Cribl LogStream Documentation Manual

Version: v3.1

Generated: 2021-08-10 21:39:04

INTRODUCTION

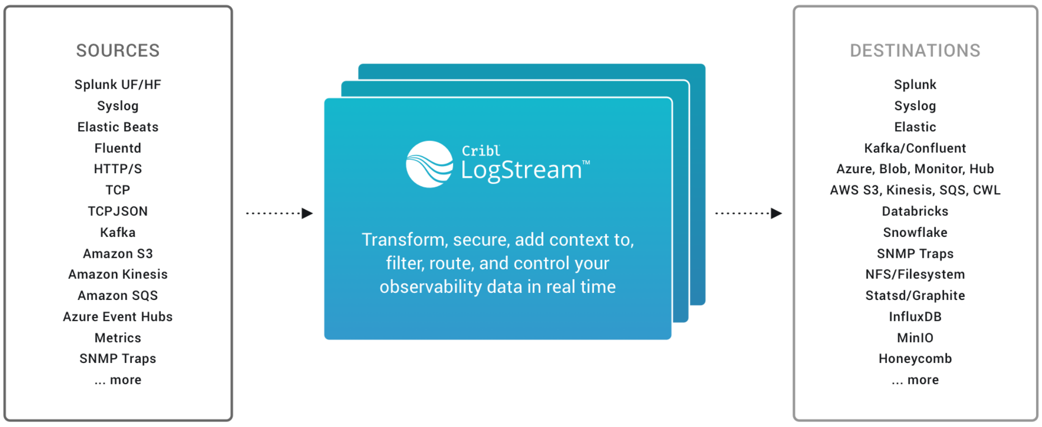
About Cribl LogStream

Getting started with Cribl LogStream

What Is Cribl LogStream?

Cribl LogStream helps you process machine data – logs, instrumentation data, application data, metrics, etc. – in real time, and deliver them to your analysis platform of choice. It allows you to:

- Add context to your data, by enriching it with information from external data sources.
- Help secure your data, by redacting, obfuscating, or encrypting sensitive fields.
- Optimize your data, per your performance and cost requirements.



Sources, LogStream, destinations

Cribl LogStream ships in a single, no-dependencies package. It provides a refreshing and modern interface for working with and transforming your data. It scales with – and works inline with – your existing infrastructure, and is transparent to your applications.

Who Is Cribl LogStream For?

Cribl LogStream is built for administrators, managers, and users of operational/DevOps and security intelligence products and services.

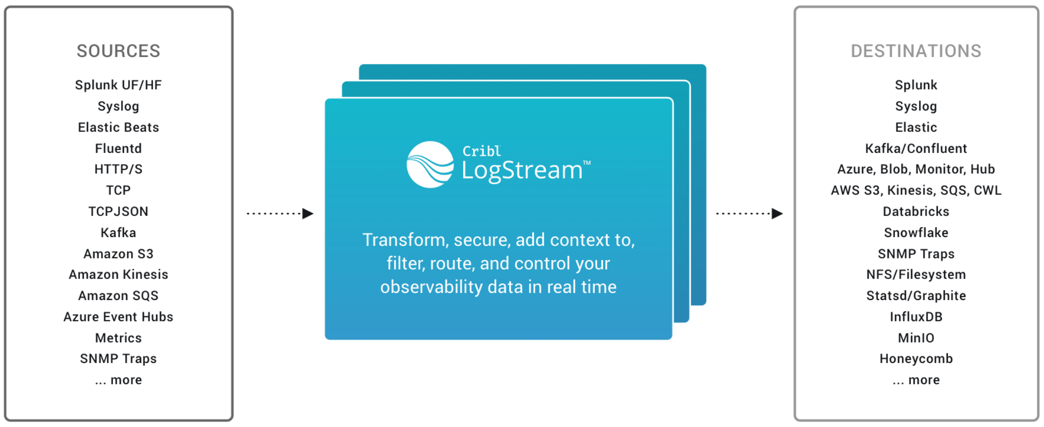
What's Next

> Basic Concepts

Basic Concepts

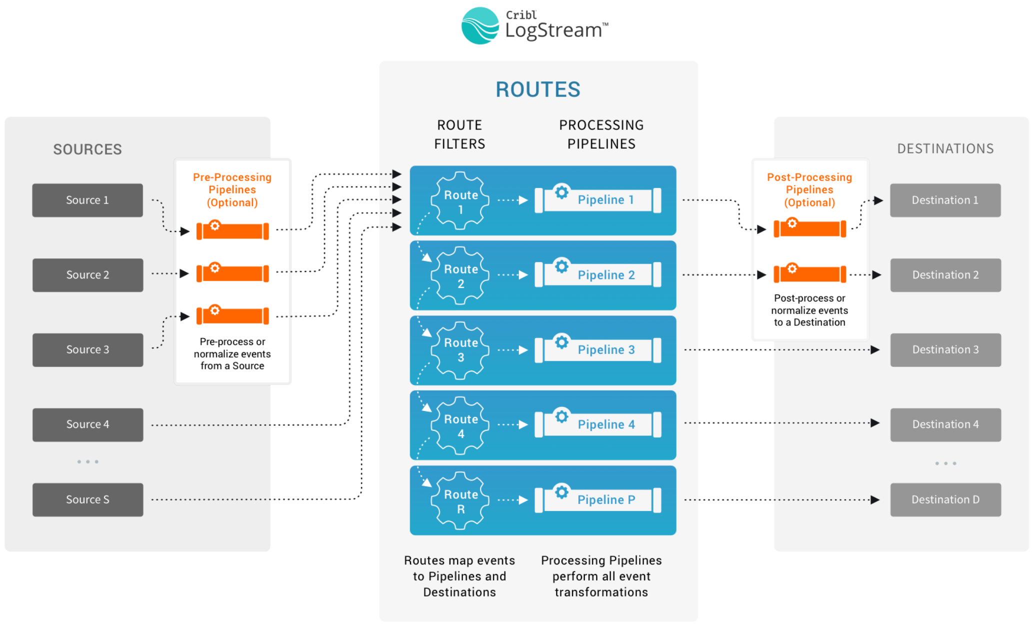
Notable features and concepts to get a fundamental understanding of Cribl LogStream

As we describe features and concepts, it helps to have a mental model of Cribl LogStream as a system that receives events from various sources, processes them, and then sends them to one or more destinations.



Sources, LogStream, destinations

Let's zoom in on the center of the above diagram, to take a closer look at the processing and transformation options that LogStream provides internally. The basic interface concepts to work with are Routes, which manage data flowing through Pipelines, which consist of Functions.



Cribl LogStream™
SOURCES
Source 1
Source 2
Source 3
Source 4
Source S
Pre-Processing Pipelines (Optional)
Pre-process or normalize events from a Source
ROUTES
ROUTE FILTERS
PROCESSING PIPELINES
Route 1
Route 2
Route 3
Route 4
Route R
Pipeline 1
Pipeline 2
Pipeline 3
Pipeline 4
Pipeline P
Routes map events to Pipelines and Destinations
Processing Pipelines perform all event transformations
DESTINATIONS
Post-Processing Pipelines (Optional)
Post-process or normalize events to a Destination
Destination 1
Destination 2
Destination 3
Destination 4
Destination D

Routes, Pipelines, Functions

Routes

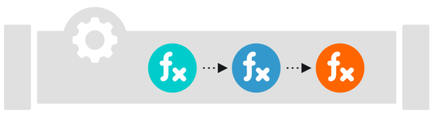
Routes evaluate incoming events against **filter** expressions to find the appropriate Pipeline to send them to. Routes are **evaluated in order**. Each Route can be associated **with only one** Pipeline and one output (configured as a LogStream **Destination**).

By default, each Route is created with its `Final` flag set to `Yes`. With this setting, a Route-Pipeline-Destination set will consume events that match its filter, and that's that.

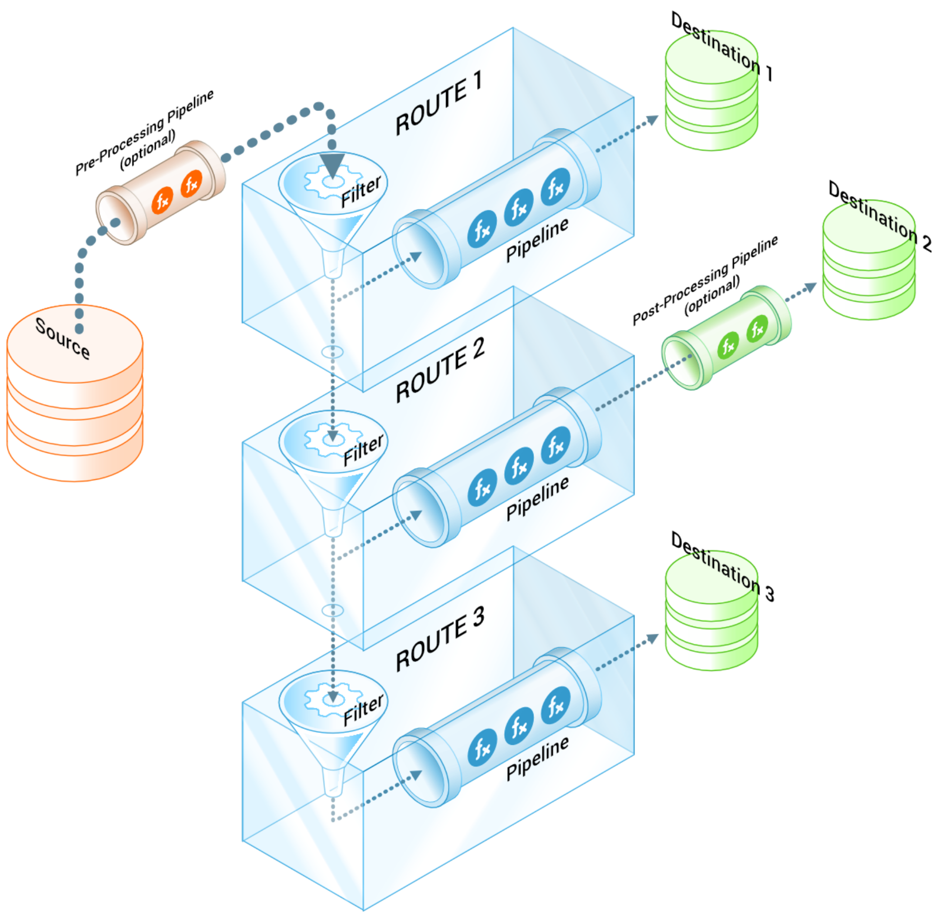
However, if you disable the Route's `Final` flag, one or more event **clones** will be sent down the associated Pipeline, while the original event continues down LogStream's Routing table to be evaluated against other configured Routes. This is very useful in cases where the same set of events needs to be processed in multiple ways, and delivered to different destinations. For more details, see Routes.

Pipelines

A series of Functions is called a Pipeline, and the order in which you specify the Functions determines their execution order. Events are delivered to the beginning of a Pipeline by a Route, and as they're processed by a Function, the events are passed to the next Function down the line.



Pipelines attached to Routes are called **processing Pipelines**. You can optionally attach **pre-processing Pipelines** to individual LogStream Sources, and attach **post-processing Pipelines** to LogStream Destinations. All Pipelines are configured through the same UI – these three designations are determined only by a Pipeline's placement in LogStream's data flow.

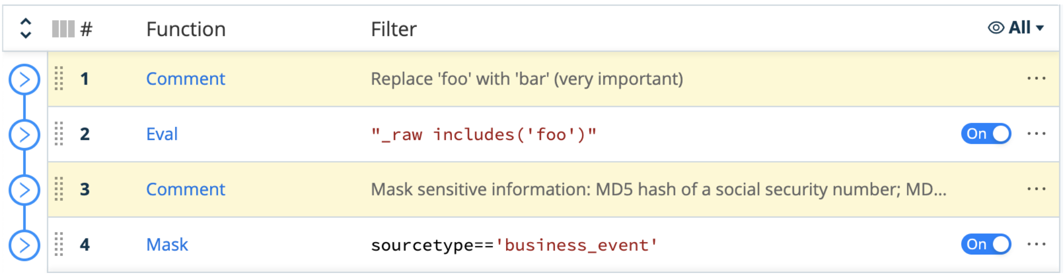


Pipelines categorized by position

Events only move forward – toward the end of a Pipeline, and eventually out of the system. For more details, see Pipelines.

Functions

At its core, a **Function** is a piece of code that executes on an event, and that encapsulates the smallest amount of processing that can happen to that event. For instance, a very simple Function can be one that replaces the term `foo` with `bar` on each event. Another one can hash or encrypt `bar`. Yet another function can add a field – say, `dc=jfk-42` – to any event with `source=*us-nyc-application.log`.



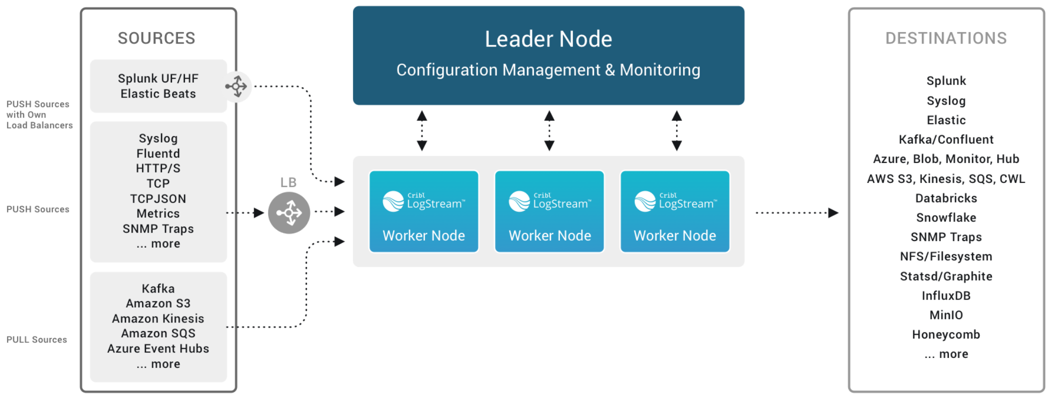
#	Function	Filter	All
1	Comment	Replace 'foo' with 'bar' (very important)	
2	Eval	`"_raw includes('foo')"`	On
3	Comment	Mask sensitive information: MD5 hash of a social security number; MD...	
4	Mask	`sourcetype=='business_event'`	On

Functions stacked in a Pipeline

Functions process each event that passes through them. To help improve performance, Functions can optionally be configured with filters, to limit their processing scope to matching events only. For more details, see Functions.

A Scalable Model

You can scale LogStream up to meet enterprise needs in a distributed deployment. Here, multiple LogStream Workers (instances) share the processing load. But as you can see in the preview schematic below, even complex deployments follow the same basic model outlined above.



Distributed deployment architecture

What's Next

> Getting Started Guide

> Deployment Types

Getting Started Guide

This guide walks you through planning, installing, and configuring a single-instance deployment of Cribl LogStream. You'll capture some realistic sample log data, and then use LogStream's built-in Functions to redact, parse, refine, and shrink the data.

By the end of this guide, you'll have assembled all of LogStream's basic building blocks: a Source, Route, Pipeline, several Functions, and a Destination. You can complete this tutorial using LogStream's included sample data, without connections to – or licenses on – any inbound or outbound services.

Assuming a cold start (from initial LogStream download and installation), this guide might take about an hour. But you can work through it in chunks, and LogStream will persist your work between sessions.

> If you've already downloaded, installed, and launched LogStream, skip ahead to Add a Source.

Requirements for this Tutorial

The minimum requirements for running this tutorial are the same as for a LogStream production single-instance deployment.

OS (Intel Processors)

- Linux 64-bit kernel >= 3.10 and glibc >= 2.17
- Examples: Ubuntu 16.04, Debian 9, RHEL 7, CentOS 7, SUSE Linux Enterprise Server 12+, Amazon Linux 2014.03+

OS (ARM64 Processors)

- Linux 64-bit
- Tested so far on Ubuntu (14.04, 16.04, 18.04, and 20.04) and Amazon Linux 2

System

- +4 physical cores = +8 vCPUs; +8GB RAM – all beyond your basic OS/VM requirements
- 5GB free disk space (more if persistent queuing is enabled)

> i We assume that 1 physical core is equivalent to 2 virtual/hyperthreaded CPUs (vCPUs). For details, see Recommended AWS, Azure, and GCP Instance Types.

Browser Support

- Firefox 65+, Chrome 70+, Safari 12+, Microsoft Edge

Network Ports

By default, LogStream listens on the following ports:

Component	Default Port
UI default	`9000`
HTTP Inbound, default	`10080`
User options	+ Other data ports as required.

You can override these defaults as needed.

Plan for Production

For higher processing volumes, users typically enable LogStream's Distributed Deployment option. While beyond the scope of this tutorial, that option has a few additional requirements, which we list here for planning purposes:

- Port `4200` must be available on the Leader Node for Workers' communications.

- Git (1.8.3.1 or higher) must be installed on the Leader Node, to manage configuration changes.

See Sizing and Scaling for further details about configuring LogStream to handle large data streams.

Download and Install LogStream

Download the latest version of LogStream at https://cribl.io/download/.

Un-tar the resulting `.tgz` file in a directory of your choice (e.g., `/opt/`). Here's general syntax, then a specific example:

```
tar xvzf cribl-<version>-<build>-<arch>.tgz
tar xvzf cribl-2.3.1-1d4e05c5-linux-x64.tgz
```

You'll now have LogStream installed in a `cribl` subdirectory, by default: `/opt/cribl/`. We'll refer to this `cribl` subdirectory throughout this documentation as `$CRIBL_HOME`.

Run LogStream

In your terminal, switch to the `$CRIBL_HOME/bin` directory (e.g,: `/opt/cribl/bin`). Here, you can start, top, and verify the LogStream server using these basic `./cribl` CLI commands:

- **Start**: `./cribl start`
- **Stop**: `./cribl stop`
- **Get status**: `./cribl status`

> For other available commands, see CLI Reference.

Next, in your browser, open `http://<hostname>:9000` (e.g., `http://localhost:9000`) and log in with default credentials (`admin`, `admin`).

Register your copy of LogStream if desired.

After registering, you'll be prompted to change the default password.

You are now ready to configure a working LogStream installation – with a Source, Destination, Pipeline, and Route – and to assemble several built-in

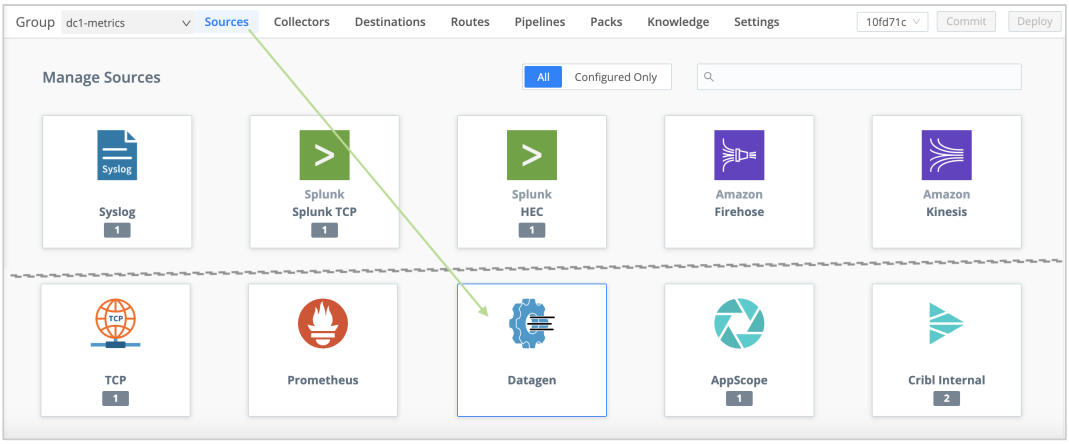
Functions to refine sample log data.

Get Data Flowing

Add a Source

Each LogStream Source represents a data input. Options include Splunk, Elastic Beats, Kinesis, Kafka, syslog, HTTP, TCP JSON, and others.

For this tutorial, we'll enable a LogStream built-in datagen (i.e., data generator) that generates a stream of realistic sample log data.



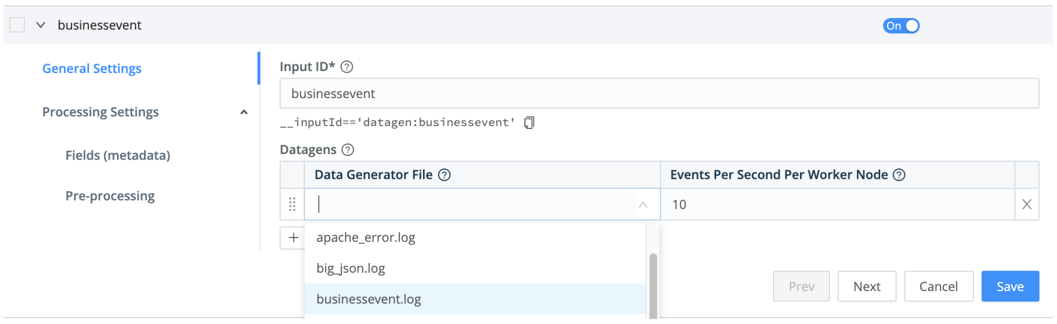
Adding a datagen Source

1. From LogStream's top menu, select **Data Sources**.
2. From the **Data Sources** page's tiles or left menu, select **Datagens**.

   (You can use the search box to jump to the **Datagens** tile.)
3. Click **+ Add New** to open the **New Datagen source** pane.
4. In the **Input ID** field, name this Source `businessevent` .
5. In the **Data Generator File** drop-down, select `businessevent.log` .

   This generates...log events for a business scenario. We'll look at their structure shortly, in Capture and Filter Sample Data.
6. Click **Save**.

The **On** slider in the **Enabled** column indicates that your datagen Source has started generating sample data.



Input ID*

businessevent

`__inputId=='datagen:businessevent'`

Datagens

Data Generator File	Events Per Second Per Worker Node
	10

apache_error.log
big_json.log
businessevent.log

Configuring a datagen Source

Add a Destination

Each LogStream Destination represents a data output. Options include Splunk, Kafka, Kinesis, InfluxDB, Snowflake, Databricks, TCP JSON, and others.

For this tutorial, we'll use LogStream's built-in **DevNull** Destination. This simply discards events – not very exciting! But it simulates a real output, so it provides a configuration-free quick start for testing LogStream setups. It's ideal for our purposes.

To verify that **DevNull** is enabled, let's walk through setting up a Destination, then setting it up as LogStream's default output:

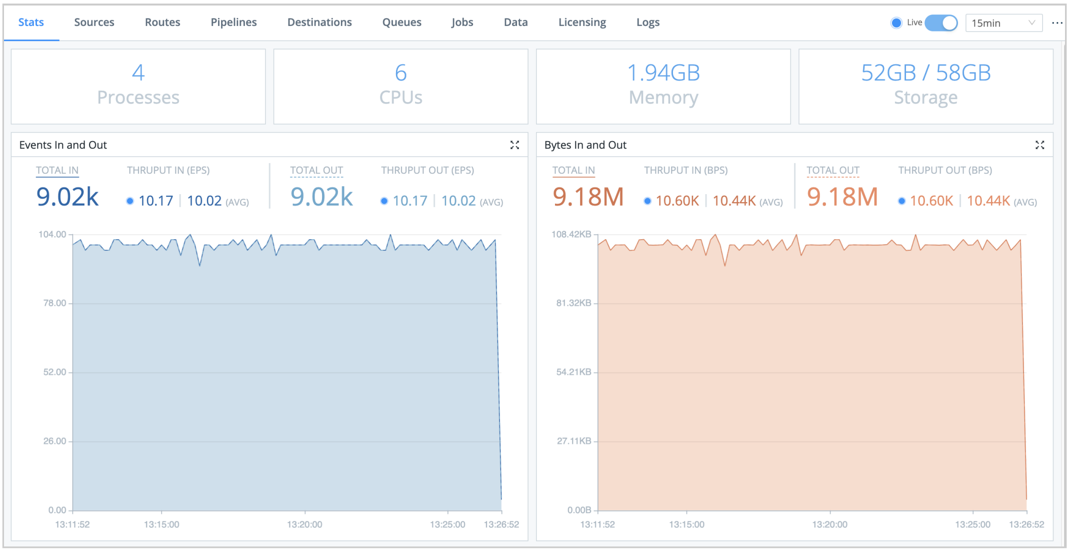
1. From LogStream's top menu, select **Destinations**.
2. Select **DevNull** from the **Data Destinations** page's tiles or left menu.

   (You can use the search box to jump to the **DevNull** tile.)
3. On the resulting **devnull** row, look for the **Live** indicator under **Enabled**. This confirms that the **DevNull** Destination is ready to accept events.
4. From the **Data Destinations** page's left nav, select the **Default** Destination at the top.
5. On the resulting **Manage Default Destination** page, verify that the **Default Output ID** drop-down points to the **devnull** Destination we just examined.

We've now set up data flow on both sides. Is data flowing? Let's check.

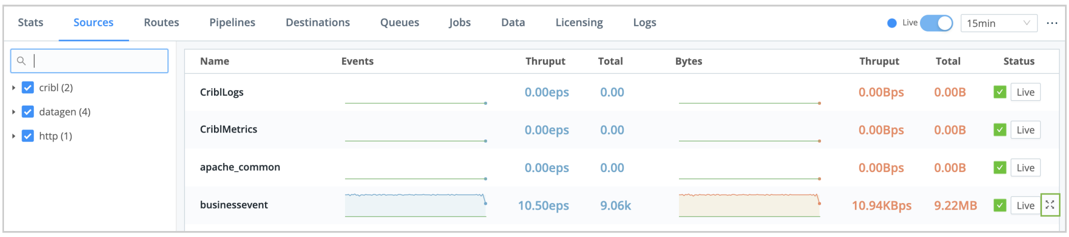
Monitor Data Throughput

From the top menu, select **Monitoring**. (On very narrow displays, you might need to select it from the ••• overflow menu.) This opens a summary dashboard, where you should see a steady flow of data in and out of LogStream. The left graph shows events in/out. The right graph shows bytes in/out.



Monitoring dashboard

Monitoring displays data from the preceding 24 hours. You can use the **Monitoring** submenu to open detailed displays of LogStream components, collection jobs and tasks, and LogStream's own internallogs. Click **Sources** on the lower submenu to switch to this view:

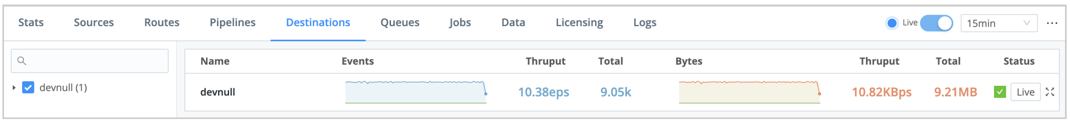


Name	Events	Thruput	Total	Bytes	Thruput	Total	Status
CriblLogs		0.00eps	0.00		0.00Bps	0.00B	✓ Live
CriblMetrics		0.00eps	0.00		0.00Bps	0.00B	✓ Live
apache_common		0.00eps	0.00		0.00Bps	0.00B	✓ Live
businessevent		10.50eps	9.06k		10.94KBps	9.22MB	✓ Live

Monitoring Sources

This is a compact display of each Source's inbound events and bytes as a sparkline. You can click each Source's Expand button (highlighted at right) to zoom up detailed graphs.

Click **Destinations** on the lower submenu. This displays a similar sparklines view, where you can confirm data flow out to the `devnull` Destination:



Name	Events	Thruput	Total	Bytes	Thruput	Total	Status
devnull		10.38eps	9.05k		10.82KBps	9.21MB	✓ Live

Monitoring Destinations

With confidence that we've got data flowing, let's send it through a LogStream Pipeline, where we can add Functions to refine the raw data.

Create a Pipeline

A Pipeline is a stack of LogStream Functions that process data. Pipelines are central to refining your data, and also provide a central LogStream workspace – so let's get one going.

1. From the top menu, select **Pipelines**.

   You now have a two-pane view, with ~~business on the left and party on the right~~ a **Pipelines** list on the left and **Sample Data** controls on the right. (We'll capture some sample data momentarily.)

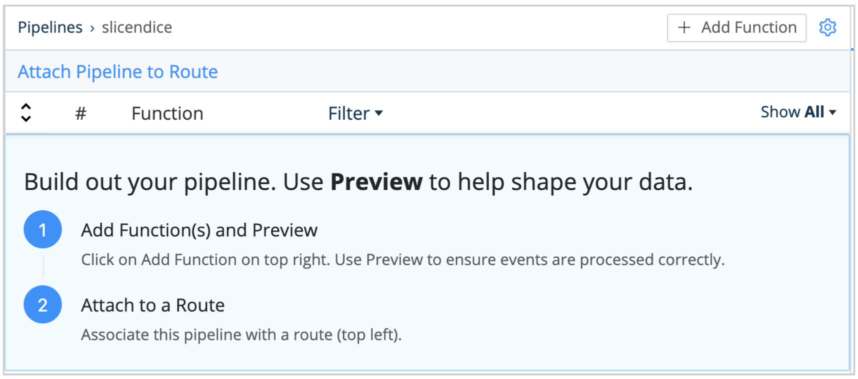
2. At the **Pipelines** pane's upper right, click **+ Pipeline**, then select **Create Pipeline**.

3. In the new Pipeline's **ID** field, enter a unique identifier. (For this tutorial, you might use `slicendice` .)

4. Optionally, enter a **Description** of this Pipeline's purpose.

5. Click **Save**.

Your empty Pipeline now prompts you to preview data, add Functions, and attach a Route. So let's capture some data to preview.



Build out your pipeline. Use **Preview** to help shape your data.

1 Add Function(s) and Preview

Click on Add Function on top right. Use Preview to ensure events are processed correctly.

2 Attach to a Route

Associate this pipeline with a route (top left).

Pipeline prompt to add Functions

Capture and Filter Sample Data

The right **Sample Data** pane provides multiple tools for grabbing data from multiple places (inbound streams, copy/paste, and uploaded files); for previewing and testing data transformations as you build them; and for saving and reloading sample files.

Since we've already got live (simulated) data flowing in from the datagen Source we built, let's grab some of that data.

Capture New Data

1. In the right pane, click **Capture New**.
2. Click **Capture**, then accept the drop-down's defaults – click **Start**.
3. When the modal finishes populating with events, click **Save as Sample File**.
4. In the **SAMPLE FILE SETTINGS** pop-up, change the generated **File Name** to a name you'll recognize, like `be_raw.log` .
5. Click **Save**. This saves to the **File Name** you entered above, and closes the modal. You're now previewing the captured events in the right pane. (Note that this pane's **Preview Simple** tab now has focus.)
6. Click **Show more** to expand one or more events.

By skimming the key-value pairs within the data's `_raw` fields, you'll notice the scenario underlying this preview data (provided by the `businessevents.log` datagen): these are business logs from a mobile-phone provider.

To set up our next step, find at least one `marketState` K=V pair. Having captured and examined this raw data, let's use this K=V pair to crack open LogStream's most basic data-transformation tool, Filtering.

Filter Data and Manage Sample Files

1. Click the right pane's **Sample Data** tab.
2. Again click **Capture New**.
3. In the **Capture Sample Data** modal, replace the **Filter Expression** field's default `true` value with this simple regex:

```
_raw.match(/marketState=TX/)
```

We're going to Texas! If you type this in, rather than pasting it, notice how LogStream provides typeahead assist to complete a well-formed JavaScript expression.

You can also click the Expand button at the **Filter Expression** field's right edge to open a modal to validate your expression. The adjacent drop-down enables you to restore previously used expressions

4. Click **Capture**, then **Start**.

   Using the **Capture** drop-down's default limits of 10 seconds and 10 events, you'll notice that with this filter applied, it takes much longer for LogStream to capture 10 matching events.

5. Click **Cancel** (and confirm your selection) to discard this filtered data and close the modal.

6. On the right pane's **Sample Data** tab, click **Simple** beside `be_raw.log` .

This restores our preview of our original, unfiltered capture. We're ready to transform this sample data in more interesting ways, by building out our Pipeline's Functions.

Refine Data with Functions

Functions are pieces of JavaScript code that LogStream invokes on each event that passes through them. By default, this means all events – each Function has a **Filter** field whose value defaults to `true` . As we just saw with data capture, you can replace this value with an expression that scopes the Function down to particular matching events.

In this Pipeline, we'll use some of LogStream's core Functions to:

- Redact (mask) sensitive data
- Extract (parse) the `_raw` field's key-value pairs as separate fields.
- Add a new field.
- Delete the original `_raw` field, now that we've extracted its contents.

- Rename a field for better legibility..

Mask: Redact Sensitive Data

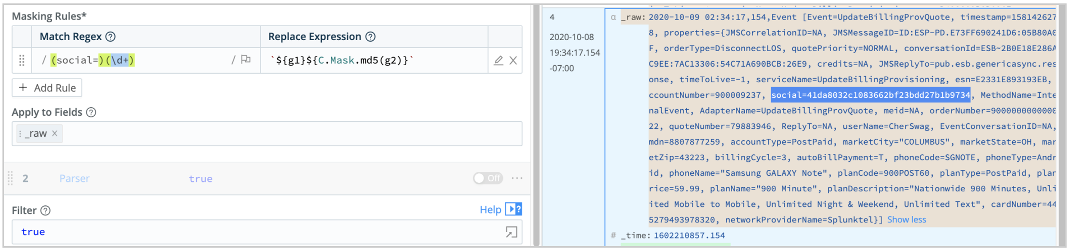
In the right **Preview** pane, notice each that event includes a **social** key, whose value is a (fictitious) raw Social Security number. Before this data goes any further through our Pipeline, let's use LogStream's `Mask` Function to swap in an md5 hash of each SSN.

1. In the left **Pipelines** pane, click `+` `+ Function` .
2. Search for `Mask` , then click it.
3. In the new Function's **Masking Rules**, click the into **Match Regex** field.
4. Enter or paste this regex, which simply looks for digits following `social=` : `(social=)(\d+)`
5. In **Replace Expression**, paste the following hash function. The backticks are literal: `` `${g1}${C.Mask.md5(g2)}` ``
6. Note that **Apply to Fields** defaults to `_raw` . This is what we want to target, so we'll accept this default.
7. Click **Save**.

You'll immediately notice some obvious changes:

- The **Preview** pane has switched from its **IN** to its **OUT** tab, to show you the outbound effect of the Pipeline you just saved.
- Each event's `_raw` field has changed color, to indicate that it's undergone some redactions.

Now locate at least one event's **Show more** link, and click to expand it. You can verify that the `social` values have now been hashed.



```
/ (social=)(\d+) /
`${g1}${C.Mask.md5(g2)}`
_raw
2  Parser  true
true
```

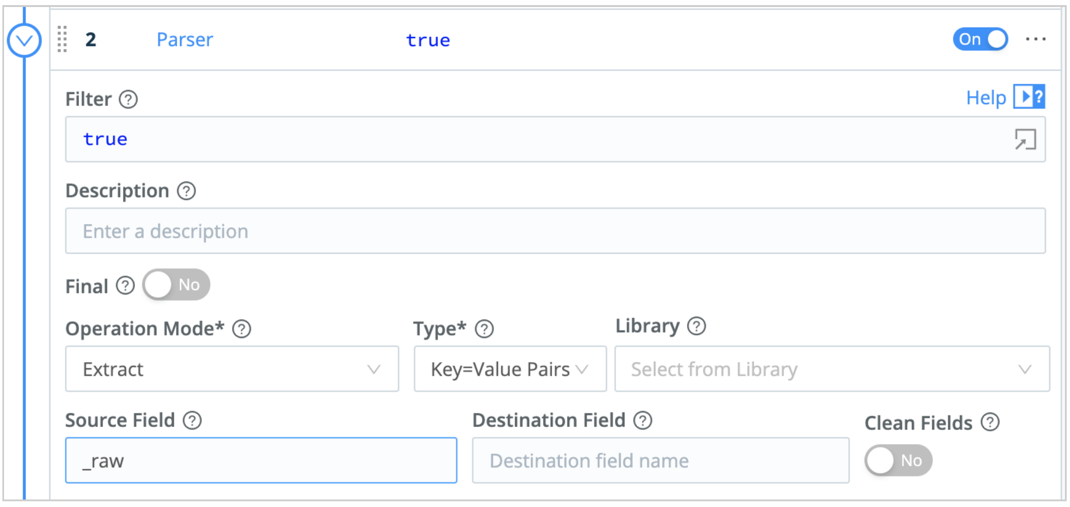
```
4
2020-10-08
19:34:17.154
-07:00
_raw: 2020-10-09 02:34:17,154,Event [Event=UpdateBillingProvQuote, timestamp=158142627
      8, properties={JMSCorrelationID=NA, JMSMessageID=ID:ESP-PD.E73FF690241D6:05B80A
      F, orderType=DisconnectLOS, quotePriority=NORMAL, conversationId=ESB~2B0E18E286A
      C9EE:7AC13306:54C71A690BCB:26E9, credits=NA, JMSReplyTo=pub.esb.genericasync.res
      onse, timeToLive=-1, serviceName=UpdateBillingProvisioning, esn=E2331E893193EB,
      ccountNumber=900009237, social=41da8032c1083662bf23bdd27b1b9734, MethodName=Inte
      nalEvent, AdapterName=UpdateBillingProvQuote, meid=NA, orderNumber=900000000000
      22, quoteNumber=79883946, ReplyTo=NA, userName=CherSwag, EventConversationID=NA,
      mdn=8807877259, accountType=PostPaid, marketCity="COLUMBUS", marketState=OH, mar
      etZip=43223, billingCycle=3, autoBillPayment=T, phoneCode=SGNOTE, phoneType=Andr
      id, phoneName="Samsung GALAXY Note", planCode=900POST60, planType=PostPaid, plar
      rice=59.99, planName="900 Minute", planDescription="Nationwide 900 Minutes, Unli
      ited Mobile to Mobile, Unlimited Night & Weekend, Unlimited Text", cardNumber=44
      5279493978320, networkProviderName=Splunktel}] Show less
# _time: 1602210857.154
```

Mask Function and hashed result

Parser: Extract Events

Having redacted sensitive data, we'll next use a Parser function to lift up all the `_raw` field's key-value pairs as fields:

1. In the left **Pipelines** pane, click `+ Function`.
2. Search for `Parser`, then click it.
3. Leave the **Operation Mode** set to its `Extract` default.
4. Set the **Type** to `Key=Value Pairs`.
5. Leave the **Source Field** set to its `_raw` default.
6. Click **Save**.



2 Parser `true` On

Filter: `true`

Description: Enter a description

Final: No

Operation Mode*: Extract

Type*: Key=Value Pairs

Library: Select from Library

Source Field: _raw

Destination Field: Destination field name

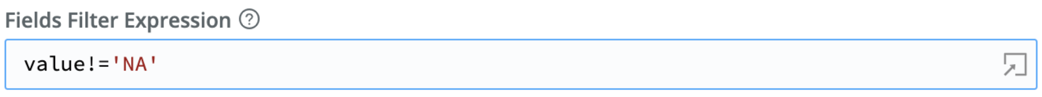
Clean Fields: No

Parser configured to extract K=V pairs from `_raw`

You should see the **Preview** pane instantly light up with a lot more fields, parsed from `_raw`. You now have rich structured data, but not all of this data is particularly interesting: Note how many fields have `NA` ("Not Applicable") values. We can enhance the **Parser** Function to ignore fields with `NA` values.

1. In the Function's **Fields Filter Expression** field (near the bottom), enter this negation expression: `value!='NA'`

   Note the single-quoted value. If you type (rather than paste) this expression, watch how typeahead matches the first quote you type.
2. Click **Save**, and watch the **Preview** pane.

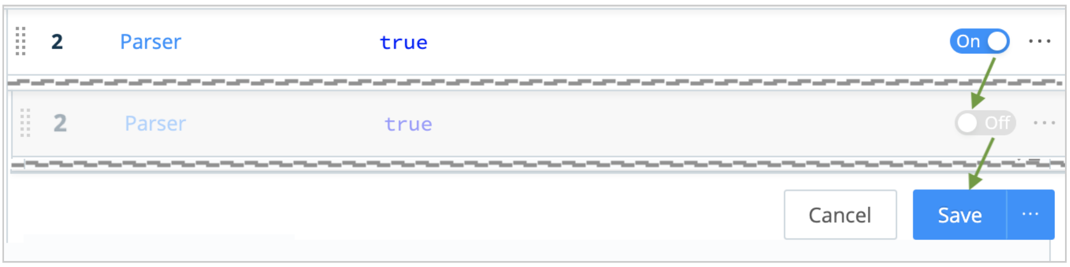

Fields Filter Expression

```
value!='NA'
```

Filtering the Parser Function to ignore fields with 'NA' values

Several fields should disappear – such as `credits`, `EventConversationID`, and `ReplyTo`. The remaining fields should display meaningful values. Congratulations! Your log data is already starting to look better-organized and less bloated.

> Missed It?
>
> If you didn't see the fields change, slide the Parser Function **Off**, click **Save** below, and watch the **Preview** pane change. Using these toggles, you can preserve structure as you test and troubleshoot each Function's effect.
>
> Note that each Function also has a **Final** toggle, defaulting to **Off**. Enabling **Final** anywhere in the Functions stack will prevent data from flowing to any Functions lower in the UI.
>
> Be sure to toggle the Function back **On**, and click **Save** again, before you proceed!



Toggling a Function off and on

Next, let's add an extra field, and conditionally infer its value from existing values. We'll also remove the `_raw` field, now that it's redundant. To add and remove fields, the **Eval** Function is our pal.

Eval: Add and Remove Fields

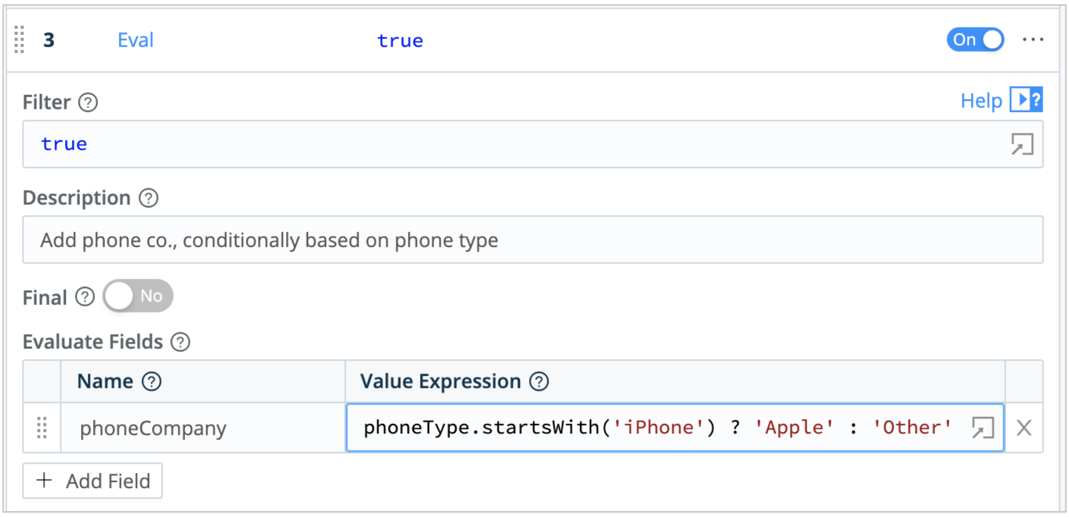
Let's assume we want to enrich our data by identifying the manufacturer of a certain popular phone handset. We can infer this from the existing `phoneType` field that we've lifted up for each event.

Add Field (Enrich)

1. In the left **Pipelines** pane, click `+ Function`.
2. Search for `Eval`, then click it.
3. Click into the new Function's **Evaluate Fields** table.

   Here you add new fields to events, defining each field as a key-value pair. If we needed more key-value pairs, we could click `+ Add Field` for more rows.

4. In **Name**, enter: `phoneCompany`.
5. In **Value Expression**, enter this JS ternary expression that tests `phoneType`'s value: `phoneType.startsWith('iPhone') ? 'Apple' : 'Other'` (Note the `?` and `:` operators, and the single-quoted values.)
6. Click **Save**. Examine some events in the **Preview** pane, and each should now contain a `phoneCompany` field that matches its `phoneType`.



3 Eval `true` On

Filter: `true`

Description: Add phone co., conditionally based on phone type

Final: No

Evaluate Fields

Name	Value Expression
phoneCompany	`phoneType.startsWith('iPhone') ? 'Apple' : 'Other'`

Adding a field to enrich data

Remove Field (Shrink Data)

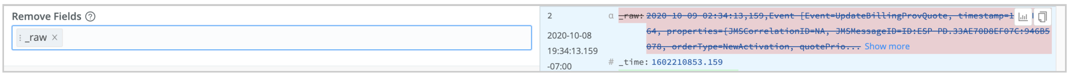
Now that we've parsed out all of the `_raw` field's data – it can go. Deleting a (large) redundant field will give us cleaner events, and reduced load on downstream resources.

1. Still in the **Eval** Function, click into **Remove Fields**

2. Type: `_raw` and press **Tab** or **Enter**.

3. Click **Save**.

The **Preview** pane's diff view should now show each event's `_raw` field stripped out.



Remove Fields

_raw

```
2
2020-10-08
19:34:13.159
-07:00
```

~~_raw: 2020-10-09 02:34:13,159,Event [Event=UpdateBillingProvQuote, timestamp=1 64, properties={JMSCorrelationID=NA, JMSMessageID=ID:ESP-PD.33AE70D8EF07C:946B5 078, orderType=NewActivation, quotePrio...~~ Show more

`_time: 1602210853.159`

Removing a field to streamline data

Our log data has now been cleansed, structured, enriched, and slimmed-down. Let's next look at how to make it more legible, by giving fields simpler names.

Rename: Refine Field Names

1. In the left **Pipelines** pane, click `+ Function`.

   This rhythm should now be familiar to you.

2. Search for `Rename`, then click it.

3. Click into the new Function's **Rename Fields** table.

   This has the same structure you saw above in Eval: Each row defines a key-value pair.

4. In **Current Name**, enter the longhaired existing field name: `conversationId`.

5. In **New Name**, enter the simplified field name: `ID`.

6. Watch any event's `conversationId` field in the **Preview** pane as you click **Save** at left. This field should change to `ID` in all events.

Drop: Remove Unneeded Events

We've already refined our data substantially. To further slim it down, a Pipeline can entirely remove events that aren't of interest for a particular downstream service.

> As the "Pipeline" name implies, your LogStream installation can have multiple Pipelines, each configured to send out a data stream

> tailored to a particular Destination. This helps you get the right data in the right places most efficiently.

Here, let's drop all events for customers who use prepaid monthly phone service (i.e., **not** postpaid):

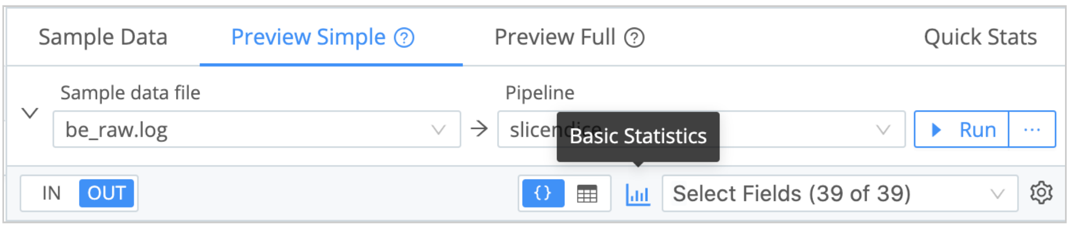
1. In the left **Pipelines** pane, click `+ Function`.
2. Search for `Drop`, then click it.
3. Click into the new Function's **Filter** field.
4. Replace the default `true` value with this JS negation expression: `accountType!='PostPaid'`
5. Click **Save**.

Now scroll through the right **Preview** pane. Depending on your data sample, you should now see multiple events struck out and faded – indicating that LogStream will drop them before forwarding the data.

A Second Look at Our Data

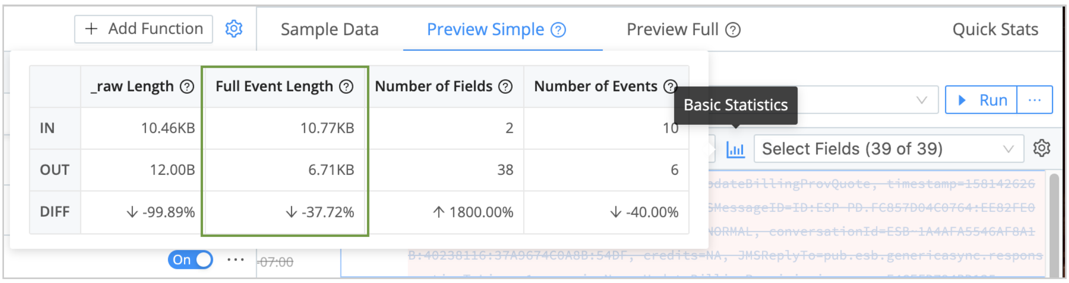
Torture the data enough, and it will confess. By what factor have our transformations refined our data's volume? Let's check.

In the right **Preview** pane, click the **Basic Statistics** button:



Displaying Basic Statistics

Even without the removal of the `_raw` field (back in Eval) and the dropped events, you should see a substantial % reduction in the **Full Event Length**.



	_raw Length	Full Event Length	Number of Fields	Number of Events
IN	10.46KB	10.77KB	2	10
OUT	12.00B	6.71KB	38	6
DIFF	↓ -99.89%	↓ -37.72%	↑ 1800.00%	↓ -40.00%

Data reduction quantified

Woo hoo! Before we wrap up our configuration: If you're curious about individual Functions' independent contribution to the data reduction shown here, you can test it now. Use the toggle **Off** > **Save** > **Basic Statistics** sequence to check various changes.

Add and Attach a Route

We've now built a complete, functional Pipeline. But so far, we've tested its effects only on the static data sample we captured earlier. To get dynamic data flowing through a Pipeline, we need to filter that data in, by defining a LogStream Route.

1. At the **Pipelines** page's top left, click **Attach Pipeline to Route**.

   This displays the **Routes** page. It's structured very similarly to the **Pipelines** page, so the rhythm here should feel familiar.

2. Click `+ Route` .

3. Enter a unique, meaningful **Route Name**, like `demo` .

4. Leave the **Filter** field set to its `true` default, allowing it to deliver all events.

   Because a Route delivers events to a Pipeline, it offers a first stage of filtering. In production, you'd typically configure each Route to filter events by appropriate `source` , `sourcetype` , `index` , `host` , `_time` , or other characteristics. The **Filter** field accepts JavaScript expressions, including AND (`&&`) and OR (`||`) operators.

5. Set the **Pipeline** drop-down to our configured `slicendice` Pipeline.

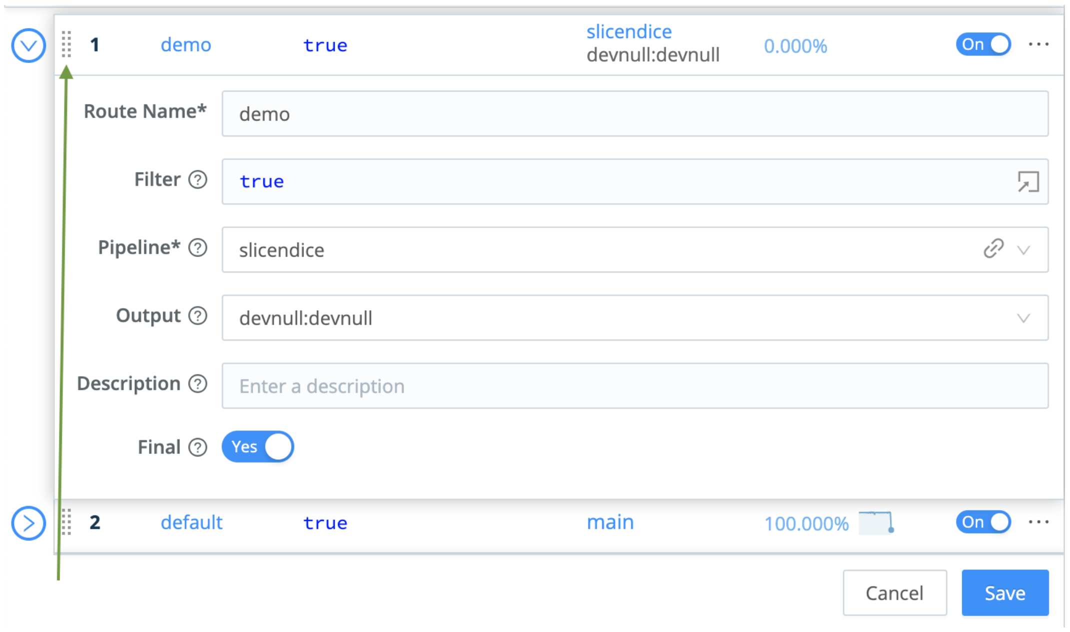
6. Set the **Output** drop-down to either `devnull` or `default` .

This doesn't matter, because we've set `default` as a pointer to `devnull`. In production, you'd set this carefully.

7. You can leave the **Description** empty, and leave **Final** set to **Yes**.

8. Grab the new Route by its left handle, and drag it above the `default` Route, so that our new Route will process events first. You should see something like the screenshot below.

9. Click **Save** to save the new Route to the Routing table.

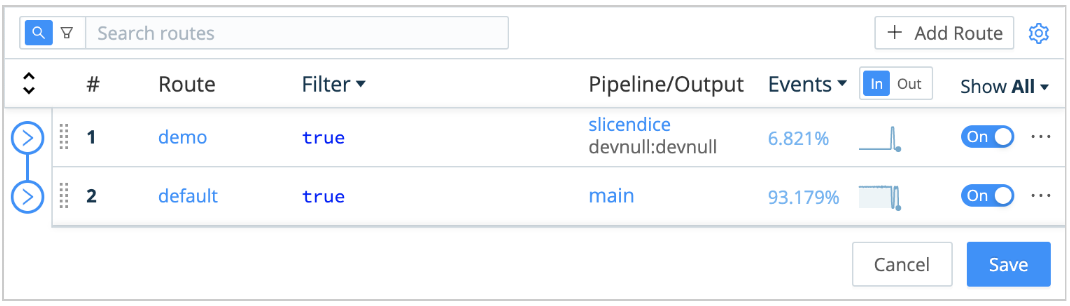


#	Route	Filter	Pipeline/Output	Events	
1	demo	`true`	slicendice devnull:devnull	0.000%	On
2	default	`true`	main	100.000%	On

Route Name*: demo
Filter: `true`
Pipeline*: slicendice
Output: devnull:devnull
Description: Enter a description
Final: Yes

Configuring and adding a Route

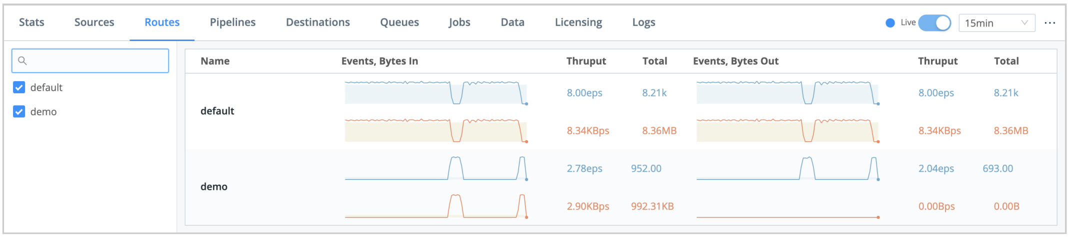
The sparklines should immediately confirm that data is flowing through your new Route:



#	Route	Filter	Pipeline/Output	Events	Show All
1	demo	`true`	slicendice devnull:devnull	6.821%	On
2	default	`true`	main	93.179%	On

Live Routes

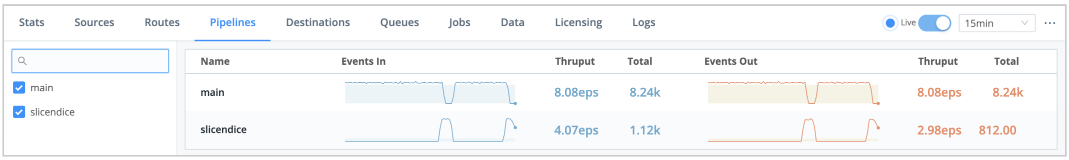
To confirm data flow through the whole system we've built, select **Monitoring > Routes** from LogStream's top menu and examine `demo`.



Name	Events, Bytes In	Thruput	Total	Events, Bytes Out	Thruput	Total
default		8.00eps	8.21k		8.00eps	8.21k
		8.34KBps	8.36MB		8.34KBps	8.36MB
demo		2.78eps	952.00		2.04eps	693.00
		2.90KBps	992.31KB		0.00Bps	0.00B

Monitoring data flow through Routes

Also select **Monitoring > Pipelines** and examine `slicendice` .



Name	Events In	Thruput	Total	Events Out	Thruput	Total
main		8.08eps	8.24k		8.08eps	8.24k
slicendice		4.07eps	1.12k		2.98eps	812.00

Monitoring data flow through Pipelines

What Have We Done?

Look at you! Give yourself a pat on the back! In this short, scenic tour – with no hit to your cloud-services charges – you've build a simple but complete LogStream system, exercising all of its basic components:

- Downloaded, installed, and run LogStream.
- Configured a Source to hook up an input.
- Configured a Destination to feed an output.
- Monitored data throughput, and checked it twice.
- Built a Pipeline.
- Configured LogStream Functions to redact, parse, enrich, trim, rename, and drop event data.
- Added and attached a Route to get data flowing through our Pipeline.

Next Steps

Interested in guided walk-throughs of more-advanced LogStream features? We suggest that next, you check out:

- LogStream Sandboxes: Work through general and specific scenarios in containers. with terminal access and free, hosted data inputs and outputs.

- Use Cases documentation: Bring your own services to build solutions to specific challenges.
- Cribl Concept: Pipelines – Video showing how to build and use Pipelines at multiple LogStream stages.
- Cribl Concept: Routing – Video about using Routes to send different data through different paths.

Cleaning Up

Oh yeah, you've still got the LogStream server running, with its `businessevent.log` datagen wtill firing events. If you'd like to shut these down for now, in reverse order:

1. Go to **Data > Sources > Datagens**.
2. Slide `businessevent` to **Off**, and click **Save**. (Refer back to the screenshot above.)
3. In your terminal's `$CRIBL_HOME/bin` directory, shut down the server with: `./cribl stop`

That's it! Enjoy using LogStream.

What's Next

> Deployment Types

DEPLOYMENT

Deployment Types

Deployment guide to get you started with Cribl

There are at least **two** key factors that will determine the type of Cribl LogStream deployment in your environment:

- Amount of Incoming Data: This is defined as the amount of data planned to be ingested per unit of time. E.g. How many MB/s or GB/day?
- Amount of Data Processing: This is defined as the amount of processing that will happen on incoming data. E.g., is most data passing through and just being routed? Or are there a lot of transformations, regex extractions, field encryptions? Is there a need for heavy re-serialization?

Single-Instance Deployment

When volume is low and/or amount of processing is light, you can get started with a single instance deployment.

Distributed Deployment

To accommodate increased load, we recommend scaling up and perhaps out with multiple instances.

Splunk App Deployment

If you have an existing Splunk Heavy Forwarder infrastructure that you want to use, you can deploy Cribl App for Splunk. See the note below before you plan.

> ⚠ Cribl App for Splunk Deprecation Notice

> Click here.

Kubernetes/Helm Deployment

You can deploy LogStream Leader Nodes (or single instances) and Worker Nodes via Cribl's Helm charts.

Docker Deployment

You can deploy LogStream instances using images from Cribl's public Docker Hub.

What's Next

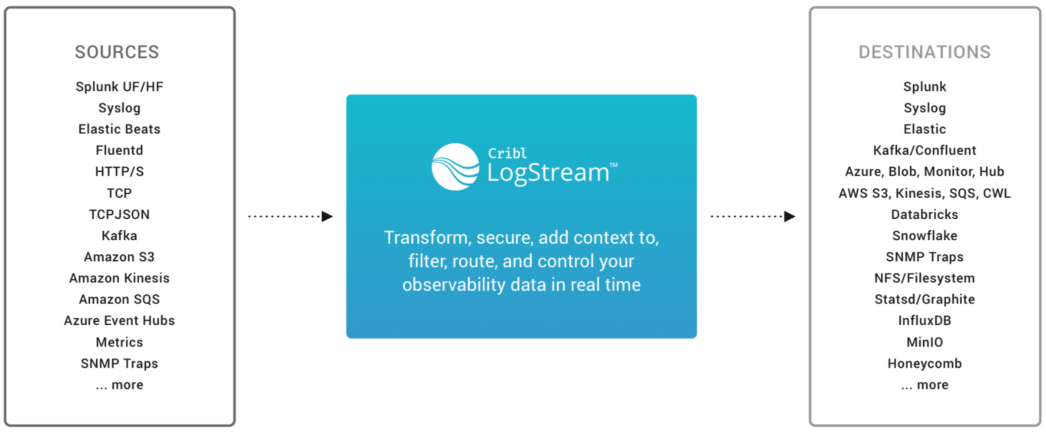
- Single-Instance Deployment
- Distributed Deployment
- Splunk App Deployment *
- Bootstrap Workers from Leader
- Kubernetes/Helm Deployment
- Docker Deployment

Single-Instance Deployment

Getting started with Cribl LogStream on a single instance

For small-volume or light processing environments – or for test or evaluation use cases – a single instance of Cribl LogStream might be sufficient to serve all inputs, event processing, and outputs. This page outlines how to implement a single-instance deployment.

Architecture



Requirements

- **OS (Intel Processors):**
  - Linux 64-bit kernel >= 3.10 and glibc >= 2.17
  - Examples: Ubuntu 16.04, Debian 9, RHEL 7, CentOS 7, SUSE Linux Enterprise Server 12+, Amazon Linux 2014.03+
- **OS (ARM64 Processors):**
  - Linux 64-bit
  - Tested so far on Ubuntu (14.04, 16.04, 18.04, and 20.04), CentOS 7.9, and Amazon Linux 2

- **System:**
  - +4 physical cores, +8GB RAM
  - 5GB free disk space (more if persistent queuing is enabled)

  > ℹ We assume that 1 physical core is equivalent to 2 virtual/hyperthreaded CPUs (vCPUs). All quantities listed above are minimum requirements. To fulfill the above requirements using cloud-based virtual machines, see Recommended AWS, Azure, and GCP Instance Types.

- **Browser Support**: Firefox 65+, Chrome 70+, Safari 12+, Microsoft Edge

Network Ports

By default, LogStream listens on the following ports:

Component	Default Port
UI	`9000`
HTTP In	`10080`
Splunk to Cribl LogStream data port	`localhost:10000` (Cribl App for Splunk)
`| criblstream` Splunk search command to Cribl LogStream	`localhost:10420` (Cribl App for Splunk)
User options	+ Other data ports as required.

Overriding Default Ports

The above ports can be overridden in the following configuration files:

- Cribl UI port (`9000`): Default definitions for `host` , `port` , and other settings are set in `$CRIBL_HOME/default/cribl/cribl.yml` , and can be overridden by defining alternatives in `$CRIBL_HOME/local/cribl/cribl.yml` .

- Data Ports: HTTP In (`10080`), TCPJSON in (`10420`) Splunk to Cribl (`10000`) : Default definitions for `host` , `port` and other settings are set in `$CRIBL_HOME/default/cribl/inputs.yml` , and can be overridden by defining alternatives in `$CRIBL_HOME/local/cribl/inputs.yml` .

Installing on Linux

- Install the package on your instance of choice. Download it here.
- Ensure that required ports are available (see Network Ports).
- Un-tar in a directory of choice, e.g., `/opt/` :
  - `tar xvzf cribl-<version>-<build>-<arch>.tgz`

Running

Go to the `$CRIBL_HOME/bin` directory, where the package was extracted (e.g.: `/opt/cribl/bin`). Here, you can use `./cribl` to:

- **Start**: `./cribl start`
- **Stop**: `./cribl stop`
- **Reload**: `./cribl reload`
- **Restart**: `./cribl restart`
- **Get status**: `./cribl status`
- **Switch a distributed deployment to single-instance mode**: `./cribl mode-single` (uses the default address:port `0.0.0.0:9000`)

> i Executing the `restart` or `stop` command cancels any currently running collection jobs. For other available commands, see CLI Reference.

Next, go to `http://<hostname>:9000` and log in with default credentials (`admin:admin`). You can now start configuring Cribl LogStream with Sources and Destinations, or start creating Routes and Pipelines.

> i In the case of an API port conflict, the process will retry binding for 10 minutes before exiting.

Enabling Start on Boot

Cribl LogStream ships with a CLI utility that can update your system's configuration to start LogStream at system boot time. The basic format to invoke this utility is:

```
[sudo] $CRIBL_HOME/bin/cribl boot-start [enable|disable] [options] [args]
```

> ℹ You will need to run this command as root, or with `sudo`. For options and arguments, see the CLI Reference.

Most, if not all, popular Linux distributions use `systemd` now to start processes at boot, while older or more obscure distributions may still use `initd`. Verify with your Linux distribution vendor if you aren't sure which method your systems use in order to know which procedure listed below to follow.

Using systemd

To **enable** Cribl LogStream to start at boot time with **systemd**, you need to run the `boot-start` command. Make sure you first create any user you want to specify to run LogStream. E.g., to run LogStream on boot as existing user `cribl`, you'd use:

```
sudo $CRIBL_HOME/bin/cribl boot-start enable -m systemd -u cribl
```

This will install a unit file (as below) and start Cribl LogStream at boot time as user `cribl`. A `-configDir` option can be used to specify where to install the unit file. If not specified, this location defaults to `/etc/systemd/system`.

If necessary, change ownership for the Cribl LogStream installation:

```
[sudo] chown -R cribl $CRIBL_HOME
```

Next, use the `enable` command to ensure that the service starts on system boot:

```
[sudo] systemctl enable cribl
```

To **disable** starting at boot time, run the following command:

```
sudo $CRIBL_HOME/bin/cribl boot-start disable
```

Note the file's default `65536` hard limit on maximum open file descriptors (known as a ulimit). The minimum recommended is 65536. Linux tracks this per user account. You can view the current soft ulimit for max open file descriptors with `$ ulimit -n` while logged in as the same user running the `cribl` binary.

Installed systemd File

```
[Unit]
Description=Systemd service file for Cribl LogStream.
After=network.target

[Service]
Type=forking
User=cribl
Restart=on-failure
RestartSec=5
LimitNOFILE=65536
PIDFile=/install/path/to/cribl/pid/cribl.pid
ExecStart=/install/path/to/cribl/bin/cribl start
ExecStop=/install/path/to/cribl/bin/cribl stop
ExecStopPost='/bin/rm -f /install/path/to/cribl/pid/cribl.pid'
ExecReload=/install/path/to/cribl/bin/cribl reload
TimeoutSec=60

[Install]
WantedBy=multi-user.target
```

> ⚠ Do NOT Run LogStream as Root!
>
> If LogStream is required to listen on ports 1–1024, it will need privileged access. You can enable this on systemd by adding this configuration key:
>
> ```
> [Service]
> AmbientCapabilities=CAP_NET_BIND_SERVICE
> ```

Using initd

To **enable** Cribl LogStream to start at boot time with **initd**, you need to run the `boot-start` command. If the user that you want to run LogStreams does not exist, create it prior to executing. E.g., running LogStream as user `cribl` on boot:

```
sudo $CRIBL_HOME/bin/cribl boot-start enable -m initd -u cribl
```

This will install an `init.d` script in `/etc/init.d/cribl.init.d`, and start Cribl LogStream at boot time as user `cribl`. A `-configDir` option can be used to specify where to install the script. If not specified, this location defaults to `/etc/init.d`.

If necessary, change ownership for the Cribl LogStream installation:

```
[sudo] chown -R cribl $CRIBL_HOME
```

To **disable** starting at boot time, run the following command:

```
sudo $CRIBL_HOME/bin/cribl boot-start disable
```

> ⚠ Do NOT Run LogStream as Root!
>
> If LogStream is required to listen on ports 1–1024, it will need privileged access. On a Linux system with POSIX capabilities, you can achieve this by adding the `CAP_NET_BIND_SERVICE` capability. For example: `# setcap cap_net_bind_service=+ep $CRIBL_HOME/bin/cribl`
>
> On some OS versions (such as CentOS), you must add an `-i` switch to the `setcap` command. For example: `# setcap -i cap_net_bind_service=+ep $CRIBL_HOME/bin/cribl`
>
> Upon starting the LogStream server, a `bind EACCES 0.0.0.0:<port>` error in the API/worker logs (depending on the service) might indicate that `setcap` did not successfully execute.

System Proxy Configuration

For details on configuring LogStream to send and receive data through proxy servers, see our System Proxy Configuration topic.

Scaling Up

A single-instance installation can be configured to scale up and utilize as many resources on the host as required. See Sizing and Scaling for details.