HANDS-ON PMM 3

Building a Professional Observability Stack

Michael Coburn
Tibor Korocz

Percona Live 2026

“Why is the database slow?”

The most common question a DBA gets asked.
Today: stop guessing.

PMM 3 is a **single pane of glass** for that question. Four angles to answer it:

- **Query Analytics** — frequency × duration × variance
- **Dashboards** — signal vs. noise, workload-aware
- **Advisors** — automated configuration + security + perf review
- **Correlation** — DB metrics against system load

And one underlying architectural change — **rootless** — that finally lets you deploy PMM on the clusters PMM 2 couldn’t reach.

What you leave with

- A working mental model for **QAN triage**: frequency × duration × variance
- A workload-aware sense of **signal vs. noise** in dashboards
- A reflex for **Advisor triage** — “real, exploitable, breaking” vs. noise
- A framework for **correlating performance signals across your stack**
- One live experience of **observing a real fault** propagate through PMM

Stop guessing. Start fixing.

Agenda (3 hours)

Time	Lab	What
0:00–0:10	0 — Pre-flight	URLs, login, sanity
0:10–0:35	1 — Rootless	Why PMM 3 unblocks restricted clusters
0:35–1:20	2 — QAN	Hunt the bottleneck
1:20–1:45	3 — Dashboards	Signal vs. noise
1:45–2:00	Break	
2:00–2:25	4 — Advisors	Audit health, triage findings
2:25–2:55	5 — Chaos	Observe a fault propagate via PMM
2:55–3:00	Wrap	Take-home

Quick
Kubernetes
refresher
PERCONA

Containers vs. VMs

A container is a **process** running in a Linux namespace, not a VM. Same kernel as the host, just sandboxed. Docker / containerd / runc are the common implementations.

	Virtual Machine	Container
Isolation	Hypervisor + guest OS	Linux kernel namespaces + cgroups
Overhead	Heavy (full OS per VM)	Light (process-level)
Startup	Minutes	Milliseconds
Use case	Whole-system isolation	App + its deps, packaged

Docker in one slide

A **Docker image** is a recipe — a versioned, layered filesystem with your app and its deps. A **container** is a *running instance* of an image (image : container :: class : object).

```
docker run -d -p 8080:80 -v /data:/srv:rw nginx
#          |  |     |         |             └── image name
#          |  |     |         └── persistent volume mount
#          |  |     └── publish container :80 to host :8080
#          |  └── detach (run in background)
#          └── create a container from the image
```

We use containers because **“works on my laptop” stops being a lie** — the same image runs identically on your laptop, in CI, and in production.

Why orchestration

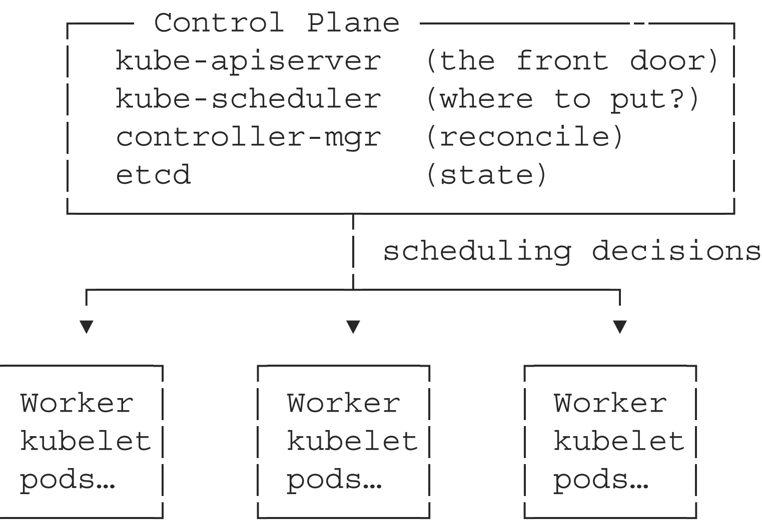
“Docker is great for 10 containers. What about 100 containers across 20 hosts?”

Orchestration handles, for you, the things you would otherwise script by hand:

- **Health checks** — restart what’s broken
- **Replicas** — keep N copies running at all times
- **Scaling** — more / fewer containers based on load
- **Rolling updates** — deploy new versions without dropping traffic
- **Service discovery** — container A finds container B by name, not IP

This is what **Kubernetes** does. It’s the de-facto orchestrator.

Kubernetes in one slide



- **Pod** = one (or a few co-located) container(s), shared network + storage
- **Node** = a VM (or bare-metal) machine running pods
- **Cluster** = a set of nodes managed by one control plane

You'll see **4 worker nodes** in your cluster today.

Kubernetes Operators

“What if Kubernetes knew how to run a database?”

An **Operator** is a controller that teaches Kubernetes a new resource type via a **Custom Resource Definition** (CRD), then runs a control loop to reconcile reality with the desired state.

```
apiVersion: pxc.percona.com/v1
kind: PerconaXtraDBCluster          # ← this CRD comes from the operator
metadata:
  name: pxc-proxysql
spec:
  pxc:
    size: 3
    image: percona/percona-xtradb-cluster:8.0
  proxysql:
    enabled: true
    size: 2
```

You declare *what you want*; the operator handles backups, rolling restarts, version upgrades, replica failover. **All four DBs in this workshop are operator-managed.**

The stack running for you

Layer	Component
Cloud	DigitalOcean Kubernetes (one cluster per attendee)
GitOps	Argo CD (App-of-Apps via ApplicationSet)
Observability	**PMM 3** (metrics → VictoriaMetrics, QAN → ClickHouse) + Grafana
Operators	Percona Operators for PG / PXC / PS
Ingress	Traefik + cert-manager
IDE	`code-server` in your browser
Chaos	`database-chaos-injector` (GitOps-driven faults)

PMM 3 supports six engines

Engine	In today's workshop
MySQL (Percona Server, PXC, Oracle MySQL)	✓ three topologies deployed
PostgreSQL (with Patroni + pgBouncer)	✓ deployed
MongoDB	covered in PMM docs, not today
Valkey	covered in PMM docs, not today
Redis	covered in PMM docs, not today
ProxySQL	✓ as a topology layer

Four databases, four different workloads

DB	Topology	Sysbench profile
PostgreSQL	Patroni + pgBouncer	`web-oltp` (pareto)
PXC	+ ProxySQL	`viral-contention` (zipfian)
PS Group Replication	+ HAProxy	`tpcc`
PS async	+ HAProxy	`analytics-scan`

Your portal

We'll share **one link** to a single-page HTML portal. Open it locally, find your attendee row.

Each row gives you:

- **Three URLs** — Web IDE, PMM, ArgoCD
- **One password** — same one unlocks all three
- **One git branch** — `env/attendee-NN` (already pushed for you)

Bookmark the portal. Re-open it any time during the 3 hours — your credentials live there, nothing to lose.

Your IDE — `code-server`

VS Code, running on **your** cluster, served to your browser.

- Open the **IDE URL** from your portal row
- Enter the **password** (same one)
- **View** → **Terminal** (or `` Ctrl+` ``) opens a shell in the pod
- Workshop repo is **pre-cloned** at `~/workspace`, on **your branch**
- `kubectl`, `yq`, `gh`, `argocd`, `git` already on `PATH` from `/shared/bin`

If your browser tab disconnects, just re-open the URL — your terminal state survives. Workspace files persist on a 5 GiB volume.

PMM
Architecture
PERCONA

PMM Server - Component-Based View

PMM Server UI

Metrics Dashboards

QAN UI

Advisors

Alerting

Backup Management

Inventory

PMM Client

PMM Server Backend

Nginx proxy

supervisord (pid 1)

Nomad

managed (API)

QAN API

VictoriaMetrics

Grafana

Persistence Layer

ClickHouse

QAN Data

VM DB

Metrics

PostgreSQL

State

State files

PMM Client - Component-Based View

PMM Client

pmm-admin >_

pmm-agent

vmagent

mysqld_exporter

node_exporter

mongodb_exporter

postgres_exporter

MySQL

Network

CPU

PMM Server Backend

Nginx proxy

managed (API)
(+authZ layer)

pull

push

QAN API

VictoriaMetrics

Grafana
(+authN layer)

Persistence Layer

PERCONA®

PMM Client-Server Interactions

Uses

PMM Client

PMM Server

Monitored system

Database/service or node

Query Analytics Metrics

pmm-agent

Invokes appropriate exporter on command

Control API

pmm-managed

Manages configuration, exposes API for other components

grafana

Data presentation and data analysis

Collects metrics

Controls

Query Analytics Metrics

exporters

Collection of programs, one for each monitored system type

Pulls from

VictoriaMetrics

Recieves and stores Prometheus-compatible metrics data

Query Analytics

Detailed database query performance analysis tool

Pulls from

Pushes to

vmagent

VictoriaMetrics agent

LAB 0
Pre-flight:
Welcome &
sanity
check
PERCONA

Lab 0 — Open three tabs, run three checks

1. Open the portal page **DOCUMENTATION**
   https://pmm-workshop-portal.sfo3.digitaloceanspaces.com/docs/index.html
2. Open the portal page for **ATTENDEE** and find your row - **percona-live-2026**
   https://pmm-workshop-portal.sfo3.digitaloceanspaces.com/attendees.html
3. Open **PMM**, **ArgoCD**, **IDE, metrics** in four new tabs
4. Log into all four with the **portal password** for your attendee environment
5. In PMM → **Inventory** → **Services**, confirm **4 services**
6. **Click one service row** — see DB status, agent health, and one-click links to its **QAN** and **Summary** dashboards

In the IDE Terminal:

```
whoami
which kubectl yq gh argocd git
kubectl get nodes
kubectl get deploy | grep -c 'sysbench-runner-.*-workload-writer'
```

Flag the instructor immediately if anything fails.

LAB 1
Secure by Design: PMM 3 Rootless
PERCONA

Why rootless matters

	PMM 2	PMM 3
Data dir UID	**0** (root)	**1000** (non-root)
Pod Security Standards	breaks “restricted”	passes “restricted”
OpenShift SCCs	requires `anyuid`	runs under `restricted-v2`
PCI / FedRAMP posture	privileged container risk	clears the bar
Agent auth	shared API keys	**service-account tokens** (RBAC-bound, rotatable)

Lab 1 — Inspect the running pod

```
# SecurityContext: runAsNonRoot, runAsUser=1000
kubectl get pod -n workshop pmm-0 -o yaml \
  | yq '.spec.containers[0].securityContext'


# What UID is the process actually running as?
kubectl exec -n workshop pmm-0 -- id


# Data dir owned by 1000:1000, not root
kubectl exec -n workshop pmm-0 -- ls -la /srv/
```

Optional: find the same `securityContext` rendered in ArgoCD → `pmm` application → Manifests tab.

Service labels in PMM

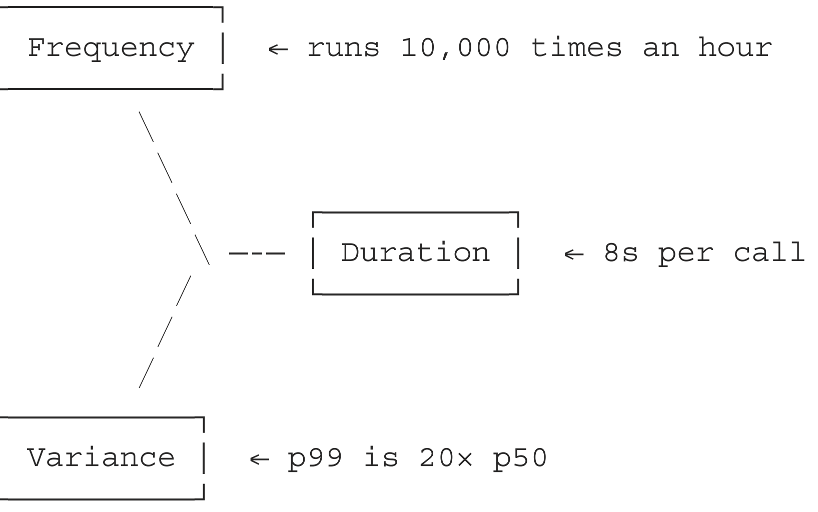
When you register a service with `pmm-admin add`, you tag it:

```
pmm-admin add mysql \
  --environment=prod  \
  --cluster=ps-gr-haproxy \
  --replication-set=ps-gr-haproxy \
  --service-name=ps-gr-haproxy-mysql-0 \
  --custom-labels='team=payments,region=eu-west'
```

These labels become **filter dimensions** in QAN and dashboards. The four DBs you'll analyze today were registered this way — every filter you'll use in Lab 2 and Lab 3 is a label that was set at registration time.

LAB 2
Hunt the Bottleneck: QAN
PERCONA

The three axes of “expensive”



- Frequency alone → death by a thousand cuts
- Duration alone → a single user staring at a spinner
- Variance alone → “it was fine yesterday”

QAN structure — what to click on

- **Service filter** — scope to one DB at a time
- **Fingerprint** — normalized query shape (literals stripped)
- **Time range** — last 15m to start; widen if data is sparse
- **Column picker** — surface Rows Examined, Rows Sent, Locks
- **Drill-in tabs** — Examples, Explain, Tables, Plan

The three-axis questions all live in this same UI.

Activity A — Heaviest query on PG (10 min)

1. PMM → **Query Analytics** (QAN)
2. Filter: `service_name = pg-patroni-pgbouncer`
3. Sort by **Query Time**
4. Click the top fingerprint → **Examples**, **Explain**, **Tables**

Is it expensive because it runs **often**, takes a **long time per run**, or **both**?

Activity B — Same workload, three engines

Why does the same “MySQL” show three different query mixes?

Service	Profile	Expect
`pxc-proxysql`	viral-contention	Row-lock-heavy updates
`ps-async-haproxy`	analytics-scan	Long range scans
`ps-gr-haproxy`	tpcc	Transactional sequences

Activity C — Spot a missing index (10 min)

Pick any query — any DB — whose **Rows Examined / Rows Sent ≫ 1.**

- Open the **Explain** tab in QAN
- Find the table scan
- **Answer**:
  - What index would help?
  - How would you confirm in QAN that it worked after deploy?

Activity D — QAN power filters (10 min)

The QAN **Filters panel** isn't just service/schema — it has boolean attributes you can check to surface specific query shapes instantly.

Try each filter, one at a time:

- ☑️ **Full Table Scan = yes** → every query missing an index, surfaced
- ☑️ **Filesort = yes** → every `ORDER BY` that couldn't use an index
- ☑️ **Tmp Disk Tables = yes** → every query that overflowed memory
- ☑️ **No Index Used = yes** → the missing-index detector, one click

Answer: which filter would you wire into your team's weekly review?

Activity E — Tag your queries (5 min, optional)

PMM 3 extracts `key=value` pairs from **SQL comments** and exposes them as **new QAN filter dimensions**:

```
SELECT * /* cluster='east', team=payments */
  FROM users WHERE last_login > NOW() - INTERVAL
7 DAY;
```

In QAN, filter by `cluster=east` the same way you filter by `service_name` today.

Answer: pick one tag your ORM could inject — `tenant_id`, `route`, `deploy_id`. Where would that filter change your triage?

LAB 3
Signal vs. Noise:
Dashboards
PERCONA

Thought Experiment

If you had a wall TV in the office, which 4 panels would you put on it — and which 4 would you never look at unless a theory said to?

- A panel that's noise during normal ops can be **critical during an incident**
- Signal is not universal — it depends on **your** workload
- Where you and your neighbour disagree is the **interesting** part

Three shapes of PMM dashboard

Shape	Example	Use it when
Overview	MySQL Instances Overview	Bird's-eye, Top-5, what's hottest right now
Summary	MySQL Instance Summary	One service, full depth, incident drill-down
Compare	MySQL Instances Compare	Up to **4 instances side-by-side** in one view

Three dashboards, six picks each

For each dashboard, pick **3 signal** and **3 noise** panels:

Dashboard	Pointed at	Signal	Noise
MySQL Overview	`ps-async-haproxy`	___	___
InnoDB Details	`pxc-proxysql`	___	___
PG Instance Overview	(PG)	___	___

Compare 4 services at once

Open **MySQL Instances Compare** — pick the three MySQL services + add PG separately for context:

- `pxc-proxysql` (viral-contention)
- `ps-gr-haproxy` (tpcc)
- `ps-async-haproxy` (analytics-scan)

The four workloads side-by-side in one view. Same row of panels for each.

Answer: which panel makes the workload differences most obvious at a glance — and which makes them look deceptively similar?

LAB 4
Audit Health: Percona Advisors
PERCONA

What Advisors are

- **Checks** written by Percona engineers — open source
- Run **on a schedule** against your registered DBs
- Categorized: **Configuration**, **Performance**, **Query**, **Security**
- Hosted on Percona Platform — PMM pulls them when **Telemetry is on**
- Output: a finding with severity, evidence, and suggested remediation

The skill we practice today is triage, not running the checks.

The triage exercise

PMM → **Advisors** → Run all categories.

For each of the 4 DBs, pick: - One **Security** finding - One **Performance** finding

Drop each into one bucket:

Bucket	Meaning
🔥 **Fix today**	Real, exploitable, breaking right now
📅 **Next sprint**	Real, but not on fire
🗑️ **Ignore**	False positive for your workload

Lab 4 — Discussion

- Show how to **disable** an Advisor that's noise for OLAP workloads
- Show how to **tune a threshold** — e.g., what "too many connections" means in your environment
- Defend a bucket assignment you **almost** put in "Fix today" but didn't

Success: at least one finding in each of the three buckets per attendee.

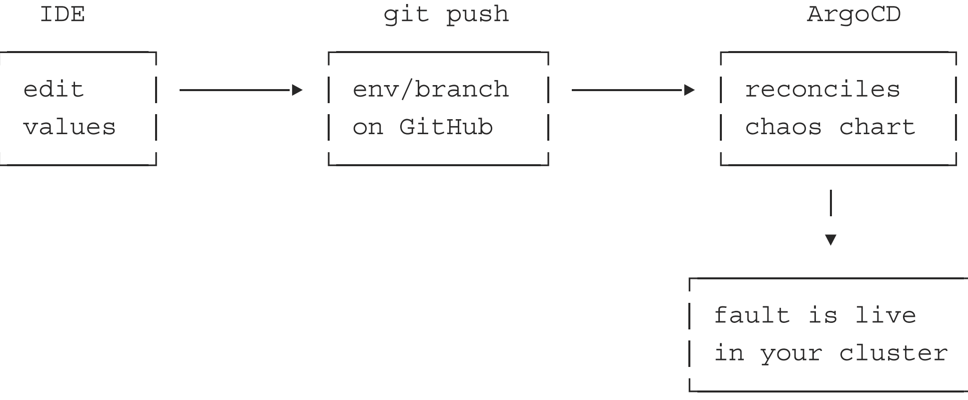
PMM 3 Alerting

Advisors find problems on a schedule.
Alerting tells you the moment they happen.

- **GA in PMM 3** (no longer technical preview)
- Built on Grafana Alerting + Percona-curated alert templates
- **Write your own templates** (YAML upload, same authoring model as Advisors)
- PMM 3 ships **Grafana 12.4**: alert state history, recording rules, silences, mute timings
- Notification channels: **email**, **Slack**, **PagerDuty**, generic **webhook**, MS Teams, Opsgenie
- Open: **PMM** → **Alerting** — browse the built-in templates, fire a test alert into a webhook of your choice

LAB 5
Correlate Load: Simulated Incident
PERCONA

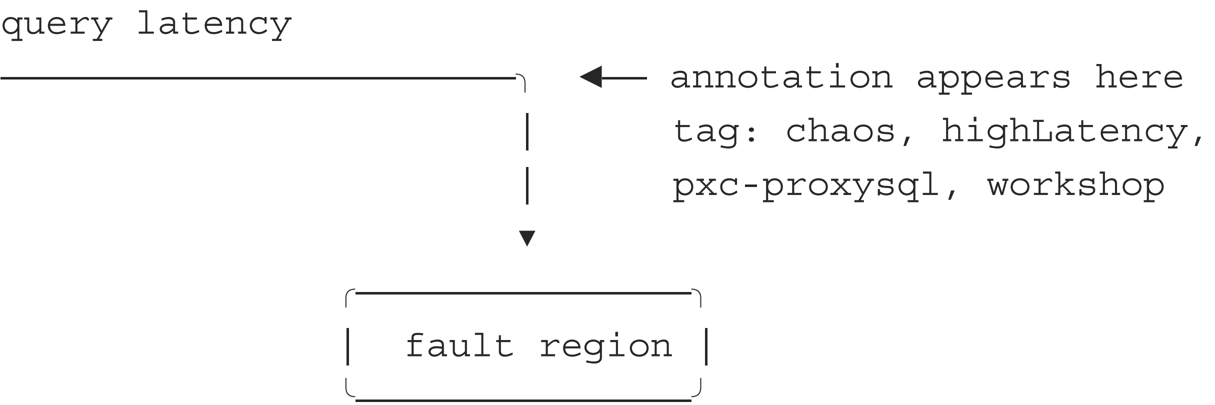
How chaos lands here



You push to Git. The fault appears. PMM reacts. You observe.

PMM marks the moment for you

The chaos-injector pod **auto-annotates** PMM the moment your fault fires. You'll see a vertical line (region annotation) on every dashboard at the fault timestamp, tagged with the fault name.



Cause and effect, visible on the same timeline. No clock math.

You can do this from any shell

```
pmm-admin annotate --tags=deploy "v1.2.3 rolled out"
pmm-admin annotate --tags=schema-change "added users.tenant_id index"
pmm-admin annotate --tags=incident "PXC node-2 OOM, restarted"
```

Wire it into your **CI/CD pipeline**. Every deploy, every migration, every schema change → a vertical line on every PMM dashboard.

Monday-morning take-home: post-incident reviews stop needing "what time was the deploy?" archaeology.

Group A — High latency on PXC

Edit deploy/workload/database-chaos-injector/values-env.yaml:

```yaml
target:
  namespace: workshop
  clusterName: pxc-proxysql
  type: pxc
faults:
  highLatency:
    enabled: true
    container: "pxc"
    podIndex: 0
    delayMs: 500
```

Then in the terminal:

```
cd ~/workspace
git add . && git commit -m "inject highLatency on pxc" && git push
```

Group A — What to watch in PMM

- **MySQL Overview** (PXC) → query latency **rises**
- **Node Summary** → disk IO is **unchanged**
- **QAN** → query times spike, but **no plan changes**

Answer: which view told you first?

Why it matters: the fault is at the **container** layer, not the disk. OS metrics will lie to you here. DB metrics tell the truth first.

Group B — Process kill on PS-GR

Edit the same file:

```
target:
  namespace: workshop
  clusterName: ps-gr-haproxy
  type: ps
faults:
  processKill:
    enabled: true
    container: "mysql"
    podIndex: 0
    signal: 9
    process: "mysqld"
```

```
cd ~/workspace
git add . && git commit -m "inject processKill on
ps-gr" && git push
```

Group B — What to watch in PMM

- **MySQL Overview** → **uptime resets**, topology graph reconfigures
- **Replication Summary** → primary election visible, GR member states transition
- **QAN** → gap in data for the killed primary; traffic shifts to the surviving member

Answer: which view told you first?

Why it matters: process kill is **dramatic** and **topology-first**. You see it everywhere at once.

Cleanup (both groups)

```
cd ~/workspace
bash scripts/cleanup-chaos.sh
git checkout
deploy/workload/database-chaos-injector/values-env.yaml
git add
deploy/workload/database-chaos-injector/values-env.yaml
git commit -m "revert chaos"
git push
```

Verify the fault Job is gone:

```
kubectl get jobs -n workshop \
  -l app.kubernetes.io/instance=database-chaos-injector
# (empty)
```

Lab 5 — Discussion (10 min)

- **Group A reports**: which dashboard told you first?
- **Group B reports**: which dashboard told you first?
- When does the **OS-level metric** tell you something the DB-level metric won't?
- When does the **DB-level metric** scream while the OS looks fine?
- Articulate the **order** in which PMM revealed your fault.

WRAP-UP
PERCONA

Take-aways

1. **Rootless** — PMM 3 deploys where PMM 2 couldn't
2. **QAN** — triage on frequency × duration × variance
3. **Dashboards** — signal is workload-dependent; build for *your* incidents
4. **Advisors** — fix today / next sprint / ignore is a real skill
5. **Chaos via GitOps** — the order of detection matters; OS ≠ DB

What else PMM does

- **Alerting** — Slack / PagerDuty / webhook (we touched it briefly)
- **Backup Management** — `xtrabackup` / `pg_basebackup` / PBM from the same UI; PMM 3.2 adds the **PBM Details dashboard** for MongoDB
- **PMM Dump** — forensic snapshot of metrics + QAN data; attach to support tickets or re-import into a different PMM
- **Custom Advisor checks** — write your own org-specific health checks (same authoring model as the curated set)
- **External exporters + Custom Queries** — bring any Prometheus exporter (`pmm-admin add external`) or any `SELECT` (metric series) into PMM. Application-specific KPIs without writing an exporter
- **Annotate from the shell** — `pmm-admin annotate --tags=deploy "..."` marks every dashboard at once (Lab 5 did this automatically)
- **K8s observability** — kube-state-metrics + cAdvisor pipe cluster state into PMM (technical preview)
- **PMM Health dashboard** — self-monitoring of your PMM stack (VictoriaMetrics, ClickHouse, Grafana, pmm-managed)
- **MongoDB / Valkey / Redis** — same observability model, just enable the right exporter

PERCONA®

Follow-up resources

- **PMM docs** →
  `docs.percona.com/percona-monitoring-and-management/`
- **PMM 3 blog series** → `percona.com/blog` (Q1 2026)
- **PMM Forum** →
  `forums.percona.com/c/percona-monitoring-and-management-pmm`
- **Percona Support Portal** →
  `percona.service-now.com` (for paying customers)

QUESTIONS?
PERCONA