TL system design

Summary:

Requirements:

1. Git, multiple commit, public accessible link ✅
2. Test code ✅
3. Dockerize with docker-compose ✅
4. README
    a. Build
    b. Test
    c. Run
    d. Mention Architectural Principles, why choose them
    e. Design considerations
    f. Coding styles, pep 8 for python ✅ (pycharm plugin install)

Exercise:

1. Keep track of all transactions involved in the Uniswap V3 USDC/ETH pool
    a. Transaction fee in USDT when Txn first confirmed on the blockchain
2. **Real time data recording + Historical batch data recording**
    a. Continuously record live txn data
    b. Process batch job requests to retrieve historical Txns for a given period of time
    c. REST API:
        i. Input: txn hash, Output: transaction fee
        ii. State clearly Interface specifications (Follow Swagger UI standards)
3. Data Source:
    a. All Historical Txns for Uniswap WETH-USDC
        i. https://etherscan.io/address/0x88e6a0c2ddd26feeb64f039a2c41296fcb3f5640#tokentxns.

    b. Get historical Txns Programmatically: Etherscan API:

        i. https://docs.etherscan.io/api-endpoints/accounts#get-a-list-of-erc20-token-transfer-events-by-address

    c. Eg. 0x8395927f2e5f97b2a31fd63063d12a51fa73438523305b5b30e7bec6afb26f48

    d. Base fee in ETH, to calculate Fee in USDT based on historical/live price for ETH/USDT

        i. Use Binance SPOT API for price: Orderbook/kline

        ii. https://binance-docs.github.io/apidocs/spot/en/#change-lo

4. Additional Requirement:

    a. **Availability, Scalability, Reliability considerations**

    b. **Decode the actual Uniswap price exectuted for each Txn.**

        i. Single Txn may contain multiple swaps. You only need to decode and save the executed price from Uniswap event. Provide REST API.

            1. Infura, Web3 packages (web3Py)

Tech Stack:

Python: Rich Library, data processing tools, SDK from Binance

DB: Redis, duplicate requests, rate limit implementation, binance api data caching, can be implemented as message queue if needed for real time streaming + fast processing.

Docker: As specified, for containerization

FastAPI (python framework): Pydantic, async support, auto generation of documentation (swagger), for high performance

POC:

1. Etherscan API ✅

    a. **Require API Key: (Store in .env file), to be replace**

    b. Rate Limit: 5 QPS, 100k Per Day

2. Binance Spot API (public) ✅

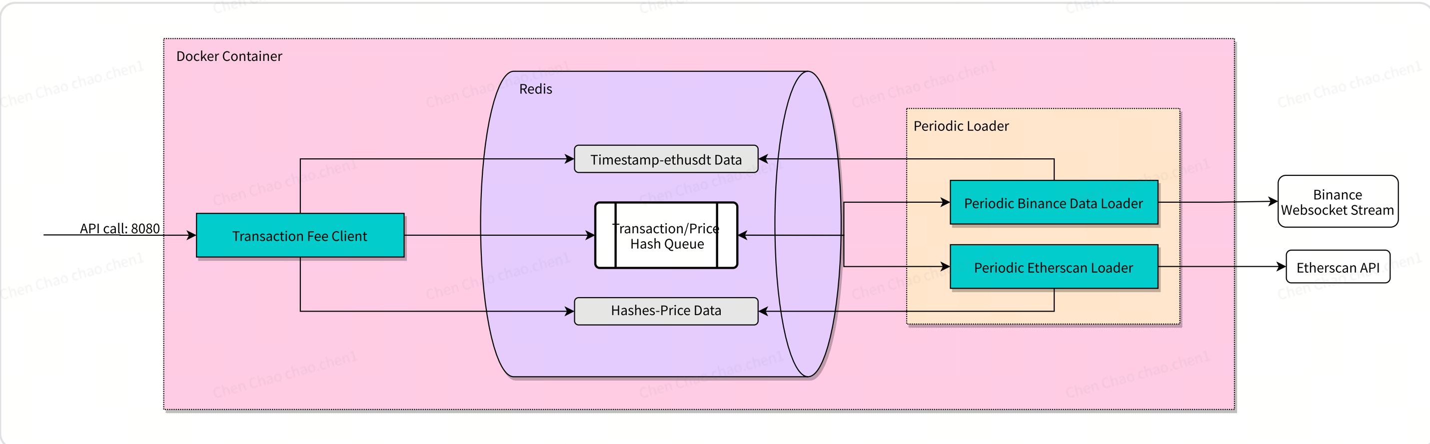
    a. Order Book (depth API)

    b. Kline (kline API)

    c. Rate limit: 1,200 requests per minute, or up to 1,800 requests per minute in bursts (QPS 20)
3. FastAPI ✅
    a. Set up an app
4. Dockerize ✅
    a. Setup Redis
    b. Run App with redis as a dependency
    c. Docker File
    d. Docker compose

High Level TD

Architectural Diagram



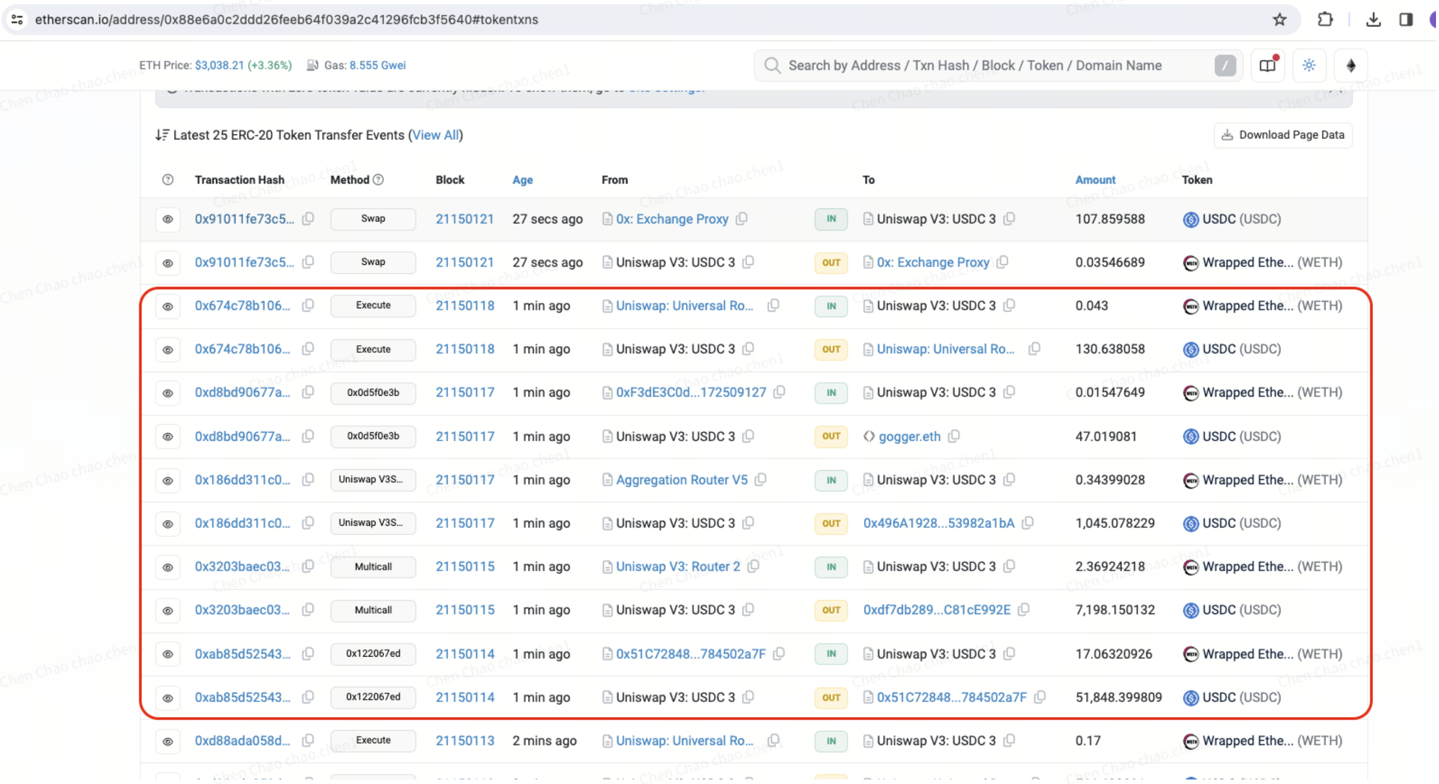
Components

1.1 Periodic Transaction Loaders

1.1.1 Etherscan Transaction Loader

We will have a transaction loader that periodically loads data from etherscan by calling the etherscan API. This loader needs to adhere to the rate limit that etherscan imposes. Based on the 100k per day requirement, we can roughly call the api at 1qps to check for new transactions. This loader will start loading on app start up, polling the api at an interval of 1s. We can also cache the latest transaction block number for future calls. Based on transaction observation, we can roughly know that this part is non-data intensive as we barely hit 5 transactions in a minute. There is also an upper bound on the ethereum chain itself which comprises of the gas fee. Higher transactions lead to higher gas leads to lower transactions.

This recursive nature will lead to a theoretical upper bound to exist. Black swan can happen. For now, we can assume that data is rather small.


etherscan.io/address/0x88e6a0c2ddd26feeb64f039a2c41296fcb3f5640#tokentxns

ETH Price: $3,038.21 (+3.36%) Gas: 8.555 Gwei

Search by Address / Txn Hash / Block / Token / Domain Name

Latest 25 ERC-20 Token Transfer Events (View All)

Download Page Data

Transaction Hash	Method	Block	Age	From		To	Amount	Token
0x91011fe73c5...	Swap	21150121	27 secs ago	0x: Exchange Proxy	IN	Uniswap V3: USDC 3	107.859588	USDC (USDC)
0x91011fe73c5...	Swap	21150121	27 secs ago	Uniswap V3: USDC 3	OUT	0x: Exchange Proxy	0.03546689	Wrapped Ethe... (WETH)
0x674c78b106...	Execute	21150118	1 min ago	Uniswap: Universal Ro...	IN	Uniswap V3: USDC 3	0.043	Wrapped Ethe... (WETH)
0x674c78b106...	Execute	21150118	1 min ago	Uniswap V3: USDC 3	OUT	Uniswap: Universal Ro...	130.638058	USDC (USDC)
0xd8bd90677a...	0x0d5f0e3b	21150117	1 min ago	0xF3dE3C0d...172509127	IN	Uniswap V3: USDC 3	0.01547649	Wrapped Ethe... (WETH)
0xd8bd90677a...	0x0d5f0e3b	21150117	1 min ago	Uniswap V3: USDC 3	OUT	gogger.eth	47.019081	USDC (USDC)
0x186dd311c0...	Uniswap V3S...	21150117	1 min ago	Aggregation Router V5	IN	Uniswap V3: USDC 3	0.34399028	Wrapped Ethe... (WETH)
0x186dd311c0...	Uniswap V3S...	21150117	1 min ago	Uniswap V3: USDC 3	OUT	0x496A1928...53982a1bA	1,045.078229	USDC (USDC)
0x3203baec03...	Multicall	21150115	1 min ago	Uniswap V3: Router 2	IN	Uniswap V3: USDC 3	2.36924218	Wrapped Ethe... (WETH)
0x3203baec03...	Multicall	21150115	1 min ago	Uniswap V3: USDC 3	OUT	0xdf7db289...C81cE992E	7,198.150132	USDC (USDC)
0xab85d52543...	0x122067ed	21150114	1 min ago	0x51C72848...784502a7F	IN	Uniswap V3: USDC 3	17.06320926	Wrapped Ethe... (WETH)
0xab85d52543...	0x122067ed	21150114	1 min ago	Uniswap V3: USDC 3	OUT	0x51C72848...784502a7F	51,848.399809	USDC (USDC)
0xd88ada058d...	Execute	21150113	2 mins ago	Uniswap: Universal Ro...	IN	Uniswap V3: USDC 3	0.17	Wrapped Ethe... (WETH)

1.1.2 Binance Spot API Loader

Note that we need to return the values stored as USDT, since what we retrieve is the ETH gas price, a conversion needs to be done. We can use the Binance spot API for that by looking at the ETH/USDT pair price. The product of the two would give the transaction price in USDT. This can be slightly data intensive. As from second to second, the price change can be quite drastic, we would want to reach second level data to achieve high precision. Since etherscan data can return epoch level timestamp, we can then use this data to retrieve the exact price of the Binance Spot pair ETH/USDT at that exact second.

The challenging part about this arises from retrieving the kline data in near real time manner, since second level data is streaming in, if we employ a rest api (pull model), the network overhead will cause a significant lag in data retrieval. We should instead use a push model by employing the binance websocket. We can, of course, do a rough estimation using minute level data, but the correctness of the price will be an issue, given the market can move very drastically within a minute.

Lets do a rough estimation. We would need to store the timestamp and the closing price, which will be two separate strings. In a single day we would have 24*60*60 = 86400 of data, each string stored in redis has roughly 80 bytes of overheads as well. We roughly assume it to be 200 bytes per two strings. Then, we will have about 17MB of data every day, which brings it to . In theory, we roughly want to have at most 25GB of data in a single redis node. Beyond that point, performance will start to degrade. For simplicity sake, we assume that our redis at most holds about 1 year worth of data, and anything beyond that point expires and cleans up itself.

Everything beyond this point, if needed, we can just query the api and cache that data in the redis, in case some other people want to use it in the future.

Discussion point as a follow up: If we want to scale this up to store all this data in memory, we can potentially have two options.

a. Redis cluster, we can horizontally scale the size by adding more redis nodes to form a redis cluster.

b. Utilize big data technology, eg, flink, to achieve real time data pipelines.

c. Just persist the data and query from harddisk works just fine. From there on, it is easy to scale with sharding based on dates.

1.2 Transaction/Price Hash Queue

Note that it is not always the case that a transaction/price is going to be in our redis after we start running the app. For simplicity sake, we would want to query this data as a batch job and inform the API caller to come back later on if the data is missing. We can simply use a message queue for this. One safe assumption is that this is probably not a critical service, otherwise it's probably better to use a log and persist for this job request, in this case for easier tracing. The caller of the API should also have fallback logic anyways.

As for implementation, redis provides publishing-subscribe architecture for real time low latency event processing. This is one of the considerations being used when choosing redis as part of the tech stack, since we can use it both for event streaming queues and data persistence.

The periodic loaders will also contain implementations to retrieve the data from the message queue, depending on what kind of data is missing.

For simplicity, we can always decouple the message consumption and job to separate classes. For now, there is no need to do so, since they are just loading data from the two separate data sources.

1.3 Transaction Fee Client

We will have a single API endpoint for this purpose. The API endpoint description is illustrated below. Calculation is done here directly

URL	POST /transaction-fee		
Input	**Type**	**Description**	**Sample**

transactionFeeRequest	TransactionFeeRequest	Request DTO
		transactionHashes: list[str] (Mandatory)
			TransactionFeeRequest:
			{
			transactionHashes: ["0x123...", "0x456...", ...]
			}

Output	**Type**	**Description**	**Sample**

transactionFeeResponse	TransactionFeeResponse	Response DTO
		transactionsList : list[TransactionsDTO]
		
		TransactionsDTO
		transactionHash: str
		success: boolean
		fee: float
		error_msg: string
			TransactionFeeResponse:
			{
			transactionsList: [
			{
			transactionHash: "0x123...",
			success: true,
			fee: 0.123
			error_msg: ""
			},
			{
			transactionHash: "0x456...",
			success: false,
			fee: 0
			error_msg: "invalid hash"
			},
			...
			}

Design considerations:

1. Scalability

- **Horizontal Scaling of Redis**: Redis is used to store price and transaction data temporarily, making it suitable for quick access. Redis can be scaled horizontally by deploying a Redis cluster, distributing the load across multiple nodes to handle an increased volume of transactions or data requests.

- **Batch Processing for Historical Data**: The system distinguishes between real-time transaction tracking and historical data retrieval. Batch processing jobs for historical data allow efficient handling of large data sets without affecting real-time processing capabilities, ensuring the system can handle both high transaction volumes and large historical queries.
- **WebSocket for Real-Time Data**: The Binance WebSocket is used for fetching real-time ETH/USDT prices, which minimizes the need for repetitive REST API calls and reduces network overhead. This setup enables the system to scale efficiently while maintaining real-time responsiveness. We can extend this idea to the evm clients and etherscan apis
- **Event-Driven Architecture**: Leveraging an event-driven approach (e.g., using Kafka or RabbitMQ) for processing transaction data and price updates would enable asynchronous processing. For example, the Etherscan transaction loader and Binance price loader could publish events when new data is fetched, and downstream services (e.g., a fee calculation service) could consume these events to perform necessary computations without blocking.

2. Availability

- **Docker and Docker Compose**: Using Docker for containerization ensures a consistent runtime environment and simplifies deployment across multiple servers. Docker Compose helps manage service dependencies like Redis and the API server, facilitating fast recovery and replication in case of a failure.
- **API Rate Limiting and Caching**: Given Etherscan and Binance have API rate limits, implementing caching with Redis and rate-limiting logic ensures that the system doesn't exceed limits and can handle request bursts by serving cached responses.

3. Reliability

- **Data Redundancy in Redis and Message Queues**: Redis provides low-latency data storage for transaction data and pricing, and using a message queue for asynchronous task processing ensures that transaction processing requests are not lost even if components go down temporarily.
- **Fallback Logic for Missing Data**: If transaction or price data is missing, the system can queue requests and notify the client to check back later. This approach allows the system to handle failures gracefully without dropping requests, enhancing reliability.
- **Error Handling and Retry Mechanism**: The system should include robust error handling for API interactions, especially with rate limits and connectivity issues. Implementing retry mechanisms and exponential backoff strategies ensures that temporary failures do not lead to permanent data loss or unhandled exceptions.

These design choices allow the system to handle large volumes of data reliably, adapt to demand variations, and maintain high availability for continuous operation.

Further Enhancements Possibilities

1. Scalability

- **Database Partitioning and Sharding**: If the transaction or price data exceeds Redis’ s capacity or the overall storage requirements grow, database partitioning can help. Sharding based on transaction date or type would allow efficient distribution across multiple storage nodes, thus handling larger data volumes while maintaining query performance.
- **Microservices Architecture**: Dividing the system into smaller, focused microservices—such as one for transaction retrieval, another for fee calculation, and another for data transformation—enables independent scaling. Each microservice can then be scaled horizontally as required, allowing the system to handle increased loads without affecting overall performance.
- **Auto-Scaling for API Instances**: Setting up auto-scaling based on traffic allows the REST API to dynamically adjust to fluctuating workloads. During high-demand periods, more instances are deployed; during low-demand times, resources scale down to optimize costs.

2. Availability

- **Load Balancing**: The REST API endpoints can be scaled by deploying multiple instances behind a load balancer, ensuring that if one instance fails, the load balancer can redirect requests to healthy instances, enhancing availability.
- **Failover and Replication in Redis**: Configuring Redis with primary-replica replication ensures that if one instance fails, another replica can immediately take over. Redis Sentinel or Redis Cluster setups can further automate failover management to enhance the service's availability.

3. Reliability

- **Data Persistence and Backups**: While Redis offers quick access to recent data, for long-term reliability, essential data should also be stored in a persistent database (e.g., PostgreSQL, MongoDB). Routine backups of transaction data ensure that historical data is never lost and can be restored if the main storage fails.
- **Monitoring and Alerting**: Implementing robust monitoring and alerting is crucial. Tools like Prometheus and Grafana can track application metrics (e.g., request latency, error rates, memory usage). Automated alerts ensure rapid response to system issues, supporting reliability and proactive troubleshooting.