traditional-price-oracles)

1-Click Price Oracle

Natively integrated price oracles available on 1-Click Rollups.

Traditional Price Oracles

Traditionally, price oracles are deployed on a per network basis (on each Rollup individually). This manual deployment process is cumbersome, expensive, and may require legal agreements to be signed. Whilst adequate for monolithic L1s, It is antithetical to the fully self-service and automated 1-Click Rollup deployment process. In addition, traditional oracles treat the networks they serve as disparate entities, completely isolated from one another. Consequently, oracles perform duplicate computation for each chain…

1. Price data is fetched from the validator network and published to each chain individually, despite the prices being the same. Furthermore, the accounts (EOAs) executing these transactions incur gas and must all be topped up periodically.
2. Upon receiving fresh price data, each chain independently verifies the signature generated by the validator network to ensure the data is legitimate.

The traditional oracle architecture, based on the assumptions of isolated blockchains, is unscalable ina world of 1000s of Rollups . Whilst monolithic L1s are in fact isolated, this is not the case for Rollups which do have something in common - the settlement layer.

Gelato 1-Click Price Oracle Module

Introducing the Gelato 1-Click Price Oracle Module which is natively integrated into the OP Stack and immediately available out of the box for all 1-Click Base L3s. Prices are published to- and verified by a single Rollup (Hub) and disseminated to all other Rollups (Spokes) whilst avoiding the aforementioned duplicate computation and all associated overhead.

The key insight making this possible is that Rollups essentially act as indexers over the L1 allowing us to consume data such as emitted events in a process known asderivation . The derivation layer is a crucial component of the Rollup architecture as it facilitates, among other things,deposits which are initiated on the L1 and executed on the L2. This is what allows users on the L1 to securely deposit tokens to the L2 - one of the key Rollup innovations.

The term L1 simply refers to the Rollupssettlement layer . This may itself be a Rollup with its own settlement layer. In our case, L1 actually refers toBase (which is an L2) and L2 to Rollups settling onBase (which are L3s). Here's how a deposit works under the hood:

1. The user invokes a smart contract which locks up tokens on the L1 and emits adeposited
2. event.
3. Whenever the Rollup enters a new epoch (new L1 block), it iterates through transaction receipts in that block and fetches alldeposited
4. event logs.
5. For alldeposited
6. event logs, the Rollup constructs special, corresponding deposit transactions which are included at the top of the next L2 block.

Steps 2 & 3 take place during thederivation process. The Gelato 1-Click Price Oracle Module deploys a forked and slightly modified OP Stack that introduces an additionalsource to the derivation layer. We first deploy anL1PriceOracle smart contract which receives price updates fromPyth every two seconds via aGelato Web3 Function . This smart contract is deployed once on the L1 and shared by all Rollups. It verifies the associated signature to ensure the price update is legitimate and emits aPricesUpdated event containing all updated price feeds. When the Rollup enters a new epoch, in addition to iterating through alldeposited event logs as before, it also iterates through allPricesUpdated event logs. Just like with deposits, the Rollup then constructs a special top-of-block transaction which updates prices on the L2. This top-of-block transaction calls aL2PriceOracle predeploy smart contract which stores prices on-chain and exposes them via aPyth-like interface .

In summary, prices are pushed to- and verified by a single smart contract living on the L1. On every epoch, Rollups essentially copy the already verified, fresh prices from L1 to L2 during derivation - similar to how deposits are handled. Since publishing & verification is done in a single place, rather than by each Rollup individually, we reduce its complexity from O(n) to O(1).

?

However, despite avoiding duplicate computation, publishing prices to an L1 such as Ethereum every two seconds is

prohibitively expensive. Instead, prices are published to a cheaper L2, in our caseBase , and consumed by 1-ClickBase L3s. Unlike traditional oracles, 1-Click Oracles actually perform better at scale since the costs of publishing & verifying the data is amortized between all Rollups. In essence, the constant cost is divided between many Rollups (1/n). In addition, whenever a new price feed is introduced on the Hub (Base), it is instantly available on all other Rollups without any additional configuration work required. This allows for the rapid introduction of new price feeds.

The diagram below visualises the architecture and illustrates the flow of an oracle price update:

1. An off-chainGelato Web3 Function
2. fetches prices fromPyth
3. every two seconds
4. . The Web3 Function implementation is availablehere
5. .
6. The fresh prices are pushed to theL1PriceOracle
7. smart contract on Base Sepolia. The smart contract implementation is availablehere
8. .
9. TheL1PriceOracle
10. smart contract calls thePyth
11. smart contract to verify the signature & update the price feed. This updated price feed is then subsequently fetched from thePyth
12. smart contract and emitted in aPricesUpdated
13. event.
14. When the Base L3 Rollup produces a new block in a new epoch (whenever there is a new Base L2 block), it indexes thePricesUpdated
15. event emitted by theL1PriceOracle
16. smart contract during derivation. The emitted event data is decoded and a system transaction is constructed to update theL2PriceOracle
17. smart contract on the L3 with the fresh prices.
18.

?

Previous Experimental Next Supported Price Feeds Last updated2 months ago