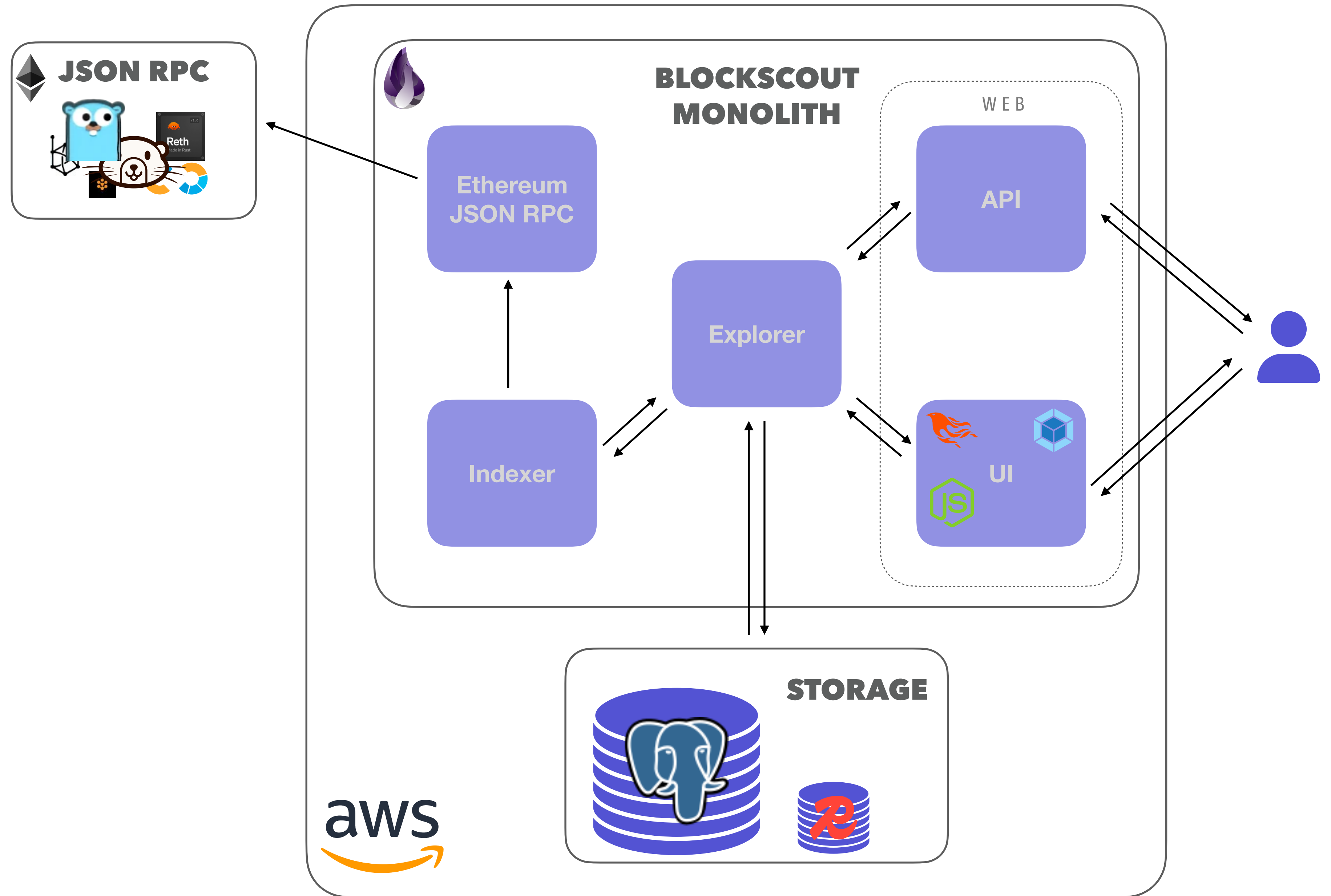


# **BLOCKSCOUT INDEXER ARCHITECTURE OVERVIEW**

VIKTOR BARANOV  
BLOCKSCOUT OFFSITE EVENT,  
ISTANBUL, OCTOBER 2024

# V1 ~~PALEOLITH~~ MONOLITH EPOCH

2018 - 2022



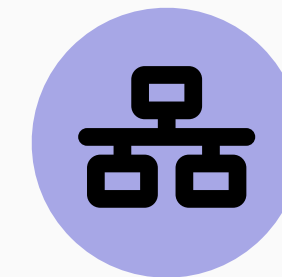
# BLOCKSCOUT V2.0 MIGRATION



FROM CLOUD TO  
ON-PREMISE

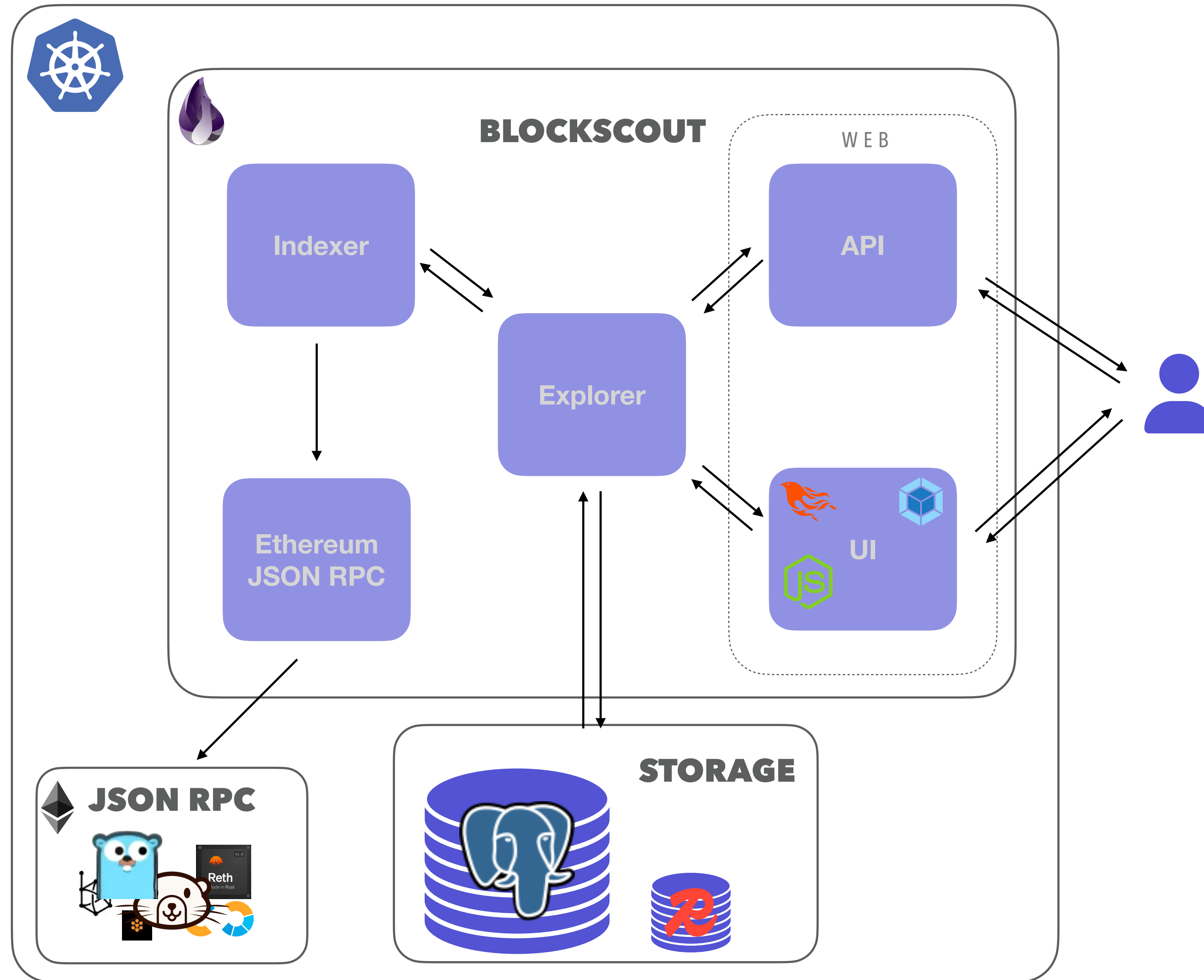


FRESH LOOK &  
FILL

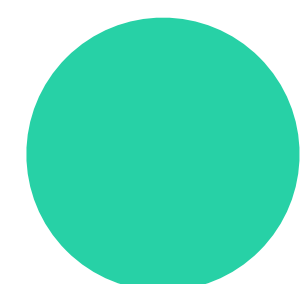


MICROSERVICE  
ARCHITECTURE

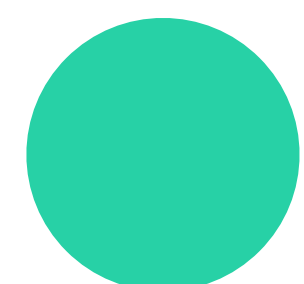
# V2



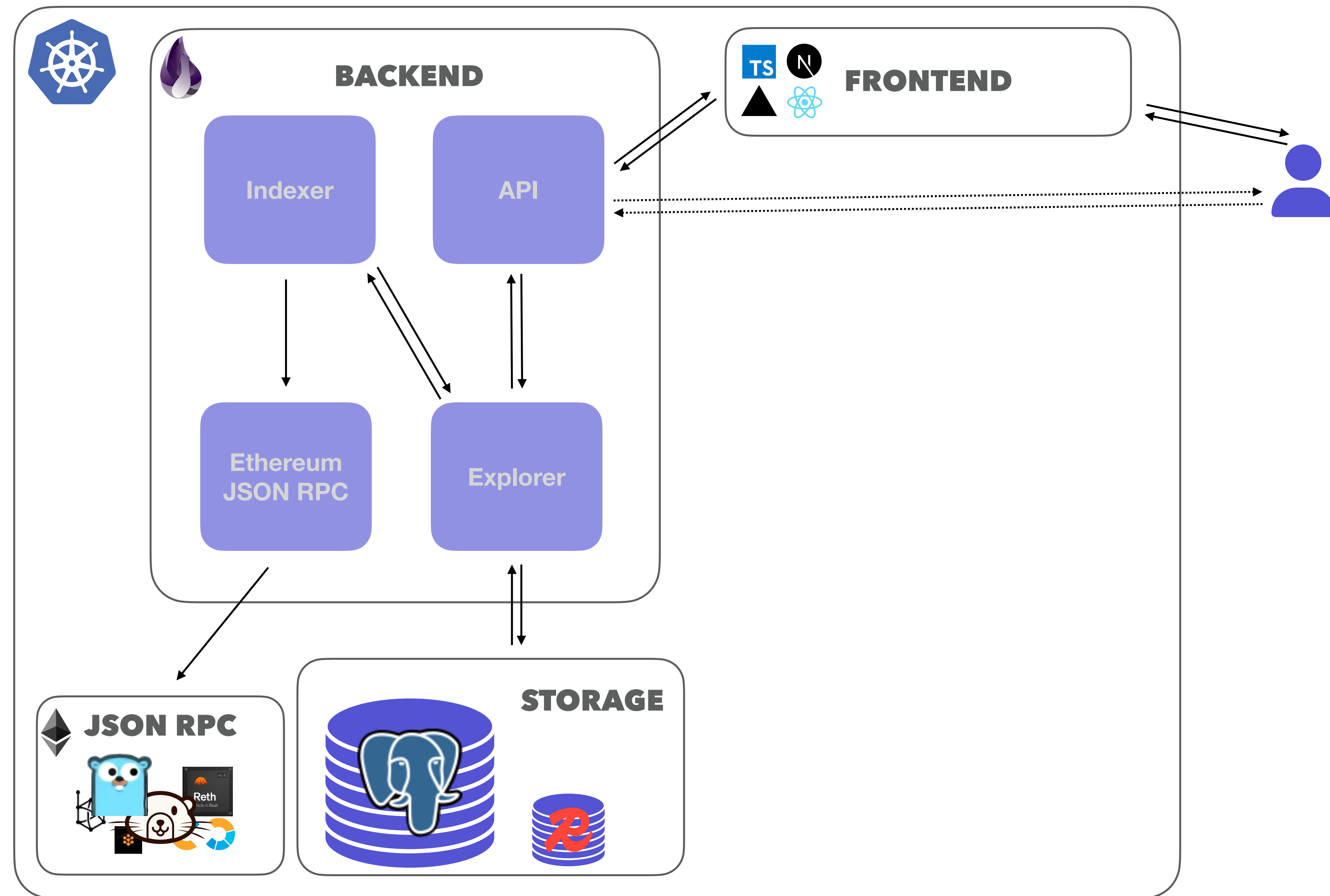
# V2



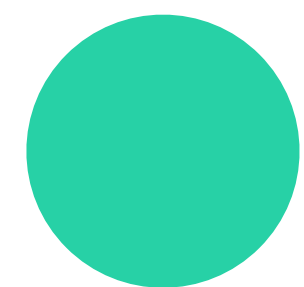
K8S



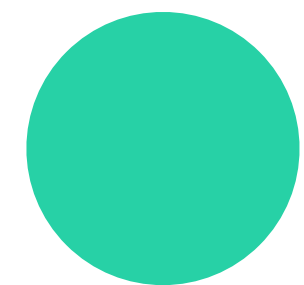
NEW FRONTEND



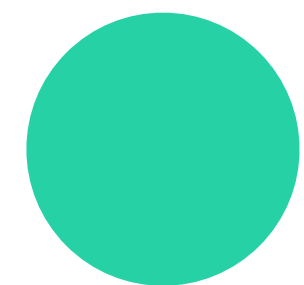
# V2



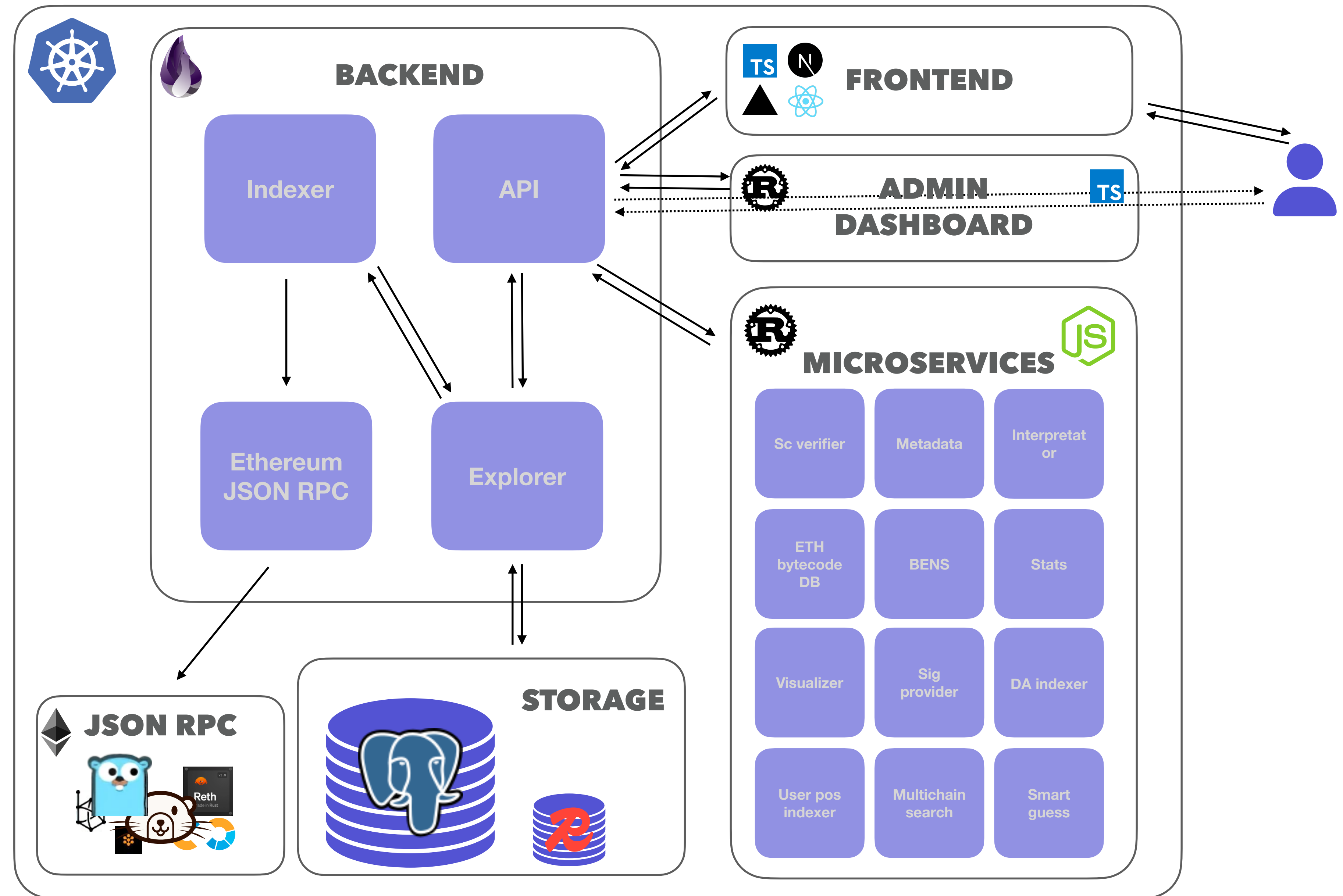
K8S



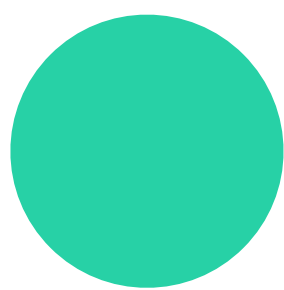
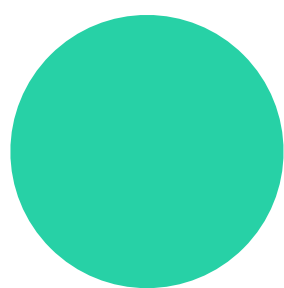
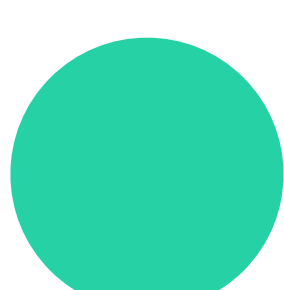
NEW FRONTEND

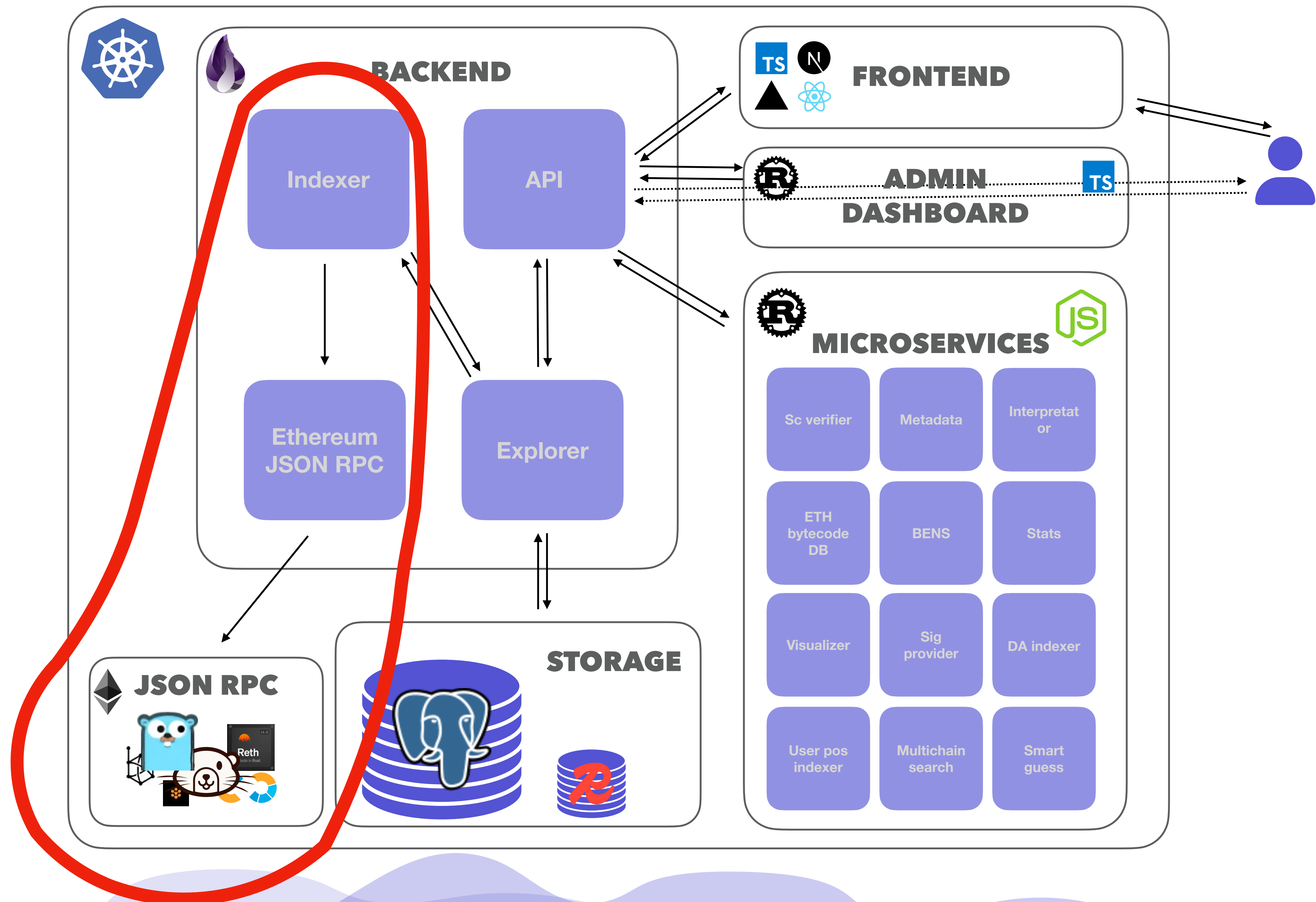


MICROSERVICES



# V2

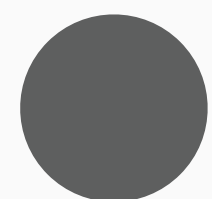
-  **K8S**
-  **NEW FRONTEND**
-  **MICROSERVICES**





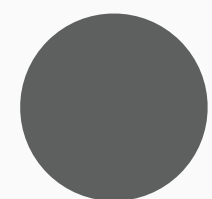
# BLOCKSCOUT INDEXER

## MAIN INDEXERS



### REALTIME

Import of new blocks data from the head of the chain

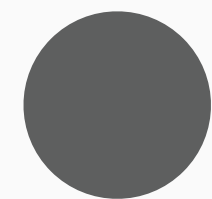


### CATCHUP

Import of blocks data down the chain

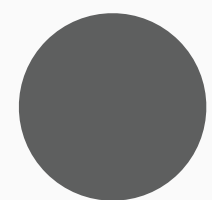
Both main indexers comprises from synchronous and asynchronous parts.

## SECONDARY FETCHERS



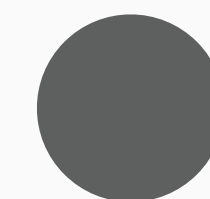
### REGULAR

- Internal transactions
- Pending transactions
- Dropped/Replaced transactions
- Contract bytecodes
- Block rewards
- Token catalog
- NFT instances
- Token/coin balances
- Uncles



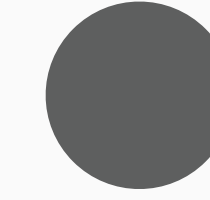
### ON-DEMAND

- Coin/token balances update
- Contract bytecodes fetch/re-check
- Contract source codes lookup
- NFT instance metadata re-fetch
- Token total supply



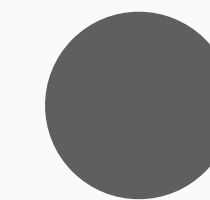
### OFF CHAIN INTEGRATIONS

- Coin / token price, market cap, tvl sources: CMC, Coingecko, Cryptorank, Defillama
- Sc verification: ETH bytecode DB, Sourcify
- Data enrichment: ENS names, public tags, AI interpreters (own, Noves Fi), sc security scanners (Solidityscan), assets portfolio (Zerion)
- Chain initialization data import: pre-mined coins, precompiled smart-contracts



### CHAIN-SPECIFIC FETCHERS

There is a dozen of chain-specific data fetchers including arbitrum, optimism, polygon edge, polygon zkevm, zksync etc.



### TEMPORARY FETCHERS

They work once in order to resolve possible data inconsistency.

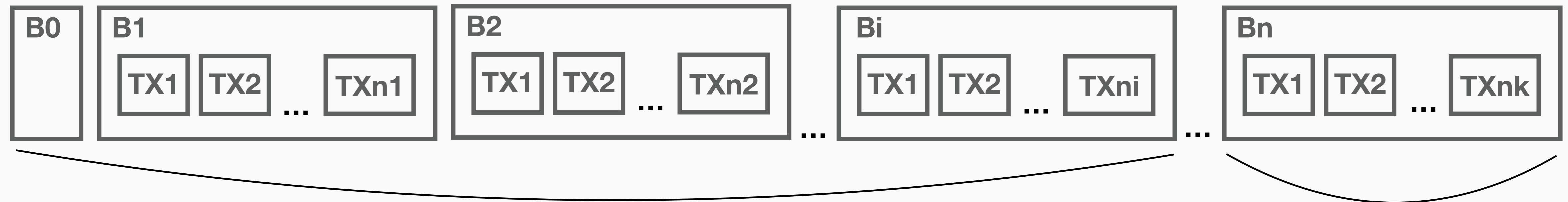


# BLOCK IMPORT

TAIL

BLOCKCHAIN

HEAD



## CATCHUP INDEXER

- At application start it calculates which block numbers are missing in the DB and generates missing block ranges which catchup indexer should fill.
- It processes blocks in batches. The batch size and concurrency are managed via env variables.



If catchup indexer failed to process block (due to reaching timeout), it is moved to another queue aka «massive blocks fetcher».

```
curl -X POST --data '[
{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["0x...", true],"id":1},
{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["0x...", true],"id":2},
...
{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["0x...", true],"id":n}
]'
```

## REALTIME INDEXER

- Connect via subscription to websockets
- Poll by HTTP requests

```
wscat -c wss://... -X
'{"jsonrpc":"2.0", "id": 1, "method":
"eth_subscribe", "params":
["newHeads"]}'
```

```
curl -X POST --data
'{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":1}'
```

**LET'S DIVE INTO  
SYNCHRONOUS PART OF  
BLOCK IMPORT...**



# BLOCK IMPORT (SYNCHRONOUS PART)

- Once **blocks** info is fetched, info about **uncles** is extracted (uncle has to nephew hashes mapping and uncles index).
- And now it checks, if the chain follows PoA «clique» consensus algorithm, than it transforms block miner address by recovering its public key from the header and extra data.
- For each block in the requested numbers range Blockscout fetches **transaction receipts** in batches
- Every tx receipt contains the list of **logs**. Blockscout parses those logs into insertable changesets.
- Then, **token transfers** are extracted from the logs. Transfers are parsed based on logs combination of topics.
- Together with transfers, **tokens'** addresses are extracted from the logs.

```
curl -X POST --data '[
{"jsonrpc":"2.0","method":"eth_getTransactionReceipt","params":["0x...", "id":1},
...
{"jsonrpc":"2.0","method":"eth_getTransactionReceipt","params":["0x...", "id":n}]'
```

💡 Block import steps also contain chain-specific requests, transformations. I do not mention them for simplification of the overview.

ERC-20 or ERC-721 (topic 0="0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef" == keccak256(Transfer(address,address,uint256)) )

ERC-20 (FT) topic 1 is not empty (to) && topic 2 is not empty (from) and amount is from log's data.

ERC-721 (NFT) topic 1 is not empty (to) && topic 2 is not empty (from) && topic 3 is not empty (instance\_id).

## ERC-1155

- single transfer (topic 0="0xc3d58168c5ae7397731d063d5bbf3d657854427343f4c083240f7aaca2d0f62" keccak256(TransferSingle(address,address,address,uint256,uint256)) )
- batched transfer (topic 0="0x4a39dc06d4c0dbc64b70af90fd698a233a518aa5d07e595d983b8c0526c8f7fb" == keccak256(TransferBatch(address,address,address,uint256[],uint256[])) )

## ERC-404

- FT transfer (topic 0="0xe59fdd36d0d223c0c7d996db7ad796880f45e1936cb0bb7ac102e7082e031487" keccak256(ERC20Transfer(address,address,uint256)) )
- NFT transfer (topic 0="0xe5f815dc84b8cecdfd4beedfc3f91ab5be7af100eca4e8fb11552b867995394f" == keccak256(ERC721Transfer(address,address,uint256)) )

## «WETH» deposits/withdrawals

- deposit (topic 0="0xe1ffcc4923d04b559f4d29a8bfc6cda04eb5b0d3c460751c2402c5c5cc9109c" keccak256(Deposit(address,uint256)) )
- withdrawal (topic 0="0x7fcf532c15f0a6db0bd6d0e038bea71d30d808c7d98cb3bf7268a95bf5081b65" == keccak256(Withdrawal(address,uint256)) )

💡 Indexer processes blocks in batches. The batch size and concurrency are managed via env variables.

# BLOCK IMPORT (SYNCHRONOUS PART)

- Next step is to fetch **block rewards**. Depending on the type of the JSON RPC client is used whether block rewards are requested from `trace\_block` method (Erigon, Nethermind, OpenEthereum (R.I.P.)) or calculated manually by summarising transaction fees from all block transactions.
- Once all responses are received from preceding JSON RPC requests we need to extract the rest of the data from the received responses:
  - Blockscout extracts **addresses** from all previous entities (blocks, logs, transfers, txs, withdrawals, block reward beneficiaries).
  - Then, address' **coin balances (historical and current)** are fetched from beneficiaries, blocks, logs, transactions, withdrawals.
  - Next is address's **token balances (historical and current)**.
- And the last part of synchronous import is **token instances**, which parsed from transfers.

```
curl -X POST --data '[
{"jsonrpc":"2.0","method":"trace_block","params":["0x..."],"id":1},
...
{"jsonrpc":"2.0","method":"trace_block","params":["0x..."],"id":n}]'
```



At the synchronous part of the block import, there only balance placeholders are inserted. And then, asynchronous process constantly monitors of those placeholders and fill them with the exact balances of the address at this block height



# JSON RPC RESPONSES TRANSFORMED TO BE INSERTED INTO THIS DB SCHEMA

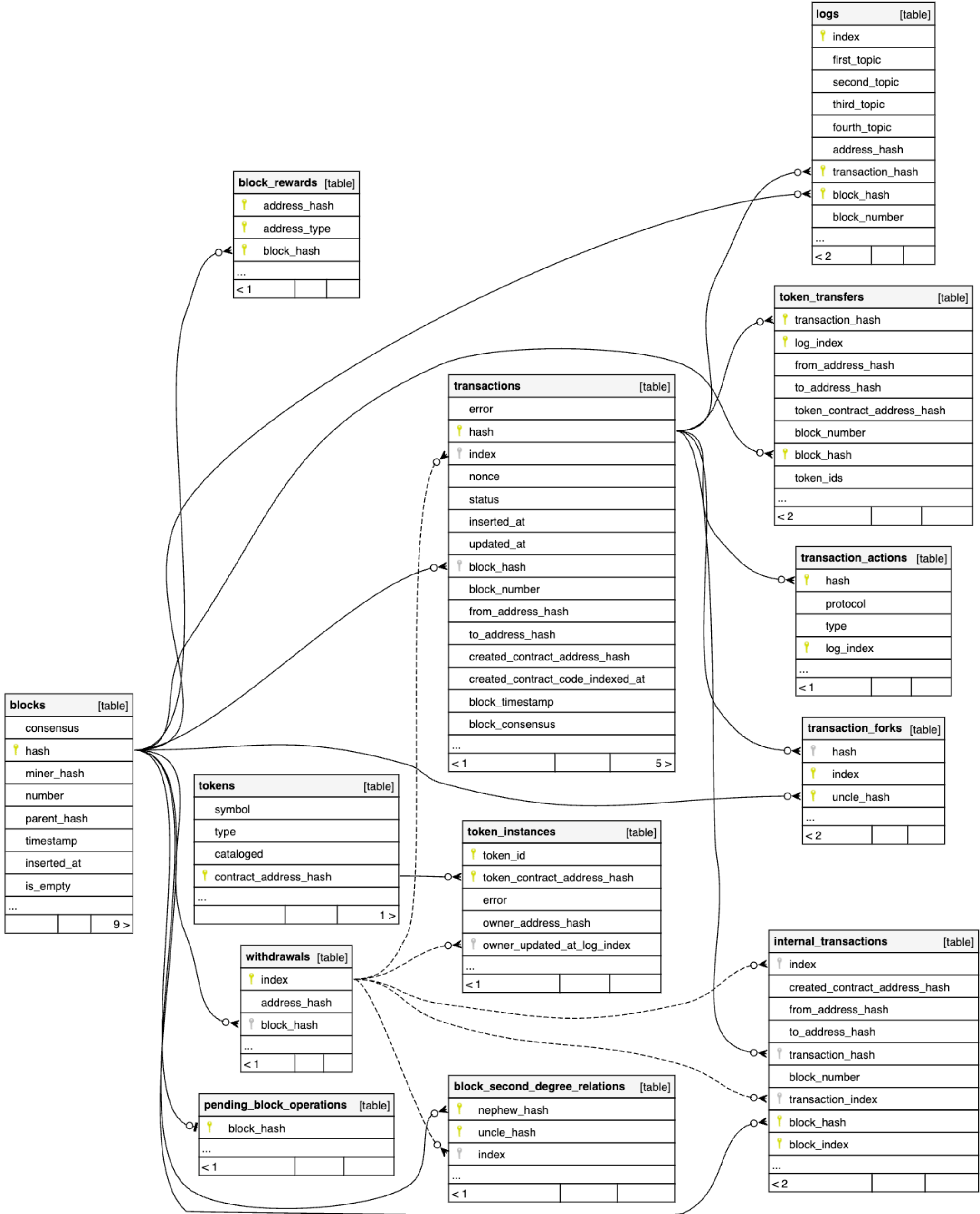
address_coin_balances_daily [table]		
address_hash	bytea[2147483647]	
day	date[13]	
value	numeric[100]	
inserted_at	timestamp[29,6]	
updated_at	timestamp[29,6]	
< 0		0 >

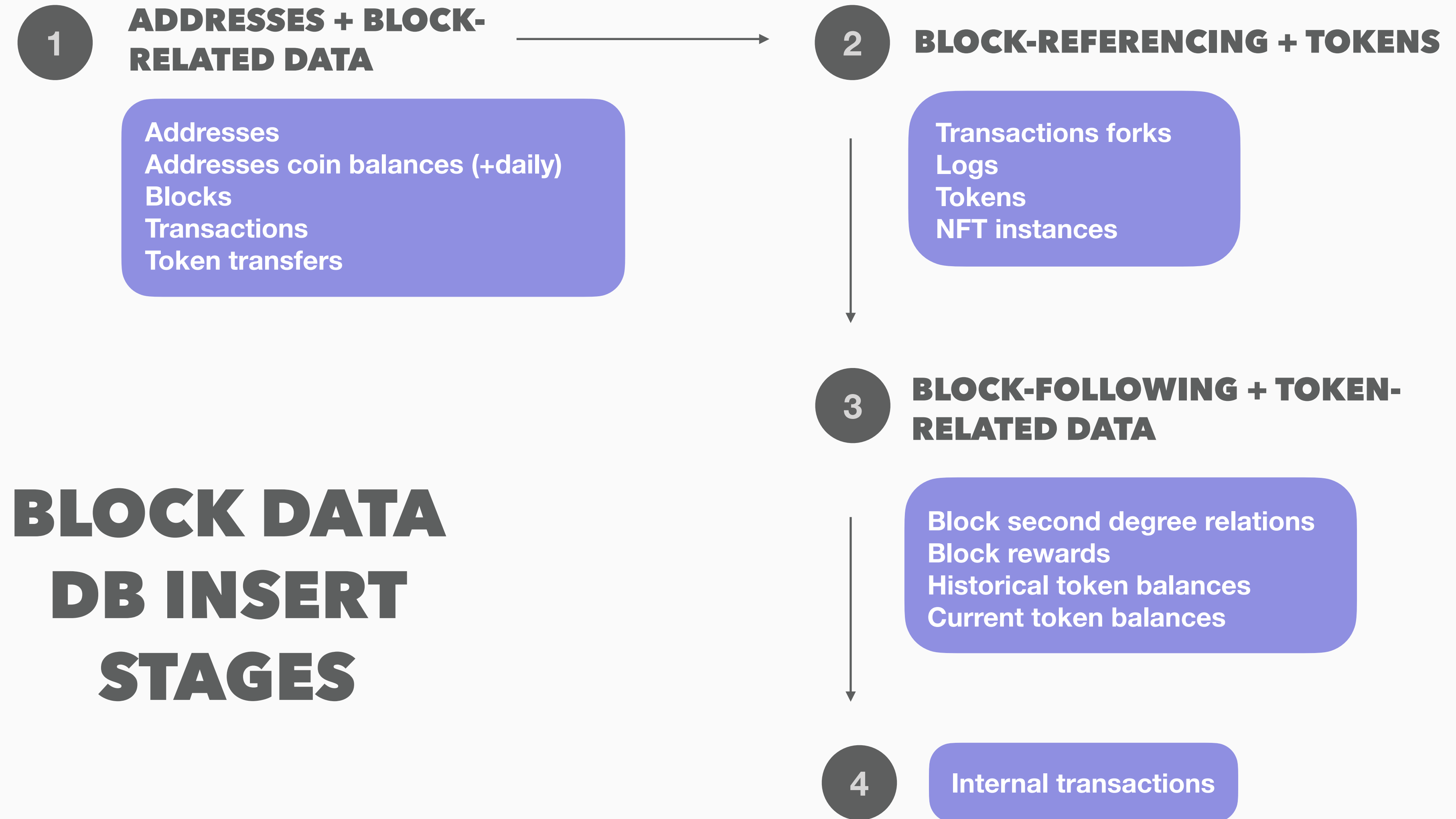
address_coin_balances [table]		
address_hash	bytea[2147483647]	
block_number	int8[19]	
value	numeric[100]	
value_fetched_at	timestamp[29,6]	
inserted_at	timestamp[29,6]	
updated_at	timestamp[29,6]	
< 0		0 >

addresses [table]		
fetches_coin_balance	numeric[100]	
fetches_coin_balance_block_number	int8[19]	
hash	bytea[2147483647]	
contract_code	bytea[2147483647]	
inserted_at	timestamp[29,6]	
updated_at	timestamp[29,6]	
nonce	int4[10]	
decompiled	bool[1]	
verified	bool[1]	
gas_used	int8[19]	
transactions_count	int4[10]	
token_transfers_count	int4[10]	
< 0		0 >

address_token_balances [table]		
id	bigserial[19]	
address_hash	bytea[2147483647]	
block_number	int8[19]	
token_contract_address_hash	bytea[2147483647]	
value	numeric[0]	
value_fetched_at	timestamp[29,6]	
inserted_at	timestamp[29,6]	
updated_at	timestamp[29,6]	
token_id	numeric[78]	
token_type	varchar[255]	
< 0		0 >

address_current_token_balances [table]		
id	bigserial[19]	
address_hash	bytea[2147483647]	
block_number	int8[19]	
token_contract_address_hash	bytea[2147483647]	
value	numeric[0]	
value_fetched_at	timestamp[29,6]	
inserted_at	timestamp[29,6]	
updated_at	timestamp[29,6]	
old_value	numeric[0]	
token_id	numeric[78]	
token_type	varchar[255]	
< 0		0 >





# ASYNCHRONOUS PART OF BLOCK IMPORT

```
defp async_import_remaining_block_data(  
  imported,  
  %{block_rewards: %{errors: block_reward_errors}} = options  
) do  
  realtime? = false  
  
  async_import_block_rewards(block_reward_errors, realtime?)  
  async_import_coin_balances(imported, options)  
  async_import_created_contract_codes(imported, realtime?)  
  async_import_internal_transactions(imported, realtime?)  
  async_import_tokens(imported, realtime?)  
  async_import_token_balances(imported, realtime?)  
  async_import_uncles(imported, realtime?)  
  async_import_replaced_transactions(imported, realtime?)  
  async_import_token_instances(imported)  
  ...  
end
```



# REGULAR FETCHERS

## INTERNAL TRANSACTIONS FETCH

This is continuous process, which checks block numbers in `pending\_block\_operations` table that internal transactions should be fetched for. This is the most time-consuming JSON RPC request in the most scenarios.

```
curl -X POST --data '[
{"jsonrpc": "2.0", "method": "debug_traceBlockByNumber", "params": ["0x...", {"tracer": "callTracer"}], "id": 1},
...
{"jsonrpc": "2.0", "method": "debug_traceBlockByNumber", "params": ["0x...", {"tracer": "callTracer"}], "id": n}]'
```

Together with `callTracer` tracer, which is default option, Blockscout also supports custom js tracer and struct/opcode logger.

In case of Erigon/Nethermind JSON RPC variants, `trace\_replayBlockTransactions` method is called instead.

## PENDING TRANSACTIONS FETCH

This is continuous process, which periodically fetches pending transactions from public mempool.

```
curl -X POST --data '{"jsonrpc": "2.0", "method": "txpool_content", "params": [], "id": 1}'
```

In case of Nethermind JSON RPC variant, `parity\_pendingTransactions` method is called instead.

## DROPPED/REPLACED TRANSACTIONS MARKING

This is continuous process, which periodically checks if the node doesn't return any info on given transaction hash, the process marks such tx as «dropped/replaced».

## CONTRACT BYTECODES FETCH

This is continuous process, which periodically checks if transactions exist in the DB where contract is created, but its byte code is not indexed yet.

```
curl -X POST --data '[
{"jsonrpc": "2.0", "method": "eth_getCode", "params": ["0x...", "0x..."], "id": 1},
...
{"jsonrpc": "2.0", "method": "eth_getCode", "params": ["0x...", "0x..."], "id": n}]'
```

# REGULAR FETCHERS

## COIN BALANCES FETCH

This is continuous process, which periodically checks coin balance placeholders in the DB and fills them via requesting `eth_getBalance` request.

```
curl -X POST --data '[
{"jsonrpc":"2.0","method":"eth_getBalance","params":["0x...", "0x..."],"id":1},
...
{"jsonrpc":"2.0","method":"eth_getBalance","params":["0x...", «0x..."],"id":n}]'
```

## BLOCK REWARDS FETCH

This is continuous process, which periodically checks if blocks exist, but corresponding rewards are not.

```
curl -X POST --data '[
{"jsonrpc":"2.0","method":"trace_block","params":["0x..."],"id":1},
...
{"jsonrpc":"2.0","method":"trace_block","params":["0x..."],"id":n}]'
```

## TOKEN BALANCES FETCH

This is continuous process, which periodically checks token balance placeholders in the DB and fills them via requesting `balanceOf` method in `eth_call` request.

```
curl -X POST --data '[
{"jsonrpc":"2.0","method":"eth_call","params":["0x70a08231...", "id":1},
...
{"jsonrpc":"2.0","method":"eth_call","params":["0x70a08231...", "id":n}]'
```

💡 `0x00fdd58e` - signature is used for NFTs (`balanceOf(address, uint256)`)

## TOKEN TOTAL SUPPLY UPDATE

This is continuous process, which periodically updates total supply of tokens, when burn or mint event is happened via requesting `totalSupply()` method in `eth_call` request.

```
curl -X POST --data '[
{"jsonrpc":"2.0","method":"eth_call","params":["0x18160ddd...", "id":1},
...
{"jsonrpc":"2.0","method":"eth_call","params":["0x18160ddd...", "id":n}]'
```

# REGULAR FETCHERS

## CATALOG TOKENS

This is continuous process, which periodically adds new tokens to catalog and performs initial metadata (name, symbol, decimals) fetching via requesting those props in [eth\\_call](#) request.

## CATALOGED TOKENS METADATA UPDATE

This is continuous process, which periodically updates tokens' metadata (name, symbol, decimals) via requesting those props in [eth\\_call](#) request. The process restarts every 48 hrs.

## BLOCK UNCLES FETCH

This is continuous process, which periodically updates block uncle info via requesting [eth\\_getUncleByBlockHashAndIndex](#) JSON RPC method.

## NFT INSTANCES AND METADATA

### REAL-TIME

It indexes metadata for new NFT instances (fast lane).



Real-time fetcher will also proceed with a single re-try on 404, 500 errors in 5 secs after the 1st try

### RE-TRY ON ERRORS

It attempts to index metadata for NFT fetching of which failed before. Exponential backoff is used to calculate time to the next retry attempt.



When reaching a week delay in retry for the given NFT instance, it becomes a constant. So, next re-tries will happen once in 7 days.



Errors "request error: 429", ":checkout\_timeout", ":econnrefused", ":timeout" are prioritized among others.

### CATCHUP

It continuously looks up token instances without errors in the DB, but also with metadata and tries to fetch that metadata (long queue).



Also fetchers-sanitisers are present, which try to fetch missing NFT instances (token transfers are detected, but corresponding token instance weren't indexed)

# ON-DEMAND FETCHERS

## COIN BALANCES FETCH

When someone opens address page ([/address/:hash](#)) or requests corresponding API endpoint ([/api/v2/addresses/:hash](#)) coin balance is updated through message to backend's web socket endpoint via requesting `eth_getBalance` JSON RPC method.

## TOKEN BALANCES FETCH

When someone triggers address page ([/address/:hash?tab=tokens](#)) or corresponding API endpoints ([/api/v2/addresses/:hash/tokens](#) or [/api/v2/addresses/:hash/token-balances](#)) corresponding token balances are updated through message to backend's web socket endpoint via requesting ``balanceOf/1`` or ``balanceOf/2`` method through `eth_call` JSON RPC request.

## CONTRACT CODE RE-CHECK

When someone opens smart-contract's address's page «Code»/«Logs» tabs ([/address/:hash?tab=contract](#) or [/address/:hash?tab=log](#)) or requests corresponding API endpoint ([/api/v2/smart-contracts/:hash](#)) contract code is re-fetched via requesting `eth_getCode` JSON RPC method through message to backend's web socket endpoint. Check is running once the last check was  $\geq 24$  hrs ago.

## CONTRACT SOURCE CODE LOOKUP IN ETH BYTECODE DB

When someone opens smart-contract's address's page «Code»/«Logs» tabs ([/address/:hash?tab=contract](#) or [/address/:hash?tab=log](#)) or requests corresponding API endpoint ([/api/v2/smart-contracts/:hash](#)), attempt is triggered to fetch verified contract source code from ETHbytecode DB Rust microservice. Check happens if the last one was  $\geq 10$  mins ago.

## CONTRACT CODE FETCH

When someone opens address page ([/address/:hash](#)) or requests corresponding API endpoint ([/api/v2/addresses/:hash](#)) contract code is fetched via requesting `eth_getCode` JSON RPC method through message to backend's web socket endpoint.

## NFT INSTANCE METADATA RE-FETCH

It re-fetches token instance metadata, if metadata exists (soon for every token instance) through «Refresh metadata» button in the UI or via triggering `POST api/v2/tokens/:hash/instances/:id/refetch-metadata` endpoint.

## TOKEN TOTAL SUPPLY UPDATE

It keeps token total supply up to date. When someone opens smart-contract's token's page ([/token/:hash](#)) or requests corresponding API endpoint ([/api/v2/tokens/:hash](#)) via requesting ``totalSupply()`` method through `eth_call` JSON RPC request.

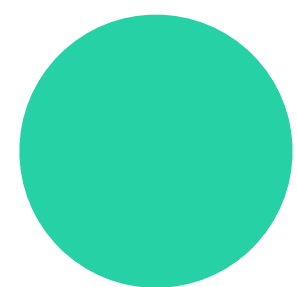
## CONTRACT DEPLOYER FETCH

To be done...

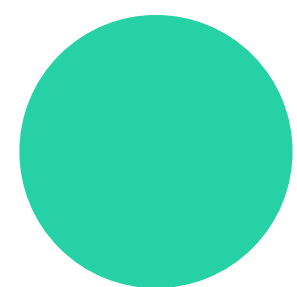
**WHAT IS NEXT?...**



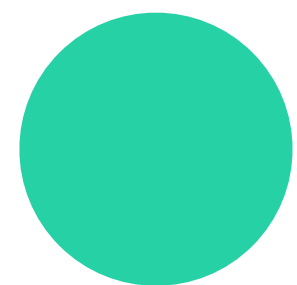
# V2



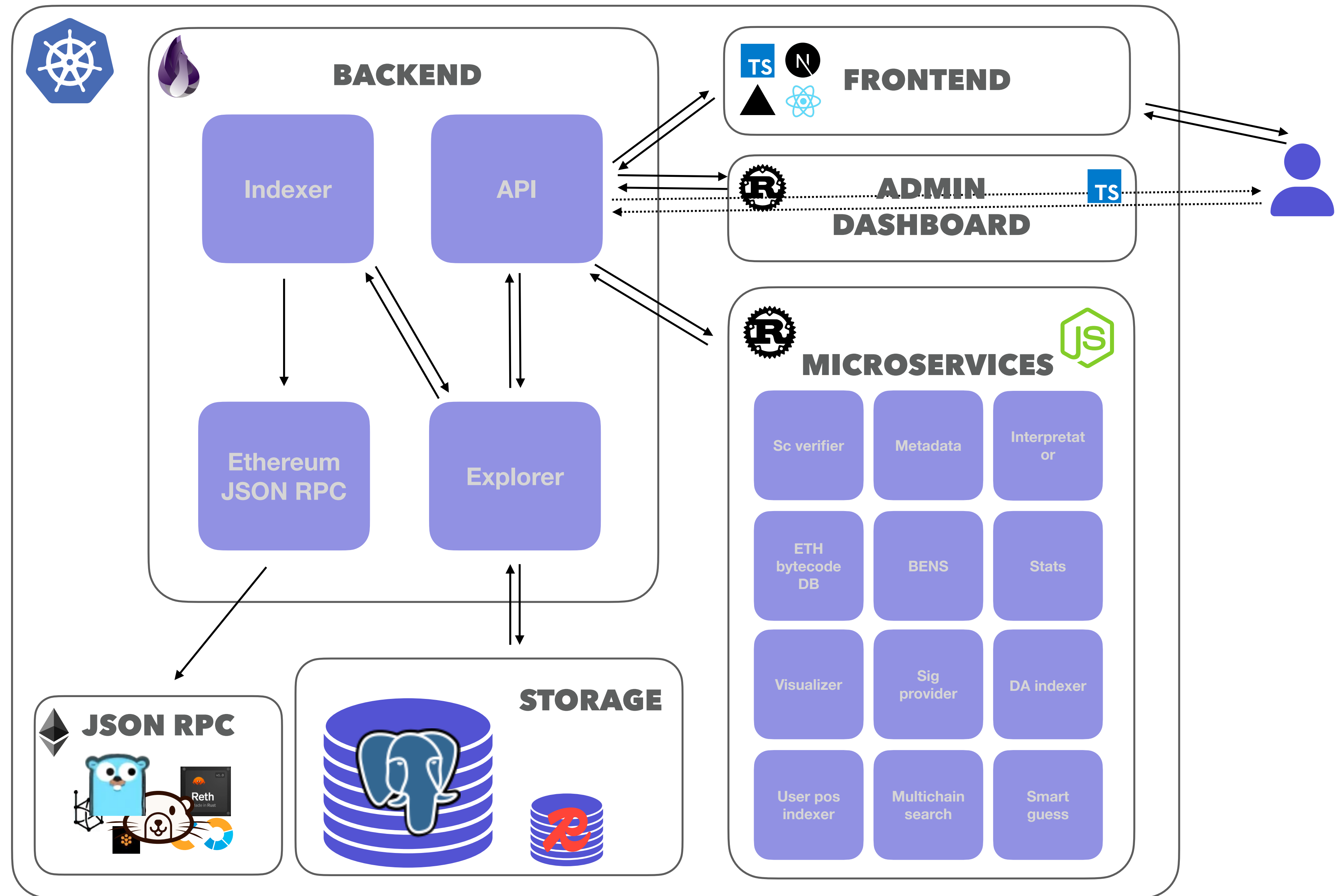
K8S



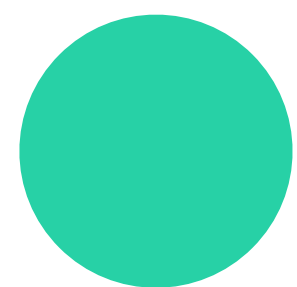
NEW FRONTEND



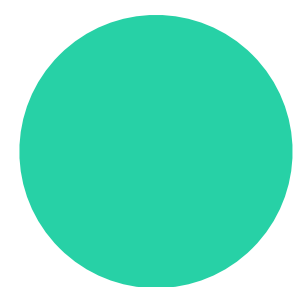
MICROSERVICES



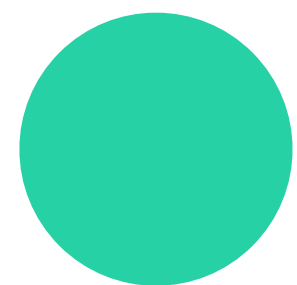
# V3



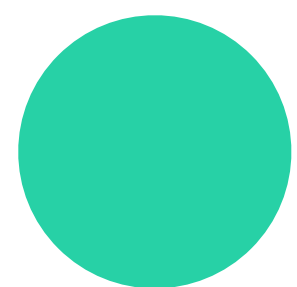
K8S



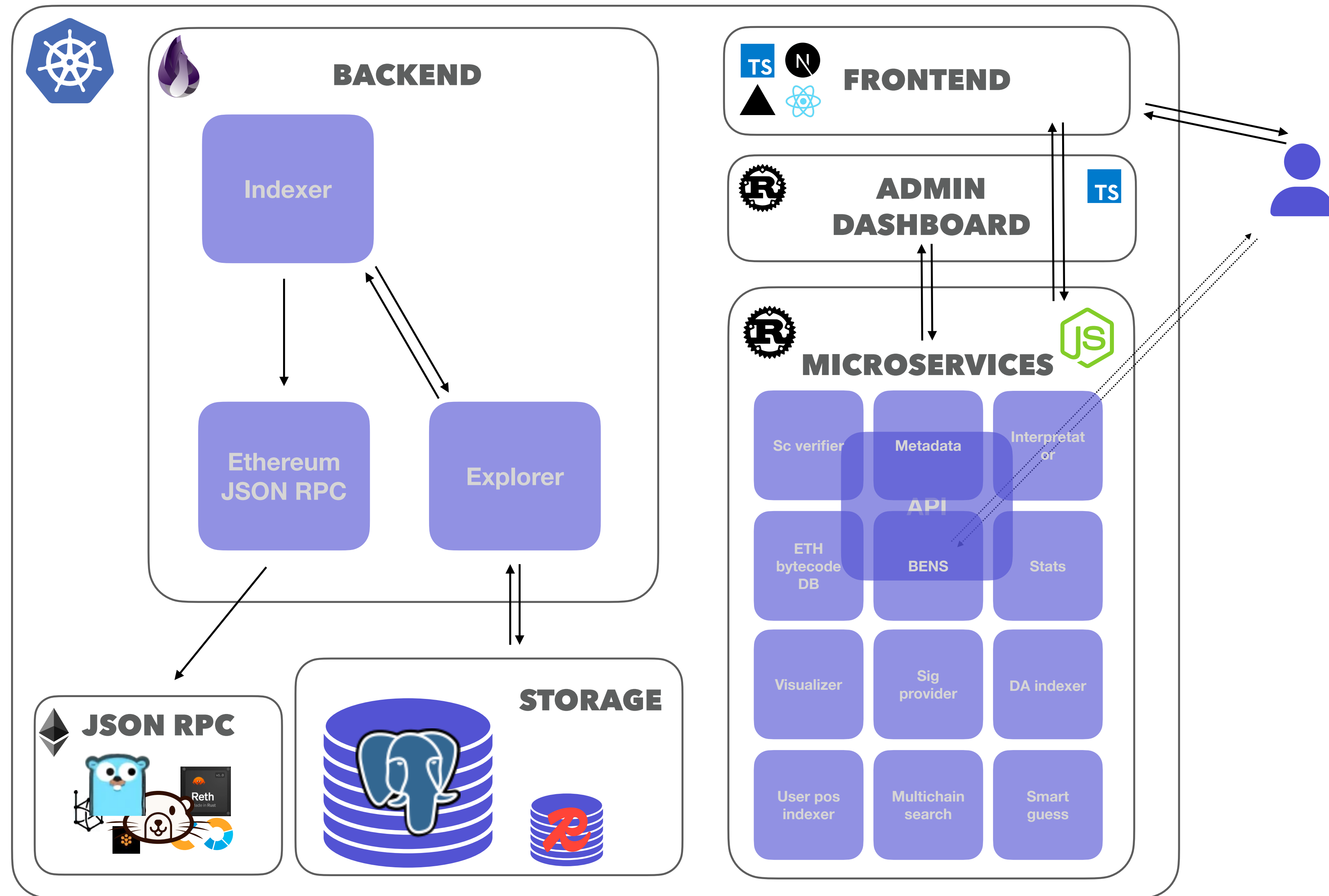
NEW FRONTEND



MICROSERVICES

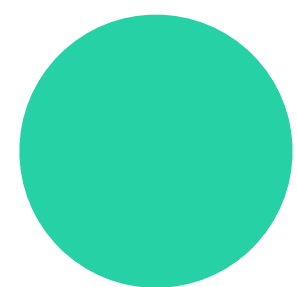


RUST API

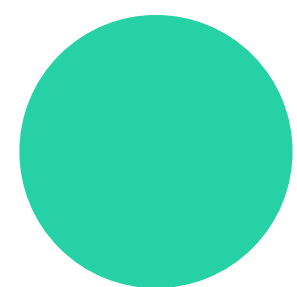




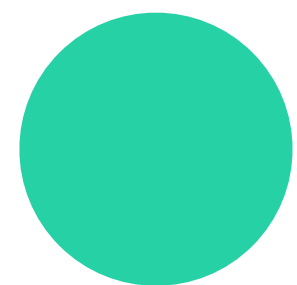
# V2



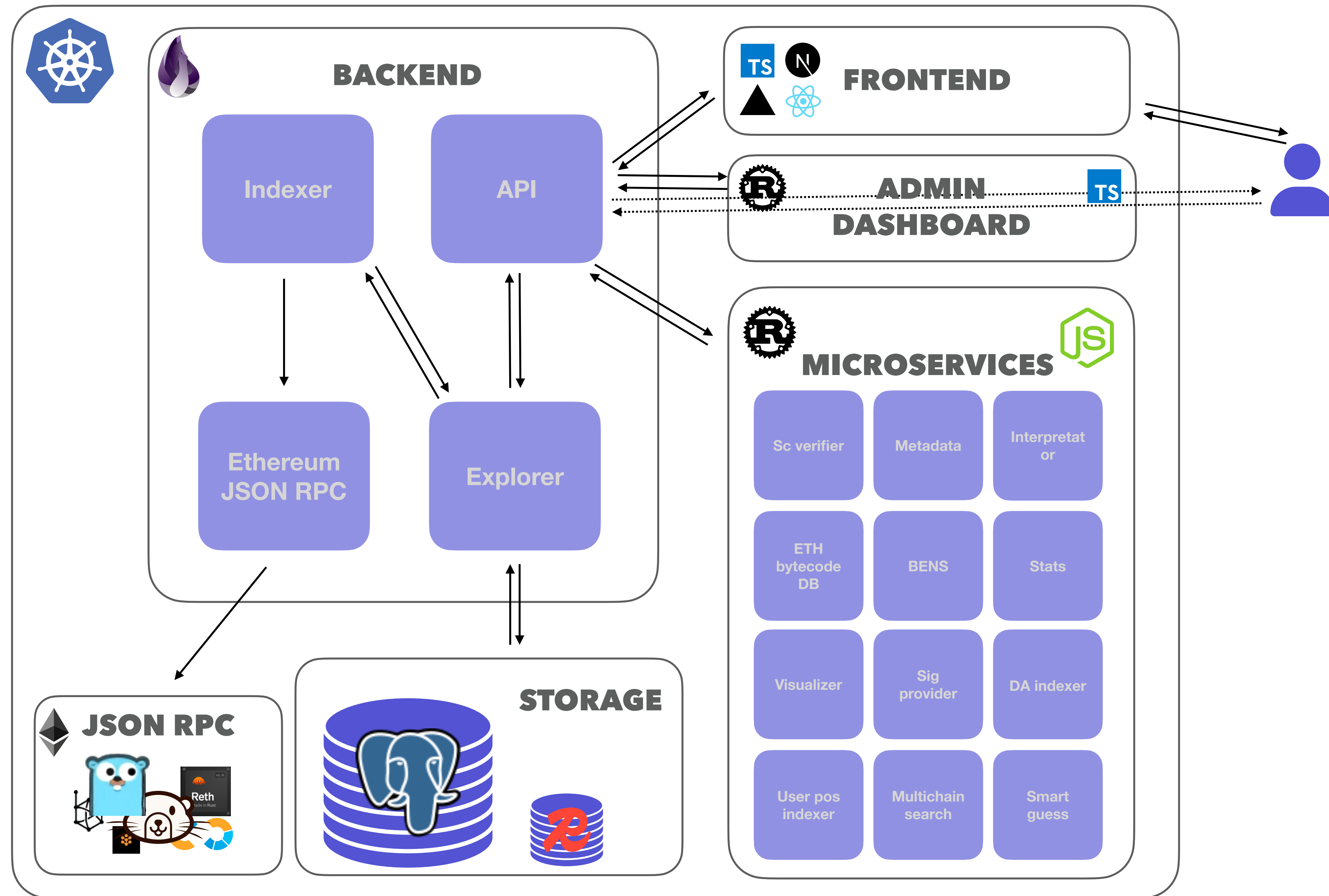
K8S



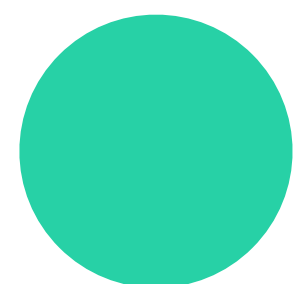
NEW FRONTEND



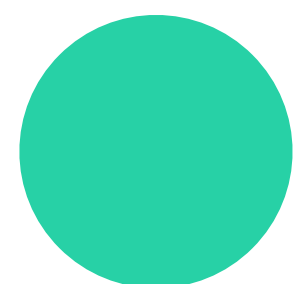
MICROSERVICES



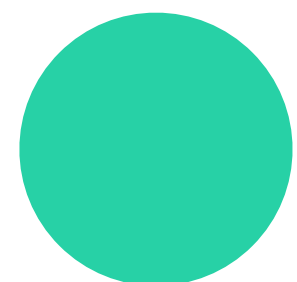
# V3



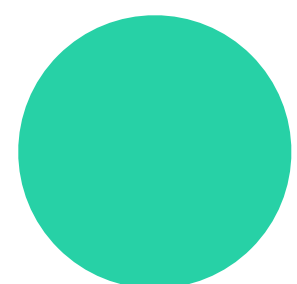
**K8S**



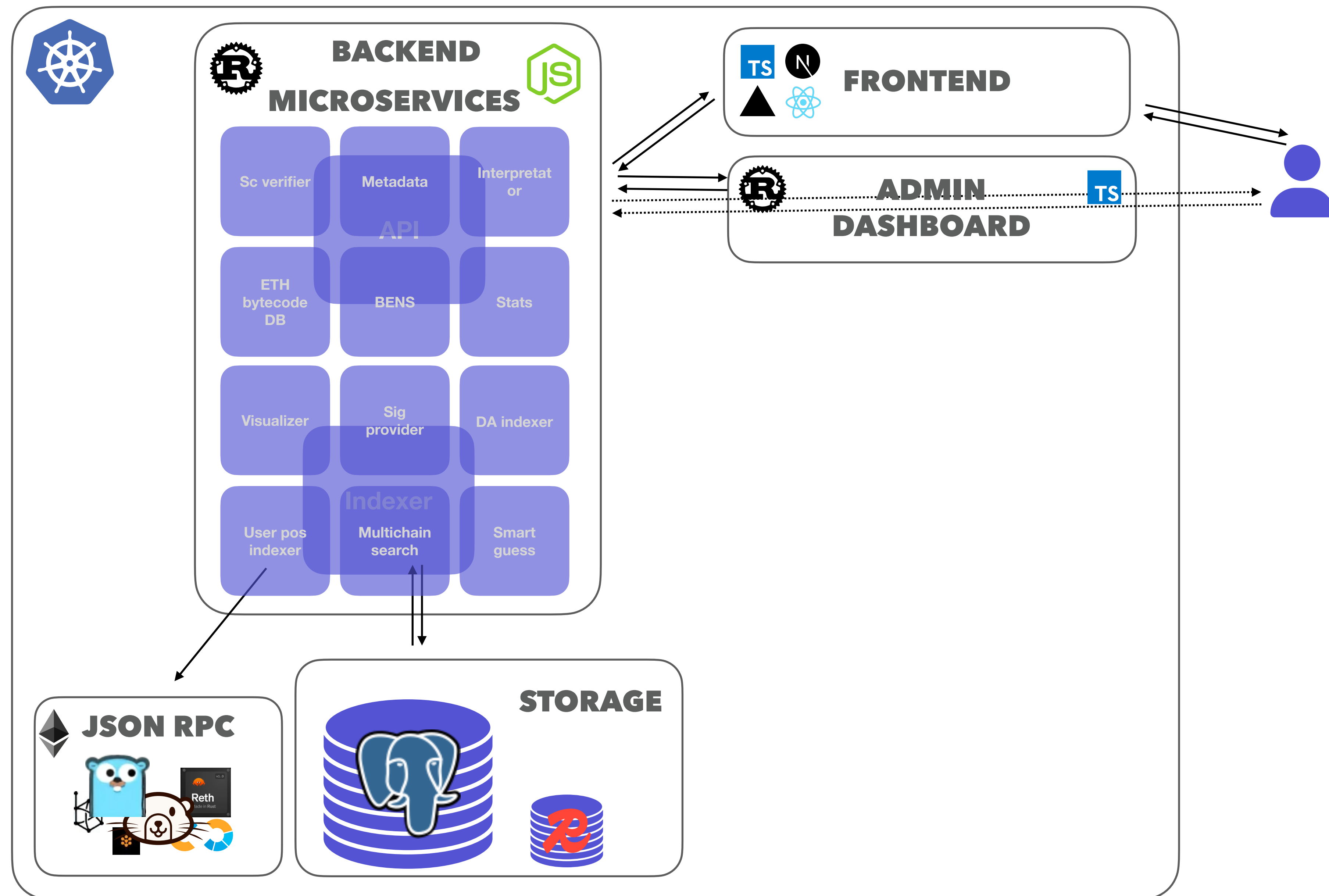
**NEW FRONTEND**



**MICROSERVICES**



**RUST BACKEND**





Blockscout

**THANKS!**