

# Loggregator Operator Handbook

A guide for operators to scale and manage logging for Cloud Foundry.

<b>Introduction</b>	<b>3</b>
<b>Service Level Indicators Overview</b>	<b>3</b>
Uptime	3
Log Message Reliability	3
Latency	4
<b>Capacity Planning</b>	<b>4</b>
Doppler Planning	4
Traffic Controller & Nozzle Planning	4
Scaling Dopplers Based on Ingress	5
<b>Black Box Monitoring Suggestions</b>	<b>6</b>
Log Stream Message Reliability Tests	6
Firehose Message Reliability Tests	7
Syslog Drain Message Reliability Tests	8
Recent Logs Latency Test	8
Container Metrics Latency Test	8
Other Black Box Monitors	8
<b>White Box Monitoring Suggestions</b>	<b>9</b>
Monitoring Drops	9
IAAS Metrics	10
<b>Defining and Monitoring Service Level Objectives</b>	<b>11</b>
Message Reliability	11
<b>Defining and Monitoring Service Level Objectives</b>	<b>12</b>
Message Reliability	12
<b>Capacity Planning Tests</b>	<b>13</b>
Test Assumptions	13
Log and Metrics Distribution	13
Deployment Assumptions	13
Infrastructure Assumptions	14
<b>Capacity Test Results</b>	<b>14</b>
Micro Test for RPS Rates	14
Previous Version Comparison	17
Small Test	18
<b>Tooling and Troubleshooting</b>	<b>19</b>
Health Endpoints	19
Log and Metric Emitters	19
Syslog Drain Counters	20
Capacity Planning Pipelines	20
Message Reliability Nozzle	20
<b>Contributing and Collaborating</b>	<b>20</b>

# Introduction

This document provides a guide for operators to plan and measure specific service reliability indicators related to aggregated logging across the Cloud Foundry platform. The guide provides an overview of where loss occurs by reviewing the results of a series of controlled tests performed with specific assumptions. As much as possible, these assumptions are designed to be specific, but realistic to typical deployments observed by the Loggregator team.

The guide assumes operators are familiar with BOSH and Cloud Foundry. It is also helpful to be familiar with the language and concepts defined in Google's Site Reliability Engineering book.

## Service Level Indicators Overview

Based on ongoing operation of the Loggregator system, feedback from operators and the open source community, the Loggregator team has monitoring for all of the following Service Level Indicators.

### Uptime

Ensuring the system is ready to receive messages serves as a foundational requirement of the system. Log message and/or metric delivery reliability cannot exceed the time for which the platform is up and ready to send logs. Unless explicitly noted in release notes, the Loggregator system is designed to remain highly available during a deployment upgrade or other individual service stop. A critical function of uptime for Loggregator is service discovery, which can sometimes operate in a degraded state. Using Loggregator to monitor its own uptime is problematic, so this is actually managed by the [BOSH System Metrics component](#).

### Log Message Reliability

Given the system is up and running, the reliability of message delivery is a critical tool for monitoring the health of your Loggregator deployment. High rates of message loss can indicate improperly scaled components, over-crowded Diego Cells, or noisy applications or services that are deployed on your platform. That said, understanding message reliability is a challenging proposition. We'll cover several black-box and white-box tools for monitoring loss as this is the primary SLI for the Loggregator system.

### Latency

Loggregator is designed to be a low-latency system with streaming interfaces. However, there are a couple of request-response endpoints that should be monitored for latency as they can cause other failures for the user.

## Capacity Planning

As a starting point, it is helpful to use a few heuristic formulas for estimating the scale of the Loggregator components. These formulas are good benchmarks, but you may find that your platform behaves differently as it grows.

Due to encryption and batching of messages, it can be difficult to predict how small differences may affect your platform. However, in our experience, the following best practices provide helpful tools for setting and monitoring Loggregator components.

## Doppler Planning

Dopplers are most directly affected by the overall log and metric volume of the platform. The following formula is recommended for determining your number of Dopplers to achieve a loss rate of <1%.

$$\text{Num Dopplers} = \text{Logs Per Second} / 2,000$$

*Our tests measured loss rates of <1% for up to 16,000 envelopes per second. It is difficult to understand the ratio of metrics to logs when you consider the system batches delivery. Therefore, we also recommend monitoring and scaling based on Doppler ingress traffic (see the Scaling Dopplers Based on Ingress section below).*

## Traffic Controller & Nozzle Planning

The number of Traffic Controllers can require scaling with the number of log streams and the Firehose subscriptions but usually is scaled in line with your Dopplers. Therefore, unless you have a special circumstance (e.g., a larger number of Firehose subscriptions), the following formula is recommended for planning for Traffic Controllers.

$$\text{Num Traffic Controllers} = \text{Num of Dopplers} / 4$$

Each nozzle deployed should be scaled to match the number of Traffic Controller instances.

$$\text{Num Traffic Nozzles} = \text{Num of Traffic Controllers}$$

## Syslog Adapters Planning and Scaling

User-provided syslog drains are managed by an independently scalable component called a Syslog Adapter. This component should be scaled depending on the number of your drain bindings. A drain binding is defined as a syslog destination associated with an application. An application can have multiple bindings, which should be accounted for in your scaling computations. In addition, this count is emitted as the `scheduler.drains` metric and the

number of adapters as the `scheduler.adapters` metric for configuring auto-scaling of these components.

$$\text{Num of Syslog Adapters} = \text{Num Drain Bindings} / 500$$

## Scaling Dopplers Based on Ingress

Monitoring for Doppler ingress requires you to sum two metrics and rate them per second. We recommend planning for max values over a two-week period and following the guidelines for scaling Dopplers.

$$\begin{aligned} \text{Num Dopplers} = & \text{doppler.ingress} + \\ & \text{DopplerServer.listeners.receivedEnvelopes} / 10,000 \end{aligned}$$

## Black Box Monitoring Suggestions

Because Loggregator provides an interface for app developers, writing and configuring blackbox monitoring tooling is easy using the CF CLI. This section describes conceptually using those tests to monitor aspects of your Loggregator Deployment.

### Log Stream Message Reliability Tests

The use of black box monitors is one of the best ways to measure message reliability of your Loggregator deployment. It does this by measuring message reliability through the logstream used when running `cf logs`. This reliability measure provides a contract with development teams as to how often they can expect to receive intended logs. It also serves as a correlated measurement for Firehose Reliability but doesn't require deployment or scaling of nozzles.

To execute a test of message reliability, start by creating an application that produces a known number of logs (there is one available in the tooling section). Next deploy that application to Cloud Foundry. Finally tail the logs on the application and pipe the results into a file. Do a wordcount on the file and you will have a single sampling of message reliability.

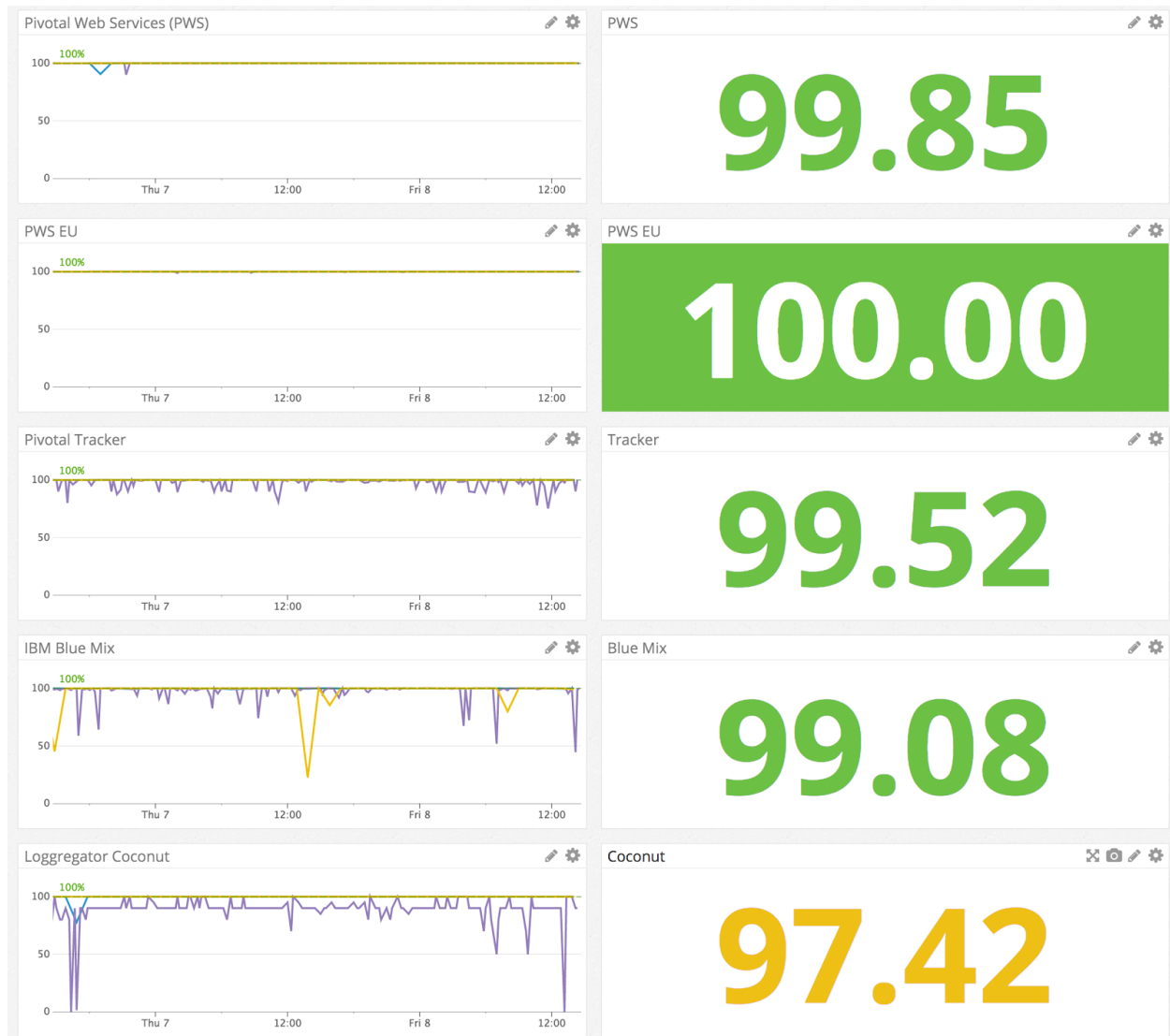
Below are a simple set of commands to do so using the `logspinner` application outlined in the tools section.

```
cd /loggregator/src/tools/logspinner
cf push logspinner
cf logs logspinner > ./tmp.txt
curl 'logspinner.coconut.cf-app.com?cycles=100&delay=10ms'
grep -wc ./tmp.txt
```

To get an accurate sampling we recommend running this test repeatedly at multiple log rates representative of your system.

***To better package this test we have included an application called [cf-logmon](#) in the tooling section that runs these tests and emits metrics through the firehose.***

To establish a Service Level Indicator we measure logs at three rates. A drip of logs at 2 logs per second, a flood of logs at 2-4M logs per second (as fast as we can send them) and a steady stream of logs at 1000/sec. We send the results of these tests directly to a monitoring system and use a weighted average of 70% drip, 20% stream and 10% flood. We have found that size of logs does not affect performance in an expected manner and we test with a consistent, but small log message for our tests.



*Dashboard of message reliability and weighted average across multiple deployments.*

## Firehose Message Reliability Tests

Measuring message loss through the firehose is more challenging than the logstream but can be done by creating a nozzle. This nozzle needs to look for, and count a known number of logs being produced similar to the log stream test. It is important that you scale the nozzle appropriately for the test. Generally we see a correlation of log stream message reliability to firehose message reliability, but firehose message reliability is consistently lower for application logs. For higher message reliability rates we recommend using syslog drains or log streams.

## Syslog Drain Message Reliability Tests

Syslog drains use a separate set of components and should be measured at the receiving end of the drain separately from the log stream. To perform this type of black box testing you'll need to create a syslog drain which receives logs on both HTTP and syslog protocol. That latter may require routing configuration for your deployment. There is a syslog server application referenced in the tooling section of this document.

The premise for the test is otherwise the same as the log stream message reliability measurements. Multiple samplings measured over time are used to determine a reliability metric. One important note: this test measures best-case reliability. The application used for the test does not need to write logs to persistent storage so it likely outperforms most real world syslog servers. Your real world message reliability rate will be dependant on the performance of the syslog server it connects to.

## Recent Logs Latency Test

The process of returning logs for the request `cf logs --recent` involves collecting logs from all the dopplers in a deployment and reassembling the results. As your deployment grows and you add more dopplers the response time for this call will increase. The latency can be measured with a simple `time` command.

```
time cf logs --recent
```

Historically the Loggregator team has monitored the upper 95th percentile of latency, although the most recent version of loggregator have added circuit breaker to return all available logs within 1500ms.

## Container Metrics Latency Test

Container metrics are collected in a similar fashion to recent logs, and they are used during the `cf push` command. This makes this indicator a critical indicator for development teams. If this call is slow developers may experience slowness when pushing their apps. This also now includes a circuit breaker timeout so monitoring latency of this specifically is not needed. It would be better to monitor latency of the full `cf push` command.

## Other Black Box Monitors

There are other black box monitoring tests that can be performed, but they are generally less interesting to operators. Some other monitors we have developed but rarely utilize are:

- Firehose message reliability nozzle (this is covered in the capacity planning test)
- Firehose latency

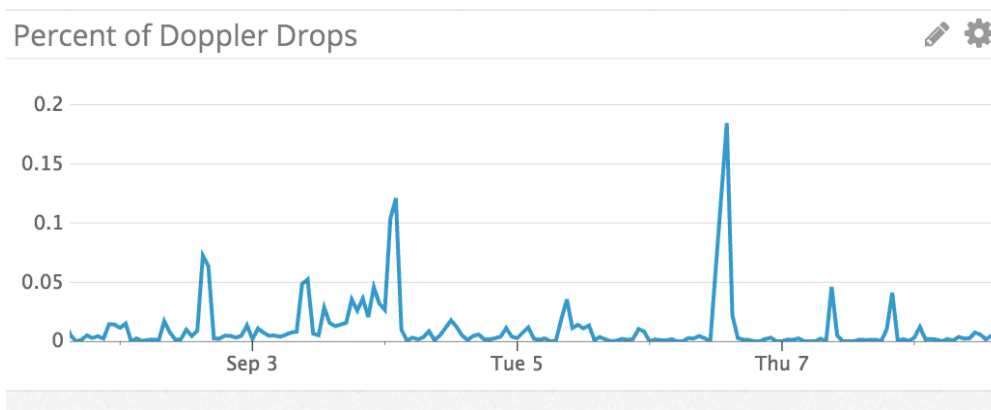


## White Box Monitoring Suggestions

In addition to providing black box monitoring tooling that works across various versions of the Loggregator system there has been considerable effort made to standardize and improve the experience of white box monitoring for operators. All whitebox monitoring suggestions are based on monitoring metrics emitted through the firehose.

### Monitoring Drops

Monitoring drops for each component is an important consideration for operators. This metric shows the number of dropped “envelopes” which could contain either a metric, or log. This helps inform where the platform is constrained and where to add more resources in the form of horizontal scaling. It’s also worth noting that drops as an individual metric is not of much value; we recommend monitoring percentage of drops based on the total ingress into the component. This should be monitored for Metron Agents, Dopplers, Reverse Log Proxies, and Adapters.



Example datadog json:

```
{
  "viz": "timeseries",
  "requests": [
    {
      "q": "( per_second(sum:datadog.nozzle.loggregator.doppler.dropped{*) /
per_second(sum:datadog.nozzle.loggregator.doppler.ingress{*) ) * 100",
      "conditional_formats": [],
      "type": "line",
      "aggregator": "avg"
    }
  ],
  "autoscale": true,
  "status": "done"
}
```

This behavior tends to be spiky in nature but this in combination with blackbox tests are helpful tools for tools for monitoring and alerting.

## IAAS Metrics

Monitoring IAAS metrics is standard practice for most operators and can be an effective predictive indicator for Loggregator as well. As a general rule most components are bound by cpu constraints and it can be helpful to compare network traffic across instances to better understand load balancing efficiency. Additionally Traffic Controller in particular can be CPU bound as your deployment and team scales in ways these guidelines don't anticipate well.

## Defining and Monitoring Service Level Objectives

Defining Service Level Objectives (SLO's) for Loggregator can be a nuanced and challenging problem for Operators to take on when first working with the Cloud Foundry. While this document describes several ways for monitoring Loggregator focusing on message reliability should be the main concern for most operators (since monitoring message reliability also serves as a valuable proxy for uptime). The Pivotal Web Services (PWS) SRE Team uses a black box monitor described above for monitoring the SLO over a 24hour, 1 week and 1 month duration as the single SLO for monitoring Loggregator.

### Message Reliability

It can be challenging to balance the expected loss rates of logs under peak load conditions with the overall needs during low volume periods. While logs are essential for debugging and monitoring the impact of loss for these functions is hard to quantify. That said a percentile based approach is a model for thinking about message reliability. Below is an example of log rate distributions for a hypothetical deployment.

Percentile	Logs Per Second Rate	Num of Dopplers for <1% Loss
95	50,000	25
<b>90</b>	<b>40,000</b>	<b>20</b>
85	32,000	16
80	25,600	13
75	20,480	10
70	16,384	8
65	13,107	7
60	10,486	5
55	8,389	4
50	6,711	3

While it can be tempting to plan for less than 1% at your 95th percentile of traffic it is twice the infrastructure cost to do so than planning for the 80th percentile of traffic. Without considering autoscaling capabilities we currently base our scaling around the 90th percentile of traffic at a 1% loss rate for the log stream.

## Defining and Monitoring Service Level Objectives

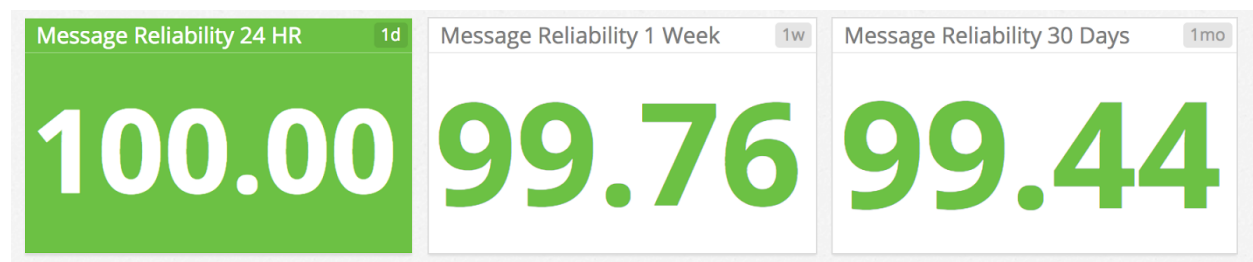
Defining Service Level Objectives (SLO's) for Loggregator can be a nuanced and challenging problem for Operators to take on when first working with the Cloud Foundry. It is recommended that you start with basic reliability monitoring and move on to more sophisticated tooling and testing as your comfort level with the product matures. In general the Loggregator system is designed to be resilient and not require significant management during stem cell upgrades, outages, or platform deployments. The main consideration for monitoring is managing cost and appropriate scaling for peak loads.

### Message Reliability

It can be challenging to balance the expected loss rates of logs under peak load conditions with the overall needs during low volume periods. While logs are essential for debugging and monitoring the impact of loss for these functions is hard to quantify. That said a percentile based approach is a model for thinking about message reliability. Below is an example of log rate distributions for a hypothetical deployment.

Percentile	Logs Per Second Rate	Num of Dopplers for <1% Loss
95	50,000	25
<b>90</b>	<b>40,000</b>	<b>20</b>
85	32,000	16
80	25,600	13
75	20,480	10

While it can be tempting to plan for less than 1% at your 95th percentile of traffic it is twice the infrastructure cost to do so than planning for the 80th percentile of traffic. Without considering autoscaling capabilities we currently base our scaling around the 90th percentile of traffic at a 1% loss rate for the log stream. We use a black box monitor described below for monitoring the SLO over a 24 hour, 1 week and 1 month duration.



## Capacity Planning Tests

To better understand the limitations of each of our components the Loggregator team maintains a set of automated tests that can be deployed to test the Loggregator system at scale and monitor message reliability of both logs and metrics.

This test is similar to our blackbox monitoring technique for log stream, but performed at scale for both logs and metrics. It works by allowing for the configuration of a baseline request per second which is then used to produce one log and one metric. After configuring the test to run at a declared rate, a pipeline is used to deploy Cloud Foundry with a set of applications that produce logs and a bosh deployment of metric emitters. The assumptions of the deployment are described below. These applications produce logs which are counted in both log streams using the CLI and through the use of a firehose nozzle. These counts and the metrics emitted by Loggregator during the test are captured in a monitoring tool.

### Test Assumptions

We have made several assumptions for our test. Where possible we have tried to make these assumptions as simple as possible to allow operators to consider what unique characteristics of their own deployment may influence how the assumptions impact the results. The assumptions are held fixed throughout our tests to allow us to isolate the performance and scaling needs for each component individually.

### Log and Metrics Distribution

For simplicity the test assumes an equal amount of logs and metrics.

### Deployment Assumptions

The test runs using [cf-deployment](#) and some basic best practices for scaling apps and diego cells. These applications all produce logs at the same rate. Additionally there is a bosh deployment of an application with a colocated metron. Other components on the platform are generally kept quiet during the test so that the majority of the traffic on loggregator can be accounted for from our controlled deployments. While this may produce an unrealistically consistent flow of logs and metrics these assumptions are designed to properly scale metron and allow us to apply load to doppler linearly for the tests.

All tests are executed with 2 firehose subscriptions and a single app stream per application. This is generally in line with most deployments we have seen. These assumptions can have impact on the scaling recommendations for Traffic Controllers but those effects are not directly measured in these tests.

All tests are executed for at least an hour, and use the new go-loggregator client inbound into metron for logs and UDP inbound into metron for metrics.

Tests were run multiple times against a scale of Requests Per Second (RPS) and Log Sizes and data collected for each test is referenced in the following [spreadsheet](#).

## Infrastructure Assumptions

All of the tests were executed on Google Cloud Platform (GCP) with the following size assumptions.

Component Name	Bosh VM Size	GCP VM Size
Doppler	m3.medium	n1-standard-1 (1 vCPU, 3.75 GB memory)
Traffic Controllers	c3.large	n1-highcpu-2 (2 vCPUs, 1.8 GB memory)
Reliability Nozzle	default	n1-standard-1 (1 vCPU, 3.75 GB memory)
Datadog Nozzle	default	n1-standard-1 (1 vCPU, 3.75 GB memory)

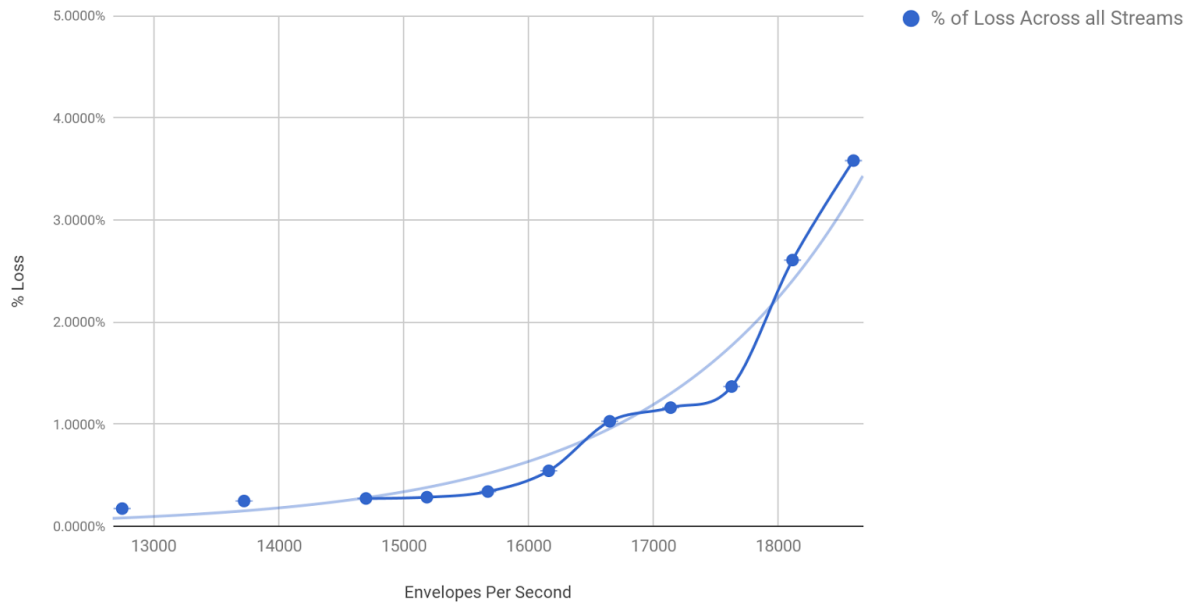
## Capacity Test Results

We executed test and three reference scales, using the rate of loss in doppler as guidance for scaling and executing the test across a range of request per second rates that account for <1% loss through 10% loss.

## Micro Test for RPS Rates

The micro scale test is designed to understand what size deployment will overwhelm a single Doppler. It's important to note however we always recommend deploying at least two Dopplers but this gives us a reference for scaling up to a small deployment with two Dopplers.

% Loss Through Firehose and App Streams For 1000 Byte Log Size

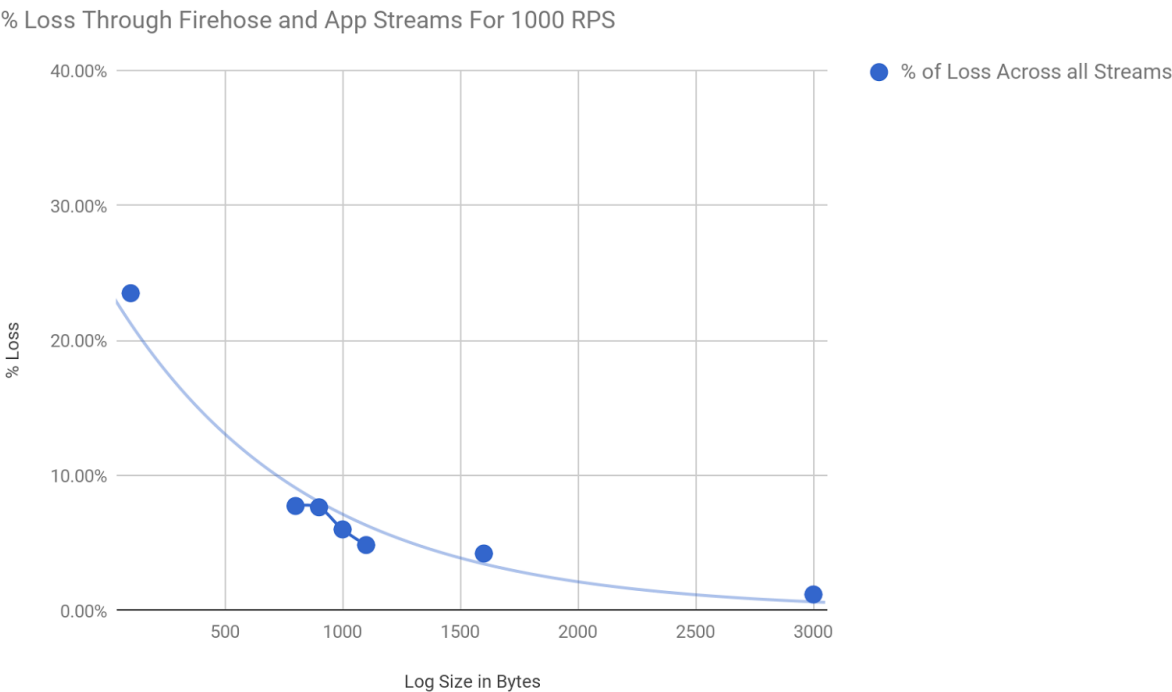


Log loss rates for a single doppler hold below 1% up to approximately 16,000 logs per second and increase exponentially for all streams after that threshold is exceeded.

Logs and Metrics Per Second	Envelopes Per Second	% of Log Loss in Firehose	% Loss in Log Stream	% of Metric Loss in Firehose	% of Loss Across all Streams	Number of Dopplers
7349	14,698	0.2566%	0.2750%	0.2847%	0.2721%	1
7593	15,186	0.2689%	0.2873%	0.2970%	0.2844%	1
7837	15,674	0.3200%	0.3200%	0.3800%	0.3400%	1
8081	16,162	0.5800%	0.2800%	0.7690%	0.5430%	1
8325	16,650	1.0700%	0.2130%	1.8000%	1.0277%	1
8569	17,138	1.0700%	0.7700%	1.6450%	1.1617%	1
8813	17,626	1.1400%	1.3500%	1.6150%	1.3683%	1
9057	18,114	2.0000%	2.8900%	2.9300%	2.6067%	1
9301	18,602	3.0400%	3.1830%	4.5200%	3.5810%	1

## Micro Test for Log Sizes

The request per second tests were executed at a consistent log size of 1000 bytes. Another micro test was executed with a single doppler across a variety of log sizes at 1000 requests per second for all tests.



The results of the log size test are surprising and somewhat counter-intuitive. That is smaller logs are more likely to produce drops in Doppler, than larger logs. This trend follows an exponential pattern with 1000 byte log sizes producing surprisingly high loss rates at 1000 logs per second.

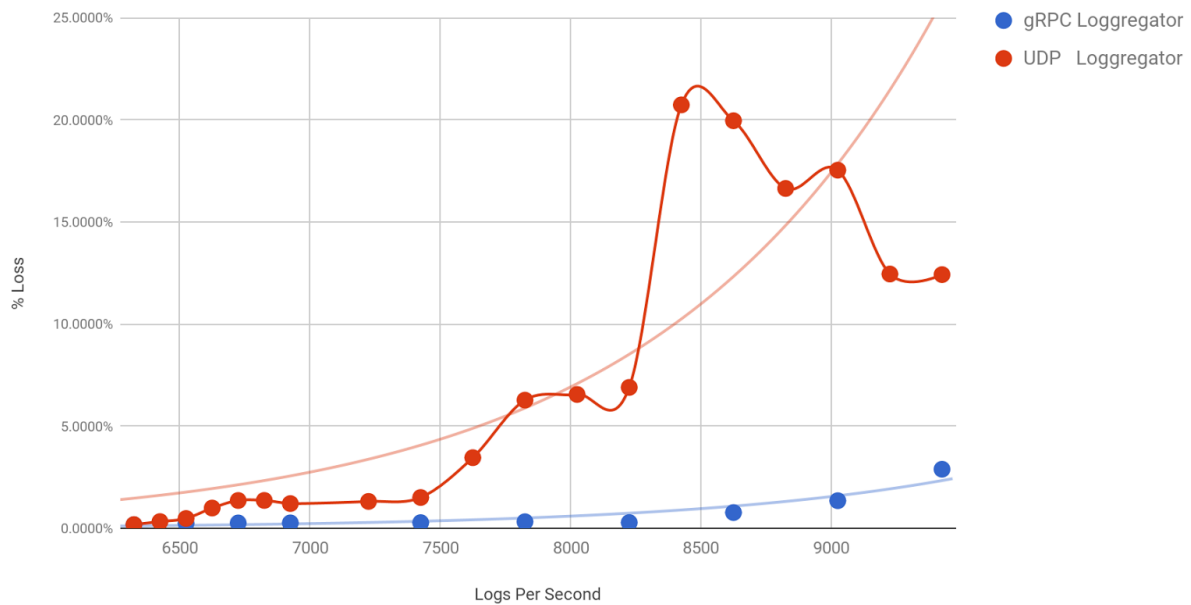
Log Size in Bytes	% of Log Loss in Firehose	% Loss in Log Stream	% of Metric Loss in Firehose	% of Loss Across all Streams	Number of Dopplers
100	37.41%	32.85%	0.25%	23.50%	1
800	12.48%	10.62%	0.10%	7.73%	1
900	12.52%	10.28%	0.10%	7.63%	1
1000	9.54%	8.35%	0.09%	5.99%	1
1100	7.77%	6.63%	0.11%	4.84%	1
1600	6.46%	5.12%	1.04%	4.21%	1
3000	2.60%	0.64%	0.28%	1.17%	1



## Previous Version Comparison

In order to best understand the impact that the switch to gRPC has had on the recent Loggregator versions these same tests were run against an older version of Loggregator that still relied on the UDP protocol.

% Loss Through App Streams For Loggregator 70 vs Loggregator 94

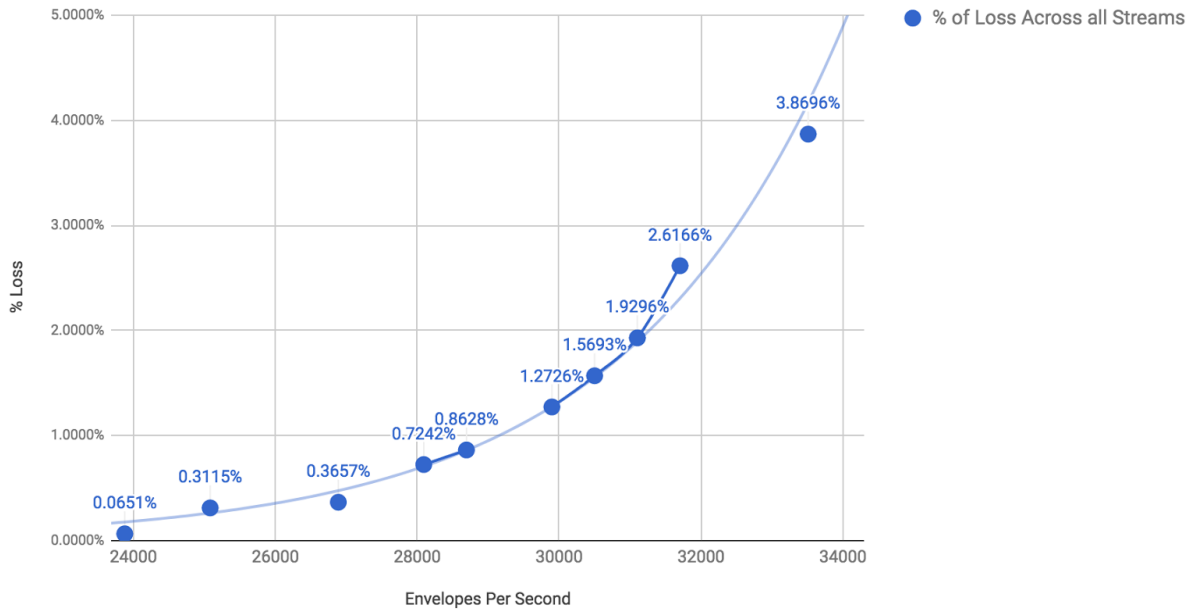


Critically these test showed a dramatic improved in reliability with gRPC and a drastically more unpredictable increase in loss for the UDP.

## Small Test

To better understand how Doppler throughput scales we repeated the test with 2 dopplers. In this test we saw similar loss rates patterns starting at about 30,000 envelopes per second, or 15,000 envelopes per second per doppler.

% Loss Through Firehose and App Streams For 1000 Byte Log Size



## Tooling and Troubleshooting

The following are a set of tools and applications that the Loggregator team has found useful. These tools are included in our repo under `/tools`.

### Health Endpoints

All loggregator components have a health endpoint available. These health endpoints can be used for debugging purposes using bosh. A summary of information that each provides is listed below:

- **Metron** - `localhost:14824/health`
  - Number of connections to dopplers
  - Number of v1 gRPC streams to dopplers
  - Number of v2 gRPC streams to dopplers
- **Doppler** - `localhost:14825/health`
  - Number of container metrics caches
  - Number of open ingress streams
  - Number of recent log caches
  - Number of open subscriptions
- **Traffic Controller** - `localhost:14825/health`
  - Number of open app streams
  - Number of open firehose streams
- **Reverse Log Proxy** - `localhost:33333/health`
  - Number of subscriptions
- **Syslog Scheduler** - `localhost:8080/health`
  - Number of Adapters
  - Number of Blacklisted or Invalid URLs
  - Number of Drains
- **Syslog Adapter** - `localhost:8080/health`
  - Drain Count

### Log and Metric Emitters

It can be helpful to emit logs and metrics at known rates. We have a variety of tools for performing both. The two most relevant are:

- [loggregator/src/tools/logspinner](#) – logspinner is intended to be pushed via `cf push` and allows you to specify cycles, delay, and text as query parameters. This is

helpful for creating tests that require deploying an app that can emit a known number of metrics in a given timeframe.

## Syslog Drain Counters

The following applications can be used together to implement a blackbox monitor for syslog drains.

- [/scalablesyslog/tools/smokes/https\\_drain](#) - This application can be deployed using cf push and details about the counter location. It's route can be created and bound as a syslog drain with the results being recorded in counter and outputted to the logs.
- [/scalablesyslog/tools/smokes/syslog\\_drain](#) - This application can be deployed using cf push but requires TCP routing is enabled to be accessible as a drain.
- [/scalablesyslog/tools/smokes/counter](#) - This application records results for both the syslog\_drain and http\_drain applications.

## Capacity Planning Pipelines

The capacity planning results outlined in this document were generated using an automated process. If you are using concourse on Google Cloud Platform these tests could easily be used to test the capacity of your deployment. These tests allow you to specify a start RPS, an ending RPS and a number of steps.

## Message Reliability Nozzle

The reliability nozzle is a scalable nozzle for counting log throughput through the firehose. This nozzle uses tags for each scaled index of the nozzle and reports the results to a monitoring tool.

## Contributing and Collaborating

Loggregator is an open source project as part of the Cloud Foundry Foundation. We welcome your feedback in [slack](#), or [github](#).