Tracer Perpetual Swaps System Architecture

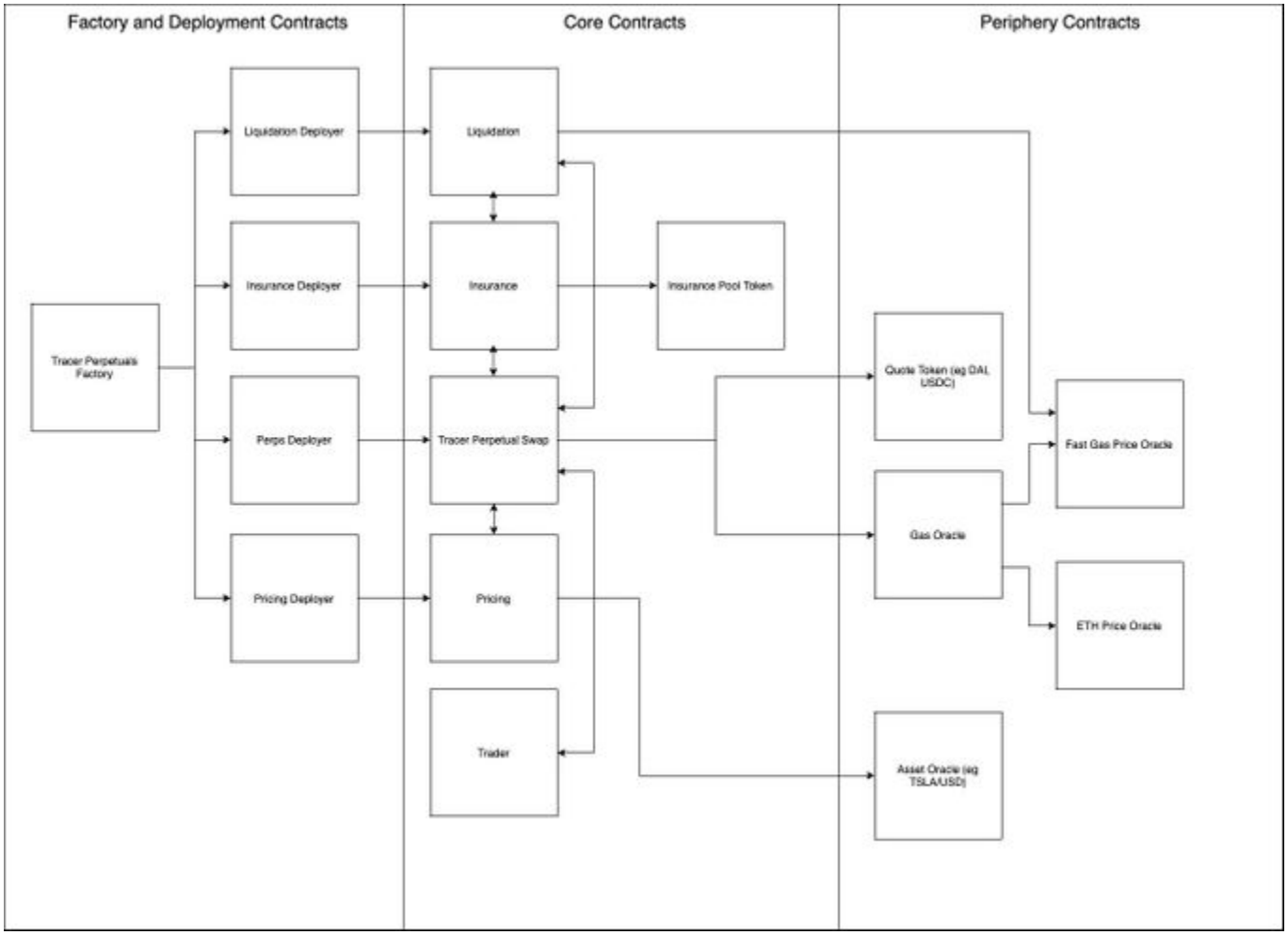
Introduction

The Tracer Perpetual Swaps smart contracts enable permissionless deployment of any perpetual swaps market, given a quote asset in the form of an ERC20, and a reliable price feed for the market.

Each market comes with all the infrastructure it needs to operate effectively in isolation to any other Tracer perpetual swap, including

- generalised trading interfaces
- a liquidation system
- a price system
- an insurance system

Architecture



The Tracer Perpetual Swaps contracts utilise a factory pattern in order to enable simple deployments of markets. In this pattern, you have a “factory” which is responsible for deploying instances of markets. In this sense, the factory and deployment contracts exist purely to support permissionless deployments of new Tracer markets and as such are not instrumental for the operations of the perpetual swaps themsleves, but are key to enabling open access to any market given a data feed.

The periphery contracts listed in the architecture diagram are instrumental pieces of each individual market, however are not stricly developed as part of the Tracer ecosystem. For a market to operate, an accurate price feed is required. Tracer initially plans on using Chainlink oracles to provide the fast gas oracle, Ethereum price oracle and asset oracle, however any contract conforming to the oracle interface may be provided.

Finally, the core contracts provide the perpetual swap functionality. Each Tracer market has its own instance of each and every core contract deployed on market creation. This allows each market to be securely isolated from all other markets, providing users cross market risk protection.

The following is a break down of the logic contained in each of the core contracts

TracerPerpetualSwap.sol

This contract holds the core logic around account state management, the execution of orders, and the enforcement of minimum margin requirements.

Most other contracts are connected to this contract as it performs the key functionality of the perpetual swap logic. All orders route through this contract.

Trader.sol

The trader contract is the current implementation of a Trading interface used in the system. See the section on generic trading interfaces for more. This contract implements EIP712 to enable an off chain order book to submit signed orders to the on chain contracts.

Pricing.sol

This contract manages all pricing state for a required market. It keeps track of all orders executed through the market and can be used to compute hourly and 24 hour averages, get the current fair price, and pull in oracle prices. This is instrumental in calculating a “fair price” for each and every market based on the perpetual price and the oracle price over time.

Insurance.sol

Insurance manages the end to end flow of the markets insurance pool and its underlying iTokens. Each insurance pool is used as insurance against each individual market, and provides protection against mass liquidations.

Liquidation.sol

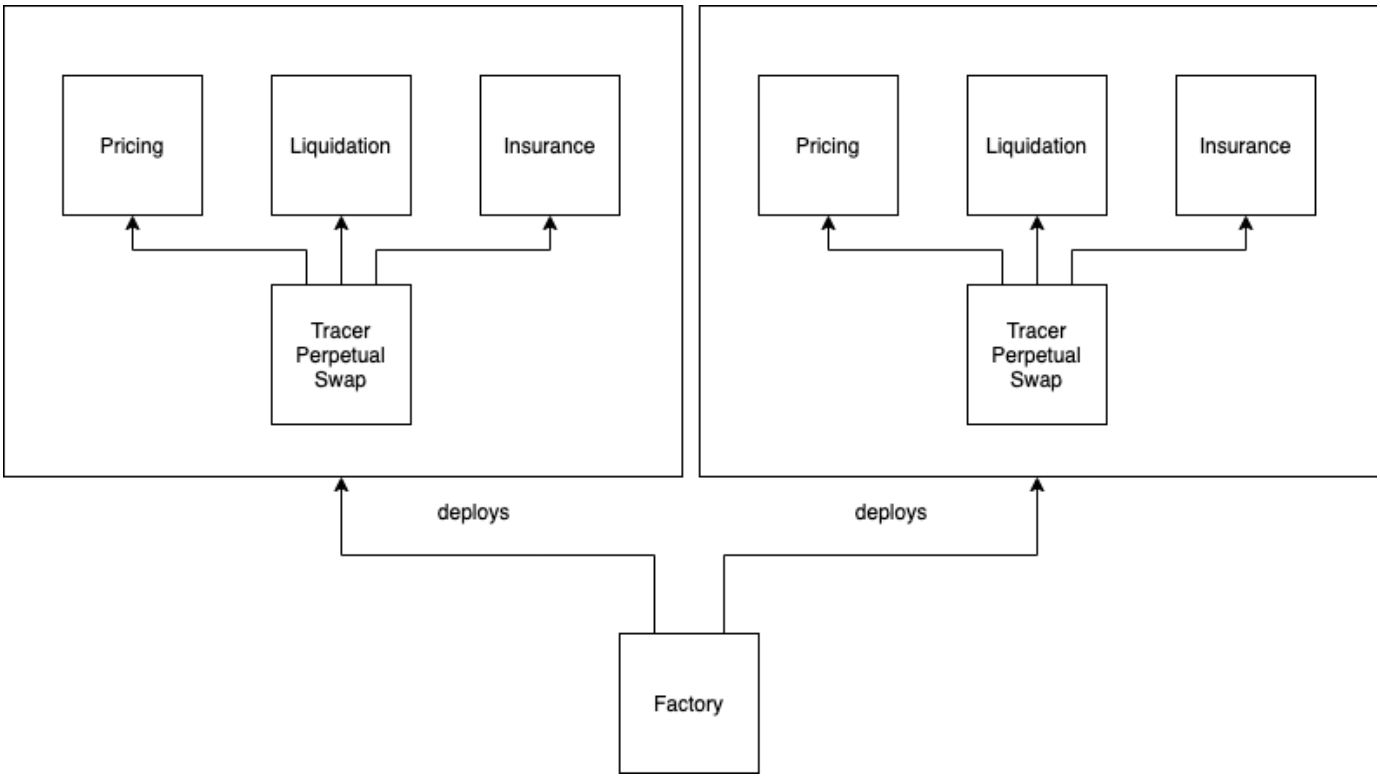
The liquidation contract manages the protocol’s liquidation mechanism, the provision of receipts to liquidators, and allows liquidators to claim slippage against these receipts, ensuring liquidation is always profitable for liquidators.

The Tracer liquidation mechanism is key in ensuring the tracer market is always correctly collateralised.

Security Model

At the heart of the Tracer Perpetual Swap contracts lies our security model. Our security assumptions for the contracts are that each individual market may be manipulated, due to the nature of permissionless deployments, however no cross market manipulation or risk should be possible. What this means for a user is, if you have deposited into a high volume market with a reliable price oracle, there is no way you may be affected by a user deploying a risky market with a manipulatable price oracle.

Each market has its own Insurance, Pricing and Liquidation state, everything you need to operate completely independently from all other markets, and state is never shared. This significantly reduces cross market risk as markets are never interacting with each other. The actual balances (account state) of each market is stored within the market itself, to allow for a more tightly coupled experience.



Trading Interfaces

One of the key pillars of Tracer is market access for all. This means that any user should be able to access any Tracer market at all times. No external system should be able to block them, no centralised entity should be able to censor them.

While this is the long term goal of Tracer, having fully accessible markets purely via smart contracts is a difficult problem to solve, due to the nature of requiring liquidity to ensure that perpetual prices are representative of the real asset price.

Having a single contract where users can simply submit any two orders and match them appears good on paper, however it leads to an array of various issues with the performance and security of our contracts.

The tradeoff between security and accessibility appears to be a bit of a catch 22, however we have proposed the following as a way around this.

Generalised trading interfaces

In our current contracts, each market owner may whitelist any trading interface they wish. Initially, Lion's Mane will develop trading interfaces that support EIP712 (see here for a write up on EIP712) and a perpetual swap specific AMM (coming soon ™). This however is not the extent of the trading experience Tracer can offer. By conforming to a simple coding interface, anyone may write their own trading contract and have it whitelisted. This means that should access to an order book or AMM be limited, new interfaces that support trading can simply be developed that are unable to be censored. This also allows us to expand the way traders interact with the protocol as the community grows, such as adding new order types.

Putting it together

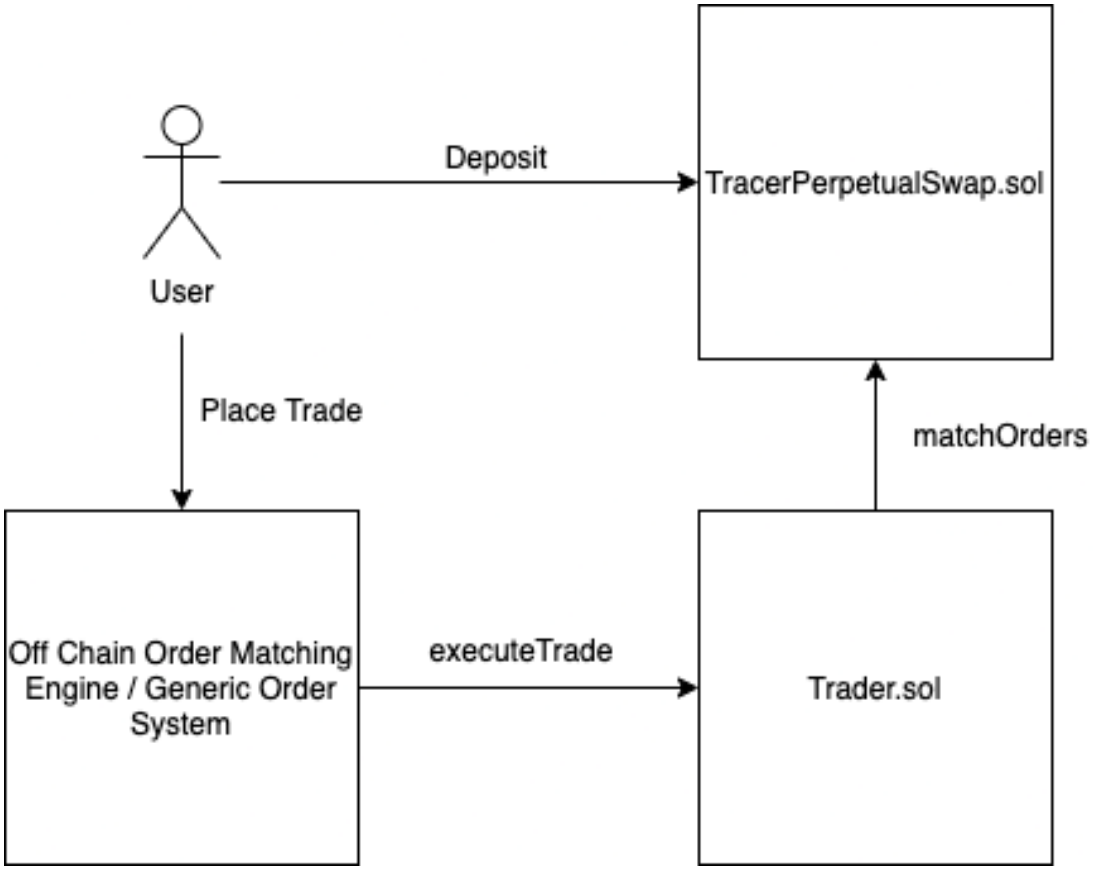
Give an example of a full E2E trading flow moving through all the contracts, with all actors (liquidators, external order books, etc)

Order Flow

The following are the steps taken by a user opening an order with a Tracer market.

1. Deposit into the market
    a. This calls deposit on the `TracerPerpetualSwap.sol` contract
2. Place an order with an external interface
    a. For this example, the off chain order matching engine, which takes in a Signed EIP712 compliant order.

    b. This implementation of a trading interface will match orders off chain, and submit a pair of orders to be executed.
3. Order is sent to the `Trader.sol` contract using the `executeTrade` function. Signature verification occurs and if passed, the order is executed on the `TracerPerpetualSwap.sol` contract.
4. The `TracerPerpetualSwap.sol` contract validates the orders, ensures that users are above margin after order execution occurs, emits events, and notifies the `Trader.sol` contract that the order was successful.



Liquidation Flow

The Tracer liquidation flow is unique in that it aims to provide as much capital back to the Trader as possible, while also maintaining riskless liquidations for liquidators in the system. This enables liquidations to be far more efficient and reliable than other systems. For more on the technicals of our liquidation system, check out the perpetual swaps whitepaper https://tracer.finance/media/whitepapers/perp-swaps/Tracer_Perpetual_Swaps.pdf.

There are a few variables that have to be defined in order to explain liquidation.

- `liquidationGasCost`: The cost, in quote tokens (usually a USD stablecoin), of calling `liquidate(…)`
    - This can calculated by multiplying the approximate GAS cost by the USD/Gas price. e.g. if the estimate is 300k gas, and 1 gas = $0.00003, 300,000*0.00003 = $9
- `trueMaxLeverage`: The maximum theoretical leverage an account can take. `trueMaxLeverage` scales between `lowestMaxLeverag` and `defaultMaxLeverage` as the insurance pool fills up to/empties out from a certain point
- `minimumMargin`: The minimum margin an account can have before somebody can call `Liquidation.liquidate(…)` on their account.
    - `minimumMargin = 6*LiquidationGasCost + notionalValue/trueMaxLeverage`
- `margin`: The total “value” of somebody’s account
    - `margin = quote + base * fairPrice`
- `escrowedAmount`: The amount of quote tokens the liquidator has to put into escrow. This is essentially their “fee” to take on an account’s position.
    - `escrowedAmount = max(0, margin - (minimumMargin - margin))`
        - where `margin` and `minimumMargin` refer to that of the liquidated (under-margined) account
    - Note that this amount decreases the further an account falls below minimum margin, and thus increases the profitability of a liquidation.

As soon as a user’s `margin` becomes less than `minimumMargin`, then anybody with sufficient collateral in a given market can call `Liquidation.liquidate(…)` with the under-margined account as the `account` parameter.

When this function is called, a few things happen, namely:

1. A “receipt” is submitted
    a. As part of this receipt, `escrowedAmount` is stored in escrow.
2. `msg.sender` gets `amount` (parameter) base units added to his position, and `totalQuote * (amount/totalBase)` quote tokens (where `totalQuote` and `totalBase` refer to the under-margined account’s total amount of quote and base respectively).

Receipts

Receipts can be summarised as follows:

- If a liquidator sells the position they liquidated, for a price lower than the `fairPrice` at time of liquidation, they can call `claimReceipt(...)` and receive a proportional amount of their `escrowedAmount` back as a slippage reimbursement.
- Any leftover `escrowedAmount` after a call to `claimReceipt(…)` is sent back to the liquidated account.
- If `claimReceipt(…)` is not called within 15 minutes, anybody can call `claimEscrow(…)` which send the escrowed amount back to the liquidated account.

Example

For example, let’s say you have two accounts, AccountA and AccountB

```
AccountA = {
    quote: $1000
    base: 0
}
AccountB = {
    quote: $-15000
    base: 7650
}
price = $2
trueMaxLeverage = 50x
liquidationGasCost = $20
```

AccountA has no positions open, but $1000 quote. AccountB is highly leveraged, with `margin = -15000 + 7650*2 = 300`, and a leverage of `notionalValue / margin = 15300 / 300 = 51x`. AccountB’s `minimumMargin = 6*20 + 15300/50 = 426` . `margin < minimumMargin` and therefore AccountB is liquidateable.

AccountA wants to liquidate AccountB’s entire position. This means AccountA will escrow `max(0, 300 - (426 - 300)) = max(0, 174) = 174`. In other words, they are getting this position, worth $300 margin, for $174.

So by the end of liquidation, The account states will look like

```
AccountA = {
    quote: $-14174
    base: 7650
}
AccountB = {
    quote: $0
    base: 0
}
```

In other words, AccountA has taken on AccountB’s position, and paid a $174 escrow amount.

Suddenly, the price drops to $1.90. The liquidator sells the whole 7650 base units he received, at the new price of $1.9, earning`7650 * 1.9 = $14535`. However, the price at time of liquidation was $2, meaning he would have earned `7650*2 = $15300`. They experienced slippage of `15300 - 14535 = $765`. The liquidator notices this, and calls `claimReceipt(…)`, providing their orders they sold with.

They will get reimbursed the $174 they escrowed. However, 174 is less than the $765 they experienced in slippage! To cover the remaining `765-174=591` of slippage experienced, the insurance pool comes in and covers the rest.

- If the amount of slippage experienced was *less* than the amount escrowed, the `escrowedAmount - slippageAmount` would have been sent back to the liquidator.
- If there was no slippage experienced, and the liquidator never called `claimReceipt(…)`, 15 minutes after liquidation the liquidated account (or anybody) would have been able to call `claimEscrow(…)` and the liquidated account would receive `escrowedAmount`.

Deleveraging

`TracerPerpetualSwaps.sol` has a function `trueMaxLeverage()`. This calculates the current theoretical max leverage a user can have. At any point in time, `lowestMaxLeverage <= trueMaxLeverage() >= defaultMaxLeverage`. This amount scales proportional to how full the insurance pool is.

- When the insurance pool is `deleveragingCliff%` full, and above, `trueMaxLeverage() == defaultMaxLeverage`
- When the insurance pool is `insurancePoolSwitchStage%` full, and below, `trueMaxLeverage() == lowestMaxLeverage`
- When the insurance pool is between `insurancePoolSwitchStage%` and `deleveragingCliff%` full, `trueMaxLeverage()` scales linearly between `lowestMaxLeverage` and `defaultMaxLeverage`
    - This linear equation is `y = mx + b = (defaultMaxLeverage - lowestMaxLeverage)/(deleveragingCliff-insurancePoolSwitchStage) * percentFull + [lowestMaxLeverage - ((defaultMaxLeverage - lowestMaxLeverage) / (deleveragingCliff - insurancePoolSwitchStage))`