GARNET | Ghanaian Academic and Research Network

DAY 1 · SESSION 5

PromQL Fundamentals

Selectors, rates, aggregation, and the patterns you will reuse forever

THE LANGUAGE

Why PromQL matters

Ad-hoc investigation

Type a query in /graph to debug a problem in real time.

Every dashboard panel

Each Grafana panel is one or more PromQL expressions.

Every alert

An alert rule is a PromQL expression that becomes true when something is wrong.

If you can't write the query, you can't build the dashboard or fire the alert.

MENTAL MODEL

The core distinction: instant vs range vectors

INSTANT VECTOR

One value per series, at one moment

```
node_load1
```

Returns: for each matching series, the most recent value

Used in: simple graphs, alert thresholds, and as input to math operations.

RANGE VECTOR

Many values per series, over a window

```
node_load1[5m]
```

Returns: for each series, all samples in the last 5 minutes

Cannot be graphed directly. Must be reduced first — usually by rate(), avg_over_time(), etc.

SELECTORS

Filtering with label matchers

`up`	All series of metric 'up'.
`up{job="node"}`	Only the node job. Equality matcher.
`up{job!="node"}`	Anything except the node job. Negation.
`up{job=~"node|snmp"}`	Regex match. Either node or snmp.
`up{job!~"^test.*"}`	Regex negation. Excludes anything starting with 'test'.

Four matcher operators: = != =~ !~

FUNCTIONS

rate() — the function you'll use most

Per-second average rate of increase, over a time window. Designed for counters.

```
rate(node_network_receive_bytes_total[5m])
```

What it does, step by step:

1. Take samples of the counter from the last 5 minutes
2. For each series, compute (last - first) / time_window
3. Detect any counter resets and account for them
4. Return one per-second value per series

WHICH ONE WHEN

rate() vs irate() vs increase()

`rate()`

Average over the window

WHEN TO USE

Default choice. Smooth graphs, alerting.

`irate()`

Last two samples only

WHEN TO USE

Volatile signals, fine-grained spikes. Don't use for alerts.

`increase()`

Total increase in the window

WHEN TO USE

Same as rate × window. Use when 'how many in last hour' is the question.

Default to rate(). Reach for irate() only when you genuinely need it (rare). Use increase() when the question is 'total in window'.

OPERATORS

Aggregation: many series → fewer

Aggregation operators reduce many series down. The 'by' / 'without' clauses control grouping.

`sum(rate(http_requests_total[5m]))`	Total requests/sec across everything.
`sum by (job) (rate(http_requests_total[5m]))`	Total requests/sec, one line per job.
`sum by (instance, job) (rate(...))`	Group by both instance AND job.
`sum without (instance) (rate(...))`	Sum across instances; keep all other labels.
`avg(node_load1) by (instance)`	Average load1 per host.

Operators: sum, avg, min, max, count, stddev, topk, bottomk, quantile

VECTOR MATCHING

Doing math between two metrics

Arithmetic operates label-by-label. Each series on the left finds its label-twin on the right.

```
node_filesystem_avail_bytes  /  node_filesystem_size_bytes

# →  fraction of disk free, per filesystem
```

Each numerator series is matched to a denominator series with the same labels.
The result has those labels too.

Common gotcha: if the two metrics have different label sets, matches fail and you get an empty result.

Use on(...) and ignoring(...) to control which labels participate in matching.

histogram_quantile()

Latency percentiles from histograms

If you only need to compute one thing from a histogram, it's a percentile.

```
histogram_quantile(
  0.95,
  sum by (le) (rate(http_request_duration_seconds_bucket[5m]))
)
```

Reads as: *the 95th-percentile request duration over the last 5 minutes.*

Three things to remember

- Always rate() the buckets first — they are counters
- Always preserve the 'le' label when aggregating
- Quantile is a fraction (0.95 = p95), not a percentage

RECIPES

Patterns to memorise

CPU usage %	`100 * (1 - avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])))`
Disk free %	`100 * node_filesystem_avail_bytes / node_filesystem_size_bytes`
Memory used %	`100 * (1 - node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)`
Network in (Mb/s)	`rate(node_network_receive_bytes_total[5m]) * 8 / 1e6`
Targets down	`up == 0`

TRAPS

Pitfalls that bite everyone

✗ `rate(node_load1[5m])`

✓ `avg_over_time(node_load1[5m])`

rate() is for counters. node_load1 is a gauge.

✗ `sum(node_cpu_seconds_total)`

✓ `sum(rate(node_cpu_seconds_total[5m]))`

Always rate counters before summing them.

✗ `http_requests_total[1m]`

✓ `rate(http_requests_total[1m])`

A range vector cannot be graphed directly.

✗ `rate(x[15s])`

✓ `rate(x[1m])`

Window must hold ≥ 4 samples; 15s isn't enough at 15s scrape interval.

OVER TO YOU

PromQL is now your tool

Selectors filter · Functions transform · Aggregators reduce · The mental model is everything