

Choose where your chain data lands

Six destinations for the same stream. What each one serves, what it asks you to operate, and how to choose.

BEFORE YOU START

WHO IT IS FOR

Teams that have a Substreams package and now have to decide what receives its output.

WHAT YOU LEAVE WITH

A destination chosen from the consumer that reads it, and a written list of what your team operates once it is running.

WHY IT MATTERS

Postgres, ClickHouse, Parquet, a Pub/Sub topic, a webhook and plain JSON are named side by side across this site, which makes them look like six settings of one feature. They are not. Two of them are a service StreamingFast operates for you. The other four are a process you run, with a cursor you keep and failures you handle. The difference is not a preference. It is a standing operating commitment.

- 01 The six destinations, and who runs the consumer
- 02 What a hosted sink removes from your side
- 03 Choose between Postgres and ClickHouse
- 04 Use Parquet when the consumer is an analytical engine

- 05 Use Pub/Sub or a webhook when the consumer is an application
- 06 Use JSON when you are writing the consumer yourself
- 07 Choose from the question the consumer asks

The six destinations, and who runs the consumer

A Substreams package produces output. A sink is the integration that carries that output to a destination. The destinations named across this site are Postgres, ClickHouse, Parquet, a Pub/Sub topic, a webhook and plain JSON. The homepage states the same list under *Sink anywhere*.

The first division to make is not technical. It is who runs the process that does the writing. Postgres and ClickHouse can be reached by Hosted Sinks, a managed service on The Graph Market that runs your sink for you. The other four destinations are reached by a sink you deploy and keep alive yourself.

DESTINATION	WHO RUNS THE CONSUMER	WHO MAINTAINS THE CONNECTOR
Postgres	StreamingFast, as a hosted sink, or you	StreamingFast, documented as officially supported
ClickHouse	StreamingFast, as a hosted sink, or you	StreamingFast, documented as officially supported
Parquet	You	Community, through substreams-sink-files
Pub/Sub topic	You	StreamingFast, documented as officially supported
Webhook	You	StreamingFast, documented as officially supported
JSON	You	StreamingFast for the ProtoJSON sink, your own code beyond it

The second division is maintenance of the connector, and it is separate from the first. The sinks documentation keeps two tables, one for official support and one for community support, and states that some sinks are officially supported by StreamingFast while others are community-driven and support cannot be guaranteed. Parquet arrives through the files sink, which sits in the community table. That is worth knowing before a destination becomes a dependency.

First decision: whether a process you operate is acceptable for this dataset. Answer that before comparing formats.

What a hosted sink removes from your side

Hosted Sinks is described in the documentation as a managed service on The Graph Market that runs your Substreams sink for you. Instead of provisioning servers, managing deployments or operating sink processes, you configure the sink in the portal and StreamingFast handles the infrastructure. It is one of two Hosted Services; the other is a Hosted Store. The documentation marks Hosted Sinks as currently in beta.

The boundary is stated plainly, and it is the part worth reading twice. A hosted sink continuously reads your package and writes the output into **your** Postgres or ClickHouse database. StreamingFast hosts the sink runner only. It does not create, host or manage the database.

What you still bring

- **The database.** One you already operate, or a managed one. The documentation provides walkthroughs for Supabase and Neon on Postgres, and ClickHouse Cloud on ClickHouse.
- **Network reachability.** The database must be reachable from StreamingFast's network, through a public hostname rather than `localhost` or a private container name.
- **Credentials.** The documentation asks for a dedicated database user whose write access is restricted to the target schema.
- **A package that outputs for SQL.** Database Changes Mode through a `db_out` module, or Relational Mappings Mode.
- **The schema.** Either already applied, or created by the package itself.

What you configure in the portal is a sink name, the package by URL or by its Substreams Registry ID, the connection details, an optional GraphQL layer, then the execution settings: start block, stop block, output module, and optional filters and parameters. Deployment moves through `Deploying` to `Deployed`, and a running pod reports its own execution state as `Initiating`, `Catching up`, `Live` or `Failing`. The detail page carries pod logs and a deployment event log, and the edit dialog changes the package, the execution settings or the connection details, after which the sink process restarts.

The optional GraphQL layer is a separate read-only deployment, powered by Hasura, that reads the same database the sink writes. It tracks the tables and views in the target schema, excludes the sink's own bookkeeping tables, generates no mutation fields, and can be added or removed without touching the pipeline. Both Postgres and ClickHouse sinks are supported.

Operating record: with a hosted sink you own a database and a schema. You do not own a sink process, a deployment or a restart.

Choose between Postgres and ClickHouse

Both engines are reached by the same sink, which is built into the `substreams` CLI as `substreams sink postgres` and `substreams sink clickhouse`, and both are offered by Hosted Sinks. The choice between them is a choice about the consumer, not about the connector.

Postgres suits a product that reads rows by key and keeps current state alongside the application's own tables. ClickHouse suits analytical queries written over wide ranges of history. Both accept the two documented mapping modes, so the mode decision is separable from the engine decision; Build a reliable blockchain-to-SQL pipeline works through that decision and the write path in detail.

Two documented differences bear on operations rather than on query style.

CONSIDER	WHAT THE DOCUMENTATION STATES
Corrections	Database Changes Mode handles reorganizations by tracking operations in a history table, receiving undo signals, reverting operations from forked blocks and cleaning up after finality. The Substreams:SQL page notes that in ClickHouse, reorganizations are currently supported with delay.
Bookkeeping in your database	The sink keeps its own tables in your schema. The GraphQL layer documentation names them as <code>_cursor_</code> , <code>_blocks_</code> and <code>_sink_info_</code> , and excludes them from the generated API. Database Changes Mode adds a <code>substreams_history</code> table for the reversible segment of the chain.

Plan capacity for those tables, and for the backup policy that now covers them. The reorganization documentation also notes that history tracking applies only in the reversible segment, so a historical backfill does not accumulate it.

The operating cost of either engine is the cost of running that engine. A hosted sink does not change it. What a hosted sink changes is whether anyone on your team is also responsible for the process writing into it.

Design question: which of your consumers reads by key, and which reads by range? If both do, say which one the schema is built for.

[7] Substreams:SQL · [8] SQL sink reorganization handling · [6] Hosted sink GraphQL layer

Use Parquet when the consumer is an analytical engine

Parquet arrives through `substreams-sink-files`, a tool that writes blockchain data into file formats: Parquet, line-based CSV, line-based JSONL, and arbitrary Protobuf turned into JSONL. Parquet is the default encoder when no `--encoder` flag is given. The sinks documentation lists this tool under community support.

The encoder derives a Parquet schema from the Protobuf message your module outputs. Table structures are inferred, or set explicitly with the `(parquet.table_name)` option; individual columns are controlled with `(parquet.column)` for type and compression, and a field can be left out with `(parquet.ignored)`. The repository documents a `tools parquet schema` command that prints the derived schema without running the extraction, which is the cheapest way to find out whether your output models the tables you expect.

Output can be a local path or a cloud bucket, addressed as `gs://`, `s3://` or `az://`. Files are written in bundles, and the number of blocks per bundle is set with `--file-block-count`.

What you operate

- **The process.** Nothing hosted writes Parquet for you.
- **The cursor.** It is saved to a file in the folder given by `--state-store`. The repository is explicit that this file must be persisted and must follow the deployment: if it is lost, the tool restarts from the beginning of the chain and redoes all previous processing.
- **The reading side.** The engine, catalogue or notebook that queries the files, and whatever registers new files as they appear.

One documented property decides most cases on its own. The repository states that the tool syncs up to the most recent final block of the chain, and that this means it is not real-time. For a warehouse load or a dataset handed to analysts, that is the correct behaviour, because no bundle on disk ever has to be unwound. For anything that reacts to the head of the chain, it is the wrong destination.

Next action: print the derived schema before writing a single file, and decide where the state store lives before the first deployment.

[9] [substreams-sink-files](#) · [1] [Consuming Substreams and sink support](#)

06 / 05

Use Pub/Sub or a webhook when the consumer is an application

These two destinations deliver messages rather than rows. Both are documented as officially supported, and both are a process you run. They differ in how many consumers can read the same output and in what happens when delivery fails.

A Pub/Sub topic

`substreams sink pubsub` publishes the output of any module to a Google Cloud Pub/Sub topic. The topic is given as an ID with `--project` or as a full `projects/<project>/topics/<topic>` path, and credentials are application default credentials. Each block message carries the attribute `type=batch` and a body holding the manifest and the blocks, where each block's `data` is that module's output as JSON.

Message ordering is enabled, and the ordering key is the output module's name, so a subscription created with message ordering enabled receives that module's messages in publish order.

Corrections are explicit. `--pubsub-undo` publishes a reorganization notification on the same topic, with attribute `type=undo`, naming the last valid block. Without that flag the cursor still moves back and the replacement blocks are published as usual, which means a subscriber that assumes append-only delivery will see the same block numbers twice and will have to reconcile them itself. A module whose output is `sf.substreams.sink.pubsub.v1.Publish` takes the other path: each of its messages is published on its own, and its reorganization message — attributes `LastValidBlock`, `Step=Undo` and the cursor, with no body — is published whether or not the flag is set.

Failure handling is also explicit. The cursor lives in `--state-file`. A payload that could not be published is kept in a `.pending` file beside it, the process exits with status 75 under the default `--pubsub-on-failure=exit`, and the next start publishes that payload before reconnecting. `--pubsub-on-failure=skip` drops a block after the last retry and continues, and an undo notification is never dropped.

A webhook

`substreams sink webhook` posts to an HTTP endpoint you supply. The payload is JSON carrying `clock`, `manifest` and `data`. Retries are configurable through `--webhook-max-retries`, `--webhook-timeout` and `--webhook-max-retry-interval`, with exponential backoff and jitter; `-1` retries indefinitely. Network errors and server errors are retried. Client errors in the 4xx range are not, because the sink treats them as permanent.

Two behaviours decide whether this destination fits. The repository states that the sink continues processing even if webhook calls fail after all retries, and that the cursor is still saved for successful blocks, so block processing is never blocked by a failing endpoint. It also lists idempotent handlers that tolerate duplicate calls as a best practice. Taken together: your endpoint is the component that must notice it missed something, and your endpoint is the component that must not double-apply what it receives twice.

CONSIDER	PUB/SUB TOPIC	WEBHOOK
Consumers	Any number of subscriptions on one topic	One endpoint per sink process
Ordering	Ordering enabled, keyed on the output module	Determined by your endpoint and its retries
Corrections	An undo notification with the flag, and always from a Publish module	Not documented; the endpoint reconciles
Undelivered output	Kept in a pending file and republished on restart	Processing continues past the failure

Operating record: name the component that deduplicates, and the component that reacts to an undo. With a message destination, neither one is the sink.

[10] Substreams:PubSub · [11] Substreams webhook sink

Use JSON when you are writing the consumer yourself

JSON is the destination with the widest range, because it covers three different situations that only look alike.

- **Inspection.** The CLI prints module output as JSON lines, which is how the chain pages on this site demonstrate a stream in one command. Nothing is stored and nothing is operated. This is the right first step on any new package.
- **JSONL files.** The ProtoJSON sink writes JSONL, and its current implementation lives in the files sink repository. The `protojson:<expression>` encoder extracts rows from a repeated field of the output message and writes one element per line, and the documented expression form today is `.<repeated_field_name>[]`. Operationally this is the files destination from the previous chapter, with a different encoder.
- **A consumer you write.** The Substreams:Stream service streams data directly into a JavaScript, Go, Rust or Python application through the SDKs.

The third case is the one to be deliberate about. Writing the consumer gives you every freedom and transfers every guarantee. The cursor, the resume position, the response to an undo signal, the idempotence of each write, the detection of a stalled stream and the restart procedure all become code your team maintains. The documented sinks already contain that work; an application reading the stream directly does not.

Choose it when the output's destination is your own process rather than a store, when the transformation that remains cannot be expressed in the package, or when no documented sink targets the system you need to reach. Choose a documented sink when the destination is one of the five above, because then the guarantees arrive with it.

Design question: if you write the consumer, which file or table holds the cursor, and who restores it?

[12] Substreams:Stream · [13] Substreams ProtoJSON sink · [9] substreams-sink-files

Choose from the question the consumer asks

Start from the query or the reaction that has to happen, then read backwards to the destination. Starting from the format produces a destination that works and a consumer that does not fit it.

WHAT THE CONSUMER DOES	DESTINATION	WHAT YOUR TEAM OPERATES
Reads rows by key, alongside application tables	Postgres	The database and its schema; the sink can be hosted
Runs analytical queries over wide ranges	ClickHouse	The database and its schema; the sink can be hosted
Loads finalized history into a warehouse or a notebook	Parquet	The sink process, the state store and the bucket
Several independent services react to the same events	A Pub/Sub topic	The sink process, the topic and every subscriber
One application reacts to events	A webhook	The sink process and an idempotent endpoint
Your own code reads the stream, or you are still exploring	JSON	Everything a sink would otherwise have handled

Three answers to write down before deploying

- **Who runs the consumer.** A hosted sink, or a process with an owner, an alert and a restart procedure.
- **What happens to a record the chain takes back.** Reverted by the sink, delivered as an undo notification, avoided by writing only finalized blocks, or reconciled by hand. One of those four, named.
- **Who deduplicates.** The destination's keys, the receiving endpoint, or nobody. If the answer is nobody, that is the first thing to fix.

Two destinations can be correct at once. A hosted sink into ClickHouse for the queries a product runs, and Parquet into a bucket for the archive an analytics team loads, are the same package with two sinks. That is cheaper than it sounds, because the modelling work is in the package, not in the destination.

The destination is also the most reversible decision in the pipeline. The package keeps producing the same output, so changing where it lands is a new sink and a backfill, not a rewrite. The decision that is expensive to reverse is the one about the model, which is upstream of everything in this guide.

For a discussion with StreamingFast, bring the destination you have chosen, who will operate the consumer, your finality requirement, and whether the database is reachable from outside your network. Those four answers are what determine whether a hosted sink applies to your case.

Next step: read the Hosted Sinks documentation if your destination is Postgres or ClickHouse, and the sink's own repository if it is not.

[3] Hosted Sinks · [1] Consuming Substreams and sink support · [14] Hosted Sinks announcement

REFERENCES

Sources and further reading

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

1. Consuming Substreams and sink support
<https://docs.substreams.dev/how-to-guides/sinks>
2. StreamingFast: Sink anywhere
<https://www.streamingfast.io/>
3. Hosted Sinks
<https://docs.substreams.dev/how-to-guides/hosted-services/hosted-sinks>
4. Hosted Services overview
<https://docs.substreams.dev/how-to-guides/hosted-services>
5. Managing your sink
<https://docs.substreams.dev/how-to-guides/hosted-services/hosted-sinks/manage-your-sink>
6. Hosted sink GraphQL layer
<https://docs.substreams.dev/how-to-guides/hosted-services/hosted-sinks/graphql-layer>
7. Substreams:SQL
<https://docs.substreams.dev/how-to-guides/sinks/sql>
8. SQL sink reorganization handling
<https://docs.substreams.dev/reference-material/sql/sql/reorg-handling>
9. substreams-sink-files
<https://github.com/streamingfast/substreams-sink-files>
10. Substreams:PubSub
<https://docs.substreams.dev/how-to-guides/sinks/pubsub>
11. Substreams webhook sink
<https://github.com/streamingfast/substreams/tree/develop/sink/webhook/README.md>
12. Substreams:Stream
<https://docs.substreams.dev/how-to-guides/sinks/stream>
13. Substreams ProtoJSON sink
<https://github.com/streamingfast/substreams/tree/develop/sink/protojson/README.md>
14. Hosted Sinks announcement
<https://www.streamingfast.io/blog/hosted-sinks-postgres-clickhouse>

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 [↗](#)