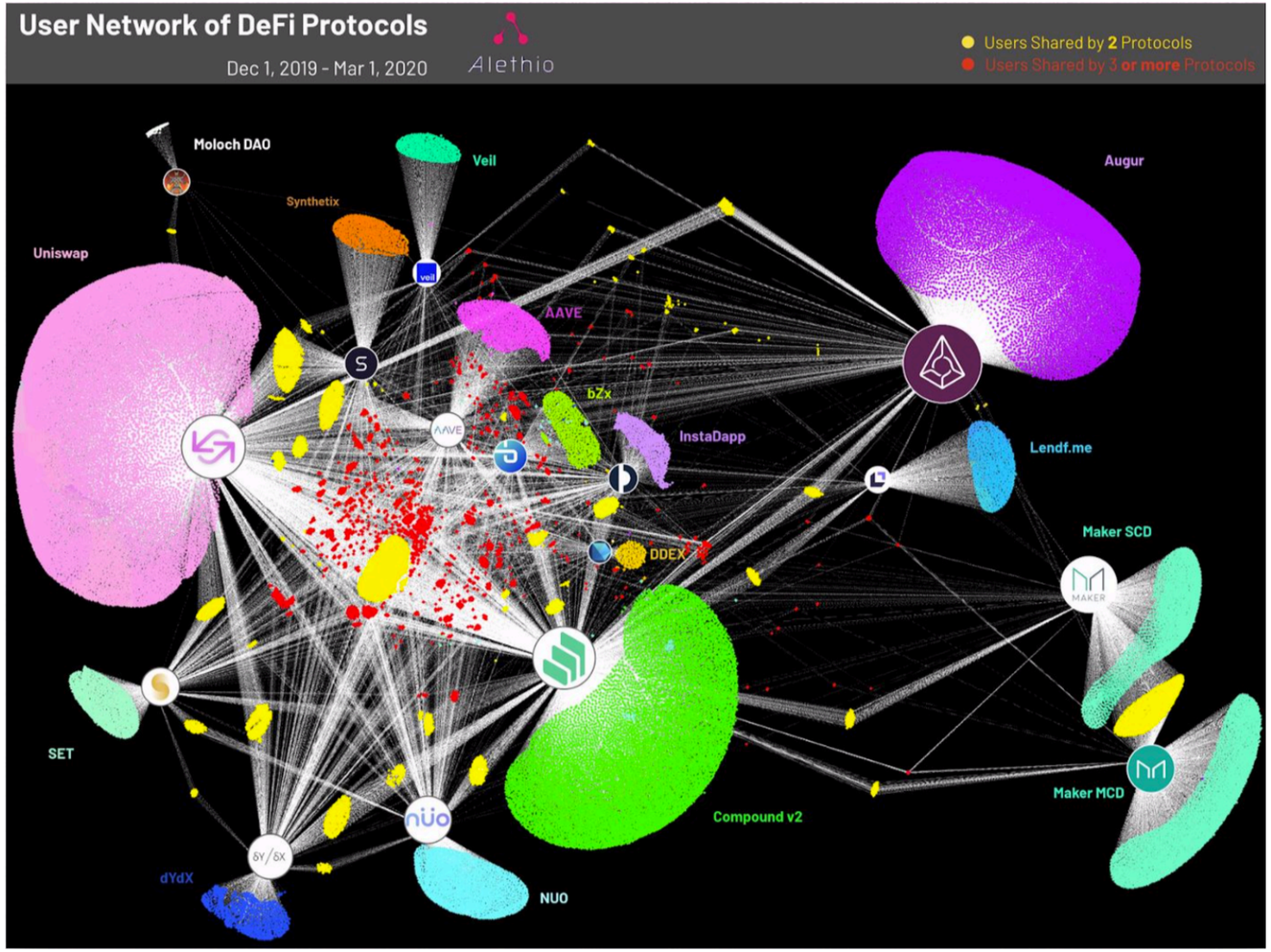


Uniswap's Metrics To Date

Quick Demo: https://uniswap.info/

Example: Uniswap

Interoperability



Optional: Generalizations of $xy = k$

- Curve: https://www.curve.fi/
- Balancer: https://balancer.finance/

DEXs: Concluding Thoughts

Desired Characteristics

- Simple — buildable as a smart contract
- Automated liquidity — no dependence on active market-makers
- No single points of control

Decentralized Lending

Eddy Lazzarin

Overcollateralized
vs
Undercollateralized

Centralized Lending

Example: Overcollateralized Margin Trading

- Trust the exchange not to get hacked, steal assets, or incorrectly calculate balances
- Borrowed assets only exist on the exchange: they can't be used on the blockchain (no protocol composability)
- Interest payments go to the exchange

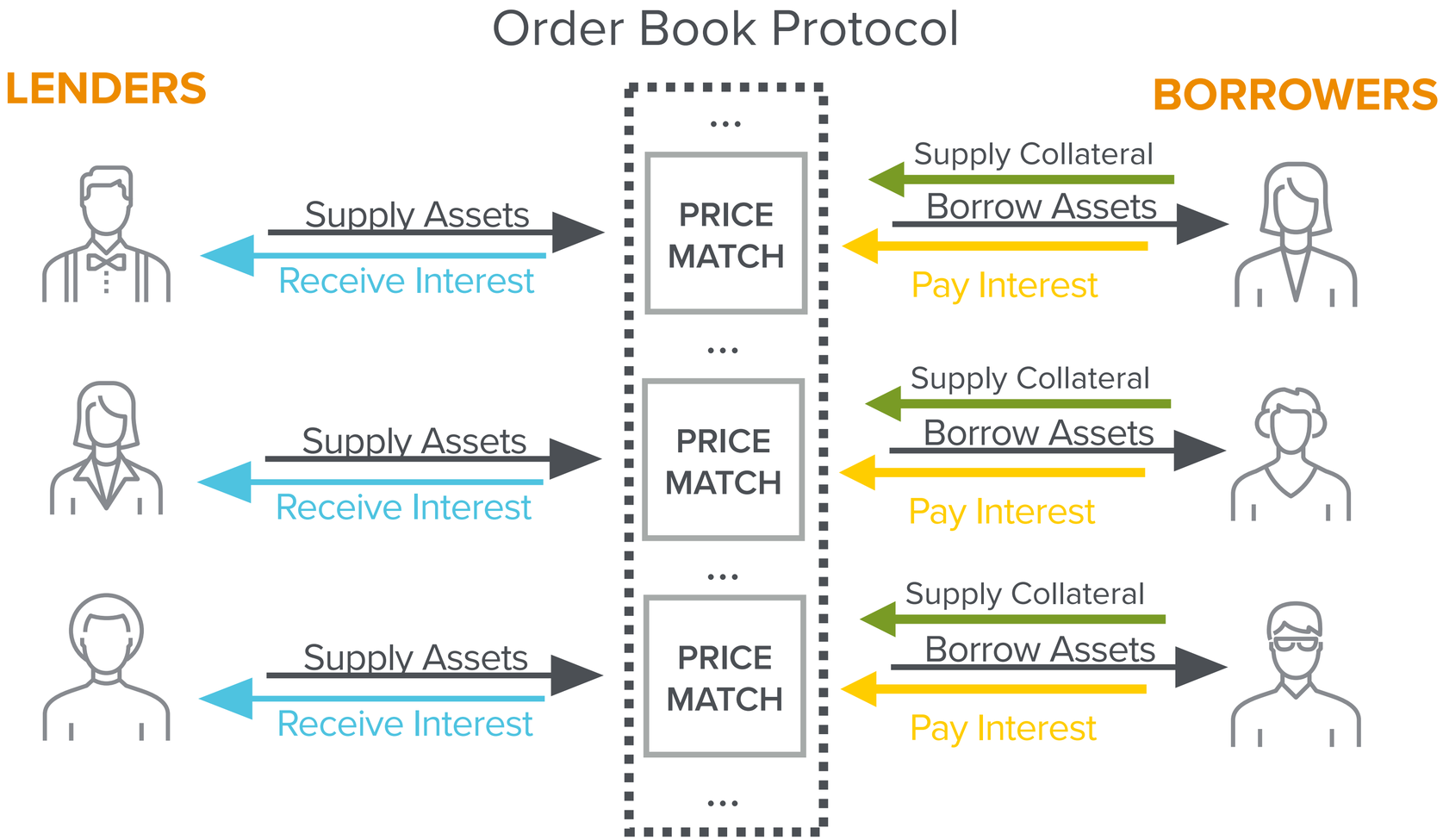
Decentralized Lending

High Level Motivation

- Minimize trust required in the counter-party
- Borrowed assets can be used freely on the blockchain (enable protocol composability)
- Interest payments go from asset borrowers to asset suppliers, neither set being permissioned

Order Book

An Early Approach



a16z

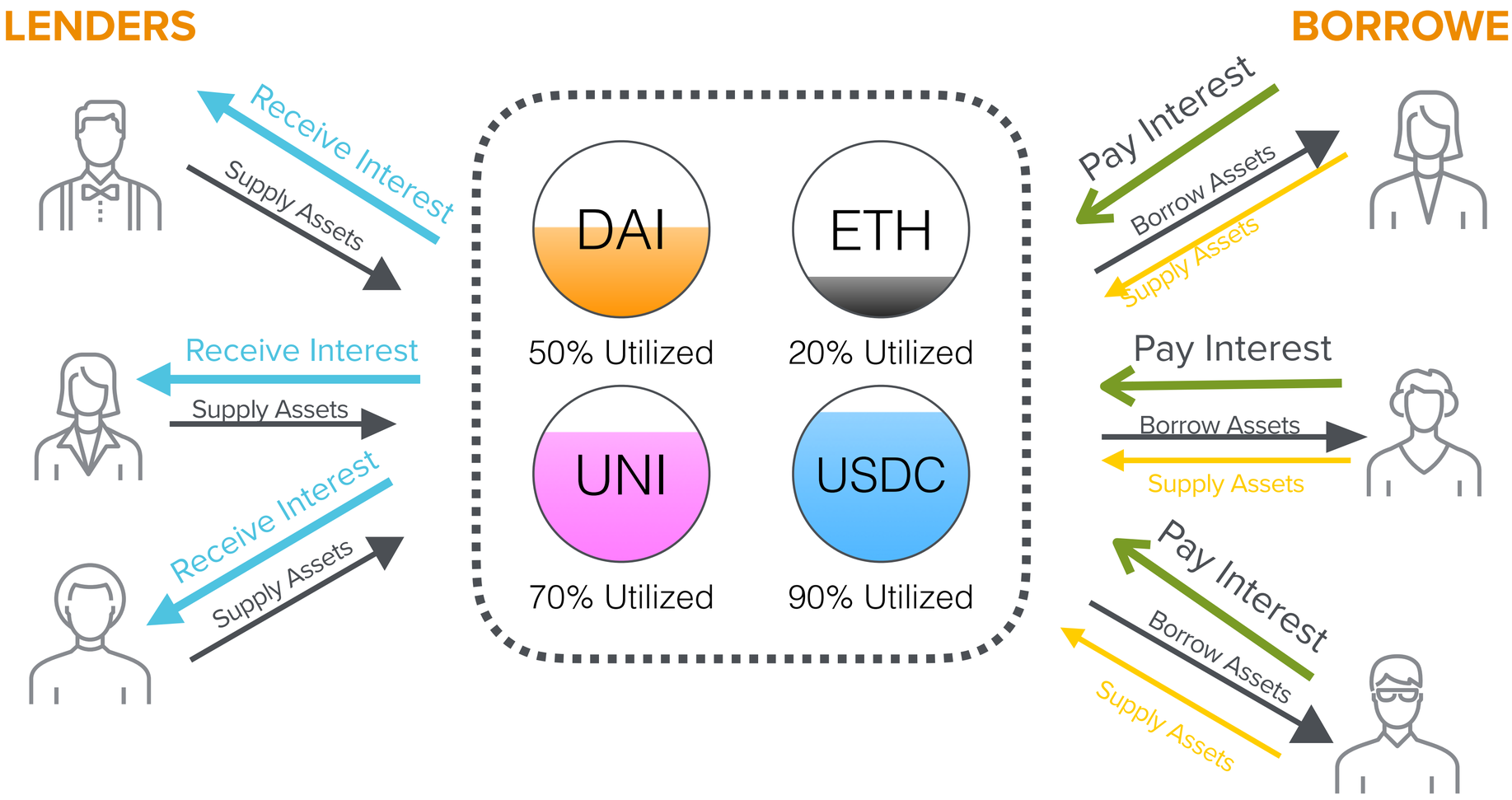
Decentralized Lending

Order Book Defects

- Fractured liquidity
  - More asset pairs thins the supply across those pairs
- Computationally expensive
  - Matching many borrowers or many lenders requires many transactions per person
- Concentrated risk
  - Lenders are exposed strictly to the risk that their matched counter-parties will default, increasing the variance of interest returns
- Fixed rates only
  - As the supply and demand to borrow a given asset changes, matched interest rates don't change, adding complexity
- Difficult withdrawal
  - A supplier must wait for their counter-parties to repay their debts (or force liquidation) to withdraw their share

Case Study: Compound

Liquidity Pool Approach



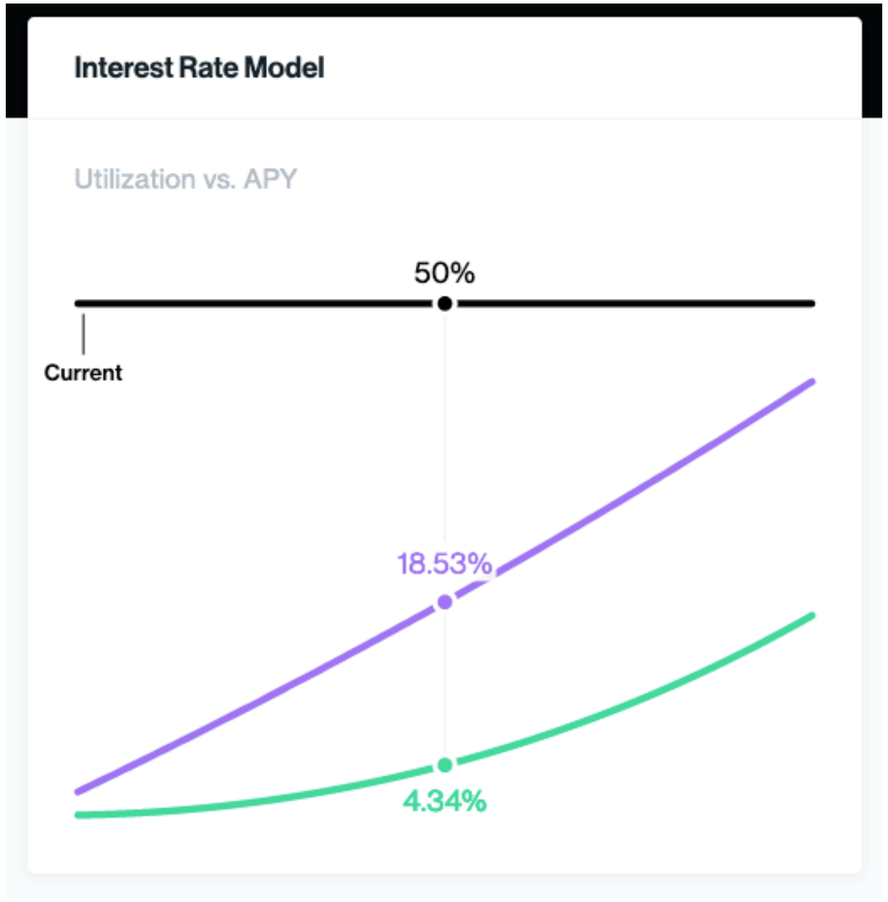
Decentralized Lending

Liquidity Pool Advantages

- Shared liquidity
  - Adding another asset no longer fractures liquidity
- Computationally simpler
  - Suppliers and borrowers can add or remove large volumes with single transactions, independent of the distribution of counter-parties
- Distributed risk
  - Risk is shared by the entire pool*
- Variable rate**
  - An order book matches supply and demand implicitly as borrowers and lenders make adjustments; the shared pool adjusts rates automatically based on existing supply and utilization
- Graceful withdrawal
  - As long as extra assets remain in the pool, a supplier can withdraw their share

Case Study: Compound

Interest Rate Curve for BAT



Utilization Rate

Borrow Interest Rate

Supply Interest Rate

Source: https://compound.finance/markets/BAT

Case Study: Compound

Step-by-step Procedure for Borrowing

- Alice supplies 2 ETH to ETH pool
  - Her total ETH-equivalent balance times the **collateralFactor** represents her total capacity to borrow: **sumCollateral**
- Alice requests to borrow 1 ETH worth of BAT
  - Since this is less than her **sumCollateral**, the borrow is valid
- Now, every block, Alice accumulates interest on what she's borrowed until it is repaid

Calculations

- supplied asset
  - 2 ETH
- collateralFactor
  - 0.6
- sumCollateral
  - 1.2
- borrowCurrent
  - 1.0

Optional Image Credit Line: URL Goes Here

Case Study: Compound

Utilization Ratio

$$U_a = Borrows_a / \left(Cash_a + Borrows_a \right)$$

Utilization Ratio

Case Study: Compound

Interest Rate Curve

$$Borrowing\ Interest\ Rate_a = Base\ Rate\ +\ U_a\ *\ Slope\ Multiplier$$

```
    function getUtilizationRate(uint cash, uint borrows) pure internal returns (IRError, Exp memory) {
        if (borrows == 0) {
            // Utilization rate is zero when there's no borrows
            return (IRError.NO_ERROR, Exp({mantissa: 0}));
        }

        (MathError err0, uint cashPlusBorrows) = addUInt(cash, borrows);
        if (err0 != MathError.NO_ERROR) {
            return (IRError.FAILED_TO_ADD_CASH_PLUS_BORROWS, Exp({mantissa: 0}));
        }

        (MathError err1, Exp memory utilizationRate) = getExp(borrows, cashPlusBorrows);
        if (err1 != MathError.NO_ERROR) {
            return (IRError.FAILED_TO_GET_EXP, Exp({mantissa: 0}));
        }

        return (IRError.NO_ERROR, utilizationRate);
    }
```

```
    /*
     * @dev Calculates the utilization and borrow rates for use by getBorrowRate function
     */
    function getUtilizationAndAnnualBorrowRate(uint cash, uint borrows) view internal returns (IRError, Exp memory, Exp memory) {
        (IRError err0, Exp memory utilizationRate) = getUtilizationRate(cash, borrows);
        if (err0 != IRError.NO_ERROR) {
            return (err0, Exp({mantissa: 0}), Exp({mantissa: 0}));
        }

        // Borrow Rate is 5% + UtilizationRate * 45% (baseRate + UtilizationRate * multiplier);
        // 45% of utilizationRate, is `rate * 45 / 100`
        (MathError err1, Exp memory utilizationRateMuled) = mulScalar(utilizationRate, multiplier);
        // `mulScalar` only overflows when the product is >= 2^256.
        // utilizationRate is a real number on the interval [0,1], which means that
        // utilizationRate.mantissa is in the interval [0e18,1e18], which means that 45 times
        // that is in the interval [0e18,45e18]. That interval has no intersection with 2^256, and therefore
        // this can never overflow for the standard rates.
        if (err1 != MathError.NO_ERROR) {
            return (IRError.FAILED_TO_MUL_UTILIZATION_RATE, Exp({mantissa: 0}), Exp({mantissa: 0}));
        }

        (MathError err2, Exp memory utilizationRateScaled) = divScalar(utilizationRateMuled, mantissaOne);
        // 100 is a constant, and therefore cannot be zero, which is the only error case of divScalar.
        assert(err2 == MathError.NO_ERROR);

        // Add the 5% for (5% + 45% * Ua)
        (MathError err3, Exp memory annualBorrowRate) = addExp(utilizationRateScaled, Exp({mantissa: baseRate}));
        // `addExp` only fails when the addition of mantissas overflow.
        // As per above, utilizationRateMuled is capped at 45e18,
        // and utilizationRateScaled is capped at 4.5e17. mantissaFivePercent = 0.5e17, and thus the addition
        // is capped at 5e17, which is less than 2^256. This only applies to the standard rates
        if (err3 != MathError.NO_ERROR) {
            return (IRError.FAILED_TO_ADD_BASE_RATE, Exp({mantissa: 0}), Exp({mantissa: 0}));
        }

        return (IRError.NO_ERROR, utilizationRate, annualBorrowRate);
    }
```

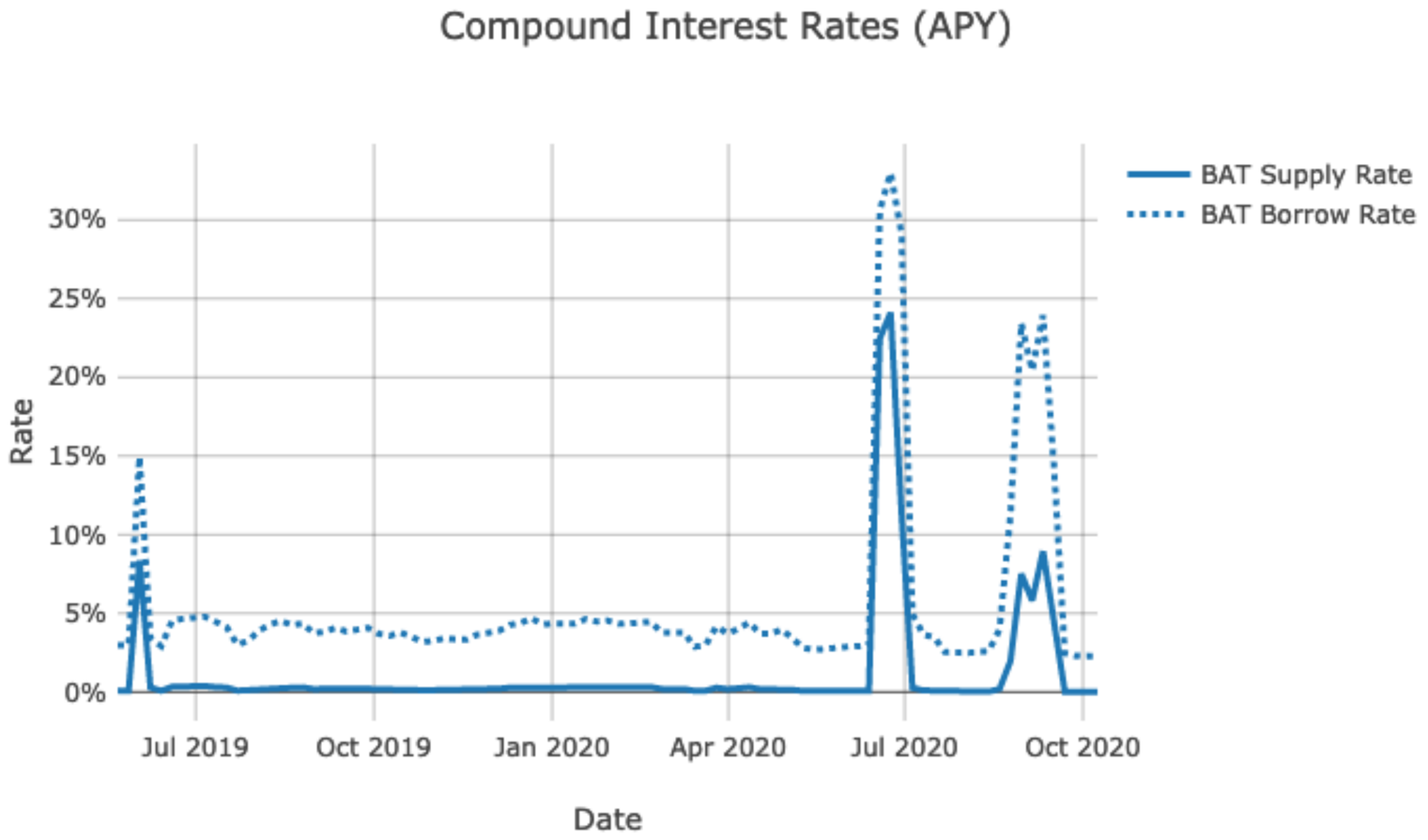
```
    /*
     * @dev Calculates the utilization and borrow rates for use by getBorrowRate function
     */
    function getUtilizationAndAnnualBorrowRate(uint cash, uint borrows) view internal returns (IRError, Exp memory, Exp memory) {
        (IRError err0, Exp memory utilizationRate) = getUtilizationRate(cash, borrows);

        // Borrow Rate is 5% + UtilizationRate * 45% (baseRate + UtilizationRate * multiplier);
        // 45% of utilizationRate, is `rate * 45 / 100`
        (MathError err1, Exp memory utilizationRateMuled) = mulScalar(utilizationRate, multiplier);

        (MathError err2, Exp memory utilizationRateScaled) = divScalar(utilizationRateMuled, mantissaOne);

        // Add the 5% for (5% + 45% * Ua)
        (MathError err3, Exp memory annualBorrowRate) = addExp(utilizationRateScaled, Exp({mantissa: baseRate}));

        return (IRError.NO_ERROR, utilizationRate, annualBorrowRate);
    }
```

```
    /*
     * @dev Calculates the utilization and borrow rates for use by getBorrowRate function
     */
    function getUtilizationAndAnnualBorrowRate(uint cash, uint borrows) view internal returns (IRError, Exp memory, Exp memory) {
        (IRError err0, Exp memory utilizationRate) = getUtilizationRate(cash, borrows);

        // Borrow Rate is 5% + UtilizationRate * 45% (baseRate + UtilizationRate * multiplier);
        // 45% of utilizationRate, is `rate * 45 / 100`
        (MathError err1, Exp memory utilizationRateMuled) = mulScalar(utilizationRate, multiplier);

        (MathError err2, Exp memory utilizationRateScaled) = divScalar(utilizationRateMuled, mantissaOne);

        // Add the 5% for (5% + 45% * Ua)
        (MathError err3, Exp memory annualBorrowRate) = addExp(utilizationRateScaled, Exp({mantissa: baseRate}));

        return (IRError.NO_ERROR, utilizationRate, annualBorrowRate);
    }
```

```
    /**
      * @notice Gets the current borrow interest rate based on the given asset, total cash, total borrows
      *         and total reserves.
      * @dev The return value should be scaled by 1e18, thus a return value of
      *      `(true, 1000000000000)` implies an interest rate of 0.000001 or 0.0001% *per block*.
      * @param cash The total cash of the underlying asset in the CToken
      * @param borrows The total borrows of the underlying asset in the CToken
      * @param _reserves The total reserves of the underlying asset in the CToken
      * @return Success or failure and the borrow interest rate per block scaled by 10e18
      */
    function getBorrowRate(uint cash, uint borrows, uint _reserves) public view returns (uint, uint) {
        _reserves; // pragma ignore unused argument

        (IRError err0, Exp memory _utilizationRate, Exp memory annualBorrowRate) = getUtilizationAndAnnualBorrowRate(cash, borrows);
        if (err0 != IRError.NO_ERROR) {
            return (uint(err0), 0);
        }

        // And then divide down by blocks per year.
        (MathError err1, Exp memory borrowRate) = divScalar(annualBorrowRate, blocksPerYear); // basis points * blocks per year
        // divScalar only fails when divisor is zero. This is clearly not the case.
        assert(err1 == MathError.NO_ERROR);

        _utilizationRate; // pragma ignore unused variable

        // Note: mantissa is the rate scaled 1e18, which matches the expected result
        return (uint(IRError.NO_ERROR), borrowRate.mantissa);
    }
}
```

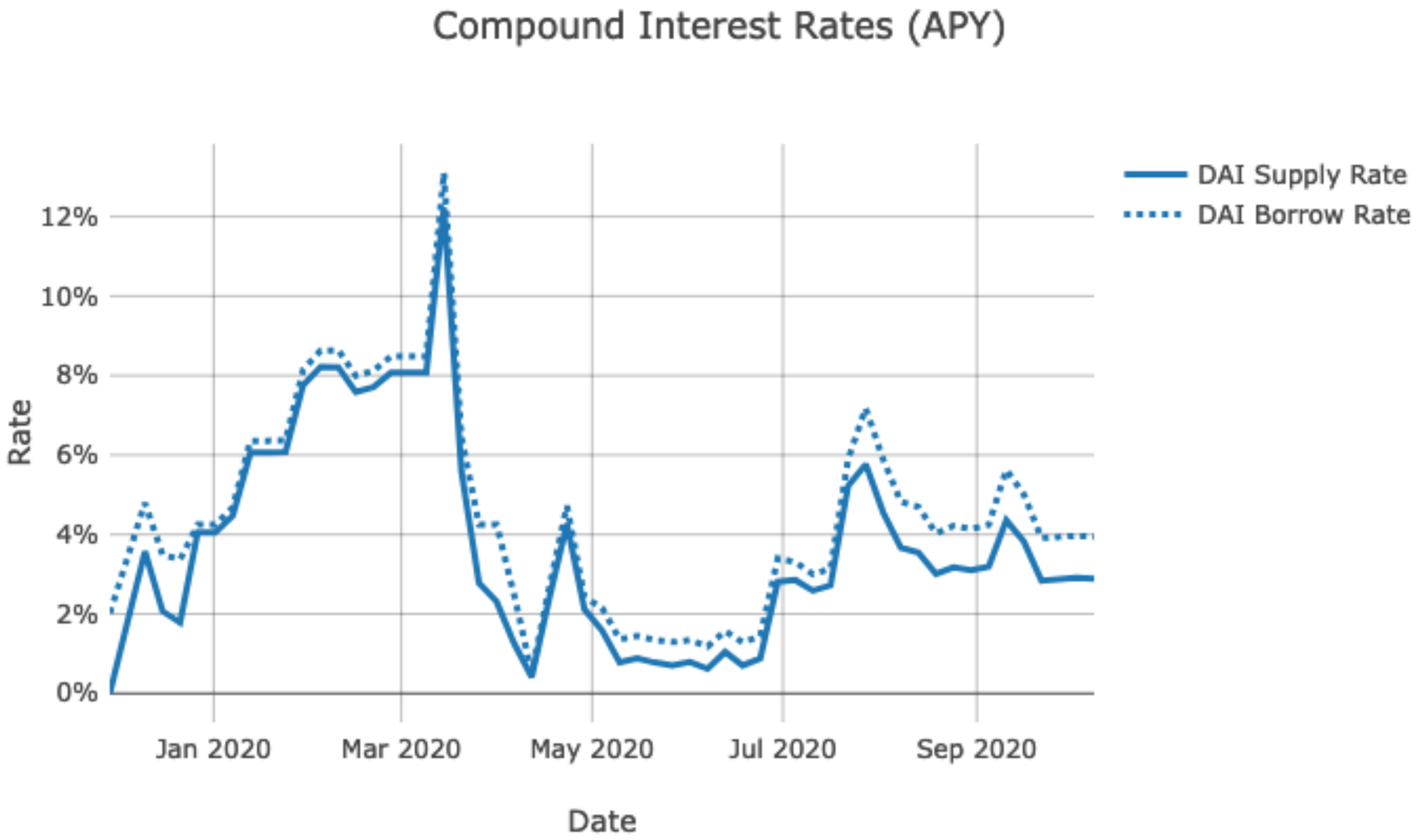
Case Study: Compound

Historical BAT Interest Rate



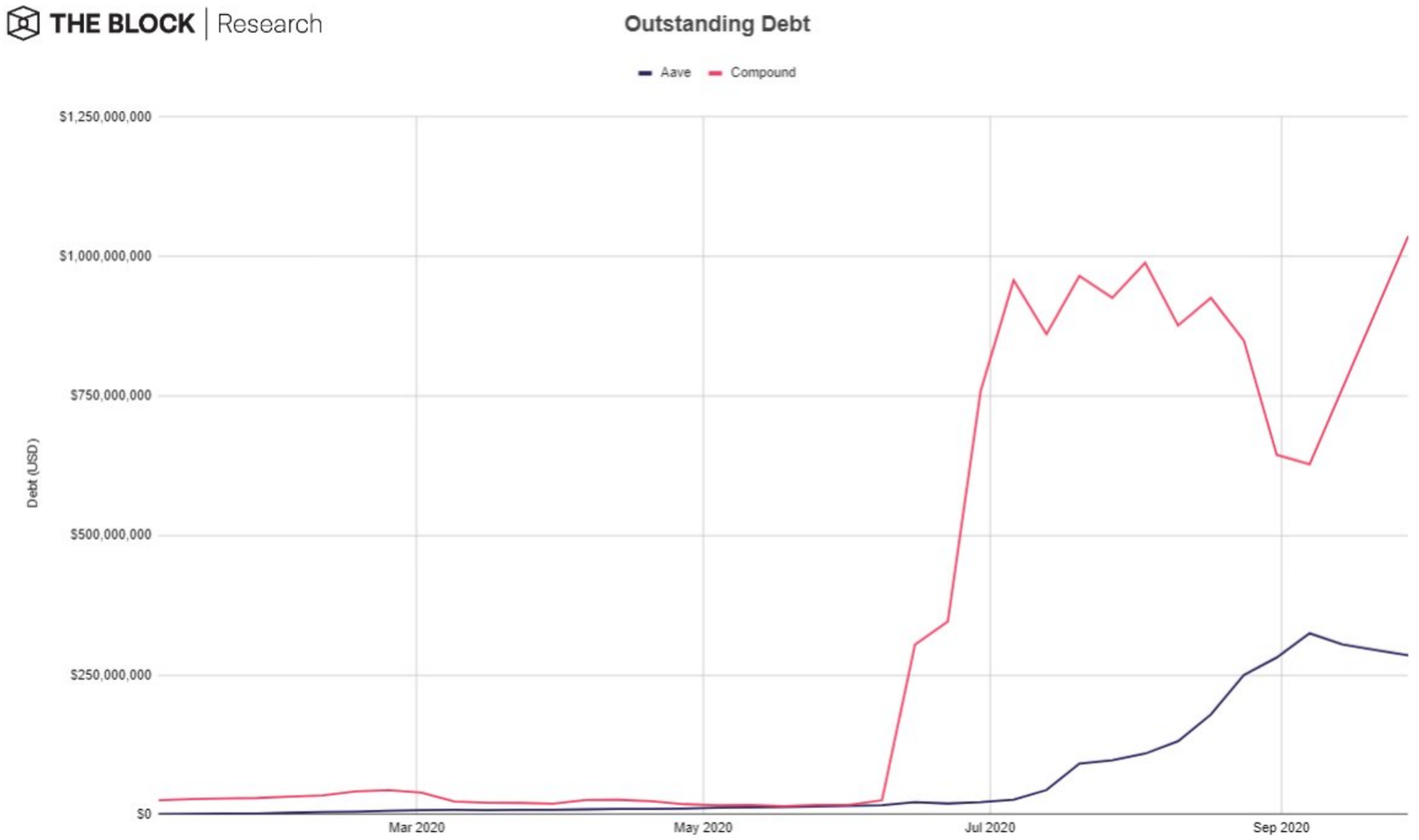
Case Study: Compound

Historical DAI Interest Rate



Case Study: Compound

Historical Borrow



Case Study: Compound

Mechanisms Not Discussed

- cTokens
- The reserve
- Governance
- Incentivized liquidity ("yield farming")
- Liquidation process
- Efforts to support undercollateralized lending

Q & A

Appendix

Links

- Uniswap Whitepaper
  - https://hackmd.io/@HaydenAdams/HJ9jLsfTz
- An Analysis of Uniswap Markets
  - https://arxiv.org/pdf/1911.03380.pdf
- Understanding Uniswap Returns
  - https://medium.com/@pintail/understanding-uniswap-returns-cc593f3499e

Appendix

Links

- Compound Whitepaper
  - https://compound.finance/documents/Compound.Whitepaper.pdf
- Compound Docs (can be used to lead you to Etherscan)
  - https://compound.finance/docs
- Compound Protocol Github
  - https://github.com/compound-finance/compound-protocol
- Compound Protocol Whitepaper Technicals
  - https://github.com/compound-finance/compound-protocol/blob/master/docs/CompoundProtocol.pdf

Optional Image Credit Line: URL Goes Here