Stream output

- When to use streaming
- Stream command execution
- Stream to client
- Stream process logs
- Handle errors
- Best practices
- Related resources

This guide shows you how to handle real-time output from commands, processes, and code execution.

When to use streaming

Use streaming when you need:

- **Real-time feedback** - Show progress as it happens
- **Long-running operations** - Builds, tests, installations that take time
- **Interactive applications** - Chat bots, code execution, live demos
- **Large output** - Process output incrementally instead of all at once
- **User experience** - Prevent users from waiting with no feedback

Use non-streaming (`exec()`) for:

- **Quick operations** - Commands that complete in seconds
- **Small output** - When output fits easily in memory
- **Post-processing** - When you need complete output before processing

Stream command execution

Use `execStream()` to get real-time output:

```
import { getSandbox, parseSSEStream, type ExecEvent } from '@cloudflare/sandbox';

const sandbox = getSandbox(env.Sandbox, 'my-sandbox');

const stream = await sandbox.execStream('npm run build');

for await (const event of parseSSEStream<ExecEvent>(stream)) {
  switch (event.type) {
    case 'stdout':
      console.log(event.data);
      break;

    case 'stderr':
      console.error(event.data);
      break;

    case 'complete':
      console.log('Exit code:', event.exitCode);
      break;

    case 'error':
      console.error('Failed:', event.error);
      break;
  }
}
```

Stream to client

Return streaming output to users via Server-Sent Events:

```
import { getSandbox } from '@cloudflare/sandbox';

export { Sandbox } from '@cloudflare/sandbox';

export default {
  async fetch(request: Request, env: Env): Promise<Response> {
    const sandbox = getSandbox(env.Sandbox, 'builder');
```

```
    const stream = await sandbox.execStream('npm run build');

    return new Response(stream, {
      headers: {
        'Content-Type': 'text/event-stream',
        'Cache-Control': 'no-cache'
      }
    });
  }
};
```

Client-side consumption:

```
// Browser JavaScript
const eventSource = new EventSource('/build');

eventSource.addEventListener('stdout', (event) => {
  const data = JSON.parse(event.data);
  console.log(data.data);
});

eventSource.addEventListener('complete', (event) => {
  const data = JSON.parse(event.data);
  console.log('Exit code:', data.exitCode);
  eventSource.close();
});
```

Stream process logs

Monitor background process output:

```
import { parseSSEStream, type LogEvent } from '@cloudflare/sandbox';

const process = await sandbox.startProcess('node server.js');

const logStream = await sandbox.streamProcessLogs(process.id);

for await (const log of parseSSEStream<LogEvent>(logStream)) {
```

```
  console.log(log.data);

  if (log.data.includes('Server listening')) {
    console.log('Server is ready');
    break;
  }
}
```

Handle errors

Check exit codes and handle stream errors:

```
const stream = await sandbox.execStream('npm run build');

for await (const event of parseSSEStream<ExecEvent>(stream)) {
  switch (event.type) {
    case 'stdout':
      console.log(event.data);
      break;

    case 'error':
      throw new Error(`Build failed: ${event.error}`);

    case 'complete':
      if (event.exitCode !== 0) {
        throw new Error(`Build failed with exit code ${event.exitCode}`);
      }
      break;
  }
}
```

Best practices

- **Always consume streams** - Don't let streams hang unconsumed
- **Handle all event types** - Process stdout, stderr, complete, and error events
- **Check exit codes** - Non-zero exit codes indicate failure

- **Provide feedback** - Show progress to users for long operations

Related resources

- Commands API reference - Complete streaming API
- Execute commands guide - Command execution patterns
- Background processes guide - Process log streaming
- Code Interpreter guide - Stream code execution output

□□□ #3
□ □□□ □□□ 2026-11-01 04:12:38 UTC
□ cocobook-sync □□□ 2026-11-04 17:48:55 UTC