Handle connectivity errors in alerts

- Understand connectivity issues in alerts
- Detect target availability with the Prometheus up metric
  - Use avg_over_time to smooth signal
  - Use synthetic checks to monitor external availability
- Manage offline hosts
- Handle query errors in Grafana Alerting
  - Design alerts for connectivity errors
  - Build a dedicated data source health alert
  - Reduce redundant error notifications
- Conclusion

Connectivity issues are a common cause of misleading alerts or unnoticed failures.

There could be a number of reasons for these errors. Maybe your target went offline, or Prometheus couldn't scrape it. Or maybe your alert query failed because its target timed out or the network went down. These situations might look similar, but require different considerations in your alerting setup.

This guide walks through how to detect and handle these types of failures, whether you're writing alert rules in Prometheus, using Grafana Alerting, or combining both. It covers both availability monitoring and alert query failures, and outlines strategies to improve the reliability of your alerts.

Understand connectivity issues in alerts

Typically, connectivity issues fall into a few common scenarios:

- Servers or containers crashed or were shut down.
- Service overload or timeout.
- Misconfigured authentication or incorrect permissions.
- Network issues like DNS problems or ISP outages.

When we talk about connectivity errors in alerting, we're usually referring to one of two use cases:

1. **Your target is down or unreachable.**
   The service crashed, the host was down, or a firewall or DNS issue blocked the connection. These are **availability problems**.
2. **Your alert query failed.**
   The alert couldn't evaluate its query—maybe because the data source timed out or an invalid query. These are **execution errors**.

It helps to separate these cases early, because they behave differently and require different strategies.

Keep in mind that most alert rules don't hit the target directly. They query metrics from a monitoring system like Prometheus, which scrapes data from your actual infrastructure or application. That gives us two typical alerting setups where connectivity issues can show up:

1. **Alert rule → Target**
   For example, an alert rule querying an external data source like a database.
2. **Alert rule → Prometheus ← Target**
   More common in observability stacks. For instance, Prometheus scrapes a node or container, and the alert rule queries the metrics later.
   In this second setup, you can run into connectivity issues on either side. If Prometheus fails to scrape the target, your alert rule might not fire, even though something is likely wrong.

Detect target availability with the Prometheus `up` metric

Prometheus scrapes metrics from its targets regularly, following the `scrape_interval` period. The default scrape interval is 60 seconds, which is generally considered common practice.

Prometheus provides a built-in metric called `up` for every scrape target, a simple method to indicate whether scraping is successful:

- `up == 1`: Your target is reachable; Prometheus collected the target metrics as expected.
- `up == 0`: Prometheus couldn't reach your target—indicating possible downtime or network errors.

A typical PromQL expression for an alert rule to detect when a target becomes unreachable is:

`up == 0`

But this alert rule might result in noisy alerts as one single scrape failure will fire the alert. To reduce noise, you should add a delay:

`up == 0 for: 5m`

The `for` option in Prometheus (or pending period in Grafana) delays the alert until the condition has been true for the full duration.

In this example, waiting for 5 minutes means the single scrape error won't result in a fired alert. Since Prometheus scrapes metrics every minute by default, the alert only fires after five consecutive failures.

However, this kind of `up` alert has a few potential downfalls:

- **Failures can slip between scrape intervals**: An outage that starts and ends between two evaluations go undetected. You could shorten the `for` duration, but this might lead to scrape failures that trigger false alarms.
- **Intermittent recoveries reset the `for` timer**: A single successful scrape resets the alert timer, which masks intermittent outages.

Brief connectivity drops are common in real-world environments, so expect some flakiness in `up` alerts. For example:

Scrape result (`up`)	Alert rule evaluation
00:00 `up == 0`	Timer starts
01:00 `up == 0`	Timer continues
02:00 `up == 0`	Timer continues
03:00 `up == 1`	Successful scrape resets timer
04:00 `up == 0`	Timer starts again
05:00 `up == 0`	No alert yet; timer hasn’t reached the `for` duration

The longer the period, the more likely this is to happen.

A single recovery resets the alert, that’s why `up == 0 for: 5m` can sometimes be unreliable. Even if the target is down most of the time, the alert didn't fire, leaving you unaware of a potential persistent issue.

Use `avg_over_time` to smooth signal

One way to work around these issues is to smooth the signal by averaging the `up` metric over a similar or longer period:

```
avg_over_time(up[10m]) < 0.8
```

This alert rule fires when the target is unreachable for more than 20% of the last 10 minutes, rather than looking for consecutive scrape failures. With a one minute scrape interval, three or more failed scrapes within the last 10 minutes now triggers the alert.

Since this query uses a threshold and time window to control accuracy, you can now lower the `for` duration (or pending period in Grafana) to something shorter—`0m` or `1m`—so the alert fires faster.

This approach gives you more flexibility in detecting real crashes or network issues. As always, adjust the threshold and period based on your noise tolerance and how critical the target is.

Use synthetic checks to monitor external availability

Prometheus often runs inside the same network as the target it monitors. That means Prometheus might be able to reach the target, but doesn't ensure it's reachable to users on the outside.

Firewalls, DNS misconfigurations, or other network issues might block public traffic while Prometheus scrapes `up` successfully.

This is where synthetic monitoring helps. Tools like the Blackbox Exporter let you continuously verify whether a service is available and reachable from outside your network—not just internally.

The Blackbox Exporter exposes the results of these checks as metrics, which Prometheus can scrape like any other target. For example, the `probe_success` metric reports whether the probe was able to reach the service. The setup looks like this:

Alert rules → Prometheus ← Blackbox Exporter (external probe) → Target

To detect when a service isn't reachable externally, you can define an alert using the `probe_success` metric:

```
probe_success == 0 for: 5m
```

This alert fires when the probe has failed continuously for 5 minutes—indicating that the service couldn't be reached from the outside.

You can then combine internal and external checks to make the detection of connectivity errors more reliable. This alert catches when the internal scrape fails or the service is externally unreachable.

```
up == 0 or probe_success == 0
```

As with the `up` metric, you might want to smooth this out using `avg_over_time()` for more robust detection. The smooth version might look like:

```
avg_over_time(up[10m]) < 0.8 or avg_over_time(probe_success[10m]) < 0.8
```

This alert fires when Prometheus couldn't scrape the target successfully for more than 20% of the past 10 minutes, or when the external probes have been failing more than 20% of the time. This smoothing technique can be applied to any binary availability signal.

Manage offline hosts

In many setups, Prometheus scrapes multiple hosts under the same target, such as a fleet of servers or containers behind a common job label. It’s common for one host to go offline while the others continue to report metrics normally.

If your alert only checks the general `up` metric without breaking it down by labels (like `instance`, `host`, or `pod`), you might miss when a host stops reporting. For example, an alert that looks only at the aggregated status of all instances will likely fail to catch when individual instances go missing.

This isn't a connectivity error in this context — it’s not that the alert or Prometheus can't reach anything, it’s that one or more specific targets have gone silent. These kinds of problems aren’t caught by `up == 0` alerts.

For these cases, see the complementary guide on handling missing data — it covers common scenarios where the alert queries return no data at all, or where only some targets stop reporting. These aren't full availability failures or execution errors, but they can still lead to blind spots in alert detection.

Handle query errors in Grafana Alerting

Not all connectivity issues come from targets going offline. Sometimes, the alert rule fails when querying its target. These aren’t availability problems—they’re query execution errors: maybe the data source timed out, the network dropped, or the query was invalid.

These errors lead to broken alerts. But they come from a different part of the stack: between the alert rule and the data source, not between the data source (for example, Prometheus) and its target.

This difference matters. Availability issues are typically handled using metrics like `up` or `probe_success` but execution errors require a different setup.

Grafana Alerting has built-in handling for execution errors, regardless of the data source. That includes Prometheus, and others like Graphite, InfluxDB, PostgreSQL, etc. By default, Grafana Alerting automatically handles query errors so you don't miss critical failures. When an alert rule fails to execute, Grafana fires a special `DatasourceError` alert.

You can configure this behavior depending on how critical the alert is and on whether you already have other alerts detecting the issue. In **Configure no data and error handling**, click **Alert state if execution error or timeout**, and choose the desired option for the alert:

- **Error (default)**: Triggers a separate `DatasourceError` alert. This default ensures alert rules always inform about query errors but can create noise.
- **Alerting**: Treats the error as if the alert condition is firing. Grafana transitions all existing instances for that rule to the `Alerting` state.
- **Normal**: Ignores the query error and transitions all alert instances to the `Normal` state. This is useful if the error isn't critical or if you already have other alerts detecting connectivity issues.
- **Keep Last State**: Keeps the previous state until the query succeeds again. Suitable for unstable environments to avoid flapping alerts.
  {{< figure src="/media/docs/alerting/alert-rule-configure-no-data-and-error-v2.png" alt="A screenshot of the `Configure error handling` option in Grafana Alerting." max-width="500px" >}}

This applies even when alert rules query Prometheus itself—not just external data sources.

Design alerts for connectivity errors

In practice, start by deciding if you want to create explicit alert rules — for example, using `up` or `probe_success` — to detect when a target is down or has connectivity issues.

Then, for each alert rule, choose the error-handling behavior based on whether you already have dedicated connectivity alerts, the stability of the target, and how critical the alert is. Prioritize alerts based on symptom severity rather than just infrastructure signals that might not impact users.

Build a dedicated data source health alert

Any alert rule query can fail to reach its data source, including one built specifically to check reachability. Prometheus-style data sources expose metrics such as `up` and `probe_success` for target-level availability, but those are still queries, and they fail the same way if the data source itself becomes unreachable. In any case, detecting that the data source itself is unreachable uses a different mechanism: the execution-error handling built into Grafana Alerting.

Create a separate alert rule dedicated to detecting when the data source itself is unreachable, distinct from any application-specific rules that happen to query it:

1. Reuse a query you already know returns data reliably when the data source is healthy. Any small, cheap query works, since the result doesn't matter—only whether the query executes.
2. In Configure no data and error handling, set **Alert state if execution error or timeout** to **Alerting**.
3. Add a pending period, for example `5m`, so the rule fires only once the data source has stayed unreachable for a sustained period, not for a single blip.

Set the same option to **Normal** or **Keep Last State** on your application-specific rules instead, so a transient error from the data source doesn't also fire a `DatasourceError` alert alongside the condition you're actually trying to detect. Which value to use isn't a one-time default—decide it per rule, based on whether that rule already has a dedicated connectivity alert covering it.

Reduce redundant error notifications

A single data source error can lead to multiple alerts firing simultaneously, sometimes bombarding you with many alerts and generating too much noise.

As described previously, you can control the error-handling behavior for Grafana alerts. The **Keep Last State** or **Normal** option prevents alerts from firing and helps avoid redundant alerts, especially for services already covered by `up` or `probe_success` alerts.

When using the default behavior, a single connectivity error will likely trigger multiple `DatasourceError` alerts.

Consider not treating these alerts in the same way as the original alerts, and implement dedicated strategies for their notifications:

- Reduce duplicate notifications by grouping `DatasourceError` alerts. Use the `datasource_uid` label to group errors from the same data source.
- Route `DatasourceError` alerts separately, sending them to different teams or channels depending on their impact and urgency.

For details on how to configure grouping and routing, refer to handling notifications and `No Data` and `Error` alerts documentation.

Conclusion

Connectivity issues are one of the common causes of noisy or misleading alerts. This guide covered two distinct types:

- **Availability issues**, where the target itself is down or unreachable (e.g., due to a crash or network failure).
- **Query execution errors**, where the alert rule can't reach its data source (e.g., due to timeouts, invalid queries, or data source outages).

These problems come from different parts of your stack, and require their own techniques. Prometheus and Grafana allow you to detect them, and combining distinct techniques can make your alerts more resilient.

With Prometheus, avoid relying solely on `up == 0`. Smooth queries to account for intermittent failures, and use synthetic monitoring to detect reachability issues from outside your network.

In Grafana Alerting, configure error handling explicitly. Not all alerts are equal or have the same urgency. Tune the error-handling behavior based on the reliability and severity of the alerts and whether you already have alerts dedicated to connectivity problems.

And don’t forget the third case: **missing data**. If only one host from a fleet silently disappears, you might not get alerted. If you're dealing with individual instances that stopped reporting data, see the Guide on handling missing data to continue exploring this topic.

□□□ #2
□ □□□ □□□ 2026-10-31 15:45:15 UTC
□ □□□ □□□ 2026-10-31 15:45:22 UTC