



SAS[®] Viya[®] Platform: Metric Monitoring

2026.03 - 2026.09

This document might apply to additional versions of the software. Open this document in [SAS Help Center](#) and click on the version in the banner to see all available versions.

Monitoring: Overview	1
SAS Viya Platform and Metrics	1
Ways to Monitor Metrics	2
Monitoring SAS Viya RabbitMQ Cluster	2
Monitoring NFS Shared Storage	6
Configure SAS Environment Manager to Access Metric Monitoring	7

Monitoring: Overview

SAS Viya Platform and Metrics

SAS Viya platform exposes metric data from applications, servers, and services through HTTP and HTTPS metric endpoints that are formatted as Prometheus metrics. The monitoring infrastructure collects data from these endpoints and you can access these metrics for observation, or you can include SAS Viya metrics into your existing monitoring solution for processing and display.

These metrics are used to monitor not only the resources that are specific to a SAS Viya platform environment, but also the performance of the Kubernetes cluster. Kubernetes pods that include the

SAS Viya components include annotations that identify the port and the path that are needed to collect these metrics.

Monitoring activities are organized into these broad categories:

Kubernetes cluster monitoring

collects metrics about the nodes, services, pods, containers, and other Kubernetes resources. The metrics are gathered from Kubernetes components (such as kubelet and kube-proxy), the monitoring components (Prometheus operator, Alertmanager, and Grafana), and the logging components (Fluent Bit and OpenSearch, if deployed).

SAS Viya platform monitoring

collects metrics about SAS Viya platform servers and services, SAS Cloud Analytic Services (CAS), Crunchy Data (PostgreSQL), and RabbitMQ.

Ways to Monitor Metrics

You can use the following tools to view and manage metrics:

- SAS Viya Monitoring for Kubernetes logging and monitoring tools
- Microsoft Azure Monitor

For more information, see [“Choosing a Monitoring Strategy for an Azure Environment” in SAS Viya Platform: Deploying SAS Viya Monitoring for Kubernetes](#).

- your preferred logging and monitoring technology

Note: The `sas-viya healthcheck system_health` CLI plug-in is no longer available. If you previously downloaded this plug-in, do not use it.

Monitoring SAS Viya RabbitMQ Cluster

Introduction

SAS Viya uses RabbitMQ as its message broker. The SAS Viya RabbitMQ cluster provides a Prometheus-style endpoint (`:15692/metrics`) that allows you to capture system behavior through health checks and usage metrics of the RabbitMQ cluster. Prometheus queries the metric endpoint and returns aggregated metrics, by default. Cluster-wide metrics are then computed from this node-specific data.

The following Grafana dashboards are available from Grafana to visualize SAS Viya RabbitMQ cluster metrics:

- RabbitMQ-Erlang, which collects and visualizes metrics that are related to the Erlang VM memory.
- RabbitMQ-Overview, which displays performance and usage metrics for RabbitMQ.

For more information on monitoring RabbitMQ with Prometheus, see [Monitoring with Prometheus and Grafana](#) and information on the prebuilt set of Grafana dashboards, see [Grafana Support](#) in RabbitMQ documentation.

Note: The SAS Viya Monitoring for Kubernetes solution includes Prometheus and Grafana dashboards for monitoring the RabbitMQ cluster. For information, see [“Metric Monitoring” in SAS Viya Platform: Deploying SAS Viya Monitoring for Kubernetes](#).

RabbitMQ Metrics to Monitor

The following tables include RabbitMQ metrics that can help with monitoring and troubleshooting the performance of your SAS Viya RabbitMQ Cluster.

Table 1 *Node-wide Totals Aggregated across Queues*

Queue Depth	Description
<code>rabbitmq_queue_messages</code>	This metric represents the total number of messages waiting in RabbitMQ queues on a node, including both messages that are ready to be delivered and messages that have been delivered but not yet acknowledged.
<code>rabbitmq_queue_messages > N</code>	<i>N</i> is your organization's threshold. It usually indicates that messages are accumulating faster than they are being processed. The expectation is that this is near to 0.
<code>rabbitmq_consumers < C</code>	<i>C</i> is specific to your deployment.
<code>rabbitmq_queue_messages_ready</code>	Total messages ready to be delivered to consumers across all queues
<code>rabbitmq_queue_messages_unacked</code>	Total unacknowledged messages across all queues

RabbitMQ queues should normally stay mostly empty because messages are consumed quickly. When the total queue depth (ready + unacknowledged messages) grows beyond a deployment-specific threshold, it indicates a backlog and may signal slow consumers, failed consumers, overloaded systems, or downstream processing problems.

RabbitMQ messages *ready* versus RabbitMQ messages *unacknowledged* tells you where a problem might be.

- High *ready* messages might mean that consumers are not keeping up with incoming message volume. Most messages are waiting in queues rather than being actively processed.
- High *unacknowledged* messages might mean that consumers have accepted many messages but are processing them slowly or are stuck waiting on some dependency.

Table 2 Cluster-wide Counters

Message Rates	Description
For the following cluster-wide counter metrics, calculate the rate of change over time using Prometheus <code>rate()</code> over a time window equal to 4 scrape intervals. This converts the counters into useful metrics such as messages received per second, acknowledged per second, delivered per second, and redelivered per second, making it easier to identify backlogs, consumer failures, or crash loops.	
<code>rabbitmq_global_messages_received_total</code>	Messages accepted from publishers
<code>rabbitmq_global_messages_acknowledged_total</code>	Messages acknowledged by consumers
<code>rabbitmq_global_messages_delivered_consume_manual_ack_total</code>	Messages delivered to consumers
<code>rabbitmq_global_messages_redelivered_total</code> example: <code>rate(rabbitmq_global_messages_redelivered_total[60s]) > 0</code>	Redeliveries If the number of message redeliveries continues increasing at a steady or high rate over time, it often indicates that one or more consumers are repeatedly failing after receiving messages, causing RabbitMQ to redeliver those messages again and again.

Table 3 RabbitMQ Node Resources

Node Resource	Description
When one or more monitored resource thresholds in a cluster exceed their configured limits, the system prevents publishers from sending additional data, messages, or events across the entire cluster until conditions return to acceptable levels. Set alerts when resources are low.	
<code>rabbitmq_process_resident_memory_bytes</code> <code>rabbitmq_process_resident_memory_bytes > .08</code>	Node memory that is in use
<code>rabbitmq_resident_memory_limit_bytes</code> <code>rabbitmq_resident_memory_limit_bytes > 0.8</code>	Memory high watermark in bytes

Node Resource	Description
<code>rabbitmq_disk_space_available_bytes</code> <code>rabbitmq_disk_space_available_bytes < 2.0</code>	Free disk on data partition in bytes
<code>rabbitmq_disk_space_available_limit_bytes</code> <code>rabbitmq_disk_space_available_limit_bytes < 2.0</code>	Free disk space low watermark in bytes

Table 4 File Descriptors

File Descriptors
When RabbitMQ reaches its file descriptor limits, new network connections are not accepted. (RabbitMQ uses file descriptors for virtually every client connection and many internal operations.)
<code>rabbitmq_process_open_fds</code> <code>rabbitmq_process_open_fds > 0.8</code>
<code>rabbitmq_process_max_fds</code> <code>rabbitmq_process_max_fds > 0.8</code>

Table 5 Node Overload Signal

Erlang Scheduler
RabbitMQ is built on Erlang, which uses lightweight processes rather than operating system threads. RabbitMQ creates many Erlang processes to handle the following: ■ Connections ■ Channels ■ Queues ■ Message routing ■ Replication ■ Management operations Erlang scheduler threads run these processes on available CPU cores. The following metric returns the number of runnable processes waiting for a scheduler thread:
<code>rabbitmq_erlang_scheduler_run_queue</code>
If the value remains greater than about one waiting process per CPU core, RabbitMQ is likely overloaded and cannot keep up with incoming work, which may lead to increased message latency, growing queues, and reduced throughput. Use this threshold or alert condition to indicate RabbitMQ may be severely CPU-constrained: <code>rabbitmq_erlang_scheduler_run_queue > (num_cpus * 1.5)</code>

Table 6 *Total Connections or Channels That are Open*

Connection / Channel Churn	Description
<code>rabbitmq_connections_opened_total</code>	Cumulative connections opened
If the number of new RabbitMQ connections being opened increases rapidly over time, clients might be repeatedly disconnecting and reconnecting instead of maintaining stable connections. Use a <code>rate()</code> calculation to determine how quickly new connections are being created. If the <code>rate()</code> is high, a reconnect loop occurs.	
<code>rabbitmq_channels_opened_total</code>	Cumulative channels opened; A channel is a lightweight virtual connection that exists inside a RabbitMQ connection.
<code>rate(rabbitmq_channels_opened_total)</code>	
If RabbitMQ is continuously opening channels at a high rate, applications may be creating channels but not properly closing or reusing them, resulting in a channel leak.	

Monitoring NFS Shared Storage

Considerations

When you use NFS as your shared storage provider for Kubernetes Persistent Volumes (PVs), capacity reporting is typically based on the underlying NFS export rather than on individual Persistent Volume Claims (PVCs). As a result, all PVCs that consume storage from the same NFS export can display the same available capacity, even though each application might be using a different amount of space.

Because of this behavior, the following is true:

- Kubernetes does not provide reliable storage consumption metrics for individual PVCs or applications that share an NFS-backed volume.
- PVC utilization percentages and capacity-based alerts can be misleading because they reflect the available space of the entire NFS export rather than actual per-application usage.
- A single application experiencing rapid growth can consume shared storage and affect other workloads using the same NFS export without triggering accurate PVC-level warnings.

Recommended Monitoring Practices

To avoid unexpected storage shortages when using shared NFS storage, follow these monitoring practices:

- Monitor storage utilization directly on the NFS server or storage platform, rather than relying solely on Kubernetes PVC metrics. Storage vendor monitoring tools or enterprise observability platforms can provide more accurate visibility into actual disk consumption.
- Establish alerts based on NFS export or file system capacity, ensuring that thresholds reflect the total shared storage usage across all consumers.
- Regularly review storage growth trends for SAS Viya workloads and shared application paths to identify abnormal consumption patterns before capacity limits are reached. Shared file storage is commonly used by multiple SAS Viya services and workloads, making proactive monitoring especially important.
- Adjust warning and critical thresholds conservatively in shared-storage deployments to provide sufficient time for remediation before the NFS file system becomes full.

Note: The Files service stores file content in Persistent Volume (PV) storage. Notifications are generated in certain situations that can help with tuning PV storage. For more information, see [“Manage Persistent Volume \(PV\) File Content Storage” in SAS Viya Platform: Files Service](#).

Configure SAS Environment Manager to Access Metric Monitoring

You can use SAS Environment Manager to access your site's metric-monitoring application. The link to the application must first be configured in SAS Environment Manager. See [“Configure This Page” in SAS Environment Manager: User's Guide](#).