Streams

- Install the capability
- Write chunks
  - Resume a producer
- Read chunks
- Serve over SSE
- Find and delete streams
  - Delete a stream when its content moves
- Storage
- On the Agent class

import { Render, TypeScriptExample } from "~/components";

<Render file="experimental" product="agents" params={{ name: "Streams", module: "agents/streams" }} />

`Streams` keeps a reply that is still being written, such as model tokens, so nobody loses it. If a user refreshes the page or drops their connection mid-reply, they reconnect and pick up where they left off, instead of losing the reply or starting over. If the Durable Object restarts mid-reply, the chunks already written are kept, and the reply can continue from there. The Think harness streams its replies this way.

`Streams` from `agents/streams` stores each stream as an ordered log of chunks in the Durable Object's SQLite database. A reader replays the chunks from any position, then follows new chunks as they are appended, until the stream ends.

Install the capability

```
import { DurableObject } from "cloudflare:workers";
import { streamText } from "ai";
import { Lifecycle } from "agents/lifecycle";
```

```
import { createAI } from "agents/models/ai-sdk";
import { Streams, sseResponse } from "agents/streams";

export class ReportAgent extends DurableObject<Env> {
  readonly streams = new Streams();
  readonly lifecycle = Lifecycle.install(this).use(this.streams);

  async onRequest(request: Request): Promise<Response> {
    const url = new URL(request.url);
    const id = url.searchParams.get("id") ?? "report";

    if (request.method === "POST") {
      const { prompt } = await request.json<{ prompt: string }>();
      this.ctx.waitUntil(this.#generate(id, prompt));
      return Response.json({ id }, { status: 202 });
    }

    // Replay what is stored, then follow live chunks until the stream ends.
    return sseResponse(this.streams, id, { request });
  }

  async #generate(id: string, prompt: string): Promise<void> {
    const stream = await this.streams.open(id);
    try {
      const ai = createAI({ binding: this.env.AI });
      const result = streamText({
        model: ai("anthropic/claude-opus-5.5"),
        prompt,
      });
      for await (const text of result.textStream) {
        stream.append(text);
      }
      stream.close();
    } catch (error) {
      stream.error(String(error));
    }
  }
}
```

Write chunks

`open(id)` returns a `StreamWriter`:

Member	Description
`append(chunk)`	Stores one chunk and wakes live readers. The write is synchronous. Returns the chunk's sequence number.
`close(options?)`	Marks the stream `completed`. Readers finish after the last chunk.
`error(reason?, options?)`	Marks the stream `errored` and stores `reason`.
`cursor`	The number of chunks stored so far, which is also the sequence number of the next chunk.
`streamId`	The stream's ID.

A chunk is any JSON value, up to 1 MiB by default. Set another limit with `new Streams({ maxChunkBytes })`.

`open()` accepts `metadata`, a JSON object stored with the stream, and `tag`. It is safe to call more than once:

- Opening a live stream again returns a writer at its current cursor.
- Opening a settled stream throws `StreamClosedError`.
- Closing or erroring a settled stream does nothing.

Resume a producer

Because the cursor is durable, a producer that restarts after an eviction can continue where the last one stopped. Start the loop at `stream.cursor`:

```
import type { Streams } from "agents/streams";

async function produce(streams: Streams, id: string, total: number) {
  const stream = await streams.open(id);
  // Chunks before the cursor were stored by an earlier run.
  for (let i = stream.cursor; i < total; i++) {
    stream.append({ step: i });
  }
  stream.close();
}
```

Read chunks

`read(id, options)` replays the stored chunks, then follows live appends, and ends when the stream settles. Each item is a `StreamChunk` with a `seq` and the `chunk` value:

```
import type { Streams } from "agents/streams";

async function collect(streams: Streams, id: string, from = 0) {
  const chunks: unknown[] = [];
  for await (const { seq, chunk } of streams.read(id, { from })) {
    chunks.push(chunk);
    console.log(`chunk ${seq}`);
  }
  return chunks;
}
```

Pass `signal` to stop following a live stream.

When each write downstream costs something, such as an SSE flush or an RPC call, read in batches. `readBatches()` yields arrays. During replay each array has up to `batchSize` chunks, 100 by default. While following a live stream, each array holds everything appended since the last one:

```
import type { StreamChunk, Streams } from "agents/streams";

async function forward(
  streams: Streams,
  id: string,
  send: (batch: StreamChunk[]) => Promise<void>,
) {
  for await (const batch of streams.readBatches(id, {
    batchSize: 50,
    onUpToDate: () => console.log("caught up"),
  })) {
    await send(batch);
  }
}
```

`onUpToDate` runs once, the first time the reader reaches the end of the stored chunks. A live stream is up to date while the reader follows it, so use it to switch a UI from replay to live.

Serve over SSE

`sseResponse(streams, id, options)` returns a `Response` that serves the whole stream as Server-Sent Events. It replays stored chunks, then follows live appends:

```
sequenceDiagram
    accTitle: Reconnect to a durable stream
    accDescr: After receiving chunk 2, a browser disconnects while the producer stores chunks 3 and 4. It reconnects with Last-Event-ID 2, replays the missing chunks, and then receives live output.
    participant B as Browser
    participant S as Streams via SSE
    S-->>B: Chunk 2, id: 2
    Note over B,S: Connection drops
    Note right of S: Producer stores chunks 3 and 4
    B->>S: Reconnect with Last-Event-ID: 2
    S-->>B: Replay chunks 3 and 4
    S-->>B: up-to-date
    Note right of S: Producer appends chunk 5
    S-->>B: Live chunk 5
    Note right of S: Producer closes or errors the stream
    S-->>B: done or error
```

Replay starts after `Last-Event-ID`, or at the sequence number in `?from=` if no valid header is present. The `from` option overrides both and names the first chunk to send. After replay, the response sends an `up-to-date` event and follows live appends. Heartbeat comments keep idle proxies from closing the connection. The response finishes with a `done` event, or an `error` event that carries the stored reason.

Each chunk's sequence number is its SSE `id`. A browser `EventSource` that reconnects sends `Last-Event-ID` on its own, so it continues from the next chunk without any client code. Pass `request` so the response reads those values and stops when the client disconnects.

Option	Description
`request`	The incoming request. Supplies `Last-Event-ID`, `?from=`, and the abort signal
`from`	The first sequence number to send
`signal`	Stops following the stream
`batchSize`	Chunks per read during replay
`heartbeatMs`	Interval between heartbeat comments

For other transports, such as a WebSocket, pipe `read()` or `readBatches()` yourself.

Find and delete streams

Method	Description
`status(id)`	Returns the stream's `StreamStatus`, or `null`
`list(options?)`	Lists streams, newest first. Filter by `state`, or by `tag`, and cap the results with `limit`
`delete(id)`	Deletes a settled stream and its chunks. Returns `false` when no stream matched

`StreamStatus` has the stream's `state` (`"streaming"`, `"completed"`, or `"errored"`), `cursor`, `tag`, `metadata`, `error`, and timestamps.

A tag groups streams that belong to one operation, such as a request ID. It is not unique. When an operation produces successive streams, for example a retried reply, tag each one and find the latest with `list({ tag, limit: 1 })`. A stream's tag is fixed when it is created.

Delete a stream when its content moves

A stream is usually temporary. Once its content has been saved somewhere else, such as a message in a conversation, delete it. `close()` and `error()` accept two options that do this in one SQLite transaction:

- `commit` runs your own synchronous writes. It must not await.
- `discard: true` deletes the stream's rows.

```
import type { StreamWriter } from "agents/streams";

function finish(stream: StreamWriter, sql: SqlStorage, text: string) {
  stream.close({
    commit: () => {
      sql.exec("INSERT INTO replies (text) VALUES (?)", text);
    },
    discard: true,
  });
}
```

Either the reply is stored and the stream is gone, or `commit` threw, nothing changed, and the stream is still live.

Storage

Chunks are stored in blocks. One row holds chunks until it reaches 256 KB, then the next append starts a new row. Each append writes one row, but a stream of thousands of chunks is only a few rows, so deleting it is cheap.

The capability does not delete old streams on its own. Delete them with `delete()`, or with `discard` when they settle.

On the Agent class

`AIChatAgent` and Think store each in-progress reply in `Streams`, and move it into the conversation when the reply ends. To add `Streams` to an `Agent`, construct it as a field and install it with `this.lifecycle.use()` in the constructor.

□□□ #2
□ □□□ □□□ 2026-10-31 23:30:22 UTC
□ □□□ □□□ 2026-10-31 23:30:23 UTC