Evolution of XRootD Monitoring

Native Support for Prometheus Metrics

Guilherme Amadio

XRootD and FTS Workshop @ CNRS Rhône Auvergne Delegation Lyon, France

16 September 2026

Outline

01 Introduction and Motivation

02 XrdMetrics Design and Architecture

03 Registered Subsystems and Metrics

04 XrdHttpMetricsExporter

05 Grafana Dashboards

06 Future: Client Monitoring

07 Summary and Conclusion

Section 1

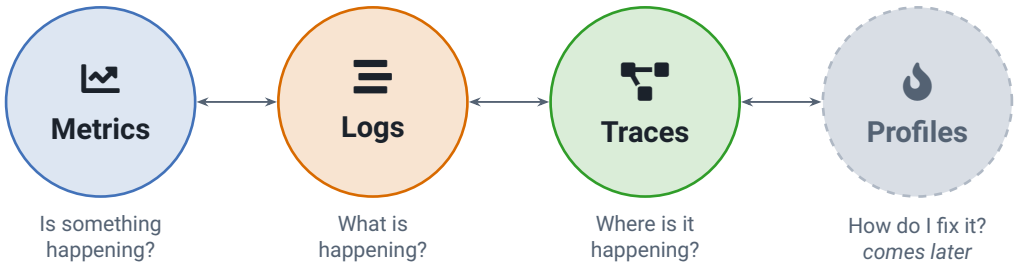
Introduction and Motivation

A Brief History of XRootD Monitoring

- XRootD has built-in monitoring since a long time
- Two main forms of monitoring: summary statistics and detailed monitoring
- Developed well before modern monitoring stacks (e.g. Grafana and Prometheus)
- For performance reasons, monitoring emitted as fire-and-forget UDP
- Historically delegated processing of detailed monitoring info to external tools
  - MonALISA — ALICE reporting via the embedded ApMon library
  - GLED — Collector behind CMS AAA monitoring
  - OSG collectors — first in Python, now written in Go
- GLED was discontinued due to performance and scalability issues
- Community wanted a native solution within the XRootD project

Cloud Native XRootD

- The community has long asked for modern monitoring built into XRootD itself
- We decided to fulfill that wish under one theme: Cloud Native XRootD
- Make XRootD interact well with a modern monitoring stack, out of the box



After the “four pillars of observability” (Grafana Labs).

Learning about Prometheus Metrics

“Prometheus Fundamentals”, by Julius Volz of PromLabs

Prometheus Metric Types

Four metric types, all built from the same one-line series format.

Type	What it holds	How you query it	XRootD examples
Counter	count, only goes up	`rate()`, `increase()`	requests, bytes read
Gauge	current value	the value itself	threads, open files
Histogram	cumulative `le` buckets	`histogram_quantile()`	I/O size distribution
Summary	in-process quantiles	the quantile directly	request latency

Counters and gauges cover most of XRootD’s series; histograms where the distribution matters.

Exposition Format: Counters and Gauges

```
# HELP xrootd_http_requests_total Total HTTP requests served
# TYPE xrootd_http_requests_total counter
xrootd_http_requests_total{verb="GET"} 187345

# HELP xrootd_sched_threads Current number of scheduler threads
# TYPE xrootd_sched_threads gauge
xrootd_sched_threads 24
```

- One line per series: `name{labels} value` — readable, and trivially parseable
- `# HELP` and `# TYPE` make the stream self-describing
- Counters only go up; Prometheus derives rates with `rate()`
- Gauges hold a current value, moving both ways

Exposition Format: Summaries and Histograms

```
# TYPE xrootd_pgread_bytes summary
xrootd_pgread_bytes{quantile="0.99"} 4194304
xrootd_pgread_bytes_sum 8123456789
xrootd_pgread_bytes_count 123456

# TYPE xrootd_http_request_duration_seconds histogram
xrootd_http_request_duration_seconds_bucket{le="0.1"} 3822
xrootd_http_request_duration_seconds_bucket{le="1"} 4510
xrootd_http_request_duration_seconds_bucket{le="+Inf"} 4600
xrootd_http_request_duration_seconds_sum 812.3
xrootd_http_request_duration_seconds_count 4600
```

- Summary: quantiles computed inside the process; cheap, but not aggregatable
- Histogram: cumulative buckets; quantiles at query time; aggregatable
- Both carry `_sum` and `_count`, so averages come for free

The Prometheus Query Language

```
# Request rate on this server, per verb, averaged over 5 minutes
rate(xrootd_http_requests_total[5m])

# The same, summed across all servers, keeping the verb label
sum by (verb) (rate(xrootd_http_requests_total[5m]))

# 99th percentile request duration, from the histogram buckets
histogram_quantile(0.99,
  sum by (le)
    ↪ (rate(xrootd_http_request_duration_seconds_bucket[5m])))
```

- Select series by name and labels; every result is a set of labeled series again
- `rate()` turns counters into per-second rates; `sum by` aggregates across servers
- Nothing is pre-registered: every dashboard panel and alert is just an expression

Section 2

XrdMetrics Design and Architecture

Initial Prototype with Claude

- First commit of the project (23 June): a flat metrics registry, built with Claude
  - Counter, Gauge, and Histogram
  - An HTTP handler to serve metrics
  - A simple bridge for `XrdMonRoll` summary statistics
  - Very useful, but still only a prototype
- Its value was in helping to understand the problems the design had to tackle later
  - The need for observed metrics in addition to native ones
  - Cardinality explosion from unbounded labels
  - Labels frozen at registration time to keep the hot path allocation-free
  - Serve `/metrics` through an `XrdHttpExtHandler` plugin
- Renamed, then deleted four days later — cheap to throw away as it was still young

Compatibility and Performance Requirements

- All supported platforms, down to Alma 8 with GCC 8.5
- ABI stability within a major version — must not break EOS, ROOT, etc
- No allocations in the hot path, scrape buffer must grow as needed and be reused
- Performance impact on the server must be minimal: no locks, only relaxed atomics
- Labels frozen at initialization time to make serialization faster for scrapes
- Current monitoring features must still work, byte for byte
  - Summary statistics from `xrd.report` used by ALICE
  - MonALISA still used and will continue to be used by ALICE
  - Detailed UDP streams also parsed by OSG collector
- And one forward-looking requirement: usable by EOS and by the client later on

Things Claude Got Wrong

- A flat, global registry — XRootD needed a hierarchy of pluggable subsystems
- `std::stringstream` scrapes — reallocates on every scrape; replaced by one reused buffer
- Awkward factories — help text after the labels, one factory for labeled and unlabeled
- Toolchain assumptions — no float `std::to_chars` on Alma 8; `"%g"` renders 1234567 as `1.23457e+06`
- An `instance` label — Prometheus silently renames it to `exported_instance`

Lack of context, not lack of skill: review supplies it, and the second iteration sticks.

Why Write Our Own Library?

We evaluated the two obvious candidates first.

OpenTelemetry C++ SDK	prometheus-cpp
• Complex, and notoriously slow	• Not available on all platforms
• Unwanted automatic instrumentation	• Tried in PR #1985, but did not land
• Depends on gRPC, protobuf, Abseil	• We kept the feel of its interfaces
• Short of maintainers	• Diverged on build, labels, and scraping

Writing `XrdMetrics` enables better integration within XRootD

Performance Considerations

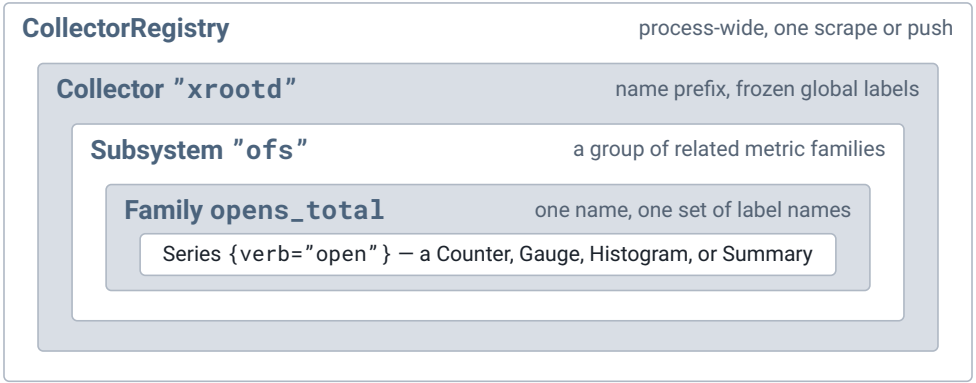
The main hot paths are metric updates on every operation, and the scrape itself. In the text format each line is `name{labels} value` — only the value changes.

```
struct Series {
  std::string           prefix;  // built once, at creation
  std::atomic<uint64_t> value;   // relaxed on the hot path
};

for (const auto &s : series) {
  out.append(s.prefix);          // pure memcpy
  appendValue(out, s.value.load(std::memory_order_relaxed));
  out.push_back('\n');
}
```

Updates use relaxed atomics; scrapes reuse the buffer and never format labels.

Hierarchy and Object Lifetime



Lifetime is strictly nested, and children hold back-pointers to parents. Never give a `Collector` a shorter life than the code registering into it.

Counters and Gauges

```
auto &opens = ofs.counter<std::uint64_t>("opens_total");
++opens;  opens += 5;  opens.value();   // the whole interface

auto &threads = sched.gauge<std::int64_t>("threads");
threads = 8;  threads += 2;  --threads; // set, up, and down

--opens;                                // does not compile
```

- A `Counter` exposes only increment; a `Gauge` also sets and decrements
- Instruments are returned by reference: look up once, cache, and update lock-free

Monotonicity is enforced by the type — misuse is a compile error

Histograms and Summaries

```
auto &h = io.histogram("size_bytes", {1, 2, 5}); // +Inf implicit
h.observe(0.5);  h.observe(1.5);  h.observe(3);  h.observe(10);
// size_bytes_bucket{le="1"} 1 ... {le="+Inf"} 4, _sum 15, _count 4

auto &s = xrootd.summary("request_bytes");
s.observe(100);  s.observe(250);  s.observe(50);
// request_bytes_sum 400, request_bytes_count 3
```

- Histogram buckets are upper bounds
- Simple usage with `h.observe(value)`
- The serializer emits cumulative `_bucket`, `_sum`, `_count`
- We decided to go with custom summaries which emit only `_sum` and `_count`
- Quantiles deliberately omitted to keep series aggregatable across servers

Observed vs Native Metrics

Native (owned)	Observed
• The metric *is* the storage • Increment the atomic directly • Prefer this	• The value is owned elsewhere • A reader runs at scrape time • Must be cheap and thread-safe

```
sched.observeGauge<std::int64_t>("threads", "worker threads")
     .add({}, [this]{ return num_Workers; }); // read at scrape
```

Needed due to ABI stability, could not change `XrdScheduler`'s counters

Portability to EOS and Beyond

`XrdMetrics` lives in `XrdUtils`, so EOS can register its own `Collector`.
EOS metrics then appear in the scrape as `eos_<subsystem>_<series>` metrics on the same `/metrics` endpoint in the MGM.

eos_exporter (today)	XrdMetrics (WIP branch)
• Separate process, calls EOS cmds	• In-process, integrated with the code
• No authentication, no rate limiting	• Auth, rate limit, TTL cache built in
• No cache: each scrape pays full price	• One scrape walks the full registry

The same path is open to ROOT and any application linking XRootD.

Measured on eospilot: 1.8 s with `XrdMetrics` vs 4.3 s with `eos_exporter`.

Section 3

Registered Subsystems and Metrics

XRootD Summary Statistics

The XRootD summary monitoring channel emits one XML record per interval over UDP.

```
xrd.report collector.example.org:3333 every 15 all -poll
```

- Each subsystem adds its own `<stats>` element:
  - `info`, `link`, `poll`, `sched`, `buff`, `xrootd`, `ofs`, `cms`, …
- Migrated summary statistics to `XrdMetrics` with source of truth reversed
  - The new `XrdMetrics` instruments own the counters
  - The legacy XML becomes a view into the metrics registry
  - Added a special serializer to emit output exactly as it currently is
  - Added byte-identity unit tests to ensure the new output matches the old one

Server Metrics: Buffers, Scheduler, Polling, Link, Process

Equivalent metrics for the XRootD summary statistics: one subsystem per component.

Subsystem	Example families	Tells you
`sched`	`threads`, `threads_idle`, `jobs_in_queue`	thread pool pressure
`buff`	`memory_bytes`, `buffers`, `requests_total`	buffer pool sizing
`poll`	`events_total`, `interrupts_total`, `attached`	socket polling activity
`link`	`connections`, `bytes_total`, `timeouts_total`	connection health
`process`	`cpu_seconds_total`, `server_info`	the process itself

General overview into the server process, first place to look when it misbehaves.

XRootD Server Metrics

Subsystem	Example families	Covers
On every server		
`xrootd`	`ops_total`, `bytes_total`, `logins_total`	the base protocol
`ofs`	`files_open`, `errors_total`, `redirects_total`	the filesystem layer
`http`	`requests_total`, `responses_total`	HTTP(S) traffic
`metrics`	`scrapes_total`, `endpoint_up`	the exporter itself
By role		
`proxy`	`opens_total`, `open_errors_total`	the proxy data path
`cache`	`bytes_total`, `data_used_bytes`	XCache (next slide)
`cms`	`nodes_active`, `selections_total`, `space_bytes`	the cluster manager

Base protocol series are flat `xrootd_*`; every other subsystem is `xrootd_<subsystem>_*`.

XCache Metrics

```
# TYPE xrootd_cache_bytes_total counter
xrootd_cache_bytes_total{origin="hit"}    812345678901
xrootd_cache_bytes_total{origin="miss"}    98765432109
xrootd_cache_bytes_total{origin="bypass"}   1234567890

# cache hit ratio over the last 5 minutes
sum(rate(xrootd_cache_bytes_total{origin="hit"}[5m]))
  / sum(rate(xrootd_cache_bytes_total[5m]))
```

- Lifecycle: `files_total{event=...}`
- Space: `disk_bytes` and `meta_bytes` by `state`
- Health: `bytes_written_total`, `checksum_errors_total`
- Need to setup and run a cache for some time to know what to monitor
- Still work in progress in need of refinement later

Extras and Future Work

- Parity with old metrics is just the beginning, we will expand it over time
- The registry offers richer functionality than the legacy summary stats could:
  - Labels, histograms, summaries, per-verb breakdowns, etc
- We plan to expand and refine the set of server health and activity metrics
  - Request latencies, I/O size distributions, error rates
- Some extras are already available with the `XrdMetrics` system
  - Reports on the exporter itself: scrapes, cache hits, rate-limited requests
  - Used in the detailed monitoring collector to report collector health metrics

We now have a solid basis upon which to build more monitoring features.

Section 4

XrdHttpMetricsExporter

Exporter Performance Considerations

- The exporter is an `XrdHttpExtHandler`
  - It runs within the server itself, no extra process to deploy or monitor
- A scrape serializes the registry into one reused buffer
  - Remains allocation-free in steady state, a strict design requirement
- A TTL cache (10 s default) makes the server resilient to heavy load
  - A cache miss serializes once under a mutex, others wait
  - Subsequent requests receive the cached metrics until next TTL window
  - Can be safely scraped by many Prometheus instances at high frequency
- The endpoint has introspection, emits metrics about the scrapes themselves
  - Exposes `scrapes_total` and `scrapes_cache_hits_total` metrics

Security and Service Protection

- Authentication: supports a simple bearer token, HTTP basic auth, or a TLS client certificate validated by `XrdHttp`
- Rate limit: 100 requests per second by default;
  - Beyond the rate limit, returns `HTTP 503` with `Retry-After` header
- Authentication runs before the limiter, so unauthenticated probes don't count
- Enforced authentication can be enabled with `metrics.requireauth`
  - Off by default for frictionless initial setup
- Failed authentications are also monitored via exposed metrics on the endpoint
  - `scrapes_rate_limited_total` and `scrapes_auth_failed_total`

Modular Serialization, Extensible to Other Output Formats

`Serializers` are modular to make it easy to add new output formats

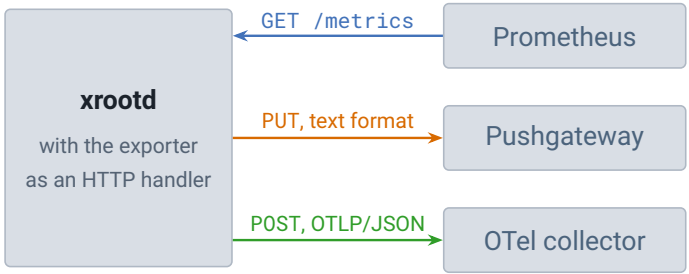
```
std::string out;       // caller owns and reuses the buffer

PrometheusTextSerializer text(out);
collector.serialize(text);  // Prometheus exposition format

OtelJsonSerializer otlp(out, "xrootd", resourceAttrs);
collector.serialize(otlp);  // OTLP/JSON resourceMetrics
```

- One registry walk can be used to serialize to multiple output formats
- Adding a new output format means just adding a new serialization function
- A `MetricSnapshot` serializer supports the legacy `xrd.report` XML view
- Easily serialize to a file by writing out the `std::string` buffer

Scrape with Pull, or Push with OTLP



Either, both, or neither push target may be active.
The pull endpoint always serves the Prometheus text format.
Flexible enough to adapt to your deployment model.

Configuration

```
http.exthandler metrics libXrdHttpMetricsExporter.so /metrics

metrics.path            <path>     # served request path (default /metrics)
metrics.instance        <name>     # instance label / resource id (default hostname)
metrics.pushurl         <url>      # Pushgateway base URL; enables Pushgateway push
metrics.pushinterval    <seconds>  # Pushgateway push period (default 30)
metrics.pushjob         <name>     # Pushgateway job label (default xrootd)
metrics.otelurl         <url>      # OTLP/HTTP metrics URL; enables OTLP push
metrics.otelinterval    <seconds>  # OTLP push period (default 30)

# Scrape endpoint authentication (pull path only)
metrics.requireauth     yes|no     # require a credential to scrape (default no)
metrics.authtoken       <token>    # accept Authorization: Bearer <token>
metrics.authpassword    <password> # accept HTTP Basic Auth with this password

# Scrape endpoint protection (pull path only)
metrics.scrapettl       <seconds>  # reuse serialized body (default 10; 0 = off)
metrics.scraperatelimit <n>        # max scrapes/s (default 100; 0 = unlimited)

# Collector-level (process-wide)
metrics.label       <key> <value> # constant global labels on every series
```

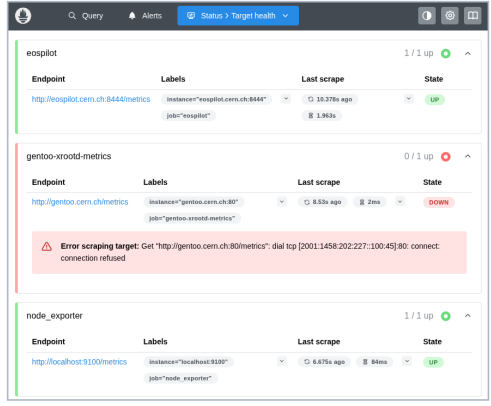
Section 5

Grafana Dashboards

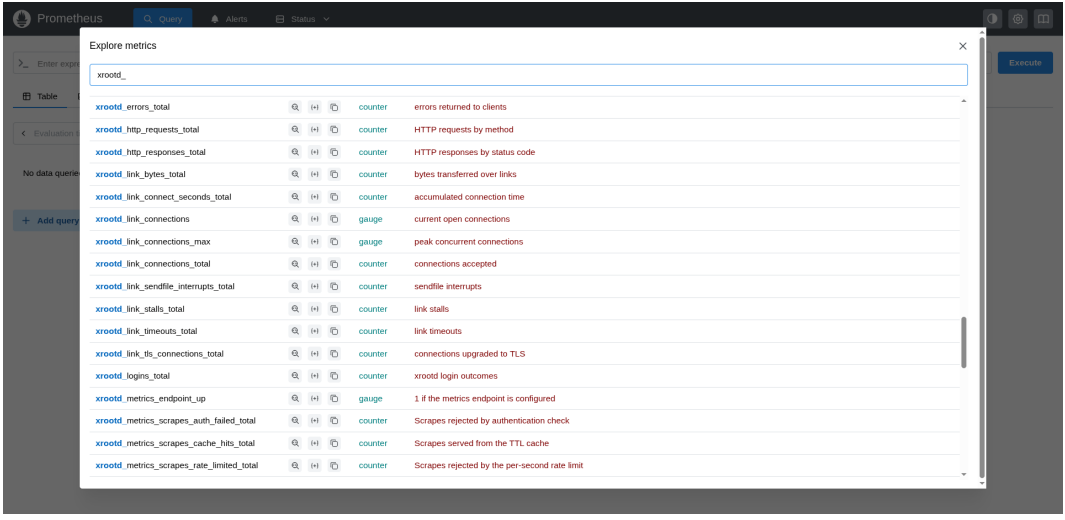
Configuring Prometheus

One scrape job in `prometheus.yml`:

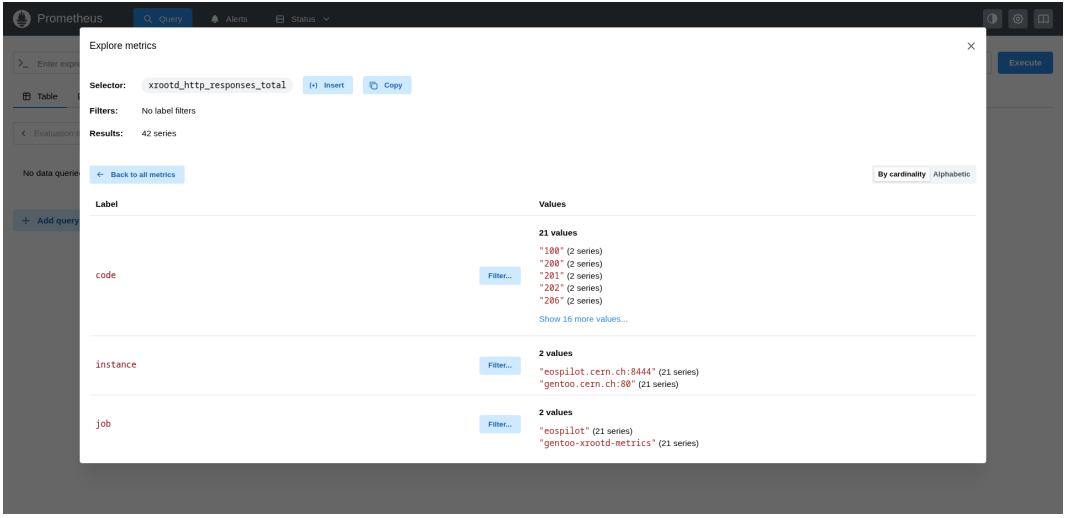
```
scrape_configs:
  - job_name: eospilot
    static_configs:
      - targets:
        ↪  ["eospilot.cern.ch:8444"]
    - job_name: gentoo-xrootd-metrics
    static_configs:
      - targets: ["gentoo.cern.ch:80"]
    # with metrics.requireauth:
    authorization:
      credentials_file: token
```



Exploring Metrics in Prometheus



Labels and Cardinality

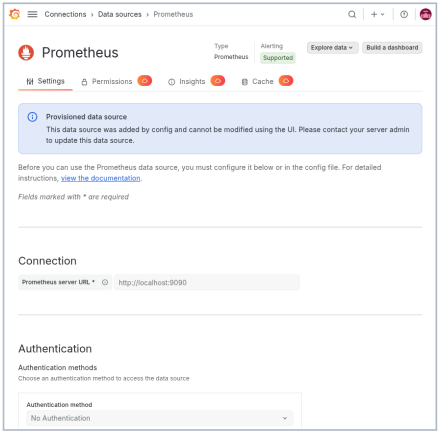


Configuring the Grafana Data Source

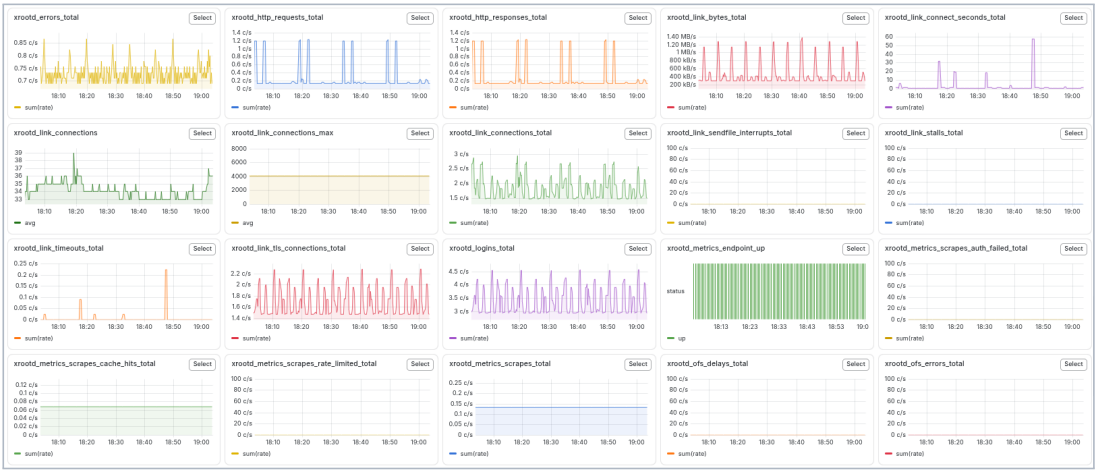
Provision it, or click it together once:

```yaml
# provisioning/datasources/prom.yaml
apiVersion: 1
datasources:
  - name: Prometheus
    type: prometheus
    url: http://localhost:9090
    isDefault: true
```

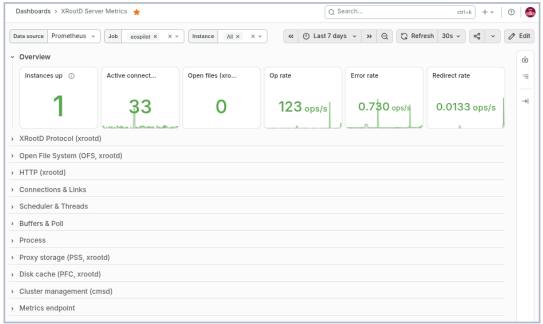
The last step of setup: the shipped dashboards ask for the data source at import time.



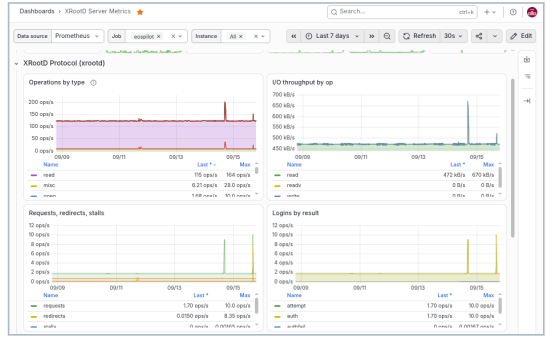
Metrics Overview



The Dashboard, Section by Section

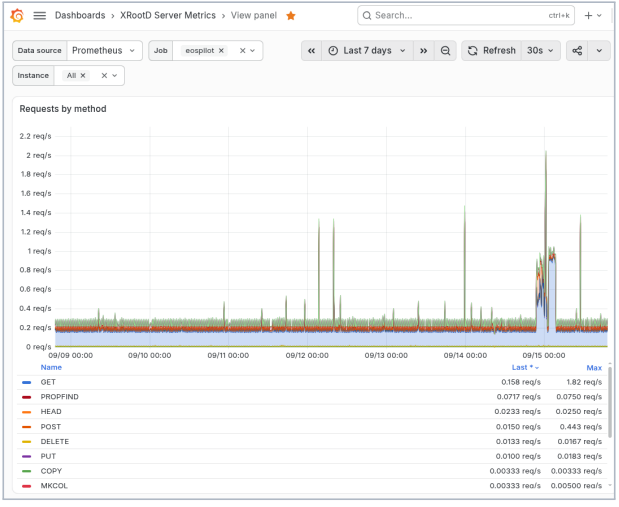


Overview, one collapsible row per subsystem.



The protocol section expanded.

HTTP Requests by Method



Section 6

Future: Client Monitoring

XrdCl Monitoring Features

- The client has had a monitoring plugin interface for years: `XrdCl::Monitor`
- Event-driven: connect and disconnect, open and close, copy begin and end, checksums, I/O errors
- Loaded through the environment: `XRD_CLIENTMONITOR` names the library, `XRD_CLIENTMONITORPARAM` its argument
- The familiar story: raw events only — aggregation and export are left entirely to the plugin

The client today is where the server was: the hooks exist, but nothing consumable comes out.

Plans for a Client Monitoring Plugin

- A `Monitor` plugin built on `XrdMetrics` — already in `XrdUtils`, linked by every client
- Aggregate in-process: counters per event, histograms for I/O sizes and latencies
- Short-lived jobs write the text format to a file at exit; long-lived clients can push, Pushgateway or OTLP
- Same metric names and labels as the server, so the same dashboards work

Nothing new to invent: the registry, the serializers, and the push transports all exist already.

Use Cases: XCache and Batch Jobs

Where client metrics pay off first.

XCache	Batch jobs
• A cache is also a client of the origins • Fetch rates, latencies, and errors • Both directions on one endpoint	• Per-job I/O accounting on the node • A metrics file at job end, or a push • Aggregate by site, VO, or workflow

The same registry serves a daemon that runs for months and a job that runs for minutes.

Section 7

Summary and Conclusion

Summary and Conclusion

- Native Prometheus metrics in XRootD is working well already
  - `XrdMetrics` registry and `XrdHttpMetricsExporter` plugin
- Made to be reusable, deployed and tested as part of EOS too
- Faster than `eos_exporter`, more robust and resilient
- Plan is to release it with XRootD 6.3 in December 2026
- The exported metrics will be expanded and refined between now and then
- XRootD 6.3-rc1 with these features to become available in our testing RPM/DEB repositories soon after XRootD 6.2 is released

Try the release candidate, and tell us what you want to see on your dashboards!

Questions?