Buf Schema Registry on-premises documentation

Dependencies

The BSR uses PostgreSQL (version 14) for storing application data, Redis (version 5) for caching, S3-compatible storage for persisting modules and plugins, as well as an Identity Provider (IdP) supporting SAML for authentication.

Interacting with the BSR via the Buf CLI requires a minimum Buf CLI version of v1.23.0.

Installation

The BSR is designed to run on Kubernetes, and is distributed as a Helm Chart and accompanying Docker images through an OCI registry. The Helm Chart and Docker images are versioned, and are expected to be used together—that is, the default values in the Chart use Docker images with the same version as the Chart itself.

1. Authenticate `helm`

To get started, authenticate `helm` with the Buf OCI registry using the keyfile that was sent alongside this documentation.

```
cat keyfile | helm registry login -u _json_key_base64 --password-stdin \
https://us-docker.pkg.dev/buf-images-1/bsr
```

2. Create a namespace

Create a Kubernetes namespace in the k8s cluster for the `bsr` Helm Chart to use:

```
kubectl create namespace bsr
```

3. Create a pull secret

Create a pull secret using the provided keyfile, this will be used by the cluster to pull images from the Buf OCI registry:

```
kubectl create secret --namespace bsr \
docker-registry bufpullsecret \
--docker-server=us-docker.pkg.dev/buf-images-1/bsr \
--docker-username=_json_key_base64 \
--docker-password=$(cat keyfile)
```

4. Configure the BSR's Helm values

The BSR is configured using Helm values through the `bsr` Helm Chart.

1. Create a file named `bsr.yaml` to store the Helm values.
   - This file can be in any location, but we recommend creating it in the same directory where the helm commands are run. It is required by the `helm install` step below.
2. Set the desired `host` .
3. Configure the chart to use the image pull secret (created above):

   ```
   host: example.com # Hostname that the BSR will be served from
   imagePullSecrets:
     - name: bufpullsecret # The image pull secret that was created above
   ```

4. Put the values from the steps below in the `bsr.yaml` file. You may skip to "Install the Helm Chart" for a full example Helm chart.

Configure object storage

The BSR requires S3-compatible object storage.

1. To configure the storage set the following Helm values, filling in your S3 variables:

   ```
   storage:
     use: s3
   ```

```
  s3:
    bucketName: "my-bucket-name"
    endpoint: "s3.us-east-1.amazonaws.com"
    region: "us-east-1"
    # forcePathStyle: false # Optional, use path-style bucket URLs (http://s3.amazonaws.com/BUCKET/KEY)
    # insecure: false # Optional, disable TLS
```

The bufd client will attempt to acquire credentials from the environment, such as instance profiles. If necessary, you may instead use an `accessKeyId` and secret pair:

1. Add the `accessKeyId` to the configuration:

```
storage:
  use: s3
  s3:
    accessKeyId: "AKIAIOSFODNN7EXAMPLE"
    bucketName: "my-bucket-name"
    endpoint: "s3.us-east-1.amazonaws.com"
    region: "us-east-1"
    # forcePathStyle: false # Optional, use path-style bucket URLs (http://s3.amazonaws.com/BUCKET/KEY)
    # insecure: false # Optional, disable TLS
```

2. Then create a k8s secret containing the s3 access secret key:

```
kubectl create secret --namespace bsr generic bufd-storage --from-literal=secret_access_key=<s3 secret access key>
```

Create a Postgres database

The BSR requires a PostgreSQL database. We support PostgreSQL **version 14.** The BSR `postgres` user requires full access to the database, and additionally must be able to create the `pgcrypto` and `pg_trgm` extensions.

1. To configure Postgres, set the following helm values:

```
postgres:
  host: "postgres.example.com"
  port: 5432
  database: postgres
  user: postgres
```

2. Then create a k8s secret containing the postgres user password:

```
kubectl create secret --namespace bsr generic bufd-postgres --from-literal=password=<postgres password>
```

Configure Redis

The BSR requires a Redis instance. We support Redis **version 5.**

To configure Redis, create a k8s secret containing the address:

```
kubectl create secret --namespace bsr generic bufd-redis \
    --from-literal=address=redis.example.com:6379 # Host
```

Optionally, authentication and TLS for Redis are also supported. These can be set with the following Helm values:

```
redis:
  # Set to true to enable auth for redis.
  # The auth token will be read from the "auth" field in the "bufd-redis" secret
  auth: true
  tls:
    # Whether to use TLS for connecting to Redis
    # Set to "false" to disable TLS
    # Set to "local" to use certs from the "ca" field in the "bufd-redis" secret
    # Set to "system" to use the system trust store
      use: "false"
```

- If authentication is enabled, the redis auth string should be added to the `bufd-redis` secret in the `auth` field.
- If TLS is enabled and `use` is set to `local` , the CA certificate(s) to trust should be added to the `bufd-redis` secret in the `ca` field.

Example of a secret containing both an authentication token and a CA certificate:

```
kubectl create secret --namespace bsr generic bufd-redis \
    --from-literal=address=redis.example.com:6379 \ # Host
    --from-literal=auth=<redis auth string> \ # Auth string
    --from-file=ca=<redis ca.crt> \ # Redis CA certificate
```

Configure SAML authentication

The BSR supports authentication using an external identity provider (IdP), through Security Assertion Markup Language (SAML).

In the SAML IdP, create a new application to represent the BSR. It should return a single sign-on URL and IdP metadata. Either a public URL or raw XML can be specified for the SAML config. If SAML is being configured in Okta, please follow our guide.

1. To configure SAML authentication in the BSR, set the following Helm values:

   ```
   auth:
     method: saml
     saml:
       # Endpoint where the XML metadata is available
       idpMetadataURL: "https://example-provider.com/app/12345/sso/saml/metadata"
       # If the authentication provider does not have a metadata url,
       # the raw XML metadata can be configured using the idpRawMetadata,
       # value instead.
       idpRawMetadata: ""
     # Optional
     # A list of emails which will be granted server admin permissions on login
     # Note that this list is case-sensitive
     autoProvisionedAdminEmails:
       - "user@example.com"
   ```

2. Additionally, a Kubernetes TLS secret named `bsr-saml-cert` containing a certificate pair is required in order for SAML to function. The certificate pair may be self-signed. Given the certificate pair, create the Kubernetes secret:

   ```
   kubectl create secret --namespace bsr tls bsr-saml-cert \
     --cert=path/to/cert/file \
     --key=path/to/key/file
   ```

Configure Ingress

The BSR uses a Kubernetes Ingress resource to handle incoming traffic and for terminating TLS. The domain used here must match the `host` set in the Helm values above.

> ℹ️ TLS is required for the BSR to function properly.
> HTTP2 is preferred to allow for gRPC support.

```
bufd:
  ingress:
    enabled: true
    className: ""   # Optional ingress class to use
    annotations: {} # Optional ingress annotations
    hosts:
      - host: example.com
        paths:
          - path: /
            portName: http
    # Optional TLS configuration for the ingress.
    # May be omitted to configure TLS termination, depending on the ingress.
    # Requires a kubernetes TLS secret.
    tls:
      - secretName: bsr-tls-cert
        hosts:
          - example.com
```

If the load balancer does not support H2C, TLS can optionally be used for communication between the load balancer and the BSR by enabling TLS on the listening ports of the `bufd` application. This requires a Kubernetes TLS secret named `bsr-tls-cert`.

```
bufd:
  tls:
    enabled: true
    # Optional. Secret name for the TLS cert
    # secretName: bsr-tls-cert
  # Optional. Used to add annotations to the ingress service.
  # May be needed for some ingress controllers to function correctly.
  service:
    annotations: {}
```

Configure Observability

The `metrics` block is used to configure the collection and exporting of metrics from your application using prometheus:

```
observability:
  metrics:
    use: prometheus
    runtime: true
    prometheus:
      podLabels: # This is required if enabling network policies.
        app: prometheus
      port: 9090
      path: /metrics
```

Trusting Additional Certificates

If you bump into issues regarding self signed certificates, such as seeing the error `tls: failed to verify certificate: x509: certificate signed by unknown authority`, you can add your root certificates on the BSR. To trust additional certificates, mount the files on the `bufd` pod and include them in the client TLS configuration.

```
bufd:
  deployment:
    extraVolumeMounts:
      - mountPath: /config/secrets/certificates/cert.pem
        name: certificate
        readOnly: true
        subPath: cert.pem
    extraVolumes:
      - name: certificate
        secret:
          secretName: tls-cert
          items:
            - key: cert.pem
              path: cert.pem
  clientTLS:
    extraCerts:
      - /config/secrets/certificates/cert.pem
```

5. Install the Helm Chart

After following the steps above, the set of Helm values should be similar to the example below:

```
host: example.com
imagePullSecrets:
  - name: bufpullsecret
storage:
  use: s3
  s3:
    bucketName: "my-bucket-name"
    endpoint: "s3.us-east-1.amazonaws.com"
    region: "us-east-1"
postgres:
  host: "postgres.example.com"
  port: 5432
  database: postgres
  user: postgres
auth:
  method: saml
  saml:
    idpMetadataURL: "https://example-provider.com/app/12345/sso/saml/metadata"
  autoProvisionedAdminEmails:
    - "user@example.com"
bufd:
  ingress:
    enabled: true
    hosts:
      - host: example.com
        paths:
          - path: /
            portName: http
    tls:
      - secretName: bsr-tls-cert
        hosts:
          - example.com
    extraVolumeMounts:
      - mountPath: /config/secrets/certificates/cert.pem
        name: certificate
        readOnly: true
        subPath: cert.pem
```

```
    extraVolumes:
      - name: certificate
        secret:
          secretName: tls-cert
          items:
            - key: cert.pem
              path: cert.pem
  clientTLS:
    extraCerts:
      - /config/secrets/certificates/cert.pem
observability:
  metrics:
    use: prometheus
    runtime: true
    prometheus:
      podLabels: # This is required if enabling network policies.
        app: prometheus
      port: 9090
      path: /metrics
```

Using the `bsr.yaml` Helm values file, install the Helm Chart for the cluster:

```
helm install bsr \
oci://us-docker.pkg.dev/buf-images-1/bsr/charts/bsr \
--version $BSR_VERSION \
--namespace=bsr \
--values bsr.yaml
```

The BSR instance should now be up and running on `https://<host>`.

Optional configuration

Maintenance mode

The BSR has a maintenance mode in which the BSR will start up, but API calls are prevented and users of the web interface are informed that maintenance is in progress, and no database/object storage writes will occur. To enable this mode, set the `maintenance` Helm value:

```
maintenance: true
```

Feature flags

Certain BSR functionality is gated behind feature flags, which can be enabled through the `featureFlags` Helm value. The currently supported feature flags are:

```
featureFlags:
  # Prevent users from creating organizations in the BSR
  # Server admins can still create organizations when this flag is enabled
  disable_user_org_creation: true
```

Automatically adding members to organizations

To automatically add all members to an organization upon login, set the `auth.autoProvisionedMembershipOrganizations` Helm value:

```
auth:
  # Map of organizations which all members will be added to on login
  autoProvisionedMembershipOrganizations:
    exampleorg: ORGANIZATION_ROLE_MEMBER
```

Other configuration

There may be additional low-level values defined in the `values.yaml` chart or Helm templates that are subject to change. Please contact us before depending on these configuration values so we can better support your needs.

Upgrading / Downgrading

Upgrading

We currently support (and test) sequential upgrades from one version to the next immediate version.

For example, if you're on version `1.0.2` then you can safely upgrade to `1.0.3` without any downtime. If you're on an older version, such as `1.0.1`, then we strongly recommend upgrading to `1.0.2` first instead of skipping directly to `1.0.3`.

Downgrading

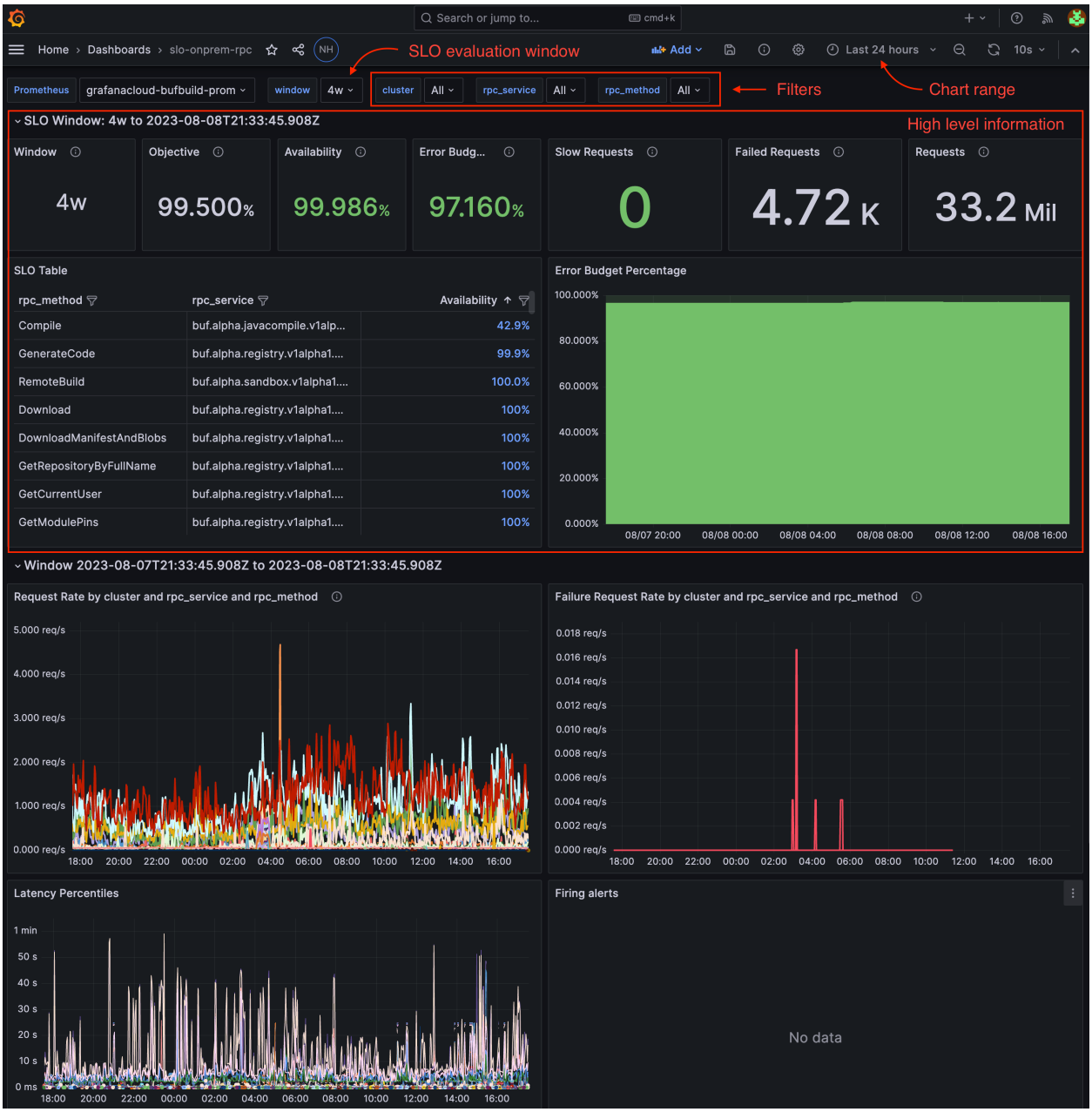
Downgrading is typically safe as long as you're not downgrading to a version that is more than one version older than your current version. For example, if you're on version `1.0.3` then you can downgrade to `1.0.2` but not `1.0.1`.

There are some exceptions to this rule, such as when a new version introduces a new feature that requires a database migration. In these cases, you may need to manually revert the database changes.

Observability

Dashboard

The dashboard included with the BSR exposes its overall health and can aid in identifying and diagnosing operational issues. There are several charts and components on the dashboard that are described in more detail below.



Service Level Objectives

The dashboard is primarily defined in terms of service level objectives (SLOs). Each RPC service and method pair has a success rate and latency objective, and the dashboard keeps track of when those objectives are met or missed. The success rate objective is currently 99.5% for all methods, and the latency objective is p95 request time < 15s for all methods except those related to code compilation which instead target p95 < 250s. The dashboard considers a request failed if it returns an unsuccessful error code (e.g. 5xx or the Connect RPC equivalent) or if it exceeds its latency target.

Filters

Filters can be set to restrict the set of data the dashboard operates on. Setting `cluster` is important for sites with multiple BSR clusters. Setting `rpc_service` and `rpc_method` is useful for drilling down into services or methods that are

be exhibiting issues.

High Level Information

The first two rows of the dashboard contain high level information to quickly provide the current operational status of the BSR.

First Row Boxes

The Availability, Error Budget, Slow Requests, Failed Requests, and Requests boxes display aggregate counts that give a very high level overview of what is happening with the BSR over the trailing SLO evaluation window.

SLO Table

This panel lists each service-method pair that has been observed, and its availability. The lowest availability pairs are listed first to call out any parts of the BSR that may be having issues.

Error Budget Percentage

This panel shows the aggregate (based on filters) error budget burn down for the BSR. Each point on this chart shows the error budget used up over the trailing SLO evaluation window.

Other Dashboard Components

Most of the other dashboard panels have titles that explain exactly what they are displaying and should be self-explanatory, others are clarified below.

Firing Alerts

Any alerts that are firing will be visible in this panel.

Config

Internal configuration for the dashboard. Currently cannot be changed by end users.

Alerts Overview

Alerts are configured for every RPC method in each deployed BSR cluster. Each method has an error budget based on an availability objective of 99.5% over the previous four weeks. An alert will fire if a given method returns errors at a rate that threatens the 99.5% objective. There are two classes of alerts defined by how rapidly they notify of errors.

High Priority Alerts: These respond swiftly, signalling immediate threats to the system. For these alerts to fire, 50% of the error budget must be consumed in the last hour and the method needs to have received at least 10 requests in that period.

Low Priority Alerts: These are designed for longer durations, capturing potential issues without responding to minor fluctuations. For these alerts to fire, 10% of the error budget must be consumed in the last 24 hours and the method needs to have had at least 10 requests in that period.

Simply put, if errors accumulate too quickly within the past one hour or 24 hours, alerts are triggered.

Diagnosing Alerts

When an alert fires, it's accompanied by specific labels to aid in your investigation: `cluster`, `rpc_method`, and `rpc_service.` These labels highlight the affected areas, helping you pinpoint the problem's origin.

Navigating the Dashboard

Once an alert is triggered:

1. **Firing Alerts Panel**: Here, you can view all active alerts, including the one that notified you.

2. **Error Rate Diagram**: Look for spikes in this chart, giving you a visual representation of when and where issues arose.
3. **Drill Down**: Utilize the provided labels (`cluster`, `rpc_method`, and `rpc_service`) to refine your search in the dashboard and focus on the affected areas.

By following these steps and utilizing the dashboard, you can swiftly identify, understand, and address any issues in your system.