

Follow the chain head without a polling loop

Open a stream, persist the cursor, handle the undo signal and join live data to the history you already have.

BEFORE YOU START

WHO IT IS FOR

Backend and data engineers who need live chain data in an application, and who retrieve it today with a request loop.

WHAT YOU LEAVE WITH

A consumer contract covering the cursor, reconnection, chain corrections, finality and the join between history and the head.

WHY IT MATTERS

A loop that returns the block you already had is a cost you can see on an invoice. A loop that misses a range reports nothing at all, and the missing records surface later as a total that does not reconcile. This guide sets out the consumer contract that removes both, and names the parts your application still owns.

- 01 State what your application needs from live data
- 02 Open a stream instead of a request loop
- 03 Persist the cursor the stream gives you
- 04 Treat a disconnection as a normal event

- 05 Act on the undo signal instead of detecting a change
- 06 Separate provisional blocks from final blocks
- 07 Run one consumer across history and the head

State what your application needs from live data

Start from what the application does with a block, not from a target delay. A notification that fires once, a balance a customer reads and a report published at the end of the day place different demands on the same stream. Write those demands down before choosing an endpoint, because they decide the rest of the design.

Three requirements are usually collapsed into one and need to stay separate: how soon a record must be visible, whether it may be read before its block is final, and what must happen to it when the chain replaces that block. The third one is the one that gets discovered in production.

REQUIREMENT	DECISION TO MAKE
Visibility	How soon after a block is produced the record must be queryable by a consumer.
Provisional data	Whether the application may expose a record whose block can still be replaced.
Correction	Which stored records, derived tables and caches must change when a block is undone.
Resume position	Which component owns the position the consumer restarts from, and where it is stored.

A request loop and a stream both answer the question "what is the latest block". Only one of them also answers "what happened while I was not connected". StreamingFast's own description of the tradeoff keeps polling as a reasonable choice for a prototype, a periodic snapshot or a value checked on a slow cadence, where a missed update is recoverable and delay does not matter. Decide which of those two questions your application actually has to answer.

Design input: one written sentence for each of the four requirements above, agreed before any endpoint or sink is selected.

[1] Substreams reliability guarantees · [2] StreamingFast: polling versus streaming blockchain data

Open a stream instead of a request loop

Consuming a Substreams package means establishing a gRPC connection with a provider, which then streams the data of every block back to your sink. The connection is long-lived: your code holds it open and reacts to what arrives, rather than deciding when to ask.

What arrives is a single **Response** message with one field set at a time: session initialisation, module progress, block data, an undo signal, or a fatal error. Each case is a branch your consumer has to implement. There is no branch in which silence means that nothing happened on the chain.

RESPONSIBILITY	REQUEST LOOP	STREAM
Deciding when data arrives	An interval you choose, tune and defend.	The provider sends a block when it exists.
Knowing what you missed	A range query you write and maintain.	The cursor returned with the data you consumed.
Learning that history changed	A comparison you implement yourself.	An explicit undo signal message.
Handling an interruption	Reconstructing the gap from the last block you remember.	Reconnecting with the stored cursor.

Decide which component holds the connection. StreamingFast documents hosted sinks that it runs for you, official sinks you run yourself, and official SDKs in Go, Rust and JavaScript for a consumer you write. The same documentation separates sinks it supports actively from community-maintained ones whose support cannot be guaranteed, so record which you selected and who maintains it.

Writing your own consumer is a legitimate choice, and it is the choice that makes the next four chapters your responsibility rather than the sink's. The cursor, the reconnection, the undo handler and the finality policy are the consumer contract. A connection that delivers blocks does not implement any of them on its own.

Decision: name the component that holds the connection, and the person who owns it when it stops.

[1] Substreams reliability guarantees · [3] Consuming Substreams and sink support · [2] StreamingFast: polling versus streaming blockchain data

Persist the cursor the stream gives you

Block data arrives as a **BlockScopedData** message carrying the module output, a clock and a cursor. The documentation is explicit about what the cursor is and what you owe it: it points to the latest piece of data consumed, and the user must persist it, so that after a disconnection the provider can resume streaming from the latest consumed block.

A block number is not a substitute. The cursor identifies a position in the stream together with the fork context that position belonged to. Storing a number gives you a height without the branch it was on, which is precisely the information a reconnection after a chain correction needs.

Decide where the cursor lives

- **In the destination, with the data.** Where the destination supports one transaction, commit the records and the cursor together. Recovery then has a single truth.
- **In the destination, in a known order.** Where that is not atomic, decide which of the two is written first, and state what recovery does with the difference.
- **Not in memory.** The Go example is blunt about it: a cursor kept in memory lasts until the process is killed, after which the consumer resumes from the start block again.
- **Not in two places.** An application checkpoint maintained next to a sink's own cursor storage gives you two positions that can disagree.

A consumer starts in one of two states, and the code should make the difference visible. With no stored cursor it begins at a requested start block; with a stored cursor it resumes from it. The Go SDK expresses exactly this distinction, with a blank cursor for the first run and a cursor loaded from storage for every run after it.

Operating record: where the cursor is stored, how it is backed up, and which write lands first when the cursor and the data cannot be committed together.

[1] Substreams reliability guarantees · [4] Substreams:Stream, Go consumer · [2] StreamingFast: polling versus streaming blockchain data

05 / 04

Treat a disconnection as a normal event

The JavaScript documentation states the operating assumption directly: because the connection is long-lived, disconnections will happen and should be taken as normal. A consumer that treats a dropped connection as an incident will page someone every week. A consumer that treats it as an expected transition will reconnect and continue.

The loop in a streaming consumer is therefore not a polling loop. It is a reconnection loop: attempt the stream, and on a retryable error wait and attempt it again from the stored cursor. The official examples are built that way, with the retryable case separated from the fatal case and a backoff between attempts. The Go SDK goes further and handles reconnection for you, which is one of the reasons to use it rather than call the service directly.

EVENT	WHAT THE CONSUMER DOES
Retryable transport error	Reconnect from the stored cursor, after a backoff.
Fatal error message	Stop, surface the error, and leave the cursor where it is.
Process restart	Read the stored cursor; use the requested start block only when none exists.
Destination refusing writes	Stop advancing the cursor until writes succeed again, so the gap stays visible.

The failure worth designing against is reconnecting at the head. Nothing errors, the stream resumes, and the blocks produced during the interruption are simply absent from the destination. StreamingFast describes that class of failure as the silent one: a handful of missing records that nobody notices until a total fails to reconcile much later.

Test to run: stop the consumer in the middle of a range, restart it, and show that the destination holds the whole interval with no gap and no duplicated record.

[5] Substreams:Stream, JavaScript consumer · [4] Substreams:Stream, Go consumer · [2] StreamingFast: polling versus streaming blockchain data

05 / 05

Act on the undo signal instead of detecting a change

A chain reorganization replaces recently accepted blocks with an alternative chain carrying more accumulated work or stake. StreamingFast states the consequence plainly: transactions in the discarded blocks may never have happened as far as the canonical chain is concerned. A swap you recorded did not happen, and a balance you computed from it is wrong.

Substreams signals this rather than leaving you to notice it. A `BlockUndoSignal` message carries the last valid block and the cursor for that block. Your consumer reverses the derived data down to that block and commits that cursor. The documentation adds a useful property: if you commit the cursors from the undo signals, a disconnection in the middle of a fork resolution brings you back exactly where you left off.

This is why a consumer registers two handlers, not one. The Go example passes a handler for block data and a handler for undo signals in the same call. A consumer with only the first handler compiles, runs, and keeps invalidated records.

What reversible writes require

The SQL sink is the worked example of the mechanism. It records every operation applied in the still-reversible segment of the chain in a history table, keeping the operation type, the table, the primary key, the previous values and the block number. On an undo signal it reverses an insert with a delete, an update by restoring the saved previous values, and a delete by reinserting the stored row. Two consequences follow directly from that design, and both apply to a consumer you write yourself.

- **Every table needs a primary key.** The sink documentation states the requirement outright: without one, there is no way to identify the row to reverse.
- **History has to be pruned.** The sink removes history for blocks that have become final, which keeps the table from growing without bound. A destination of your own needs the same cleanup, and the capacity to hold that history in the meantime.

Confirm the behaviour of the exact sink and release you deploy. The documentation notes, for instance, that the ClickHouse sink supports reorganization handling with a slight delay compared with PostgreSQL. And test the undo path deliberately: StreamingFast's own advice is that reorganizations are rare enough that untested handling usually contains bugs.

Design question: for every table your consumer writes, name the operation that reverses each write and the key it reverses by.

[6] StreamingFast: what a blockchain reorg is, and how indexers handle it · [1] Substreams reliability guarantees · [7] SQL sink reorganization handling · [4] Substreams.Stream, Go consumer

Separate provisional blocks from final blocks

Each block data message carries a final block height alongside the output and the cursor. That field is what lets a consumer answer, at the moment it writes, whether the record it is writing can still be taken back. Keep that answer with the record rather than inferring it later.

From there the policy is a product decision, not a technical one.

POLICY	WHAT THE APPLICATION MUST THEN PROVIDE
Publish only final data	An accepted delay between a block being produced and its records becoming visible.
Publish provisional data	A visible status on the record, and a correction path to every report, cache and downstream consumer derived from it.

StreamingFast compares the two common implementations. Waiting a fixed number of confirmations is simple, adds delay, and still fails when a reorganization runs deeper than the depth you chose. Cursor-based handling instead relies on the stream telling you explicitly which blocks were undone, which is the approach the documentation and the blog both describe.

At the very head, a block can arrive in parts

On networks that produce them, Substreams delivers partial blocks as ordinary block data messages flagged as partial, with an ordinal for the part and a marker on the last part of a given height. The documentation

describes this as a beta feature available on a single endpoint, requested through a flag on the request, and it comes with constraints a consumer has to respect.

- Modules must not perform block-level aggregation, because they run repeatedly over increments of the same height.
- Modules must not use the block hash, which changes between versions of the same partial block.
- A consumer should only take decisions on the block hash for a full block or for the last partial of a height.
- An undo signal is sent for a reorganization, or when sent partials are discarded and replaced; it is not sent between partials of the same height.

Partial blocks make the head more responsive and make the provisional nature of head data unavoidable. Adopt them when the application has something useful to do with a transaction before its block is complete, and not because they are the newest option available.

Publication rule: every record the application exposes should be traceable to a block and to the policy under which it was published.

[1] Substreams reliability guarantees · [8] Substreams partial block support at the chain head · [6] StreamingFast: what a blockchain reorg is, and how indexers handle it

05 / 07

Run one consumer across history and the head

History and the head are the same request. A start block in the past and no stop block produce one stream that delivers the historical range and then continues at the head, through the same modules, the same output and the same destination code. There is no second implementation for live data, which removes the most common source of disagreement between a backfill and a running pipeline.

The transition has a documented boundary. Forward parallel execution runs ranges in parallel from the requested start block up to the last known final block, also called an irreversible block, or the request's stop block if that is lower. It occurs in production mode only, while backward parallel execution occurs in both production and development mode. Past that boundary the output is linear, because there is no finished chain left to divide into segments.

Record what the consumer knew at the transition: the cursor at which it reached the head, and the last block durably written to the destination. Those two values are what a later investigation needs, and neither can be reconstructed afterwards from the destination alone.

Watch what the consumer reports

The SQL sink's stream summary is a useful model for what to observe in any consumer, including one you wrote. Three of its fields carry most of the operational meaning.

- **Whether the stream is live.** The summary reports whether the stream is considered to be at the chain head. That flag is what separates catching up from following.
- **Undo signals received.** A rate that stays at zero through a historical range and becomes non-zero at the head means reorganizations are being handled, which is the expected behaviour rather than an alarm.
- **The last block delivered.** The documentation is careful to note that this is what has been downloaded, not what has been committed. Keep delivered and committed as two separate numbers in whatever you alert on, because the gap between them is the thing you actually want to know about.

Assign an owner to each of those, plus the cursor storage, the undo handler and the finality policy. An alert should carry enough context to tell an unavailable provider apart from a destination that has stopped accepting writes, since the correct response to the two is different.

For a discussion with StreamingFast, bring the networks, the package or modules, the destination, the finality policy and what the application does with a correction. Those are the inputs that make it possible to assess a streaming deployment against a concrete workload rather than against a general description.

Next step: review the reliability guarantees documentation with your cursor handler and your undo handler open next to it, and confirm that each one does what the contract requires.

[9] Substreams architecture and parallel execution · [10] SQL sink progress and performance output · [1] Substreams reliability guarantees

REFERENCES

Sources and further reading

Documentation reviewed on 29 October 2026. Consult the current documentation for the release and service you use.

1. Substreams reliability guarantees
<https://docs.substreams.dev/reference-material/core-concepts/reliability-guarantees>
2. StreamingFast: polling versus streaming blockchain data
<https://www.streamingfast.io/blog/common-questions/polling-vs-streaming-blockchain-data>
3. Consuming Substreams and sink support
<https://docs.substreams.dev/how-to-guides/sinks>
4. Substreams:Stream, Go consumer
<https://docs.substreams.dev/how-to-guides/sinks/stream/go>
5. Substreams:Stream, JavaScript consumer
<https://docs.substreams.dev/how-to-guides/sinks/stream/javascript>
6. StreamingFast: what a blockchain reorg is, and how indexers handle it
<https://www.streamingfast.io/blog/common-questions/what-is-a-blockchain-reorg>

7. SQL sink reorganization handling

<https://docs.substreams.dev/reference-material/sql/sql/reorg-handling>

8. Substreams partial block support at the chain head

<https://docs.substreams.dev/reference-material/chain-support/flashblocks>

9. Substreams architecture and parallel execution

<https://docs.substreams.dev/reference-material/core-concepts/architecture>

10. SQL sink progress and performance output

<https://docs.substreams.dev/reference-material/sql/sql/progress-info>

The planning checklists are editorial recommendations. Technical product statements refer to the documentation linked in each chapter. The guides do not report measured pipeline performance.

● NEXT

Run it on your own chain.

Get an API key ↗