STREAMING FAST

GUIDE 07 · AGENT ACCESS

Give a coding agent access to chain data

The surfaces an agent can read, the specification and skills that are published, and what still needs a person.

BEFORE YOU START

WHO IT IS FOR	Developers who want a coding assistant to find, inspect and run blockchain data packages without being taught the stack again in every conversation.
WHAT YOU LEAVE WITH	A named surface for every question your agent will ask, the commands that produce machine-readable output, and a written boundary around what the agent cannot reach.
WHY IT MATTERS	A documentation page written for a person states a schema in prose. A machine consumer needs the same facts at an address it can request, in a format it can parse, with a credential it never has to hold. This guide covers the surfaces that are published today, and names the ones that are not.

CHAPTERS

01 The four surfaces an agent can read

02 Query the registry through its published specification

03 Inspect the package before the agent builds on it

04 Install the skills instead of pasting instructions

05 Choose an output encoding the agent can consume

06 Keep credentials out of the agent's context

07 State the limits of the agent's access

07 / 01

The four surfaces an agent can read

An agent working on a Substreams project reads four surfaces. Each one answers a different kind of question, and none of them substitutes for another.

SURFACE	WHAT IT CARRIES
The registry search API	Which packages exist, for which networks, and the exact reference to run.
The documentation in Markdown	Every page is served again at the same address with `.md` appended, and the whole set is indexed in one file.
The package itself	The module graph, the module kinds, the output types and the module hashes, read through the CLI.
The installable skills	The development knowledge that is in neither the registry nor the reference pages: manifests, chain-specific decoding, sink schemas, testing.

Decide which surface answers which question before handing over a task. "Find a package for this protocol on mainnet" is the registry. "What does this module output" is the package. "How do I write a store module" is a skill. An agent that has only the reference pages will reimplement a published package, because nothing in front of it said the package exists.

What separates these surfaces from a documentation site is addressability. The documentation serves a Markdown twin of each page and lists the full set in `llms.txt`, so retrieving a page does not mean parsing a rendered layout to find the schema inside it. The registry website organizes the same packages for a person, through datasets, chains, packages and projects; the search API is the same inventory without the layout.

> **Design output:** one line per question your agent is expected to answer, naming the surface that answers it.

[1] Substreams registry OpenAPI specification · [2] Substreams documentation index · [3] Substreams introduction · [4] Substreams package registry

Query the registry through its published specification

The registry publishes an OpenAPI document at `substreams.dev/v1/registry/openapi.yaml`. It describes itself as the public, agent-facing API of the package registry, served as REST and JSON, and it declares one operation: `GET /v1/registry/packages`.

The parameters it accepts

- **`query`** — free-text search over package name and keywords.
- **`network`** — matches a package whose latest release declares that network, default or not.
- **`organization`** and **`user`** — filter by publishing organization slug or publisher login.
- **`featured`** — restrict to featured packages.
- **`page`** and **`pageSize`** — one-based paging; the page size defaults to 24 and is capped at 100.

The response carries the matching `packages`, a `hasMore` flag and a `recommendations` list. Each package entry names the organization, the source repository, the latest version, the default network, every network the latest release declares, and two references to that same release.

FIELD	WHAT THE AGENT DOES WITH IT
`spkg`	The full package URL. The specification says to use this when executing a command programmatically.
`reference`	The short `slug@version` form. The specification says to use this when displaying a command to a human.
`networks`	Every network the release declares. Set the network explicitly when the default is not the one you want.
`orgScoped`	The package is reachable only through its organization, because another package holds the bare slug.

Recommendations are not search results. The specification states that they are curated, hand-authored answers to the query, returned alongside the ranked results, present only on the first page, and that the same package may appear in both lists — which is expected and is not a duplicate to filter out. An agent that de-duplicates by slug can discard the entry that was chosen by hand, together with the `reason` line written to be shown verbatim.

The search operation declares no security scheme, so a lookup works before anyone has provisioned a key. The specification also declares exactly one operation: there is no documented endpoint that returns a package's module graph. That question is answered against the package, in the next chapter.

> **Agent instruction:** search with the filters, execute with `spkg`, display `reference`, and record the version that was resolved.

> [1] Substreams registry OpenAPI specification · [5] StreamingFast: agentic Substreams development

---

07 / 03

Inspect the package before the agent builds on it

A package is a single file carrying its manifest, modules and protobuf definitions, with the extension `.spkg`. It is the unit the registry distributes and the unit the CLI runs, from a local path or a remote URL.

`substreams info` prints the contents of one for inspection. The documented output names the package, its version and its documentation string, then for each module the initial block, the kind, the output or value type, the update policy and the module hash. That is the contract a consumer has to match, available without streaming a single block. `substreams graph` prints the module dependency graph in Mermaid syntax, which is a graph an agent can read as text. `substreams inspect` reaches into the package file structure and is documented as a debugging tool.

Every command that accepts a manifest also reads one from standard input when the manifest path is `"-"`: `build`, `run`, `gui`, `info`, `graph`, `pack`, `protogen`, `inspect`, `registry publish`, `registry verify`, `service deploy` and `service update`. An agent that generates a manifest can pipe it straight into the command. One documented consequence: in standard input mode, paths inside the manifest resolve against the current working directory rather than the manifest's original location.

Keep the module hash next to the version in whatever the agent writes down. A package reference identifies a release; the hash identifies the module whose output you actually consumed.

> **Review question:** can the agent state the kind, output type and hash of every module it consumes, before it runs anything?

> [6] Substreams CLI reference · [7] Substreams packages reference

---

07 / 04

Install the skills instead of pasting instructions

StreamingFast publishes its Substreams expertise as agent skills: open-source knowledge packages, under the Apache 2.0 licence, in one repository. They are installed, not pasted, so the knowledge arrives with the project instead of with the conversation.

How they are installed

- As a Claude Code plugin: `claude plugin marketplace add streamingfast/substreams-skills`, then `claude plugin install substreams-dev@streamingfast-substreams`.

Installing the base plugin pulls the skills it defines.

- From a clone, without installing: `git clone` the repository, then start the assistant with `claude --plugin-dir ./substreams-skills`.
- In Cursor, by adding the skill directory path in settings. In VS Code 1.107 and later, skills are an experimental feature that takes configured paths.

The repository ships twelve skill directories. The README documents the nine below, which split along the lines a real project splits along: cross-cutting development, the two chain families, the sinks, the deployment paths, the account API and testing.

SKILL	WHAT IT COVERS
`substreams-dev`	Manifests, module graphs, protobuf schema design, and routing to the chain-specific packs.
`substreams-ethereum`	EVM contracts: ABI code generation, event decoding, contract calls, token metadata stores.
`substreams-solana`	Solana programs: instruction walking including inner instructions, program filters, account layouts.
`substreams-sql`	Database Changes and from-proto mapping modes, PostgreSQL and ClickHouse schema design.
`substreams-sink`	Consuming output in Go, JavaScript, Python or Rust, with cursor and reorganization handling.
`substreams-sink-deploy-local`	Running a sink binary yourself: sink selection, production patterns, common pitfalls.
`substreams-hosted-sink`	Deploying and operating a sink on hosted infrastructure through the Portal API.
`thegraph-market-api`	Account access: authentication, usage and billing reads, and the hosted deployment lifecycle.
`substreams-testing`	Unit, integration and performance testing, and continuous integration.

The published lists do not agree with each other. The documentation page on agent skills describes three, the repository's README documents nine, and its `skills/` directory holds twelve: `substreams-bitcoin`, `substreams-convert` and `substreams-dev-legacy` ship without an entry of their own. Read the directory, pin the commit you installed, and treat any page's list as a snapshot of the day it was written.

A skill changes what the assistant knows, not what it is allowed to reach. Installing one grants no credential, no network access and no deployment right. Those are the subject of chapter six.

> **Decision record:** which skills are installed, at which commit, and who is responsible for updating them.

> [8] Substreams: agent skills · [9] streamingfast/substreams-skills · [5] StreamingFast: agentic Substreams development

07 / 05

Choose an output encoding the agent can consume

`substreams run` takes an output flag. The documented options are `ui`, a formatted interface that displays progress information and execution logs; `json`, an indented stream of data that displays neither; and `jsonl`, the same data with one output per line. Request `jsonl` for a machine consumer: it is the form that can be read incrementally, and the form that does not interleave logs with data the agent is parsing.

An encoding carries decisions with it. The documented ProtoJSON example writes JSONL in which address fields hold base64 strings rather than hexadecimal — `"from": "ZfU/nt+BtrSyp9QMPKVgVNTJO5o="` — because that is how ProtoJSON encodes bytes. An agent that assumes a `0x` prefix will compare strings that never match, and report an empty result instead of an error. Decide where that conversion happens, once, and state it in the schema rather than in the consumer.

Name the destination before the agent starts

- **A stream the agent parses.** Fine for inspection and for a narrow task. Nothing survives the end of the process.
- **Files.** Output lands in a directory, with the block range in the filename, and can be re-read without touching the service again.
- **A database.** The destination becomes the surface: the assistant queries SQL afterwards instead of re-reading a stream, which is usually what makes an agent useful on the second question.

The sink documentation also marks which integrations are officially supported and which are community-maintained. That distinction belongs in the agent's instructions, not in its guesses.

> **Output contract:** the encoding, the field types after encoding, the destination, and the one place a conversion is allowed to happen.

> [6] Substreams CLI reference · [10] Substreams ProtoJSON sink · [11] Consuming Substreams

---

07 / 06

Keep credentials out of the agent's context

Streaming access and account access are two different credentials, obtained two different ways. Both are designed so that a password never reaches the assistant.

For streaming, `substreams auth` opens a browser so a person signs in, picks an organization if they belong to more than one, and selects an API key. The documentation is explicit about the mechanism: the CLI retrieves the selected key over the API rather than having you copy and paste it, exchanges it for a JWT, and writes the token to `.substreams.env` as `SUBSTREAMS_API_TOKEN`. Other commands read that file

when it sits next to the manifest or in the working directory. The documentation also says to add it to `.gitignore`. If a token already exists, `substreams auth --paste` accepts a JWT or an API key, and the environment variable can be set directly.

For account operations, the account skill documents the OAuth 2.0 device authorization grant: the agent starts a device-code login, hands the person a URL and a code to approve in their browser, waits for them to confirm the approval, then holds a short-lived organization-scoped access token alongside a rotating refresh token. The repository states that this is the Portal administration login only, and that it is separate from the streaming authentication produced by `substreams auth`.

Two rules worth writing down

- **A token file in the working directory is a token the agent can read.** Scope the key to the task, and keep a separate key for anything the agent drives.
- **An expiry is a feature.** Record the lifetime, so a stalled agent fails on a dead token instead of resuming a session nobody is watching.

> **Access record:** which credential, which scope, which expiry, which file it lands in, and who can revoke it.

> [12] Substreams CLI authentication · [9] streamingfast/substreams-skills

---

07 / 07

State the limits of the agent's access

Everything above is documented. Several things an agent would find convenient are not, and the useful habit is to write that difference down instead of letting the assistant infer it.

- **The registry specification covers search.** One documented operation. Package inspection runs through the CLI against the package, not through an HTTP endpoint, and an agent should not be told otherwise.
- **The account API has no published specification at a documented address.** The account skill carries inlined snapshots of the upstream definitions and says so: when a call returns `invalid_argument`, or a documented field is missing, the skill may be out of date, and there is no automatic check for that drift.
- **What is published here is skills, not a protocol server.** The agent surface of the registry itself is the OpenAPI document above. MCP exists elsewhere in the same family: the Token API on The Graph Market announces itself MCP-ready for AI agents, and Pinax publishes an MCP client for the services reachable through the market. A community MCP server for searching the registry also exists, published by another author under the MIT licence; it is a third-party project, and presenting it to your team as one is still the right call.
- **The documentation list trails the repository.** That is true of any fast-moving surface, and it is the reason to pin a commit rather than cite a page.

Keep the confirmation step for anything destructive. The hosted deployment skill documents an explicit confirmation protocol before a destructive mutation, which means the published design already assumes a person is reachable at that moment. Do not remove that assumption to make an automation tidier.

Last, an agent that compiles is not an agent that is right. The surfaces in this guide let it find a package, read a schema, write a consumer and deploy a sink, once a person has approved the credential in a browser. None of them tells you whether the resulting rows mean what your product thinks they mean. Reconcile the output against real transactions before the dataset is treated as true, whoever wrote it.

> **Next step:** read the registry specification, install the skills, then write the one page that lists the surfaces the agent may read, the credentials it may hold, and the actions that still need a person.

> [1] Substreams registry OpenAPI specification · [9] streamingfast/substreams-skills · [13] Community MCP server for the registry · [14] The Graph Market: Token API · [15] Pinax MCP client

---

REFERENCES

Sources and further reading

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

1. Substreams registry OpenAPI specification
   https://substreams.dev/v1/registry/openapi.yaml
2. Substreams documentation index
   https://docs.substreams.dev/llms.txt
3. Substreams introduction
   https://docs.substreams.dev/
4. Substreams package registry
   https://substreams.dev/
5. StreamingFast: agentic Substreams development
   https://www.streamingfast.io/blog/leveling-up-agentic-substreams-development
6. Substreams CLI reference
   https://docs.substreams.dev/reference-material/command-line-interface
7. Substreams packages reference
   https://docs.substreams.dev/reference-material/manifest-and-components/packages
8. Substreams: agent skills
   https://docs.substreams.dev/how-to-guides/develop-your-own-substreams/general/agent-skills
9. streamingfast/substreams-skills
   https://github.com/streamingfast/substreams-skills

10. Substreams ProtoJSON sink
https://docs.substreams.dev/how-to-guides/sinks/protojson

11. Consuming Substreams
https://docs.substreams.dev/how-to-guides/sinks

12. Substreams CLI authentication
https://docs.substreams.dev/how-to-guides/installing-the-cli/authentication

13. Community MCP server for the registry
https://github.com/PaulieB14/substreams-search-mcp

14. The Graph Market: Token API
https://thegraph.market/token-api

15. Pinax MCP client
https://github.com/pinax-network/pinax-mcp

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 ↗