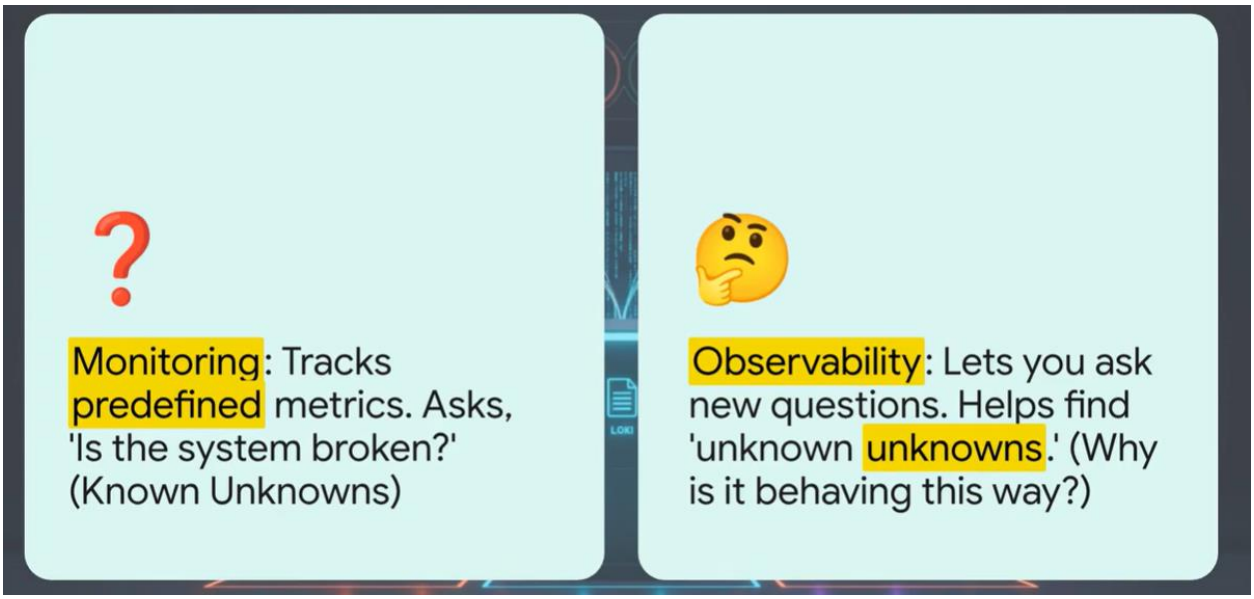
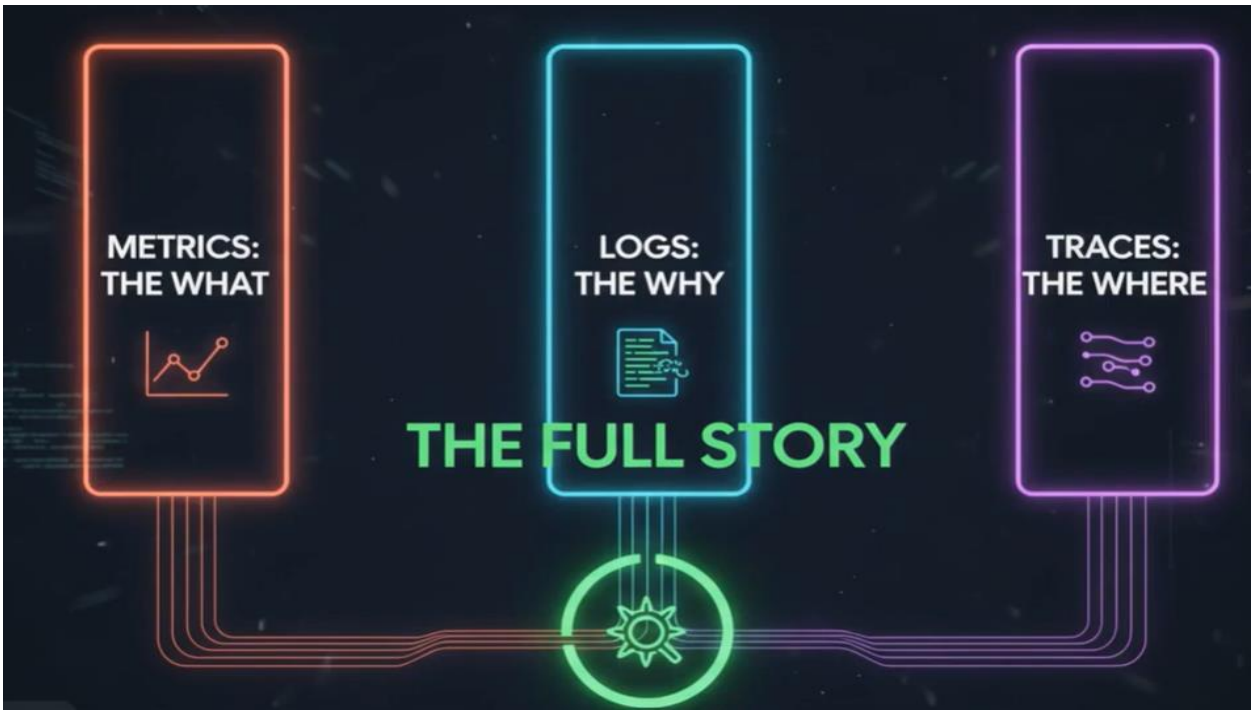
The Grafana LGTM Stack

- Loki: for logs
- Grafana: for visualization
- Tempo: for traces
- Mimir / Prometheus: for metrics





Metrics

Metrics tell you what is happening

Logs:

Logs is tell you why is happening

Traces:

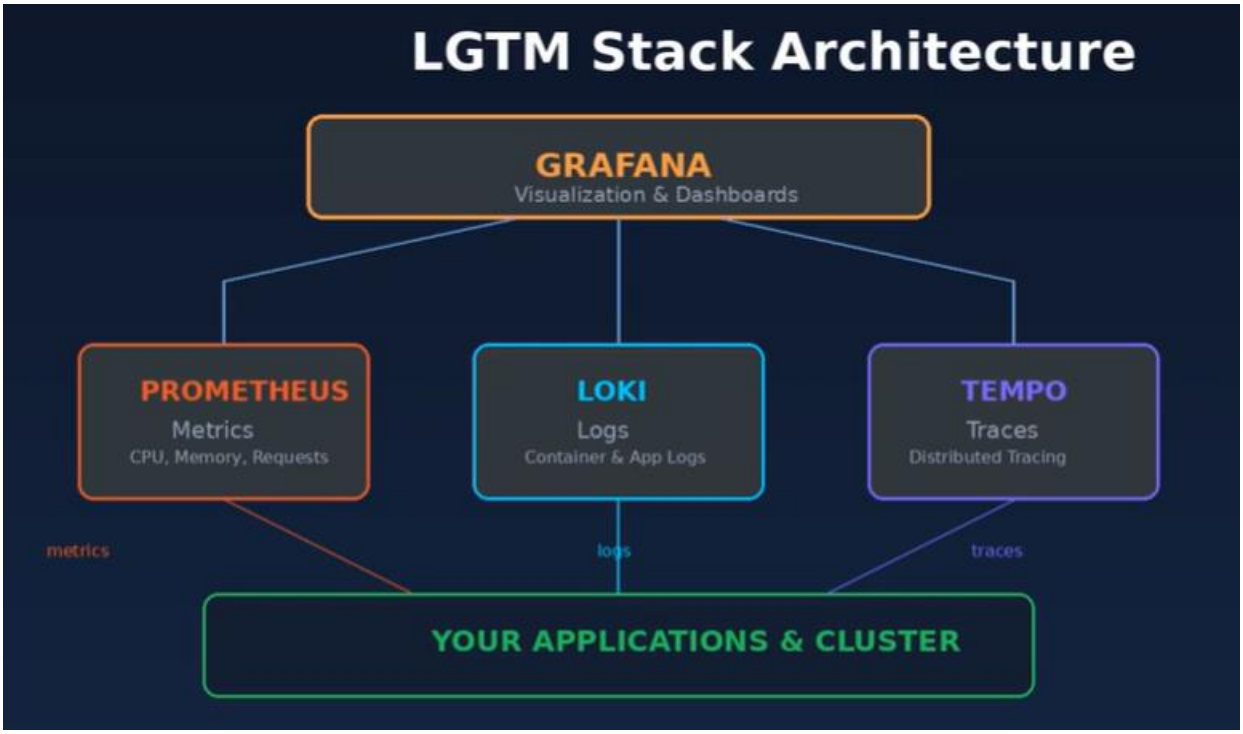
Traces tell you where is happening

Terminal - observability-lab

```
# What You'll Have in 5 Minutes:

┌──────────────────────────────┐
│  3-Node Kubernetes Cluster   │
│  ├── 1 Control Plane (the brain)  │
│  ├── 2 Worker Nodes (run your apps) │
│  ├── Helm (package manager)  │
│  └── Ingress (traffic routing)  │
└──────────────────────────────┘

# PLUS: Live nginx deployment demo!
```

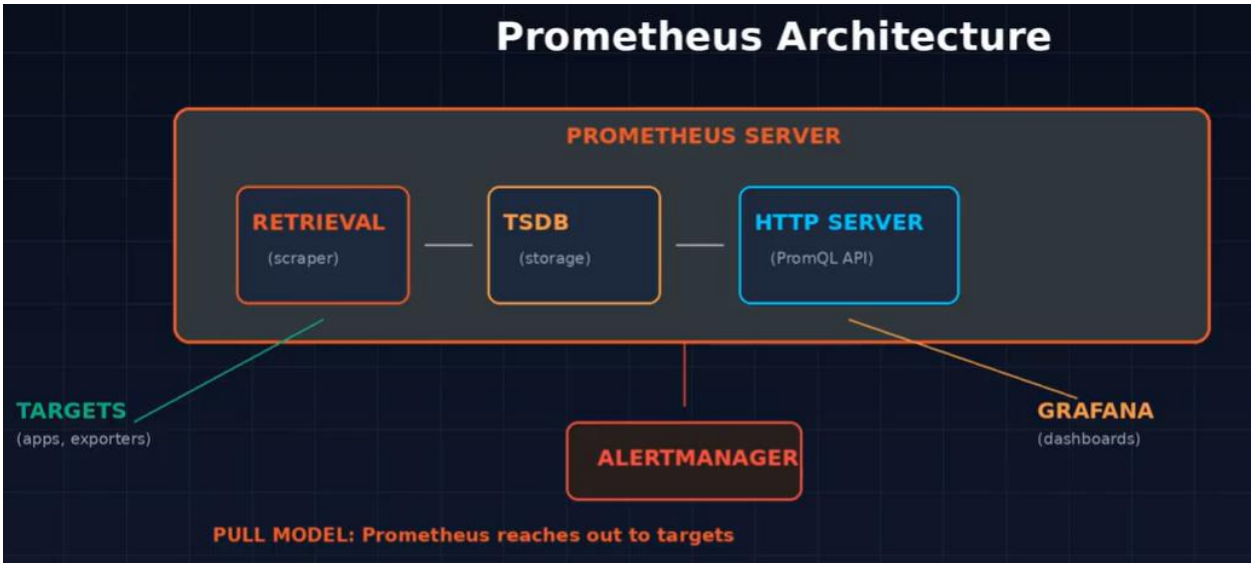


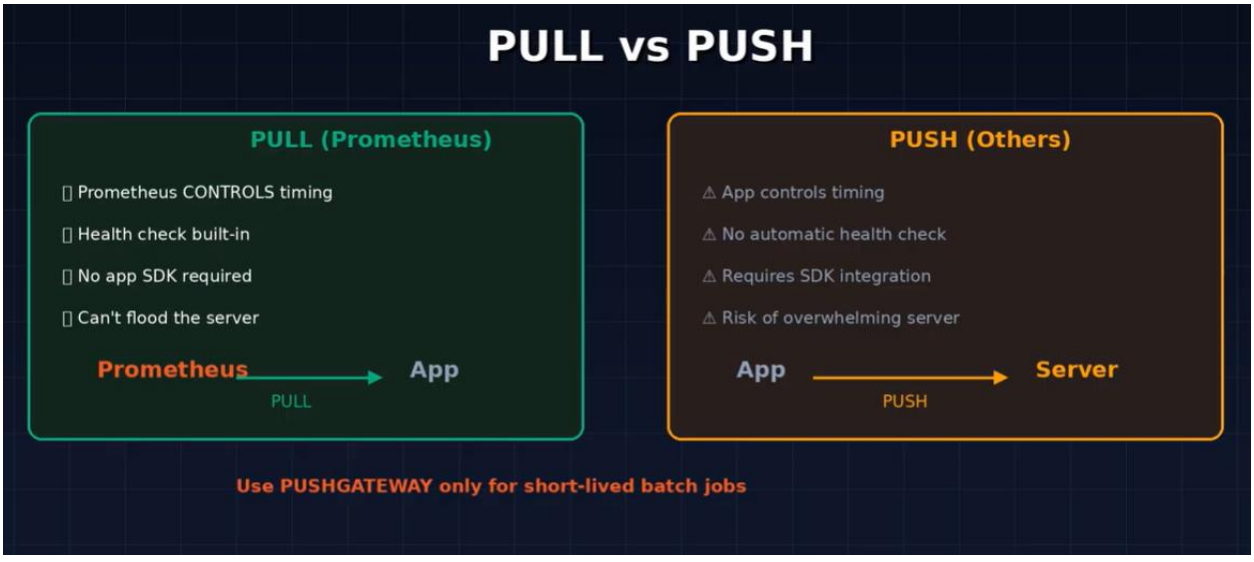
What You Built!

- ✓ **Prometheus**
  Scraping metrics every 15 seconds
- ✓ **Grafana**
  Pre-configured with all data sources
- ✓ **Loki**
  Collecting logs from every pod
- ✓ **Tempo**
  Ready for distributed traces

PROMETHEUS

How Metrics Collection REALLY Works





COUNTER

Only goes UP (or resets to zero)

Examples:

- `http_requests_total`
- `errors_total`
- `bytes_sent_total`

Use rate() - never raw value!

GAUGE

Goes UP and DOWN

Examples:

- `memory_usage_bytes`
- `cpu_percent`
- `queue_depth`

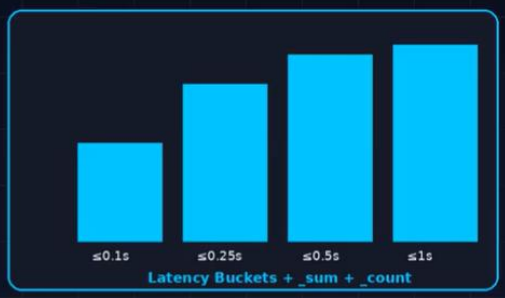
Current value is meaningful

HISTOGRAM

Counts values in BUCKETS

Examples:

- `request_duration_seconds_bucket`
- `_sum`
- `_count`



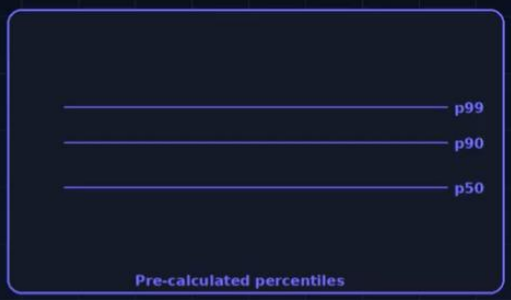
Use histogram_quantile() for percentiles

SUMMARY

Pre-calculated percentiles (client-side)

Examples:

- `gc_duration_seconds{quantile="0.99"}`



Prefer histogram for new metrics

Metric Naming Conventions

- **snake_case:** lowercase with underscores
- **Include units:** _seconds, _bytes, _total
- **Namespace prefix:** prometheus_, node_, http_

Good Examples:

```
http_requests_total
node_memory_Active_bytes
request_duration_seconds
```

Bad Examples:

```
httpRequests (wrong case)
request_time (no unit)
memory (too vague)
```

Labels = Dimensional Data

One metric name, infinite dimensions:

```
http_requests_total{method="GET", status="200"} 15234
http_requests_total{method="GET", status="500"}    23
http_requests_total{method="POST", status="201"}  890
```

Each unique label combination = separate time series

⚠ **HIGH CARDINALITY WARNING**

NEVER use as labels: user_id, request_id, trace_id, or unbounded values

1 million users = 1 million time series = dead Prometheus!

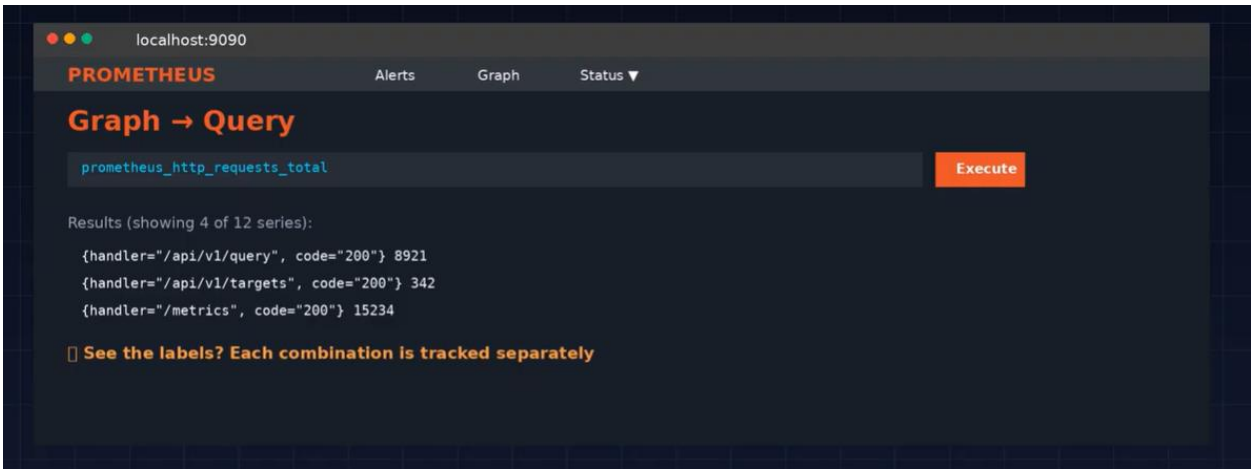
localhost:9090

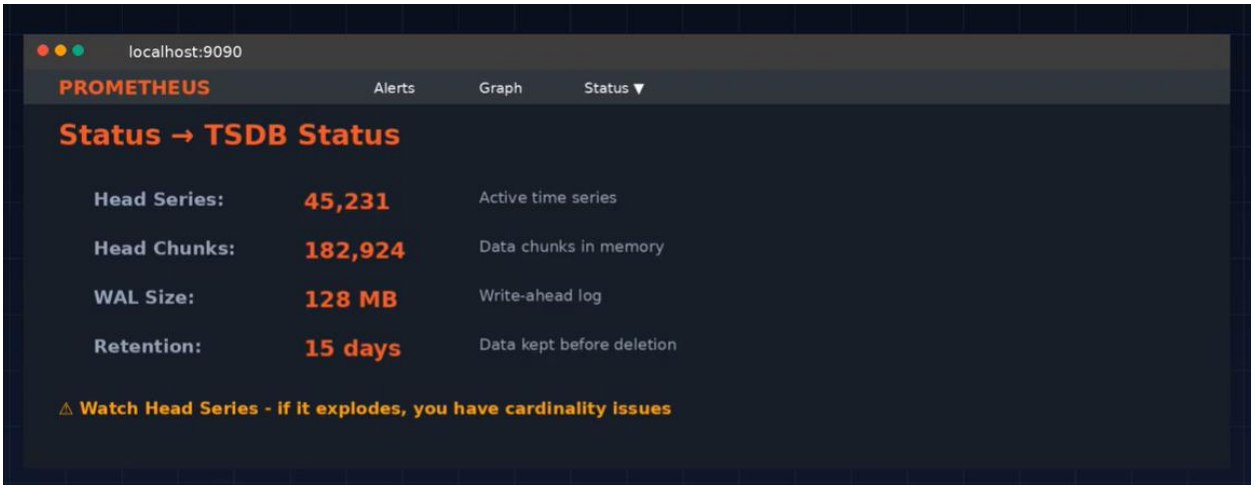
PROMETHEUS Alerts Graph Status ▼

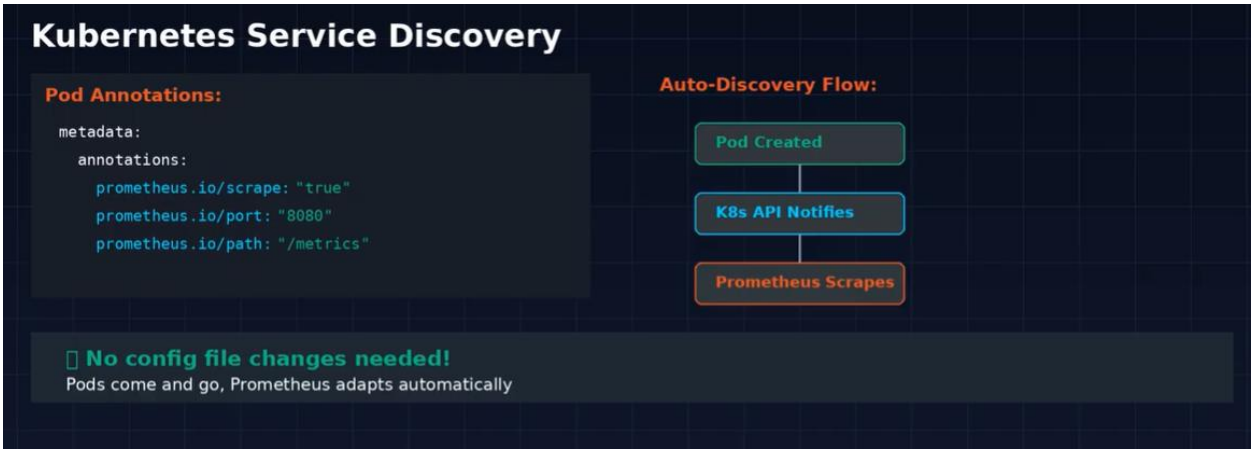
Status → Targets

Endpoint	State	Labels	Last Scrape
prometheus-server:9090/metrics	UP	job="prometheus"	2s ago
node-exporter:9100/metrics	UP	job="node"	5s ago
kube-state-metrics:8080/metrics	UP	job="kube-state"	3s ago

All targets healthy = Prometheus is collecting data


localhost:9090
PROMETHEUS
Alerts
Graph
Status ▼
Graph → Query
prometheus_http_requests_total
Execute
Results (showing 4 of 12 series):
{handler="/api/v1/query", code="200"} 8921
{handler="/api/v1/targets", code="200"} 342
{handler="/metrics", code="200"} 15234
See the labels? Each combination is tracked separately


localhost:9090
PROMETHEUS
Alerts
Graph
Status ▼
Status → TSDB Status
Head Series: 45,231 Active time series
Head Chunks: 182,924 Data chunks in memory
WAL Size: 128 MB Write-ahead log
Retention: 15 days Data kept before deletion
⚠ Watch Head Series - if it explodes, you have cardinality issues


Kubernetes Service Discovery
Pod Annotations:
metadata:
annotations:
prometheus.io/scrape: "true"
prometheus.io/port: "8080"
prometheus.io/path: "/metrics"
Auto-Discovery Flow:
Pod Created
K8s API Notifies
Prometheus Scrapes
No config file changes needed!
Pods come and go, Prometheus adapts automatically

Critical Settings

SCRAPE INTERVAL

Default: 15 seconds

Lower = more data, more storage

Recommended: 15-30 seconds

RETENTION

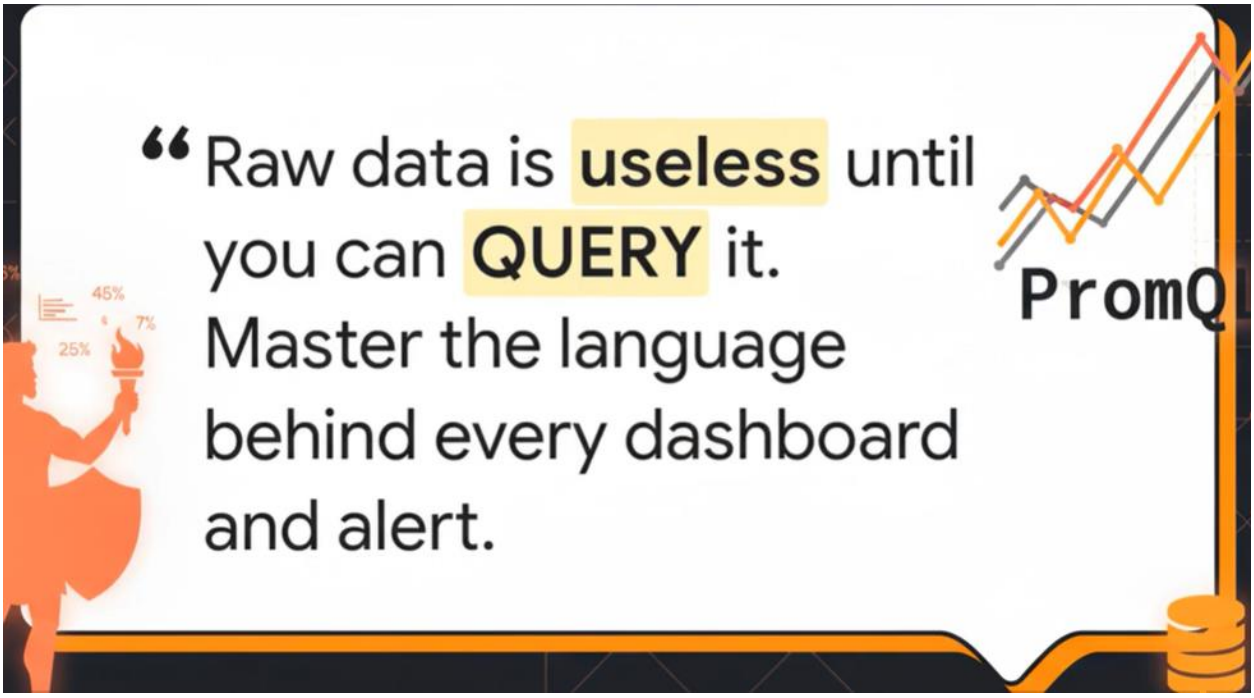
Default: 15 days

Older data deleted automatically

Long-term: Use Thanos or Mimir

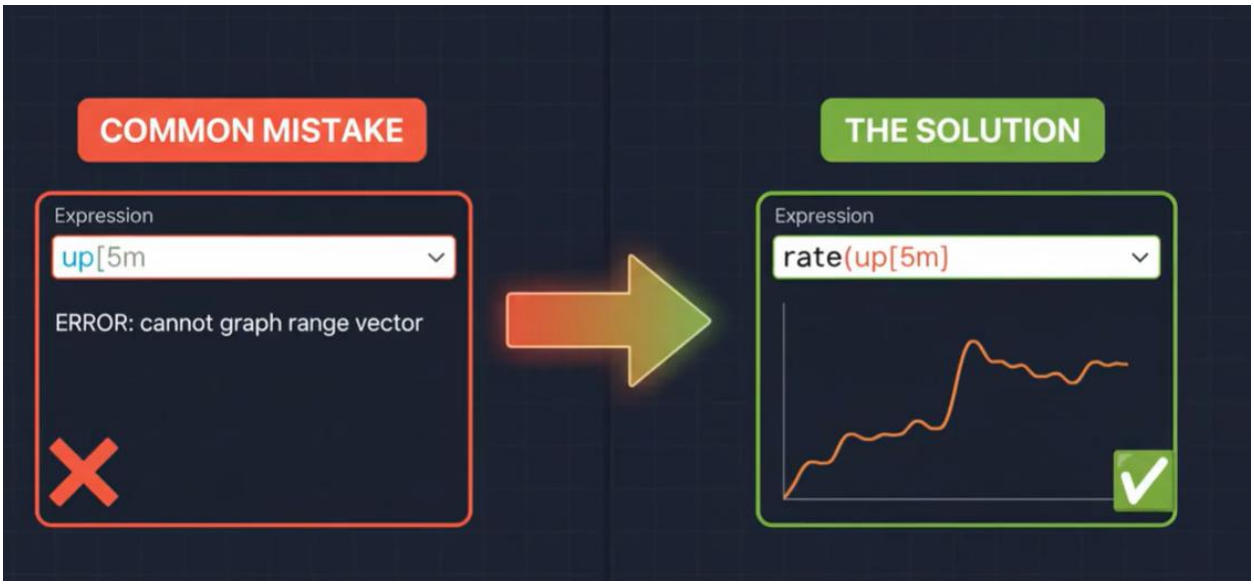
Helm values.yaml:

```
server.global.scrape_interval: 15s
server.retention: 30d
```

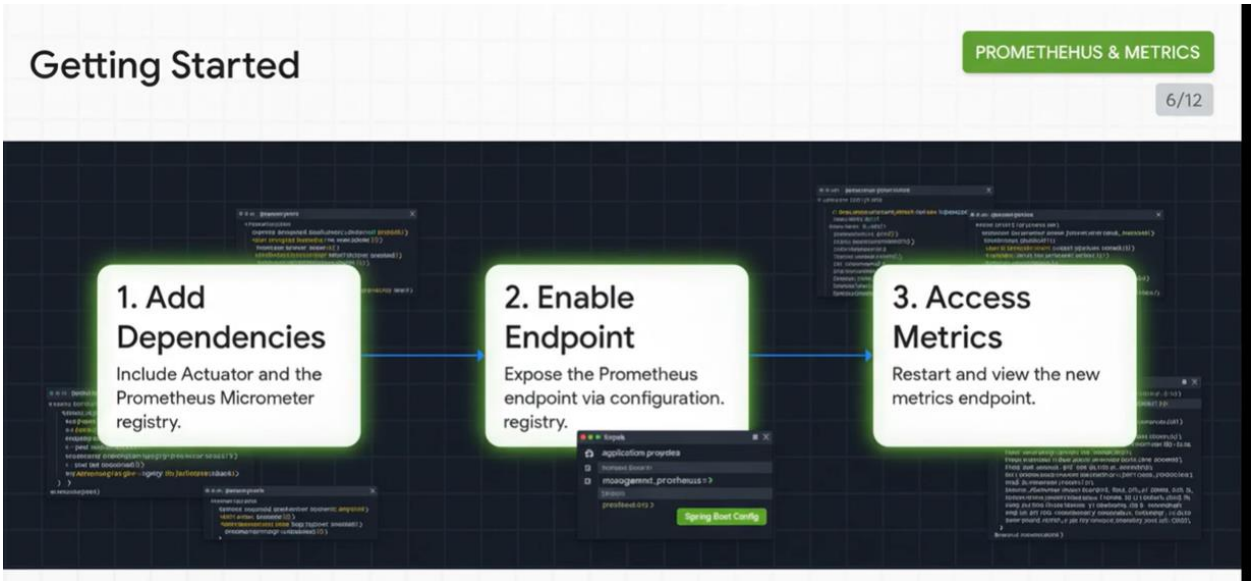


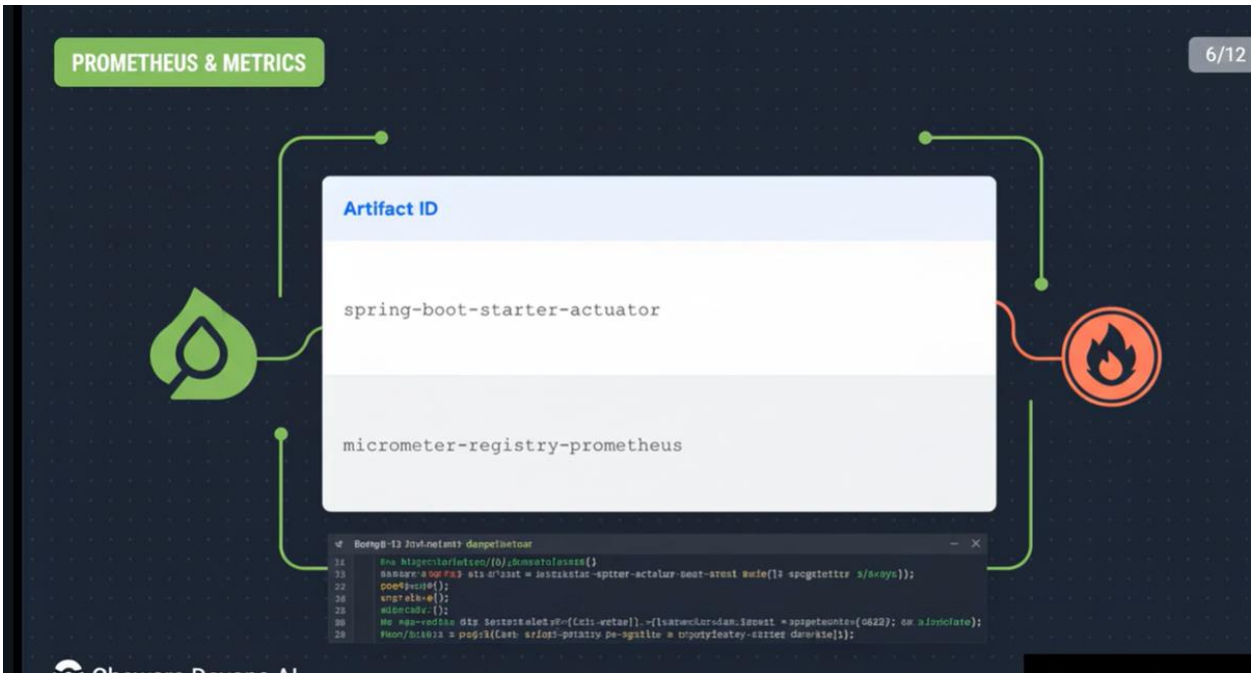
Four Ways to Filter

- =: Exact match (e.g., `job="prometheus"`)
- !=: Not equal (e.g., `job!="test"`)
- =~: Regex match (e.g., `job=~"prod.*"`)
- !~: Negative regex match (e.g., `job!~".*test.*"`)


COMMON MISTAKE
Expression
up[5m
ERROR: cannot graph range vector
THE SOLUTION
Expression
rate(up[5m])


1
Activating the Actuator
Your First Step
ServiceMontor

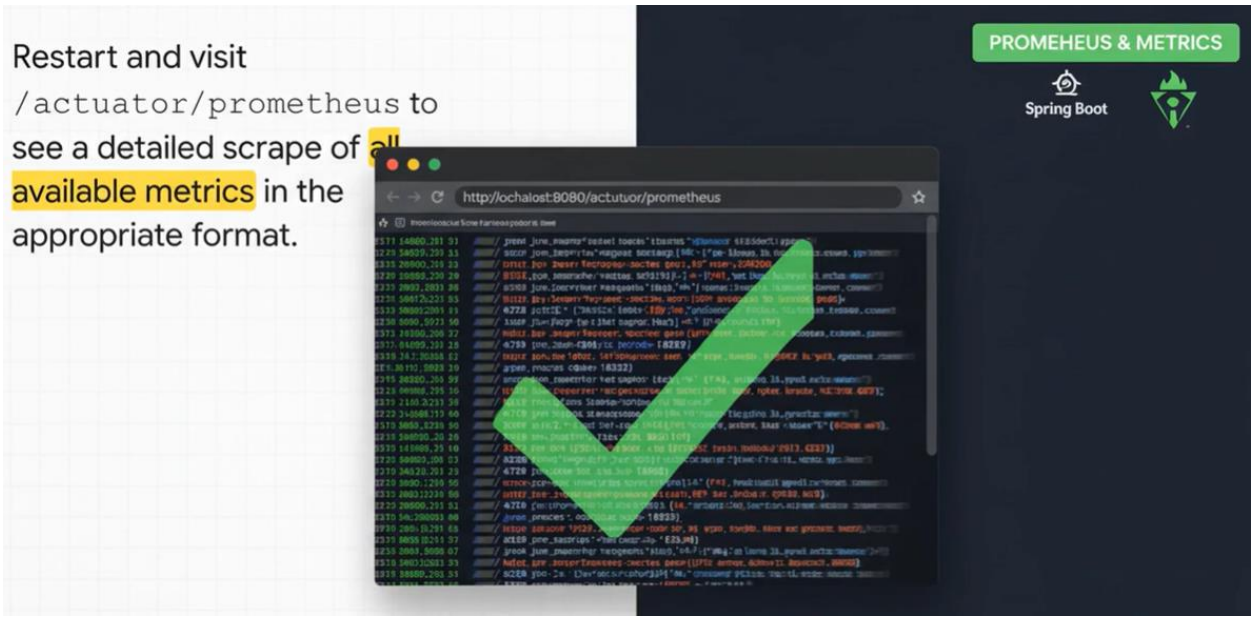

Getting Started
PROMETHEHUS & METRICS
6/12
1. Add Dependencies
Include Actuator and the Prometheus Micrometer registry.
2. Enable Endpoint
Expose the Prometheus endpoint via configuration. registry.
3. Access Metrics
Restart and view the new metrics endpoint.



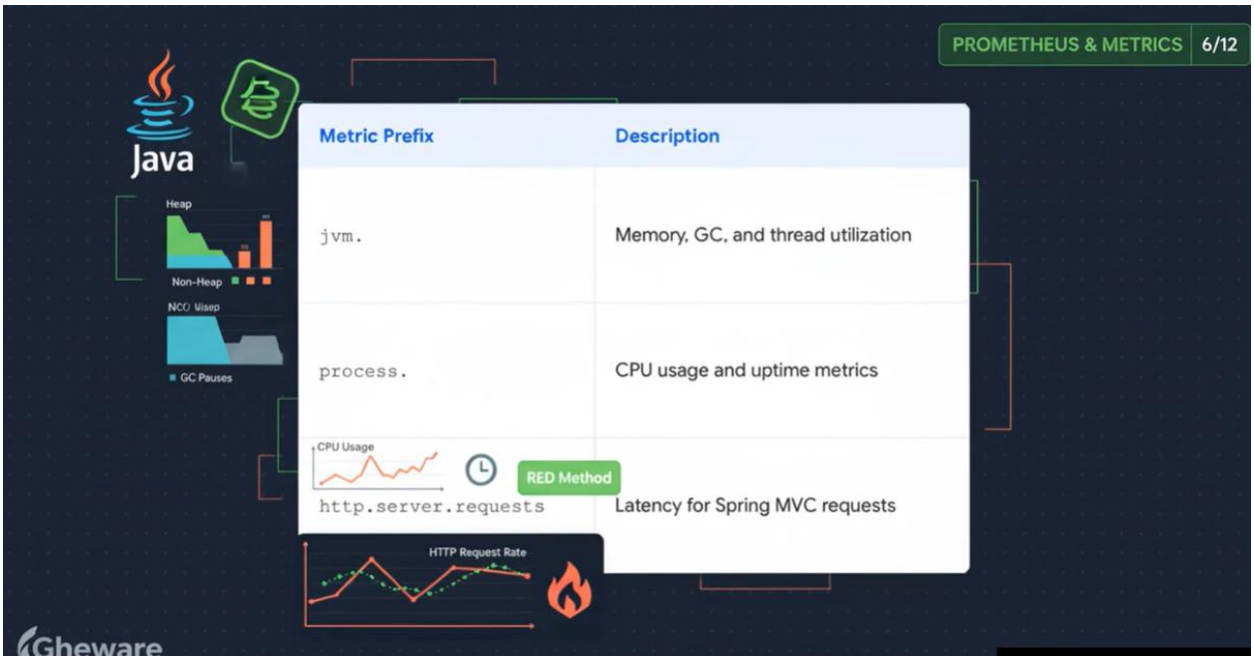
dpoints.web.exposure.include=heal

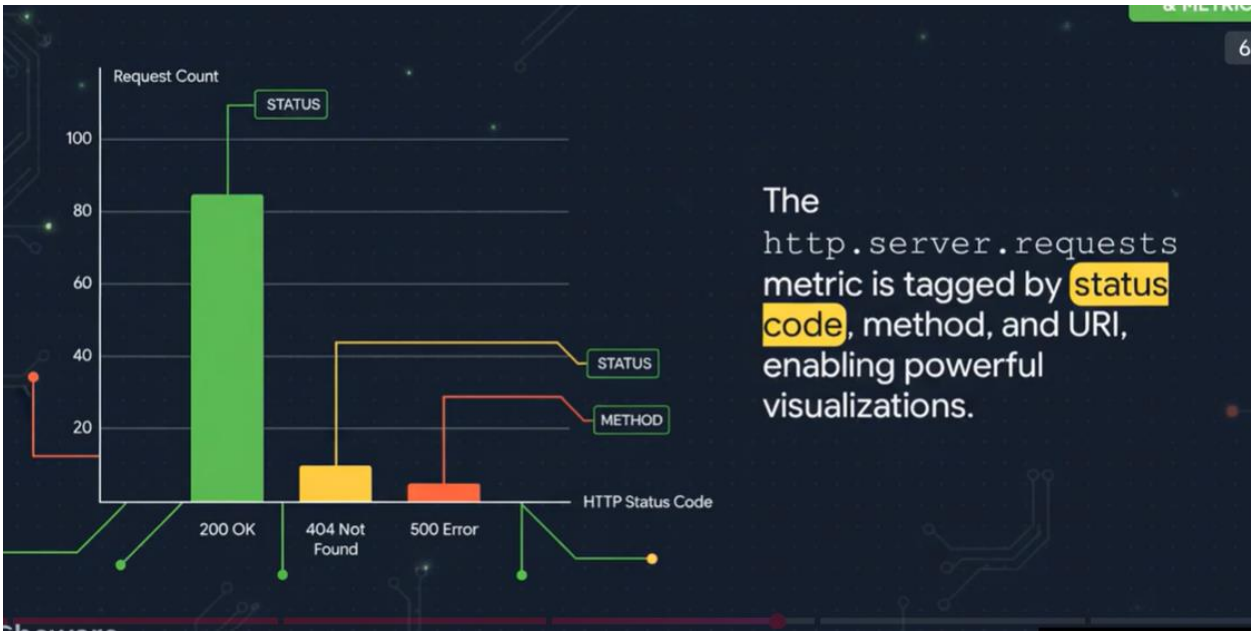
tion to expose web endpoints.

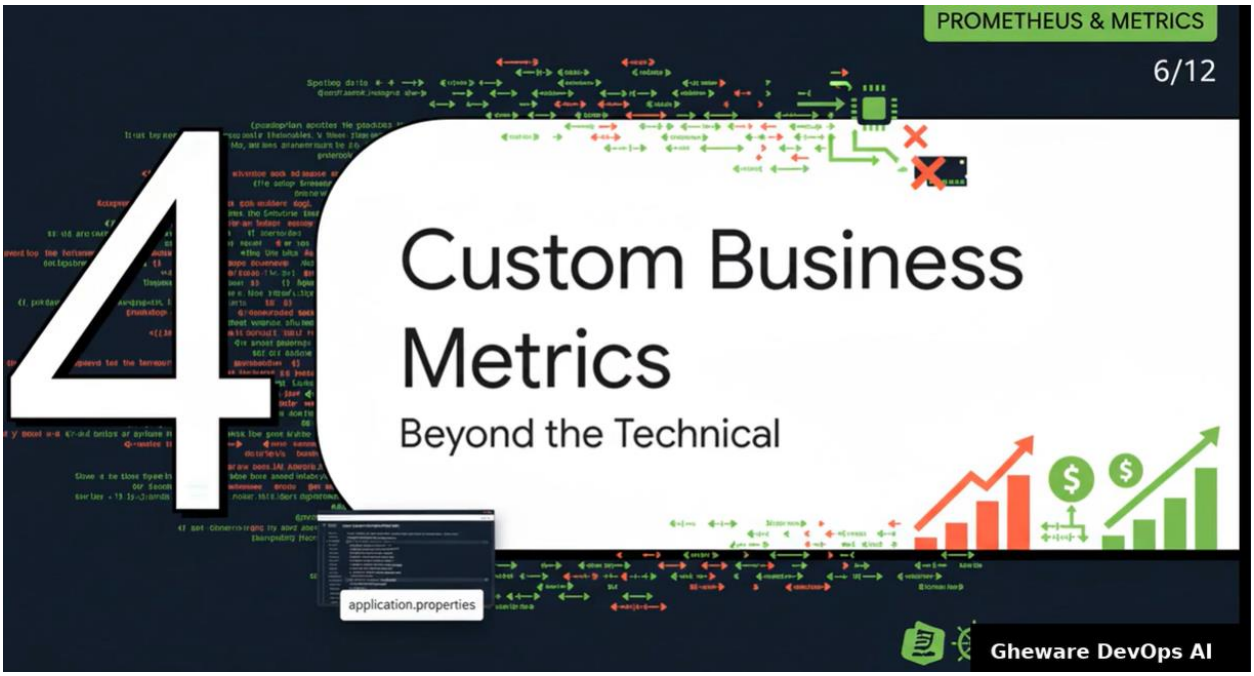
Restart and visit /actuator/prometheus to see a detailed scrape of all available metrics in the appropriate format.

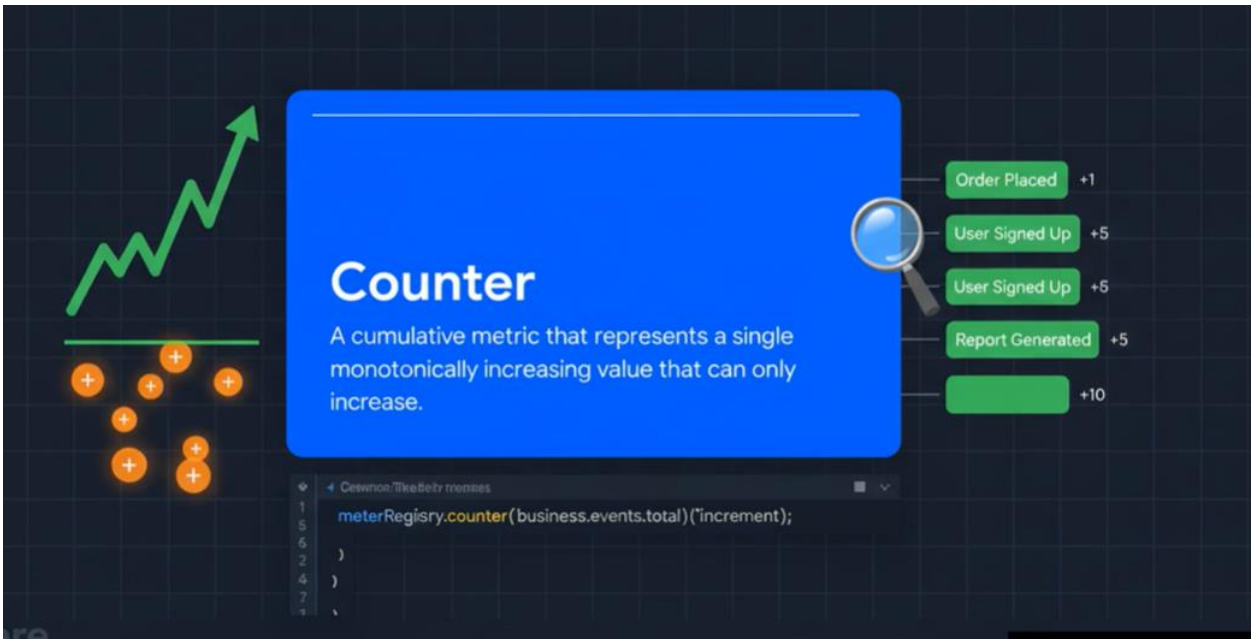



2
Out-of-the-Box Metrics
What You Get for Free

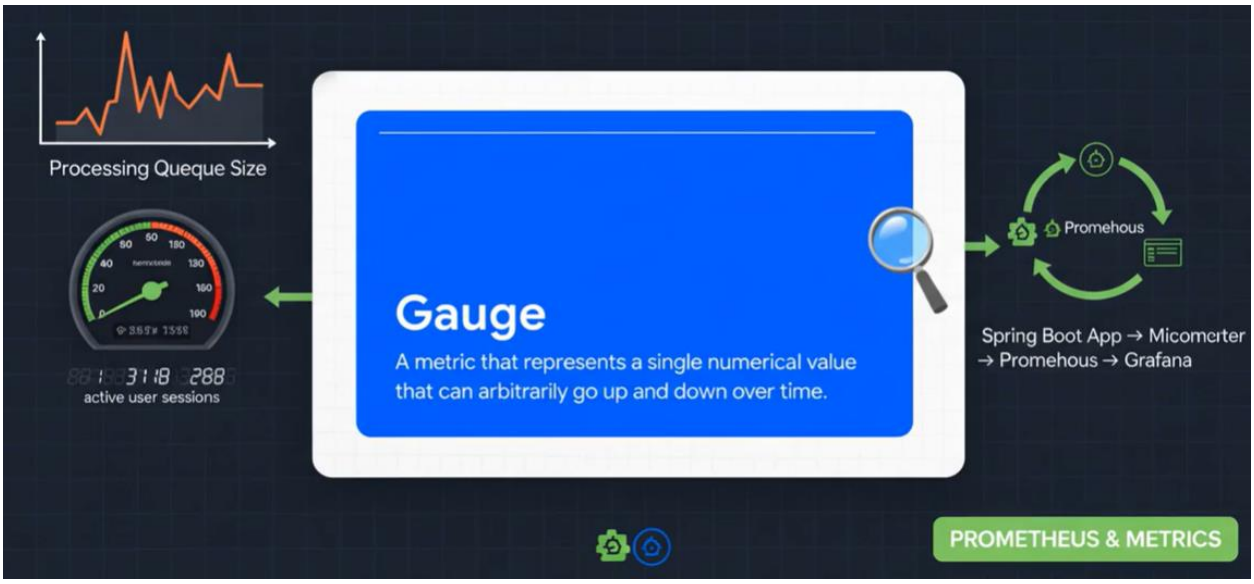

PROMETHEUS & METRICS 6/12
Java
Heap
Non-Heap
GC Pauses
Metric Prefix
Description
jvm.
Memory, GC, and thread utilization
process.
CPU usage and uptime metrics
CPU Usage
RED Method
http.server.requests
Latency for Spring MVC requests
HTTP Request Rate
Gheware


Request Count
STATUS
100
80
60
40
20
STATUS
METHOD
HTTP Status Code
200 OK
404 Not Found
500 Error
The http.server.requests metric is tagged by status code, method, and URI, enabling powerful visualizations.


PROMETHEUS & METRICS
6/12
4
Custom Business Metrics
Beyond the Technical
application.properties
Gheware DevOps AI



To register custom metrics, inject `MeterRegistry` into your component, then build and increment your counter.


Processing Queque Size
active user sessions
Gauge
A metric that represents a single numerical value that can arbitrarily go up and down over time.
Promehous
Spring Boot App → Micometer → Promehous → Grafana
PROMETHEUS & METRICS


PROMETHEUS & METRICS
6/12
5
Prometheus &
Grafana
Full Visibility
Grafana

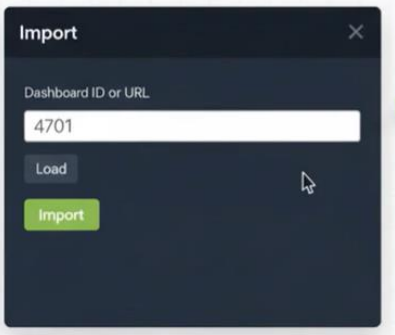
Kubernetes Discovery

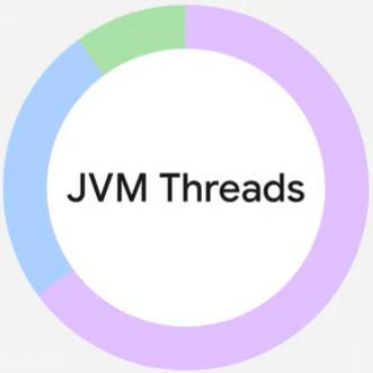
```
apiVersion: V
  aetertion: Pod
  himd
      my.spring-Boot-app
    annotations:
    annotations:
      prometheus.io/path: "/actuator/prometheus"
      prometheus.io/path: "/actuator: "8880"
      containers:
    spec: containers: ...
  ):
):
```

Auto-Discovery

Use a prebuilt dashboard for Micrometer instrumented applications to get started instantly without custom configuration.

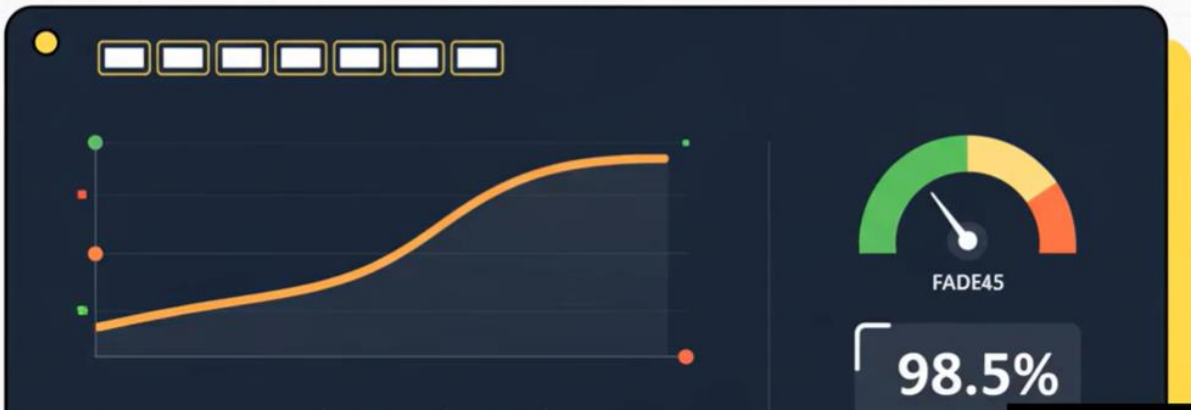
PROMEHEUS & METRICS

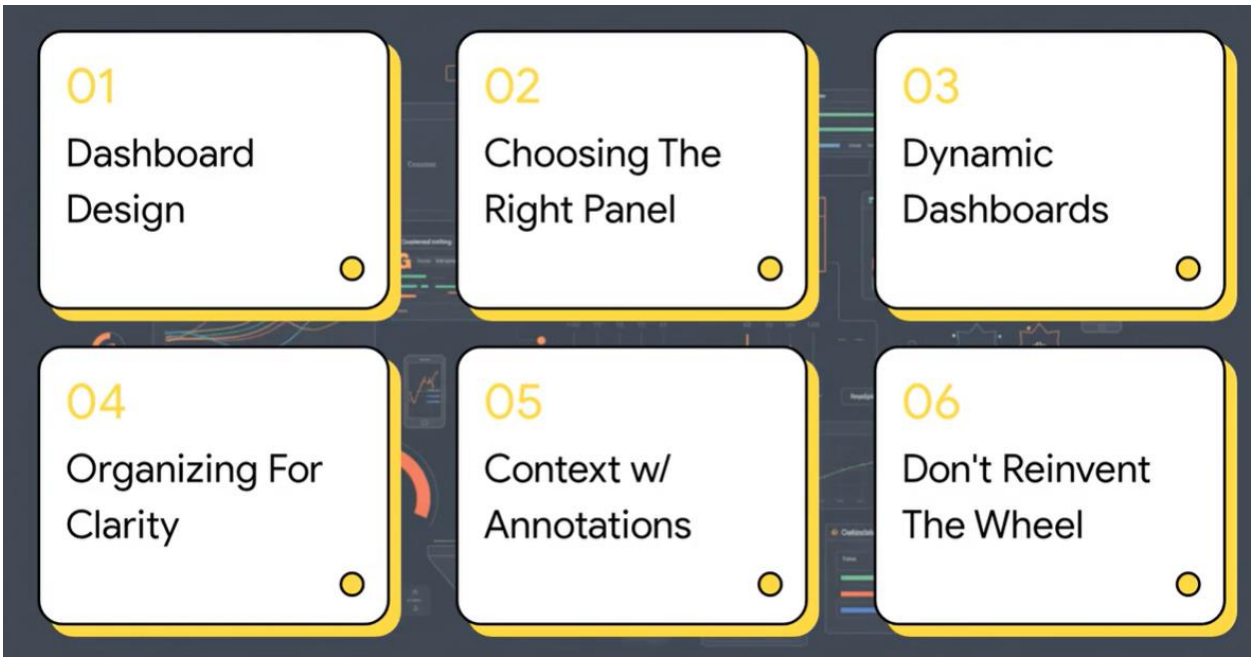


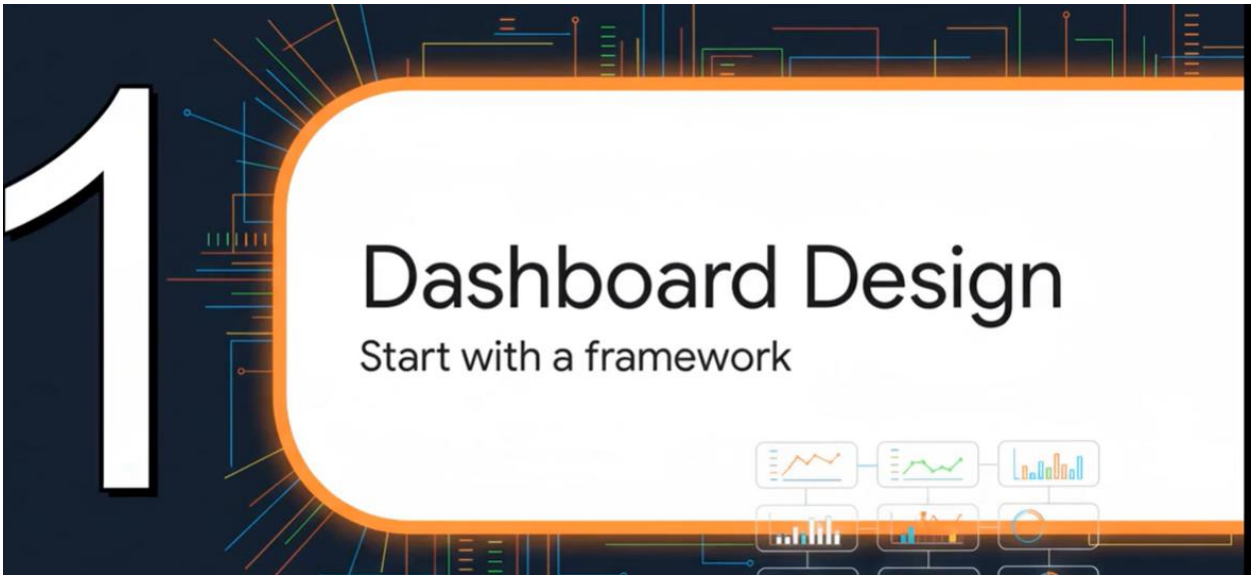


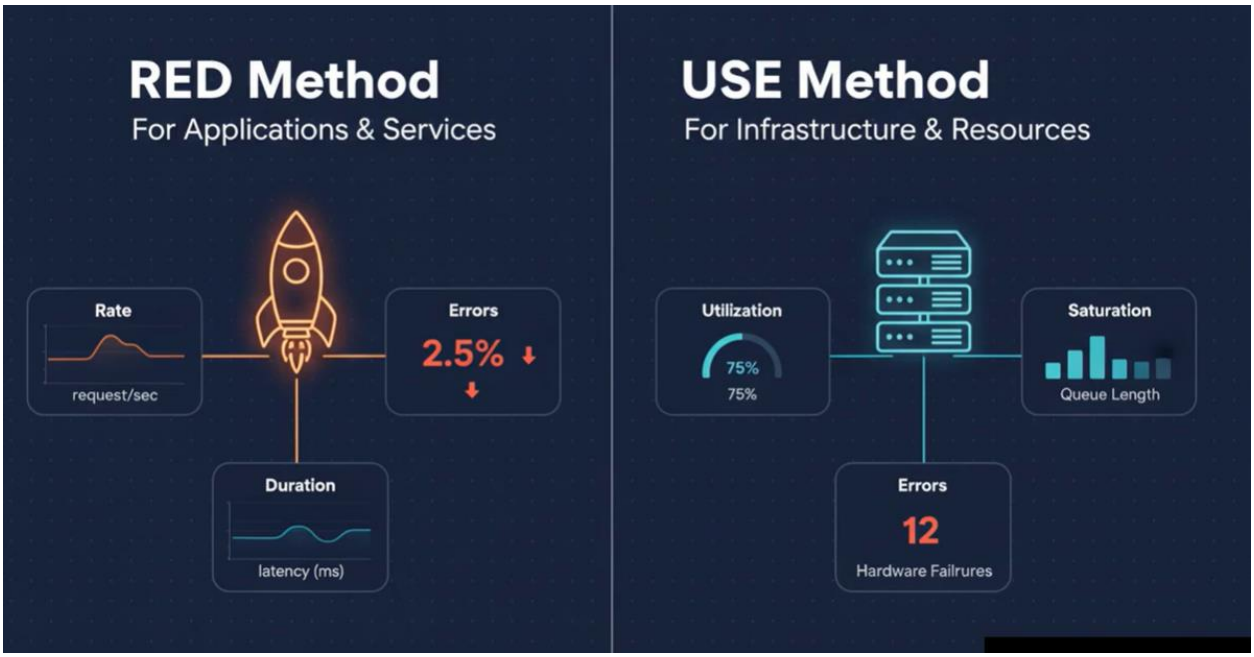
Instantly visualize key performance indicators like JVM Thread States, Garbage Collection, CPU Usage, and more.

Production Grafana Dashboards




01
Dashboard Design
02
Choosing The Right Panel
03
Dynamic Dashboards
04
Organizing For Clarity
05
Context w/ Annotations
06
Don't Reinvent The Wheel


1
Dashboard Design
Start with a framework

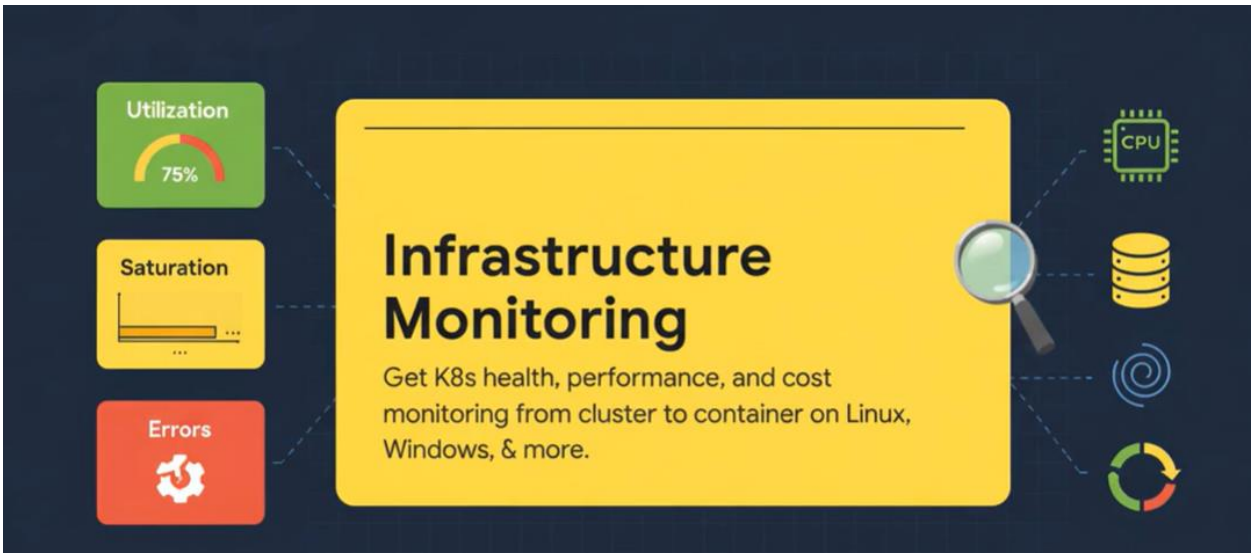


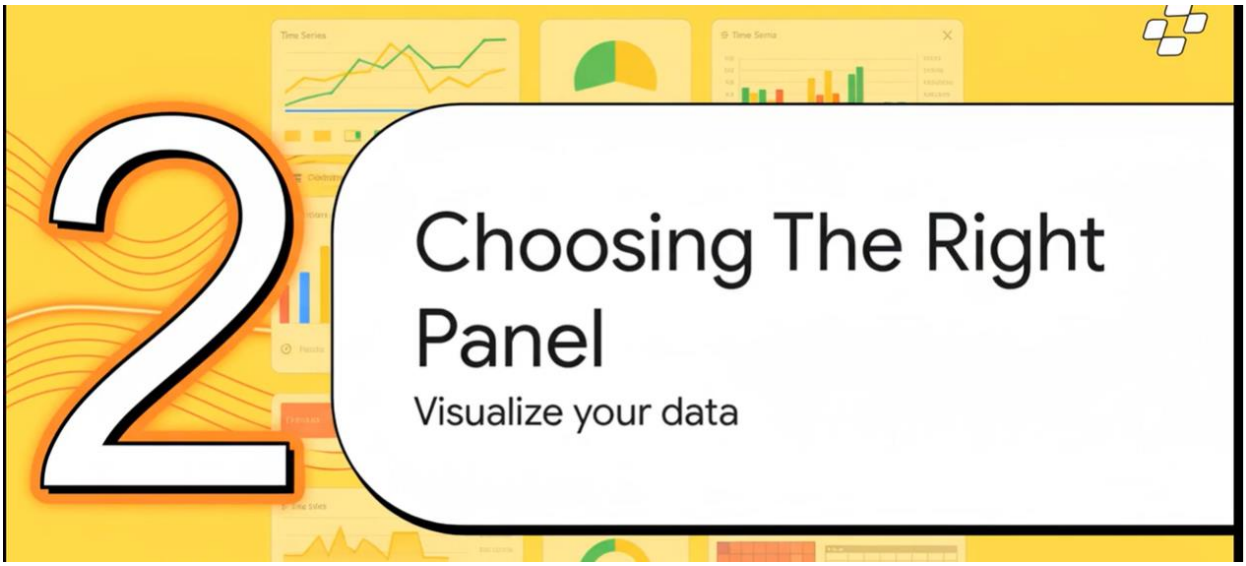
Any service we are running need to know three things

Rate: How much traffic are we getting

Errors: How much traffic is failing

Duration: How long is it taken


Utilization
75%
Saturation
Errors
Infrastructure Monitoring
Get K8s health, performance, and cost monitoring from cluster to container on Linux, Windows, & more.
CPU


2
Choosing The Right Panel
Visualize your data

Essential Panel Types

Time Series

Stat

Gauge

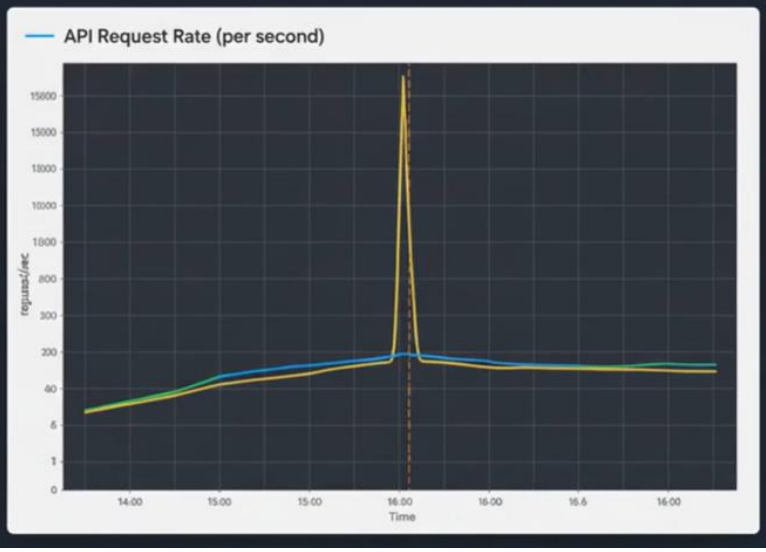
Table

Heatmap

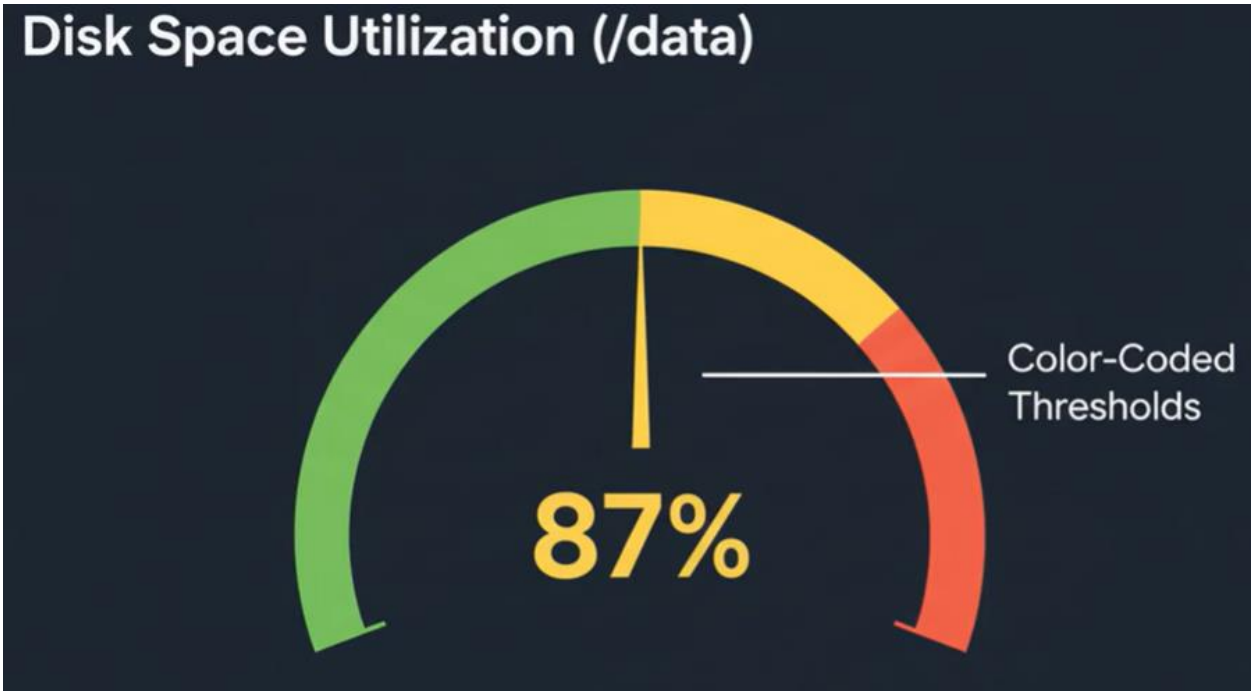
Bar Gauge

Time Series

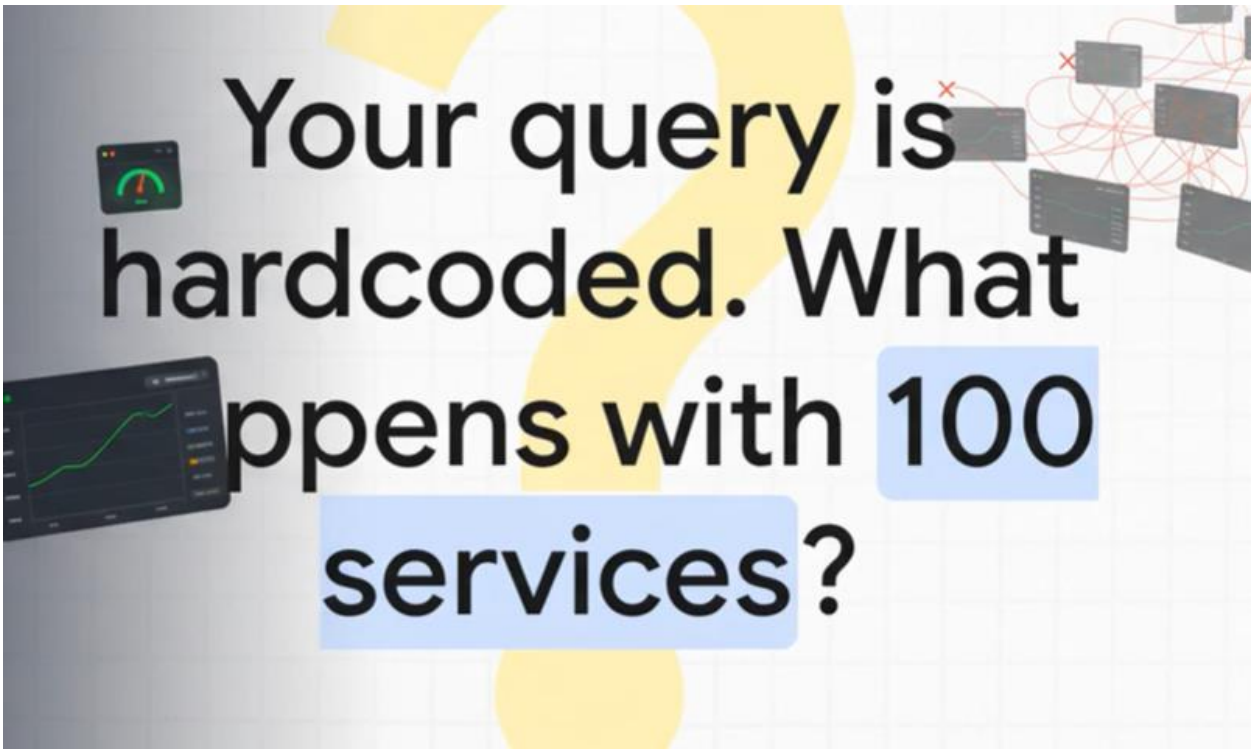
The default choice for tracking metrics over time.

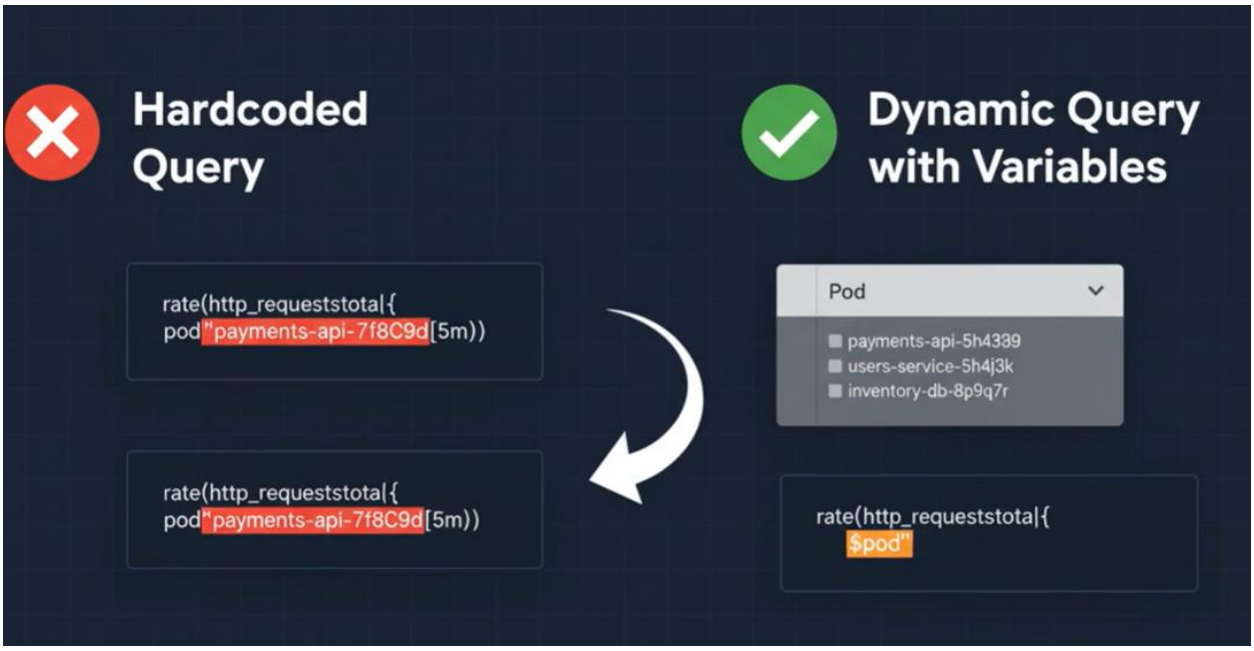


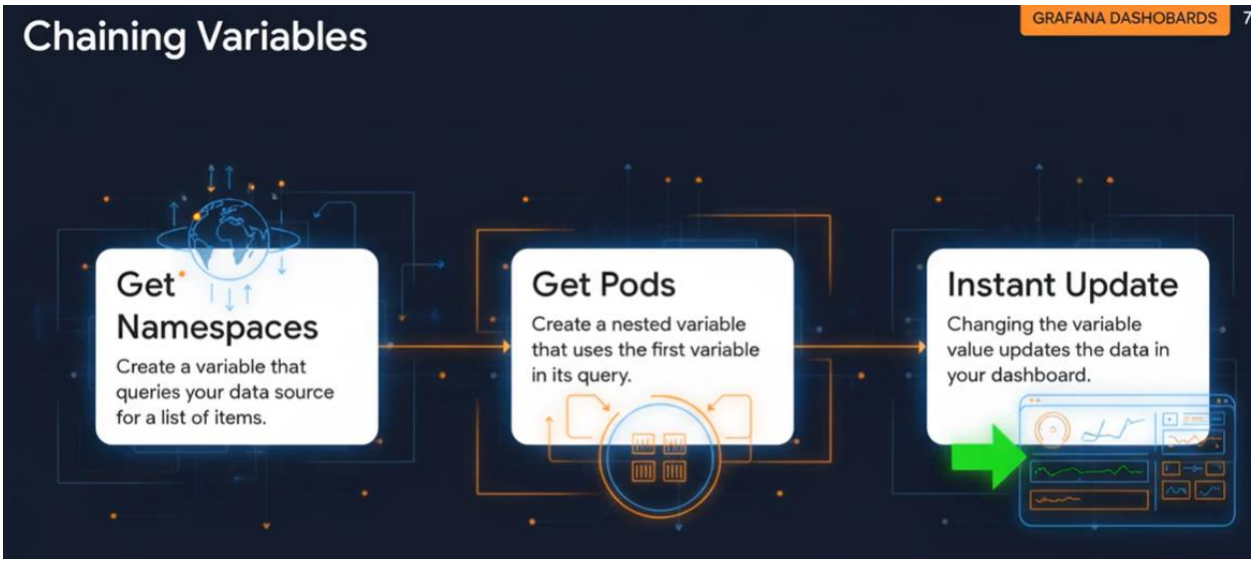

Stat Panel: At-a-glance KPIS that demand attention.
Availability SLO
99.98%
Requests (24hr)
2.45M
Error Rate
1.7%


Disk Space Utilization (/data)
87%
Color-Coded
Thresholds

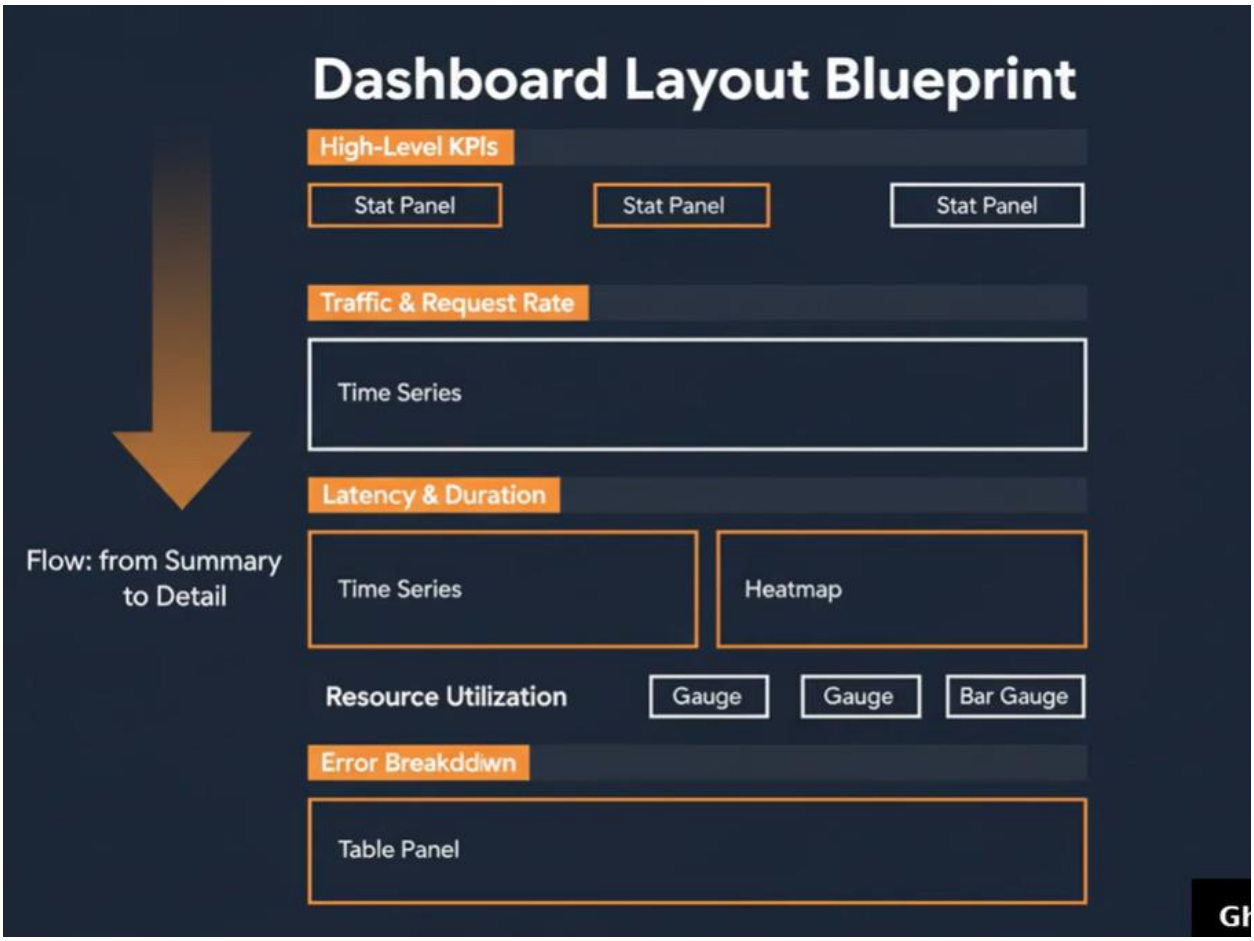
The Table panel is perfect for displaying detailed, multi-dimensional data clearly.

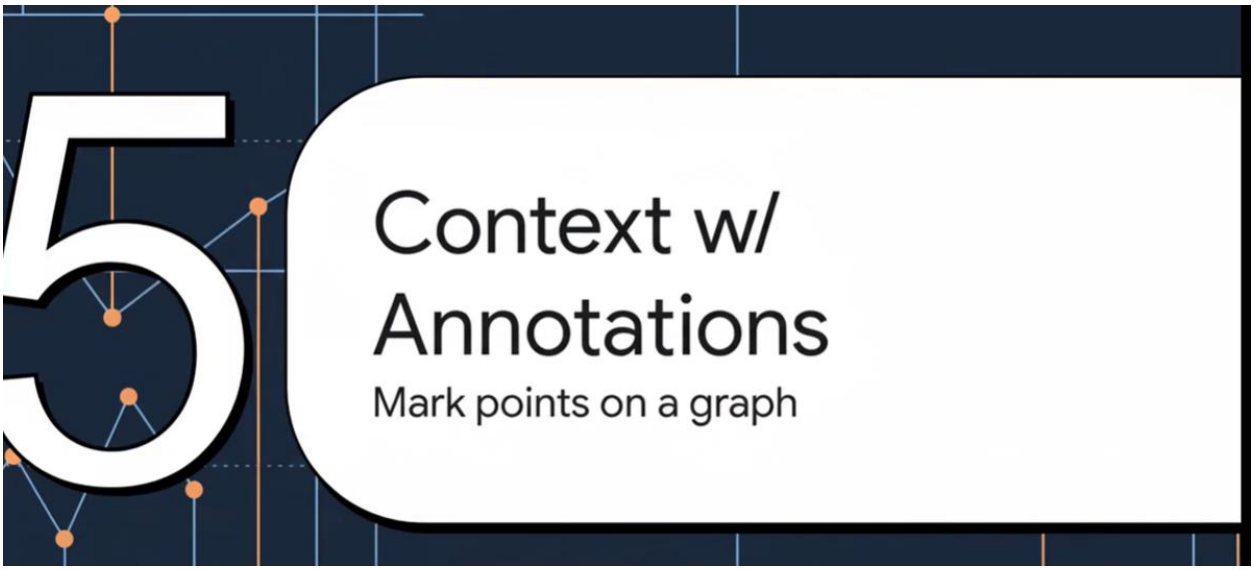

Your query is hardcoded. What ppens with 100 services?


Hardcoded Query
rate(http_requeststotal{
pod"payments-api-7f8C9d[5m))
rate(http_requeststotal{
pod"payments-api-7f8C9d[5m))
Dynamic Query with Variables
Pod
payments-api-5h4339
users-service-5h4j3k
inventory-db-8p9q7r
rate(http_requeststotal{
$pod"


Chaining Variables
GRAFANA DASHOBARDS
Get Namespaces
Create a variable that queries your data source for a list of items.
Get Pods
Create a nested variable that uses the first variable in its query.
Instant Update
Changing the variable value updates the data in your dashboard.


7/12
4
Organizing For Clarity
Structure into rows


Dashboard Layout Blueprint
High-Level KPIs
Stat Panel
Stat Panel
Stat Panel
Traffic & Request Rate
Time Series
Latency & Duration
Time Series
Heatmap
Resource Utilization
Gauge
Gauge
Bar Gauge
Error Breakddwn
Table Panel
Flow: from Summary to Detail


5
Context w/ Annotations
Mark points on a graph


GRAFANA DASHBOARDS
7/12
Event
An event occurs, like a new deployment or an alert firing.
Annotation
An annotation is created via the UI, API, or from a data source query.
Correlation
The event marker appears on graphs, providing context to data changes.
Depaloyment v1.2
Promethous Alert
FADE2
Promethous Alert


6
Don't Reinvent The Wheel
Use community dashboards
1860
9998


12
Name
Use Case
Kubernetes Monitoring
K8s health & performance
MongoDB Data
Visualize MongoDB data
Jira Data
Visualize Jira data
InfluxDB Dashboards
Monitor telegraf metrics

You can visualize data...
but how do you get alerted?

Alerting: Prometheus & Grafana


ALERTING
8/12
SYSTEM HEALTH: OK
Your dashboards are green,
but are you watching 24/7?
Let's build alerts that wake you
only when it truly matters


ALERTING
8/12
1
The Alerting Paradox
Fatigue vs. Action
CRITICAL
CRITICAL
TOO MUCH NOISE
FIRING
slack
JUST
Gheware
Devops AI
Gheware DevOps AI

Too Many Alerts

Too Many Alerts: Leads to alert fatigue, ignored notifications, and missed critical incidents.

Too Few Alerts

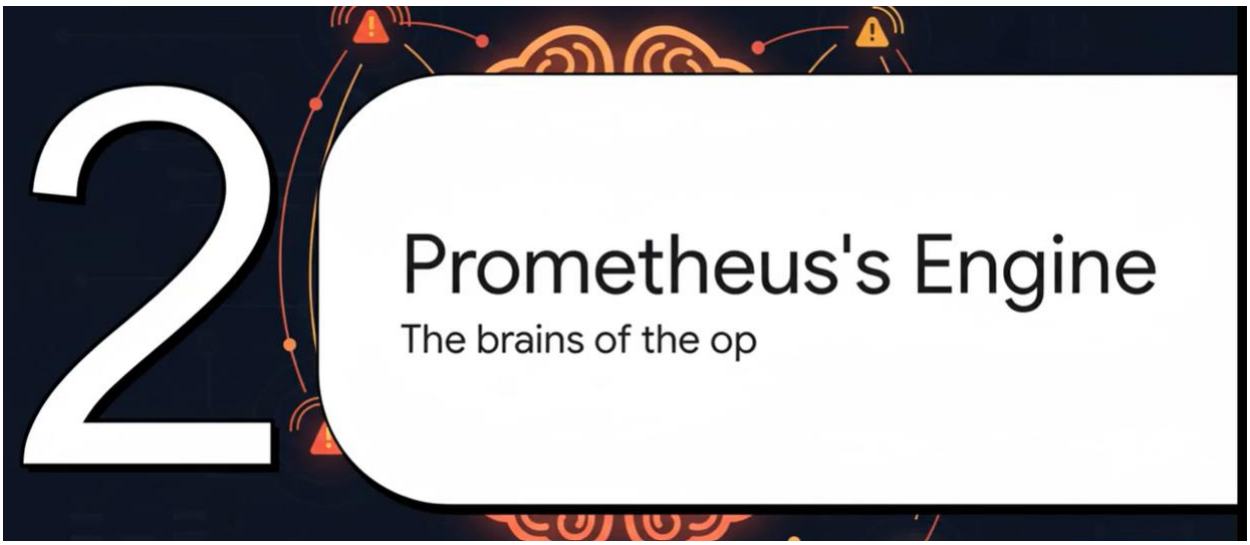
ALERTING

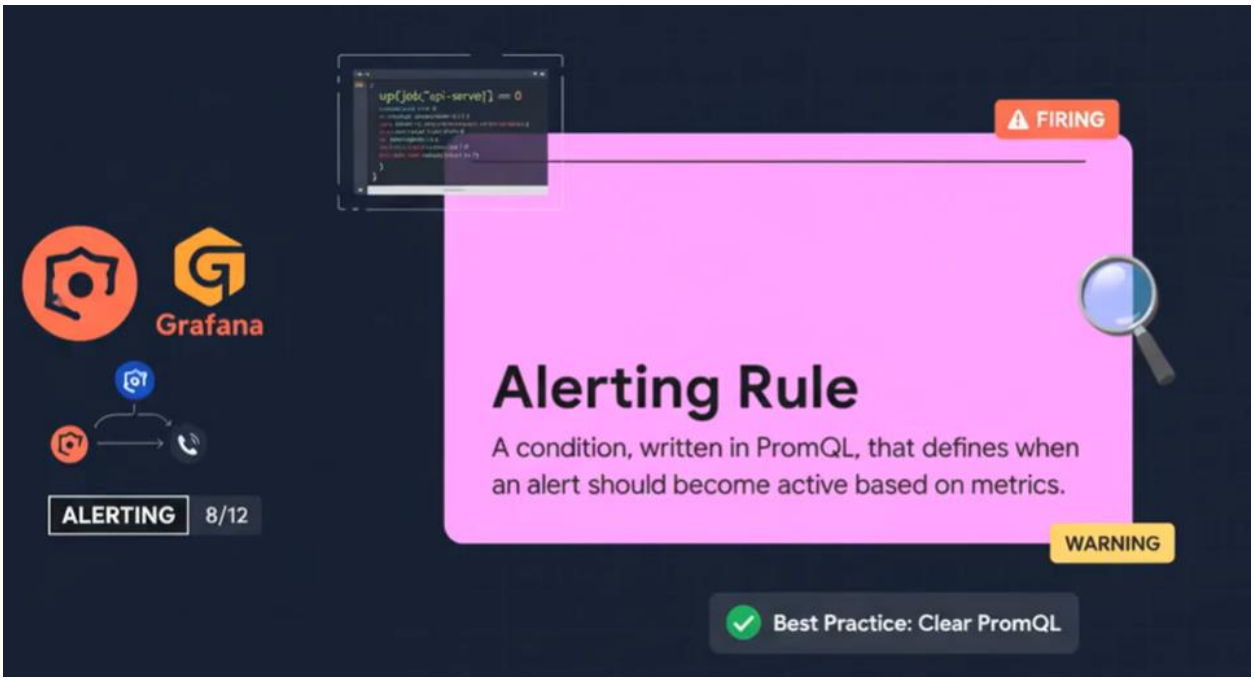
8/12

“Every page should require human intelligence to resolve.

Gheware


ALERTING
8/12
The Three Golden Rules of Alerting
1. Alert on SYMPTOMS
High Latency
CPU Usage High
2. Every alert must be ACTIONABLE
CRITICAL: Service X Down
Down
View Runbook
CPU is high
3. Alerts must be URGENT
FIRING
Service X Down
CPU is high


2
Prometheus's Engine
The brains of the op



Anatomy of an Alert Rule

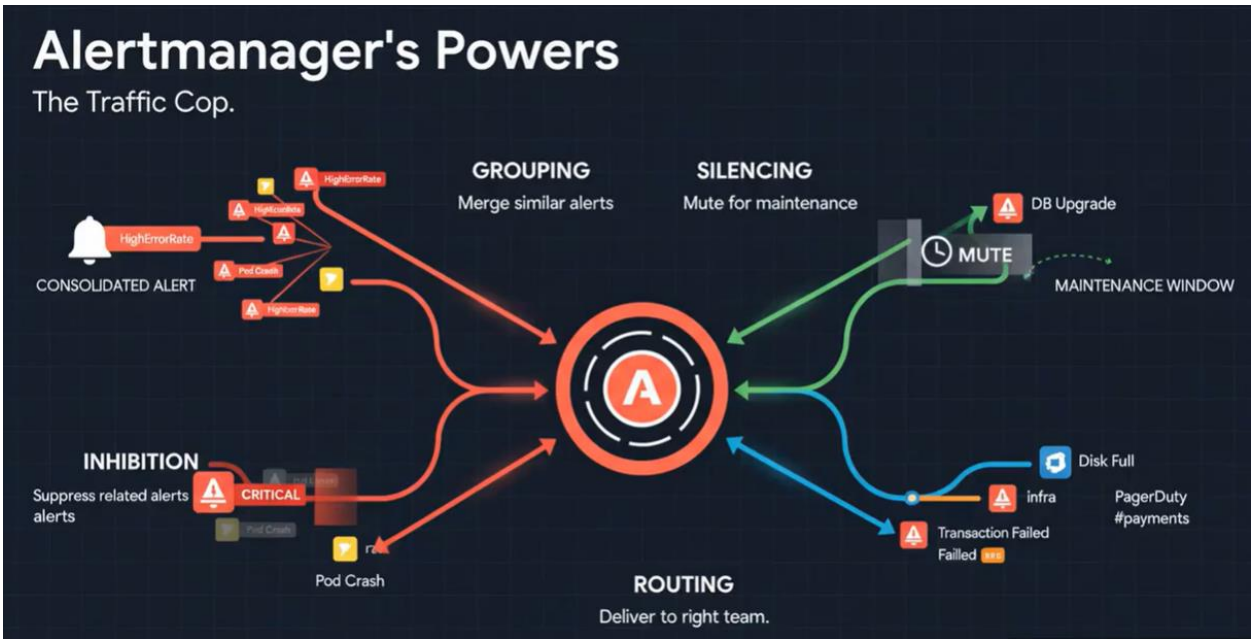
- expr: The PromQL query defining the failure condition.
- for: Duration to prevent alerts from brief, flapping spikes.
- labels: Metadata for routing and severity (e.g., `severity: page`).
- annotations: Human-readable info, like a summary or runbook link.

Alerting Flow

1. Evaluate Rules
Prometheus evaluates alerting rules against metrics.

2. Fire Alerts
Active alerts are sent to Alertmanager.

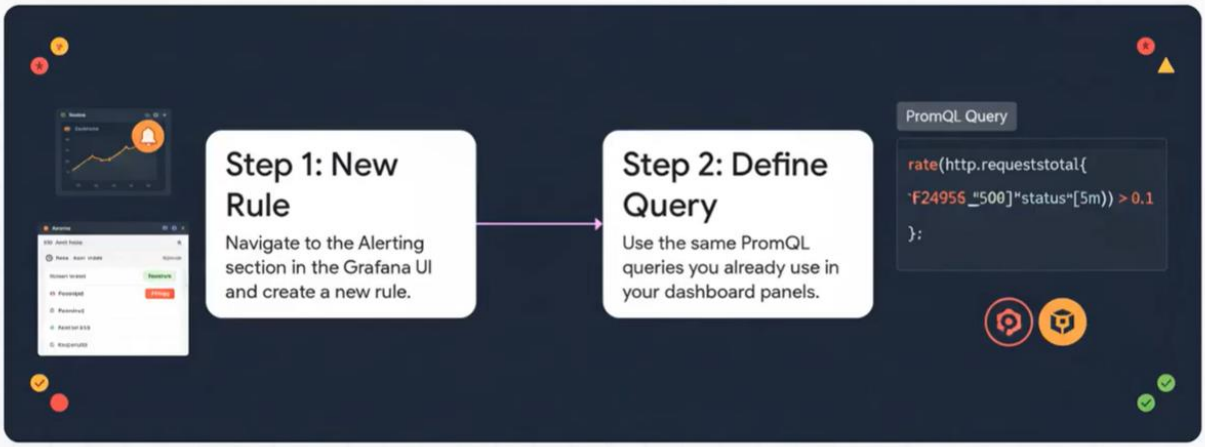
3. Route Notifications
Alertmanager processes and routes the alerts.

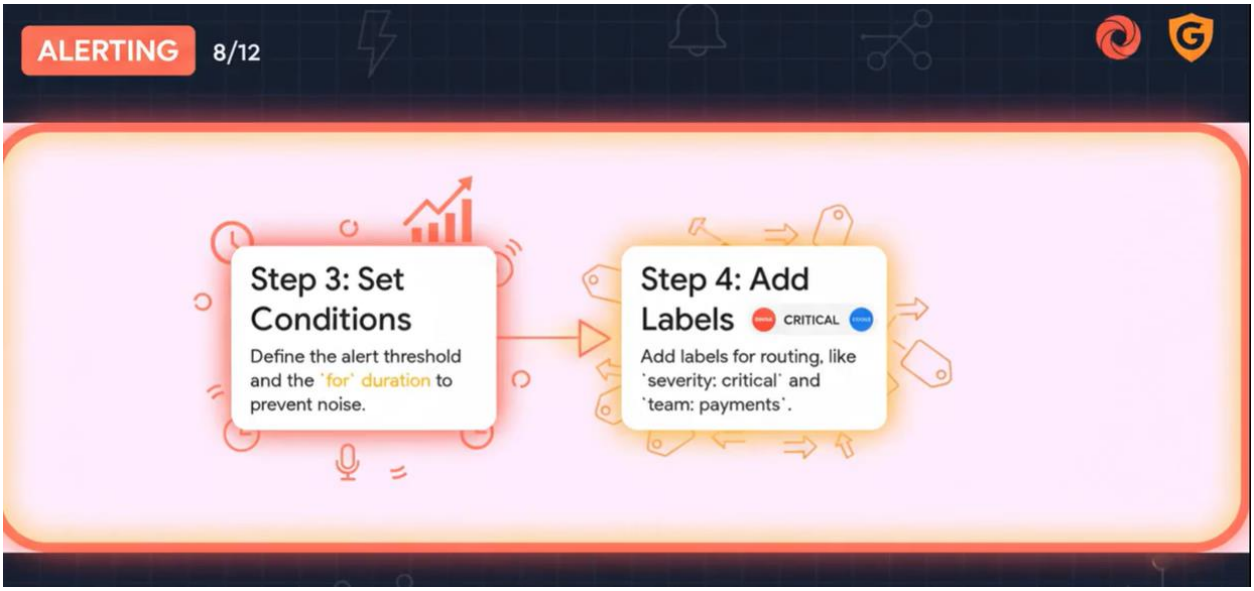

Alertmanager's Powers
The Traffic Cop.
GROUPING
Merge similar alerts
SILENCING
Mute for maintenance
HighErrorRate
CONSOLIDATED ALERT
DB Upgrade
MUTE
MAINTENANCE WINDOW
INHIBITION
Suppress related alerts
CRITICAL
Disk Full
infra
PagerDuty
#payments
Transaction Failed
Pod Crash
ROUTING
Deliver to right team.

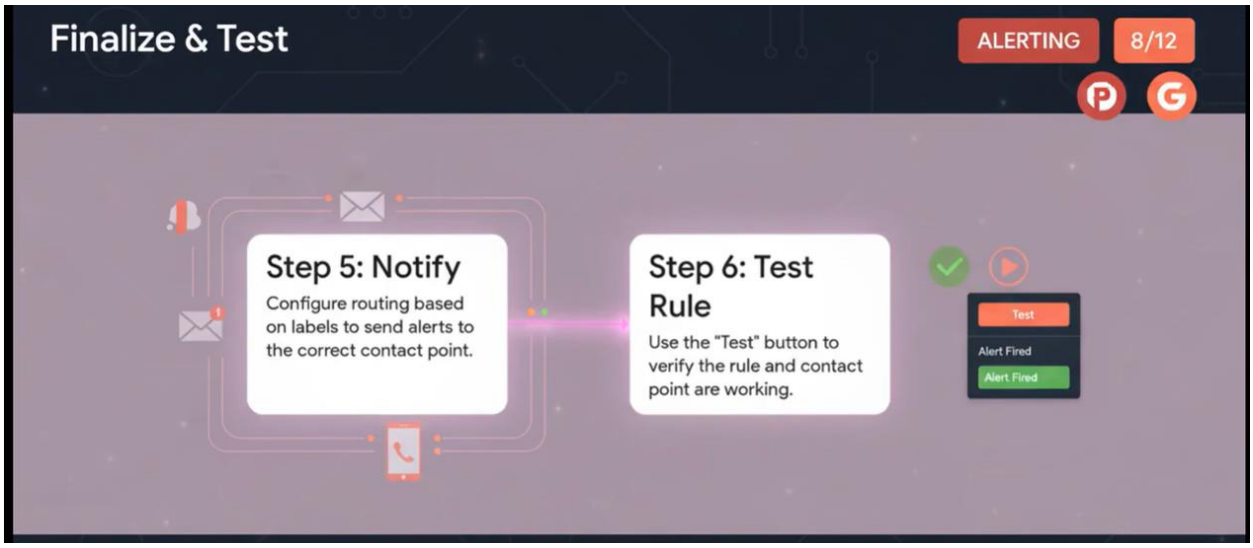

CRITICAL ALERT
HighErrorRate
Grafana: Alerting UI
A Hands-On Tutorial

Create Alert in Grafana

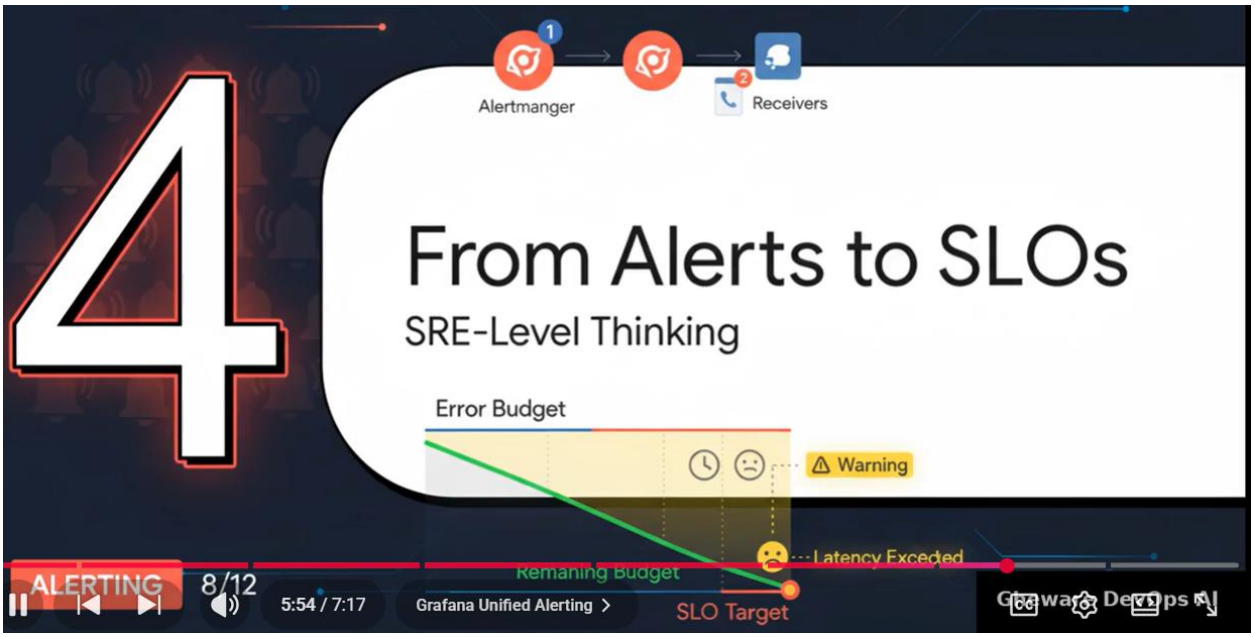
ALERTING 8/12

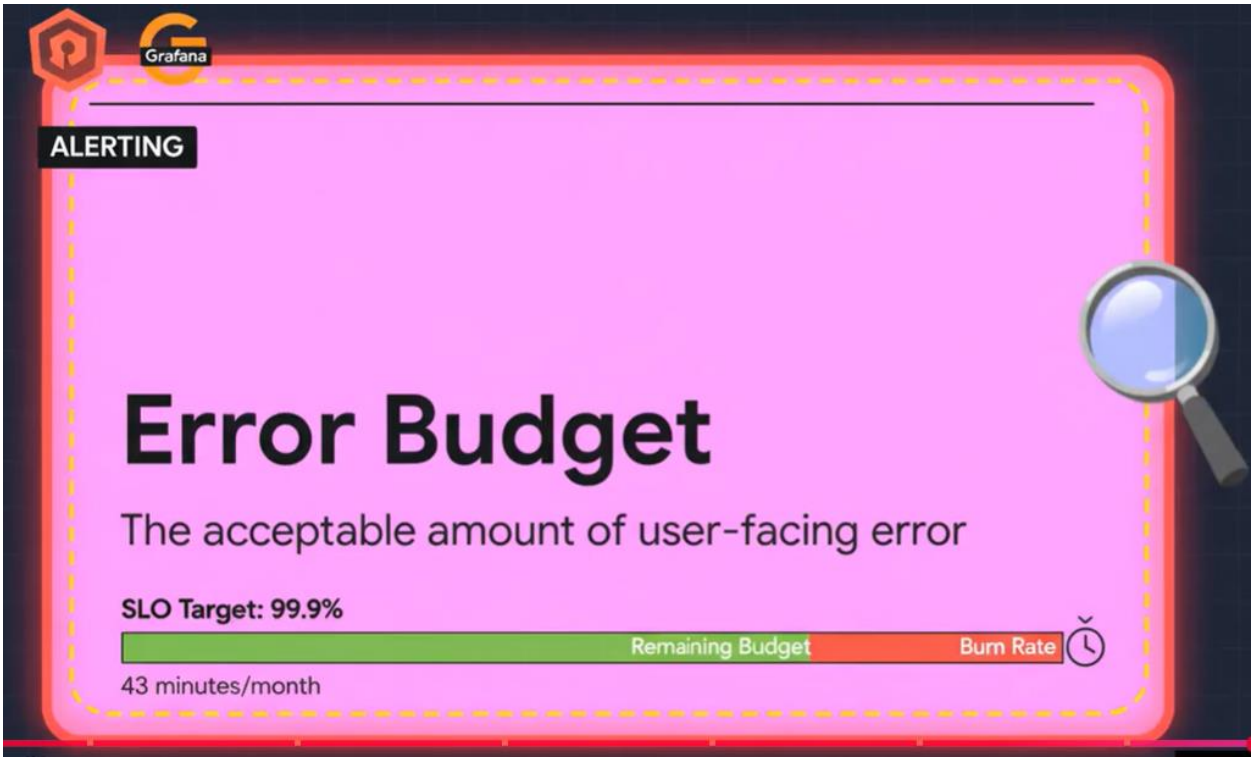


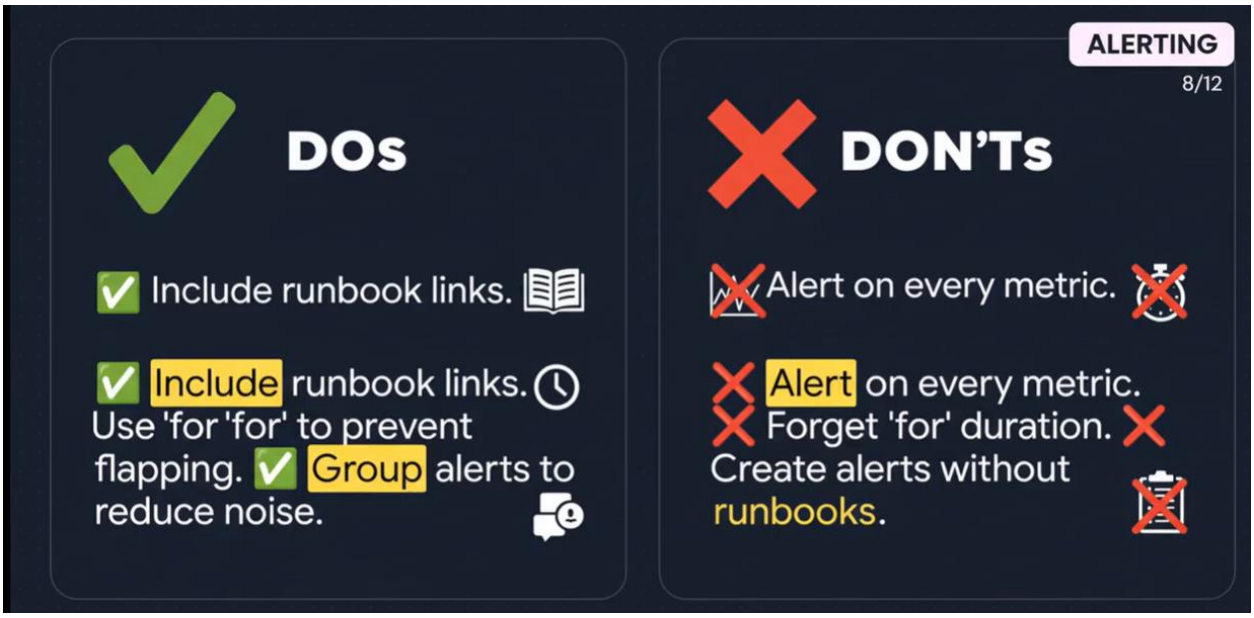




The end result: a clear, actionable notification in Slack with a summary, description, and severity to kickstart the investigation. 👩‍🚒


4
Alertmanger
Receivers
From Alerts to SLOs
SRE-Level Thinking
Error Budget
Warning
Latency Exceeded
Remaining Budget
SLO Target
ALERTING
8/12
5:54 / 7:17
Grafana Unified Alerting


Grafana
ALERTING
Error Budget
The acceptable amount of user-facing error
SLO Target: 99.9%
Remaining Budget
Burn Rate
43 minutes/month

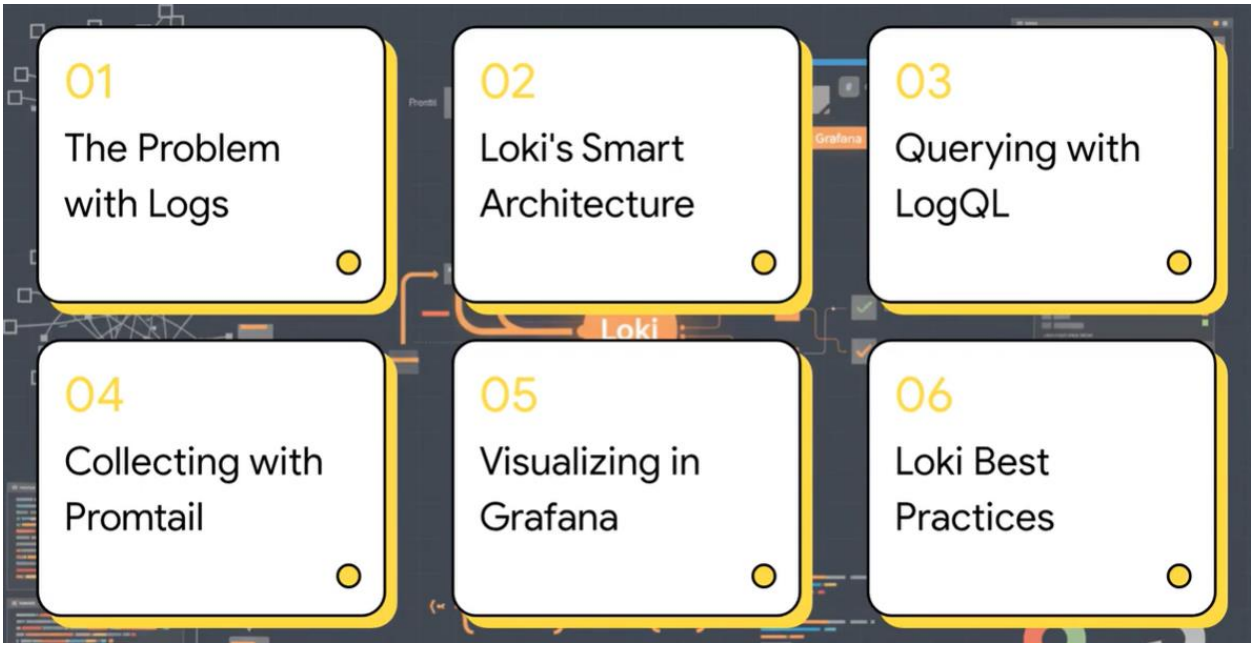

ALERTING
8/12
DOs
✅ Include runbook links.
✅ Include runbook links. Use 'for 'for' to prevent flapping. ✅ Group alerts to reduce noise.
DON'Ts
Alert on every metric.
Alert on every metric. Forget 'for' duration. Create alerts without runbooks.


Your alerts show what is broken. But do you know WHY?

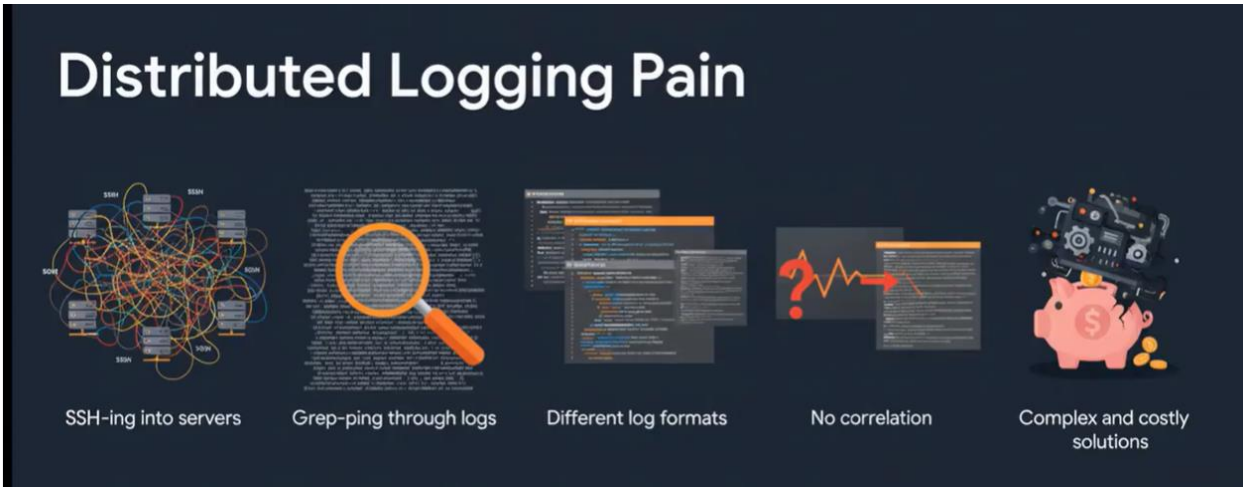
Chaos to Clarity with Loki

Alerts fired. You know WHAT broke. But do you know WHY?

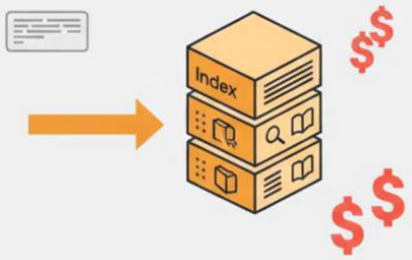

Before: The Hunt
After: The Promise of Loki
From many tools to one unifed view
Grafana
Structured Logs


01
The Problem with Logs
02
Loki's Smart Architecture
03
Querying with LogQL
04
Collecting with Promtail
05
Visualizing in Grafana
06
Loki Best Practices
Loki


LOKI LOGGING
9/12
1
The Problem with Logs
Slow, expensive, & complex

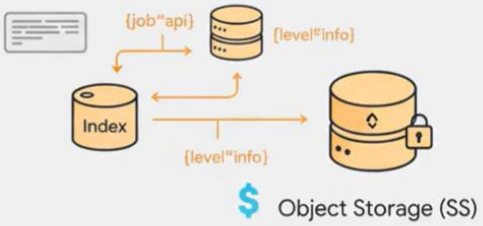

Distributed Logging Pain
SSH-ing into servers
Grep-ping through logs
Different log formats
No correlation
Complex and costly solutions

ELK Stack: Full-Text Indexing

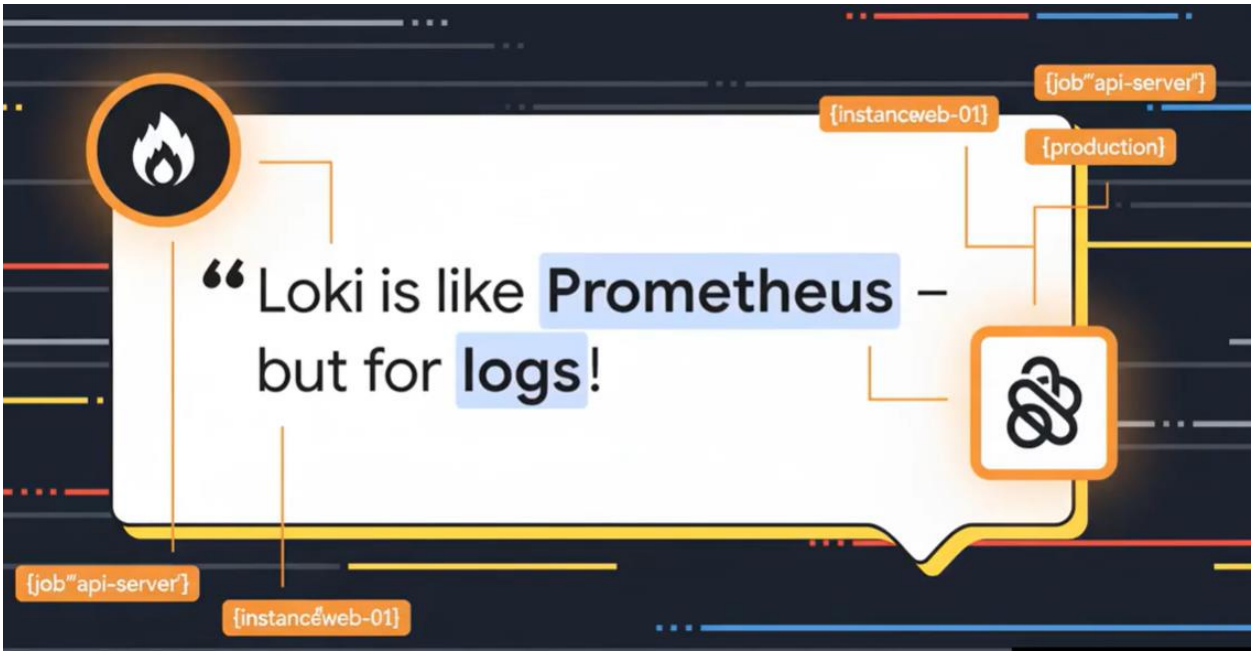


Indexes everything. High storage, powerful search.

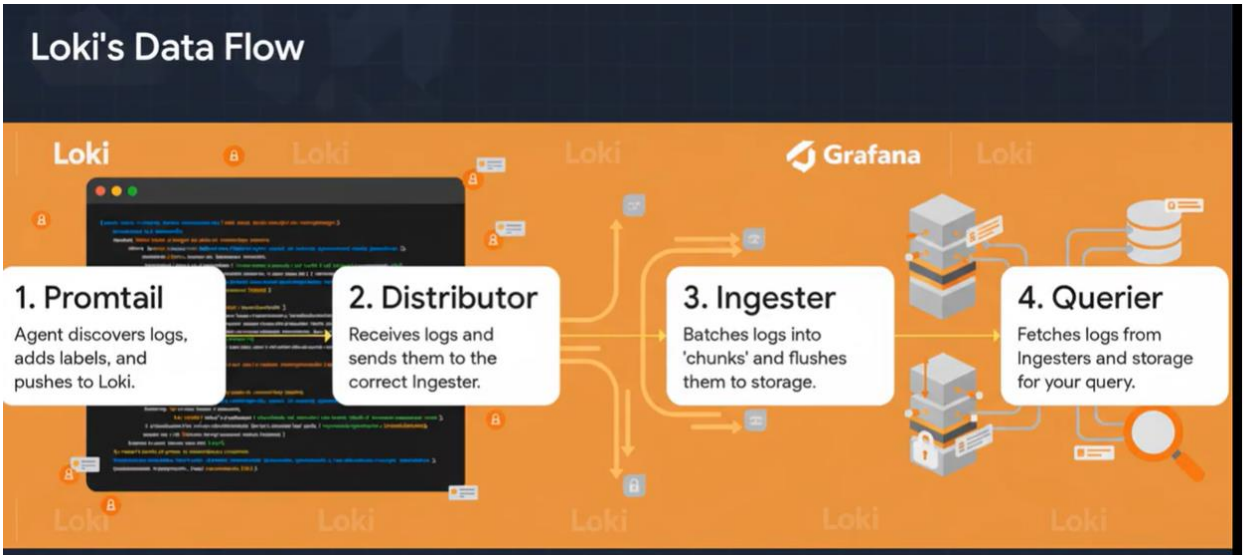
Grafana Loki: Label Indexing

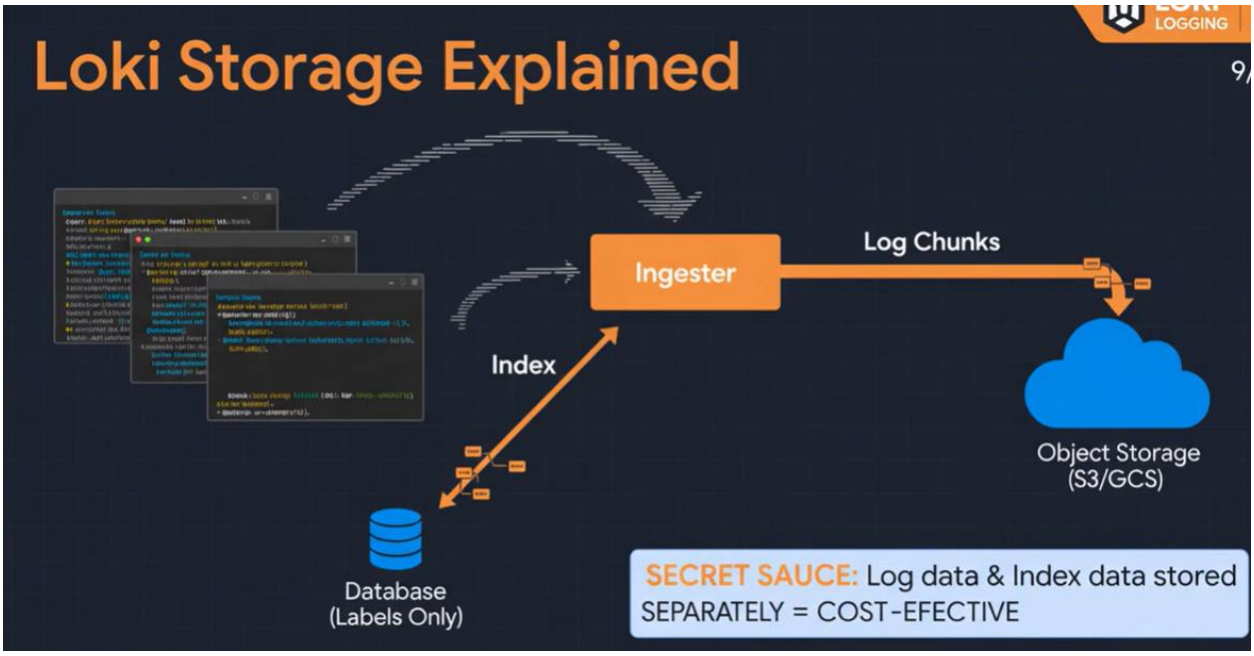


Indexes only labels. Low storage, cost-effestive.

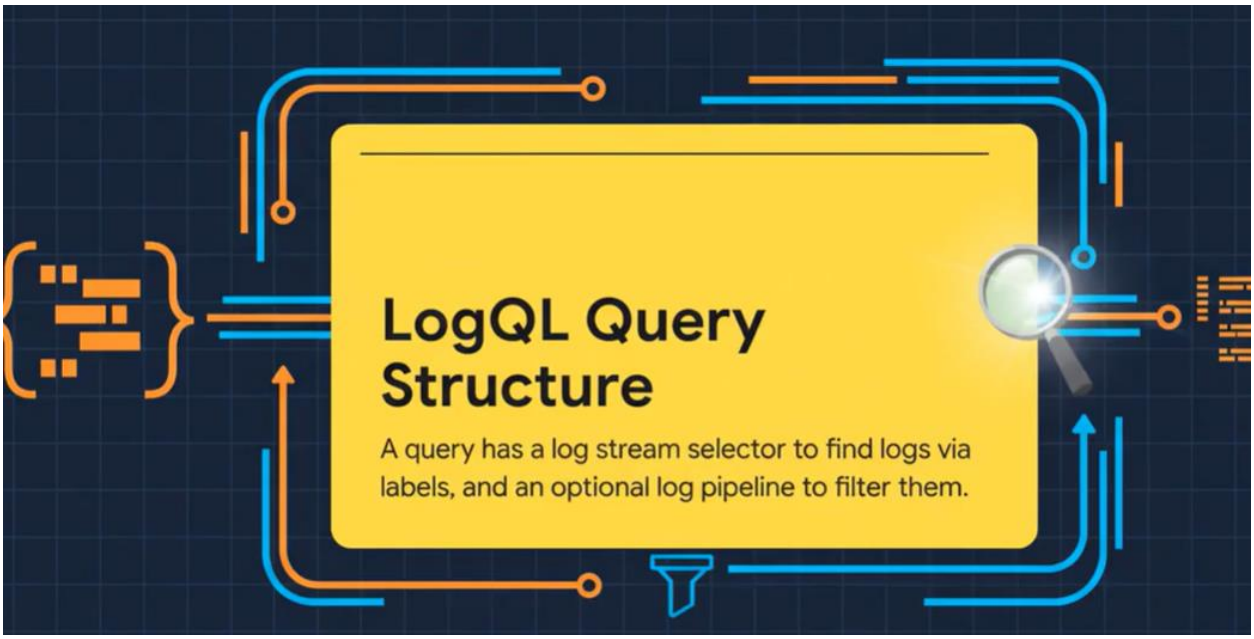



2
Promtail
Loki's Smart
Loki
Architecture
Decoupled for scale


Loki's Data Flow
Loki
Grafana
1. Promtail
Agent discovers logs, adds labels, and pushes to Loki.
2. Distributor
Receives logs and sends them to the correct Ingester.
3. Ingester
Batches logs into 'chunks' and flushes them to storage.
4. Querier
Fetches logs from Ingesters and storage for your query.


LOGGING
Loki Storage Explained
Ingester
Log Chunks
Index
Object Storage
(S3/GCS)
Database
(Labels Only)
SECRET SAUCE: Log data & Index data stored
SEPARATELY = COST-EFECTIVE


Promtail
Loki
Grafana
3
Querying with
LogQL
Connecting to your logs
{job"-api} l: cntonl:
{job"-api} l "error"
Gheware DevOps AI

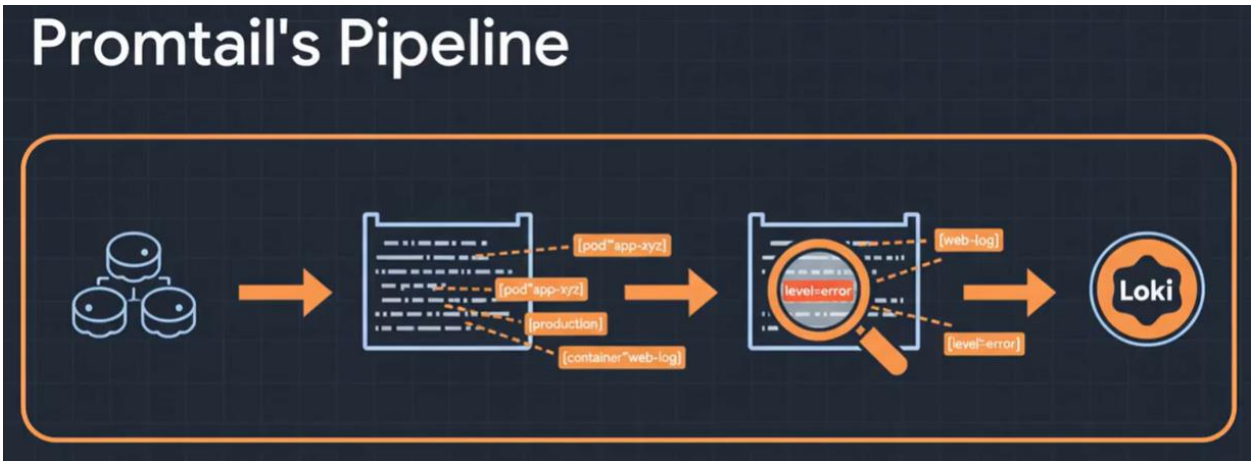


Operator	Description
=	Exactly equal
!=	Not equal
=~	Regex matches
!~	Regex does not match
`	=`
`	~`

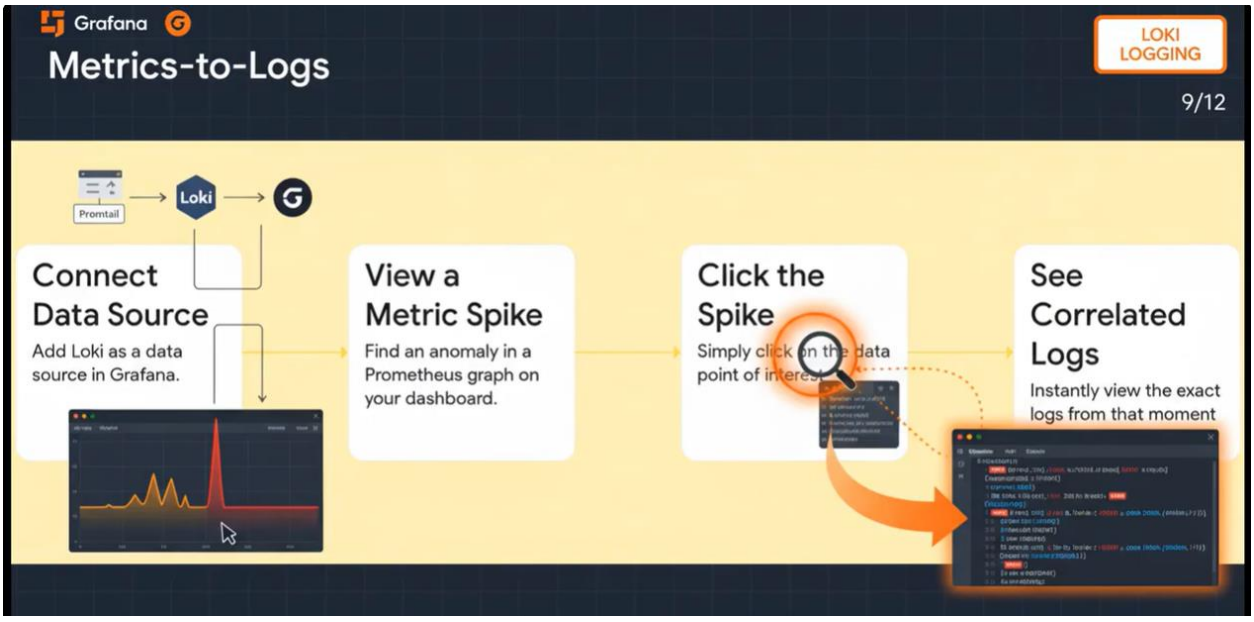
Beyond Basic Queries

- Parse logs on the fly with `| json` or `| logfmt`.
- Filter on newly extracted fields, like `duration > 500ms`.
- Create metrics from logs, just like in Prometheus!

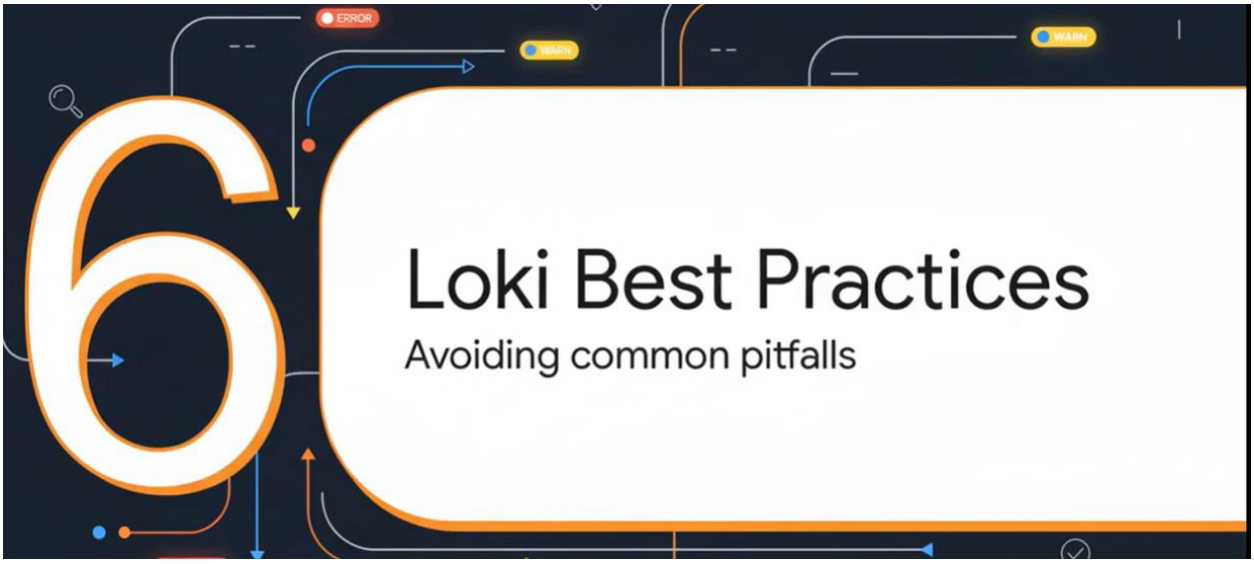
A simple YAML file tells Promtail where to send logs (`clients`) and what to scrape (`scrape_configs`), just like Prometheus.


Promtail's Pipeline
[pod="app-xyz"]
[pod="app-xyz"]
[production]
[container="web-log"]
level=error
[web-log]
[level="error"]
Loki

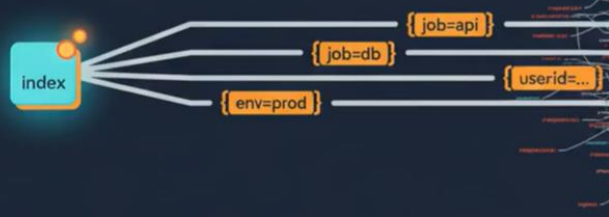

5
Visualizing in Grafana
Metrics and Logs Unified


Grafana
Metrics-to-Logs
LOKI LOGGING
9/12
Promtail
Loki
Connect Data Source
Add Loki as a data source in Grafana.
View a Metric Spike
Find an anomaly in a Prometheus graph on your dashboard.
Click the Spike
Simply click on the data point of interest
See Correlated Logs
Instantly view the exact logs from that moment

Your Prometheus metrics and Loki logs, side-by-side. Go from 'What broke?' to 'Why?' in seconds, without ever leaving your dashboard.



DO: Low Cardinality



Few, distinct streams → Efficient Index.

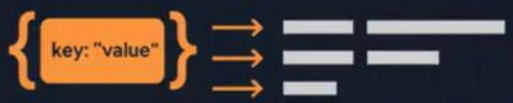
DON'T: High Cardinality

Chaotic, infinite streams → Kills performance.

Production Takeaways

Manage Cardinality

Keep label values limited and stable.



Use Structured Logs

JSON or logfmt makes parsing easy.

Optimize Queries

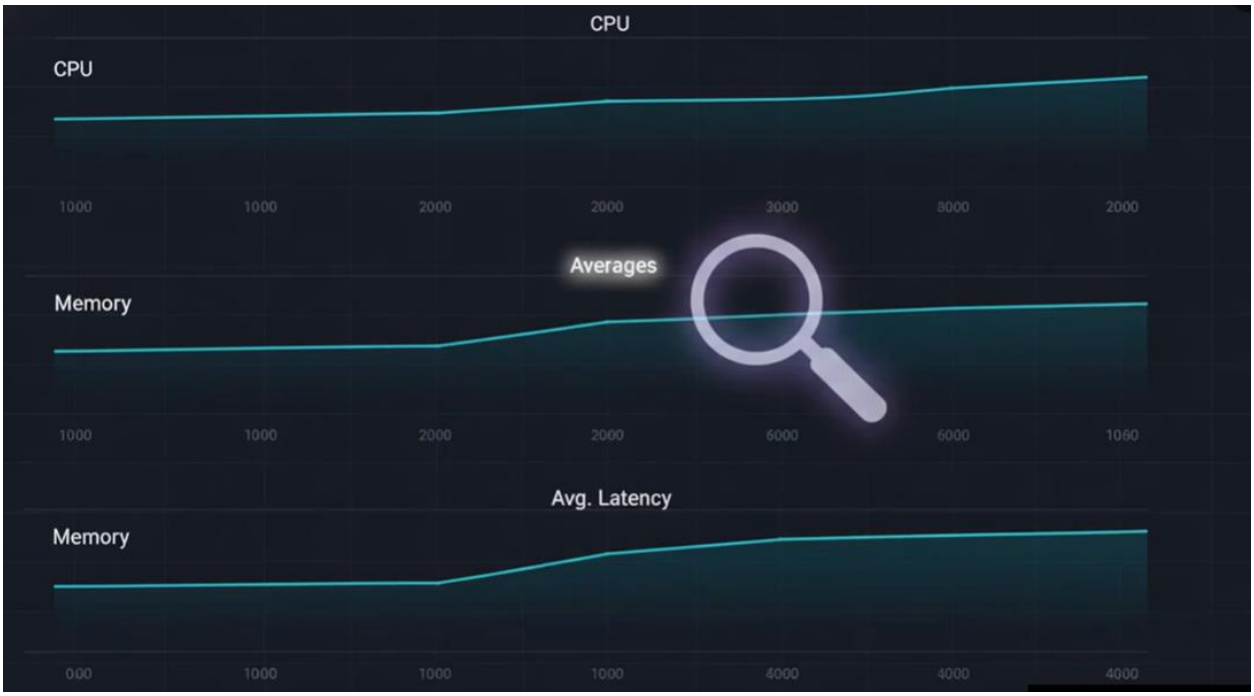
Start with restrictive label selectors.

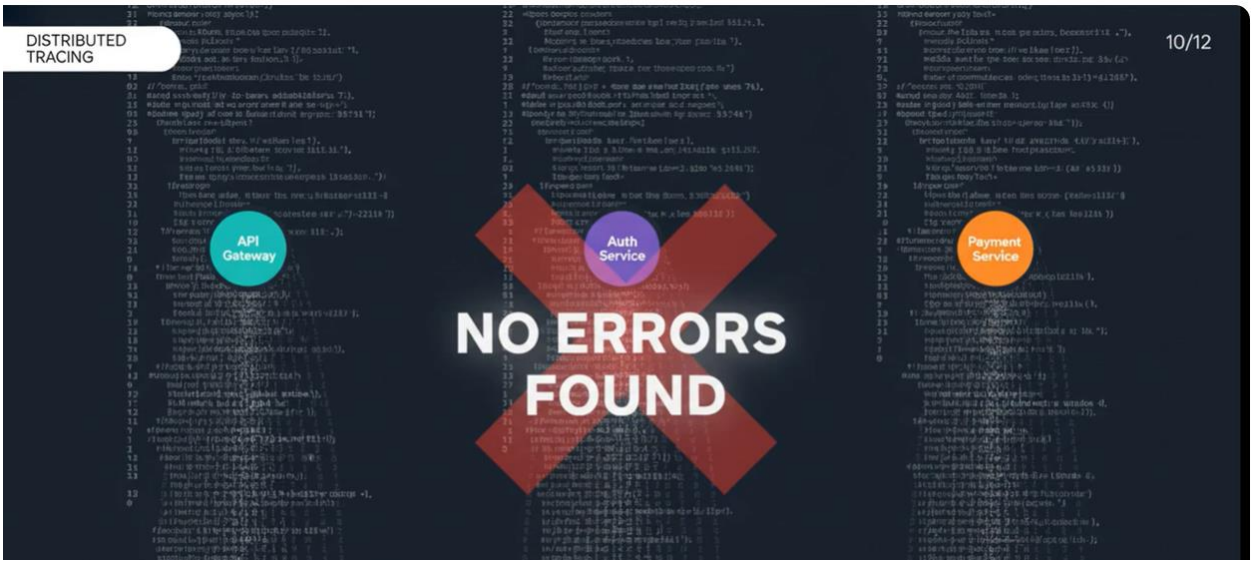
Configure Retention

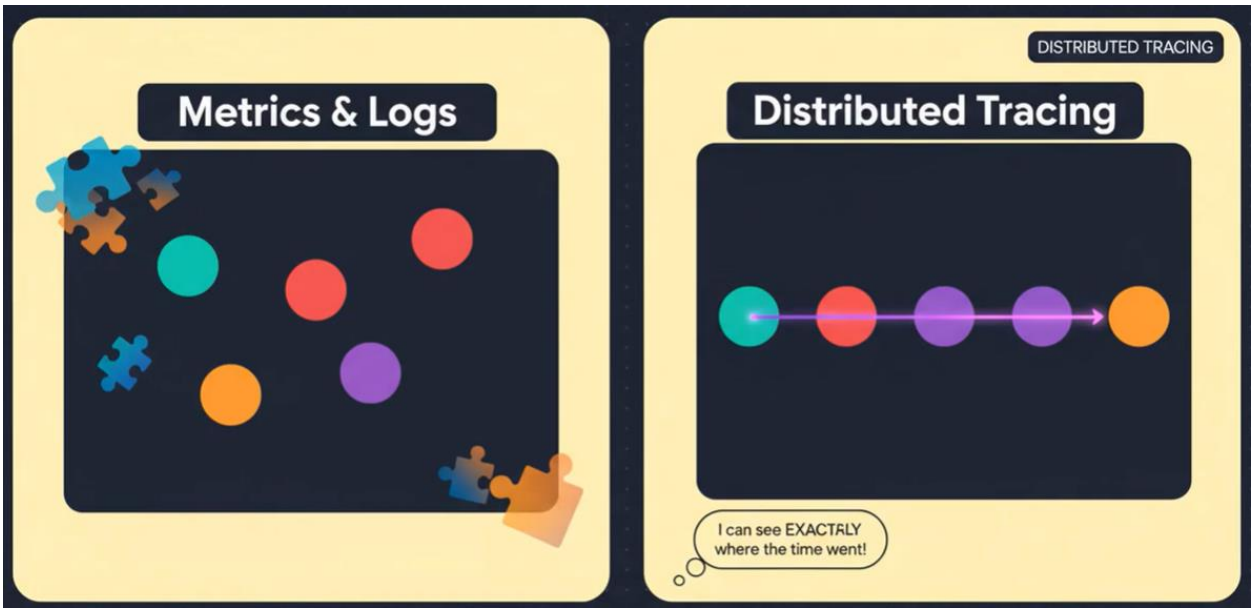
Lifecycle Management

You have logs. Can you now trace a request across your system?

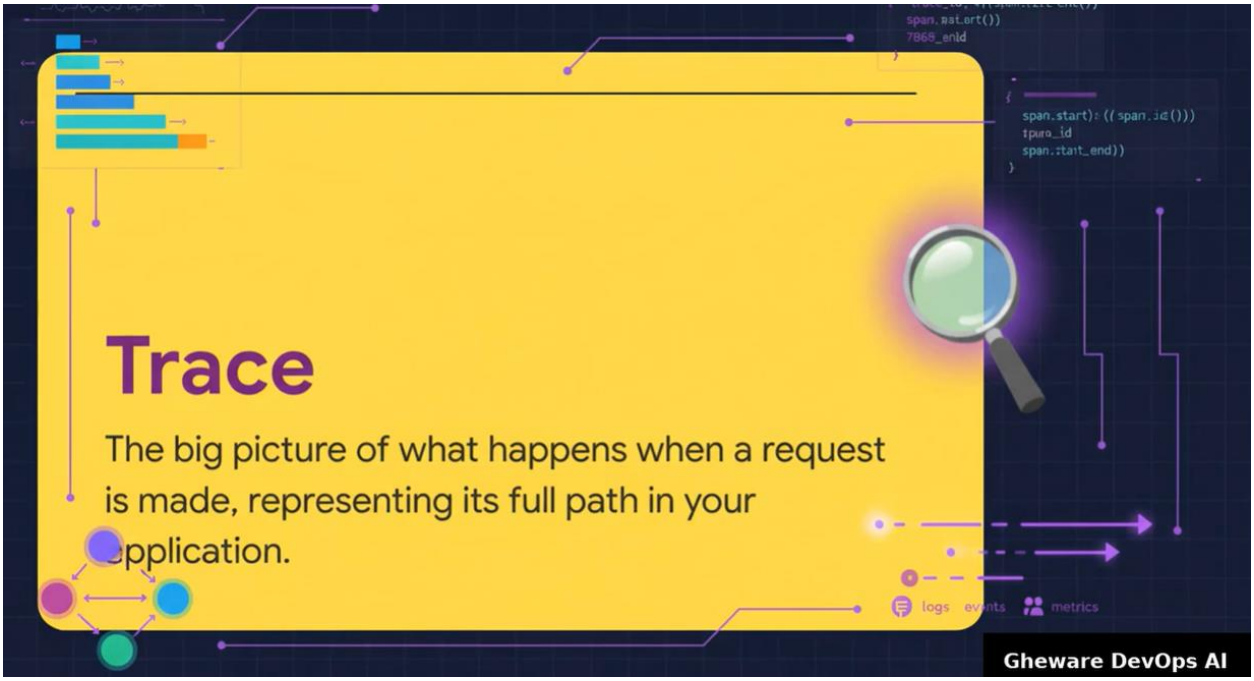
WHERE did those 3 seconds go?

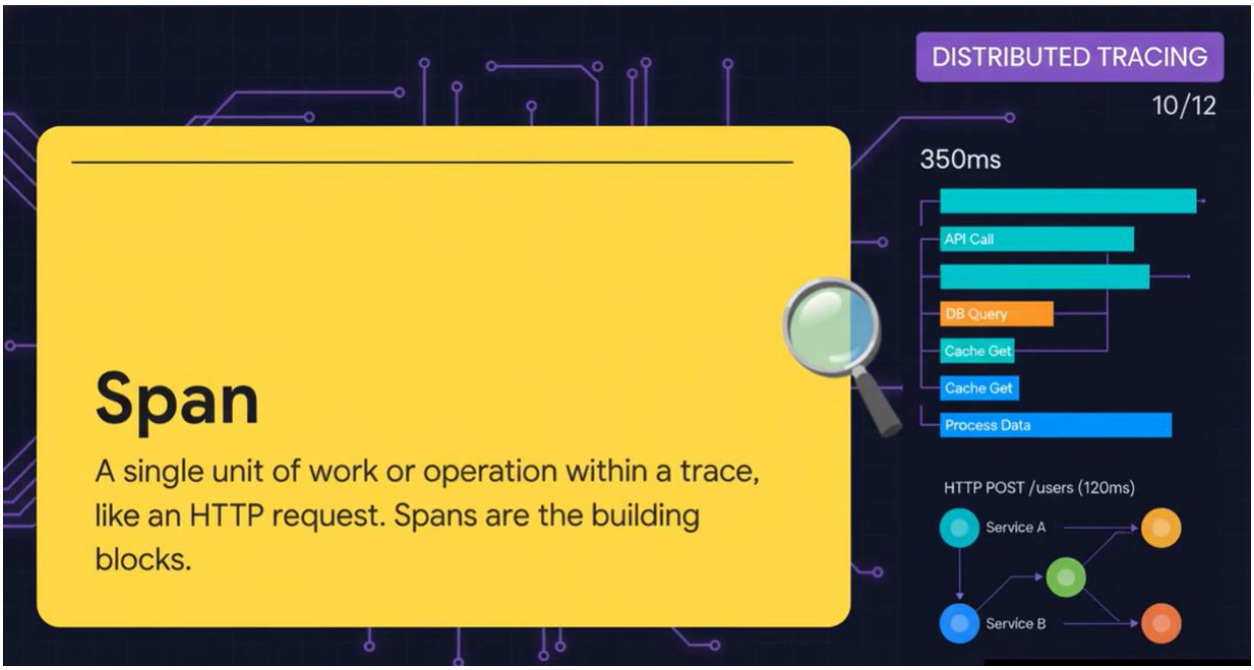


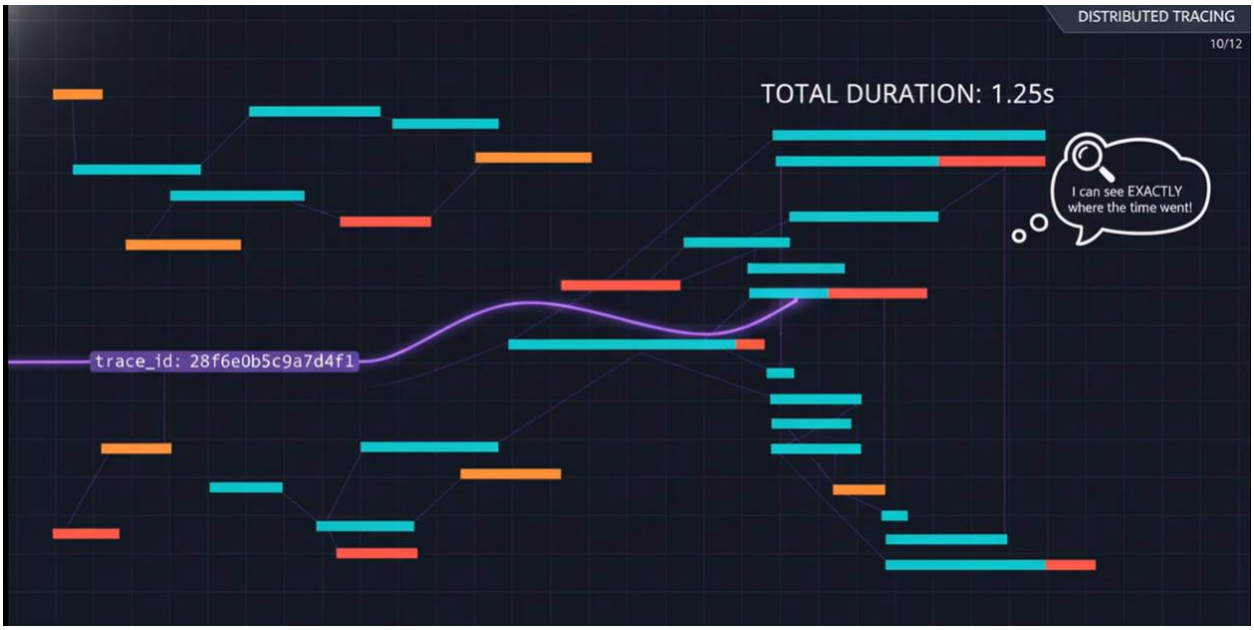

DISTRIBUTED TRACING
10/12
API Gateway
Auth Service
Payment Service
NO ERRORS FOUND

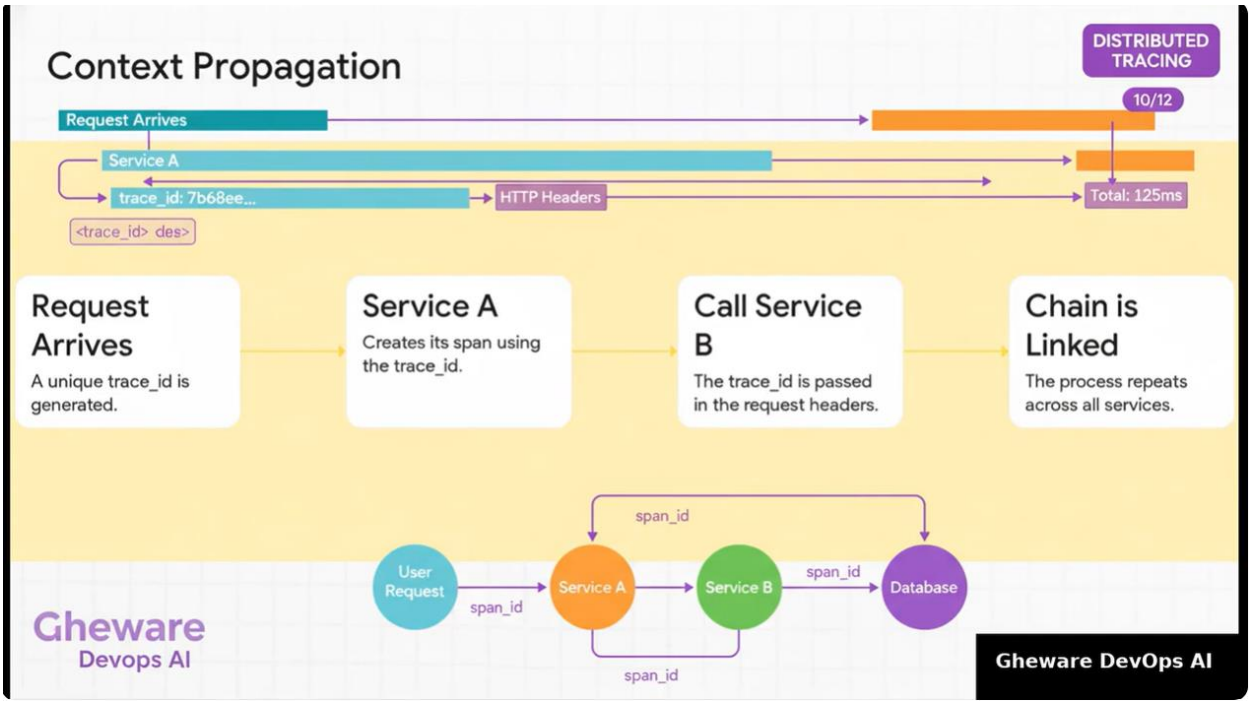

Metrics & Logs
DISTRIBUTED TRACING
Distributed Tracing
I can see EXACTLY where the time went!


10
The Breakthrough
Following the Clues

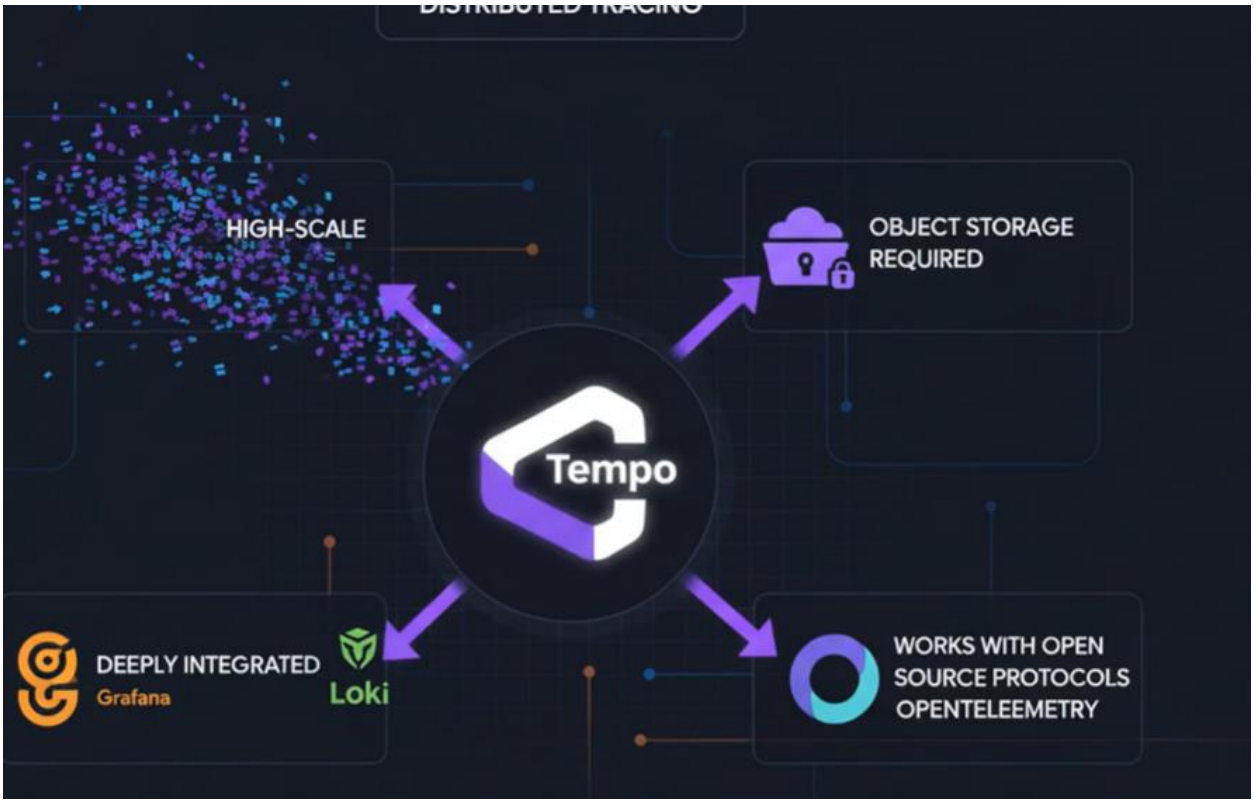

Trace
The big picture of what happens when a request is made, representing its full path in your application.
logs
metrics
Gheware DevOps AI


DISTRIBUTED TRACING
10/12
Span
A single unit of work or operation within a trace, like an HTTP request. Spans are the building blocks.
350ms
API Call
DB Query
Cache Get
Cache Get
Process Data
HTTP POST /users (120ms)
Service A
Service B


DISTRIBUTED TRACING
10/12
TOTAL DURATION: 1.25s
I can see EXACTLY where the time went!
trace_id: 28f6e0b5c9a7d4f1


Context Propagation
DISTRIBUTED TRACING
10/12
Request Arrives
Service A
trace_id: 7b68ee...
HTTP Headers
Total: 125ms
<trace_id> des>
Request Arrives
A unique trace_id is generated.
Service A
Creates its span using the trace_id.
Call Service B
The trace_id is passed in the request headers.
Chain is Linked
The process repeats across all services.
span_id
User Request
span_id
Service A
Service B
span_id
Database
span_id
Gheware
Devops AI
Gheware DevOps AI

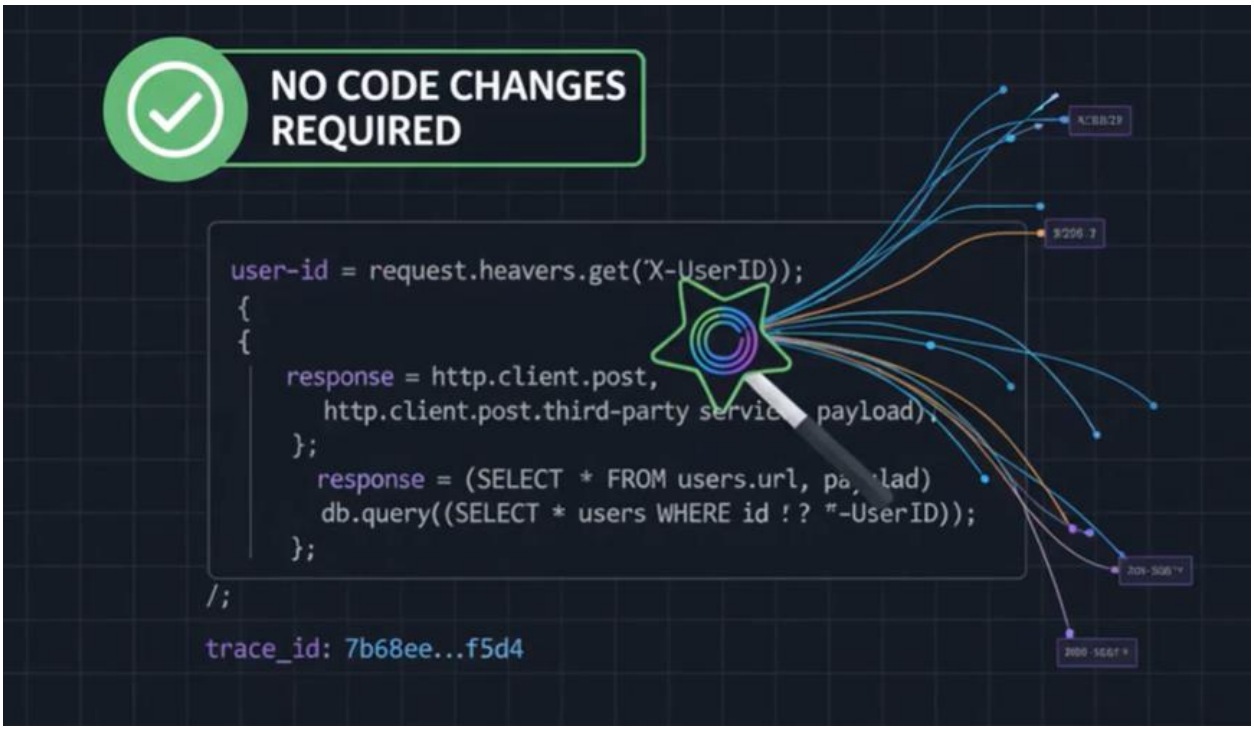

10/1
2
Your Toolkit
Tempo & OpenTelemetry
F2.49.5C
Gheware Devops AI
Gheware DevOps AI

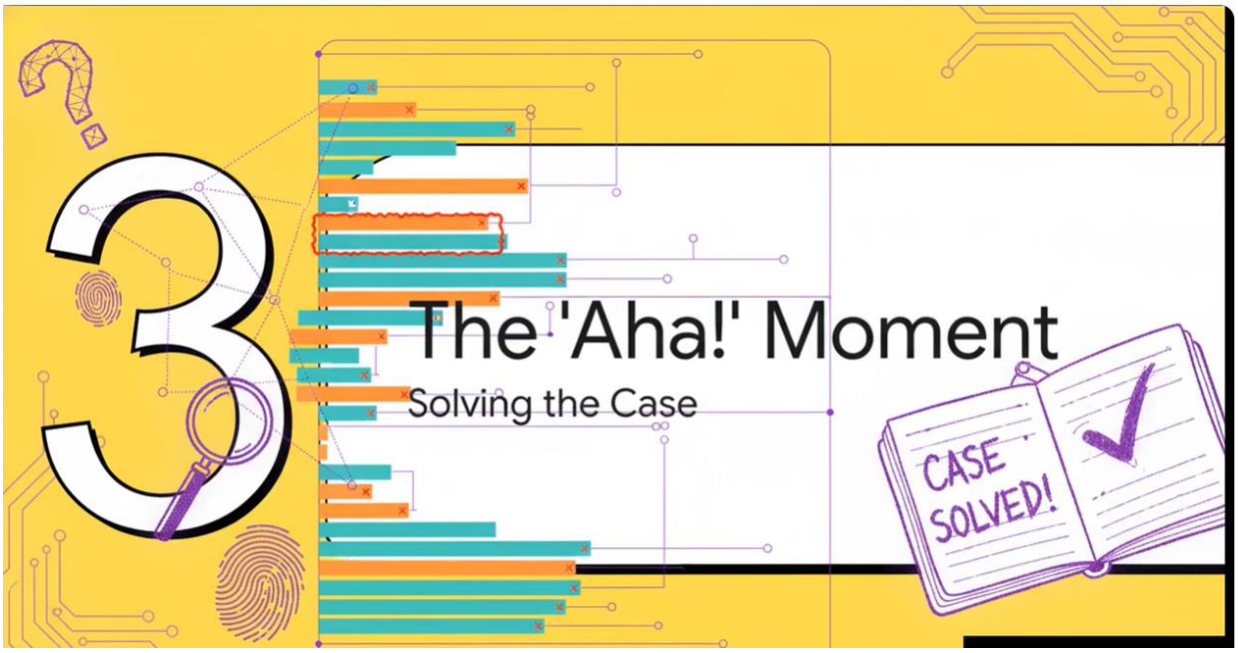


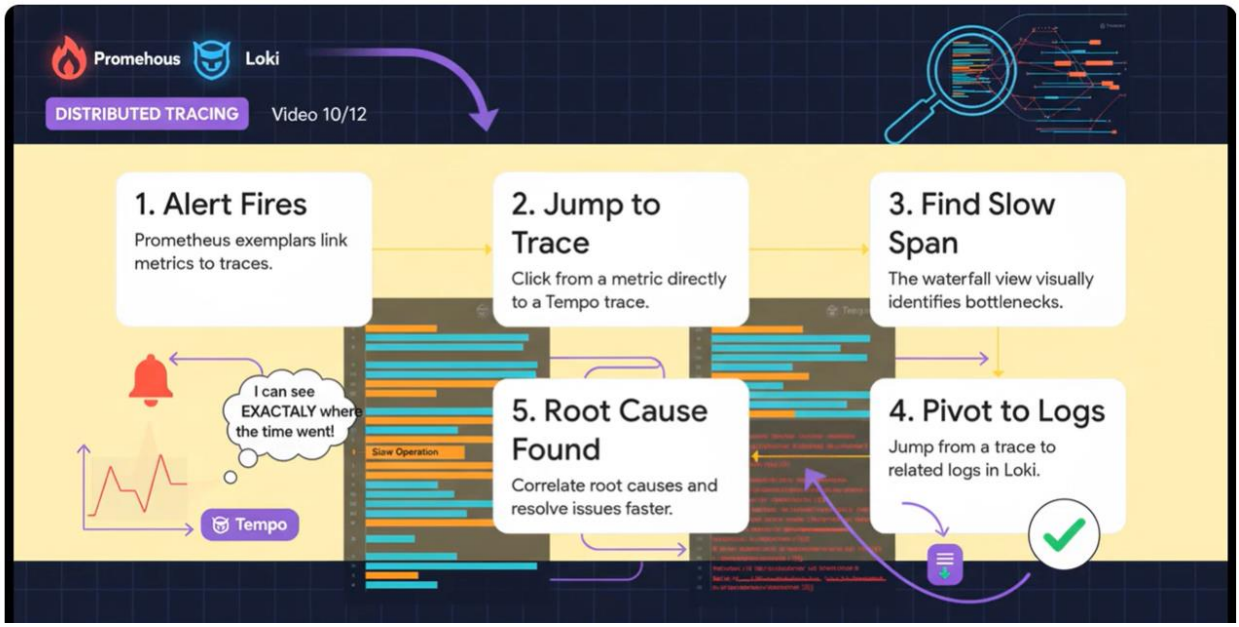
Tempo is the backend, but your applications must first be instrumented to generate and emit traces.

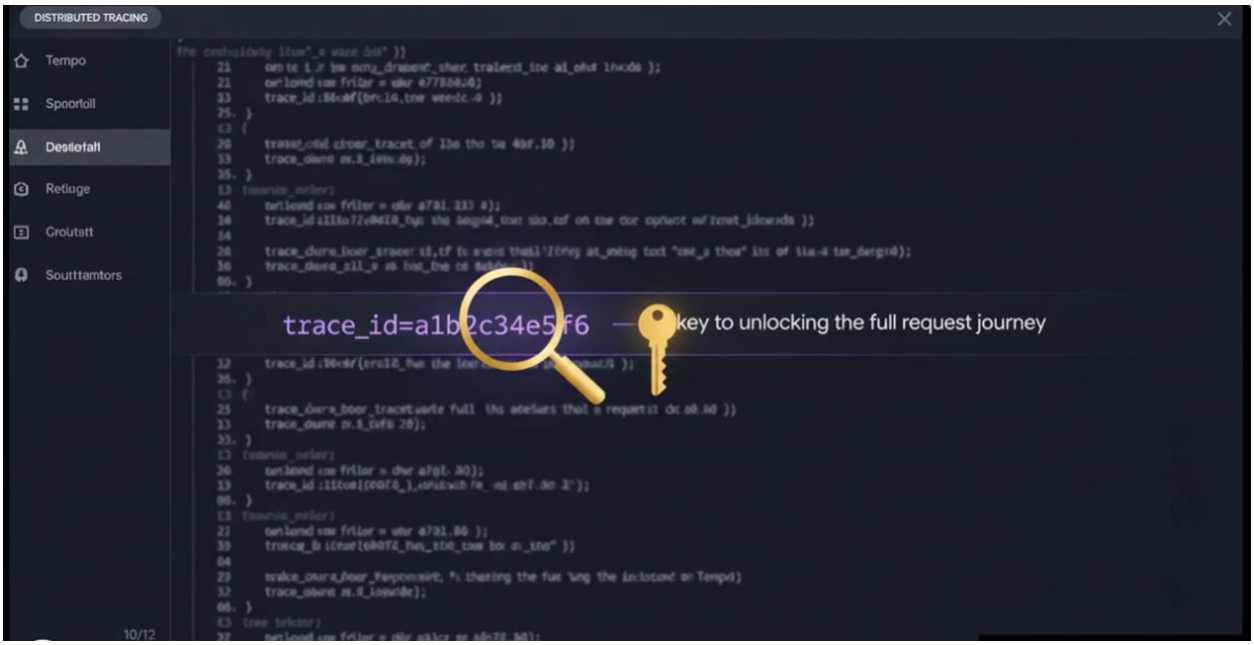
OpenTelemetry (OTel)

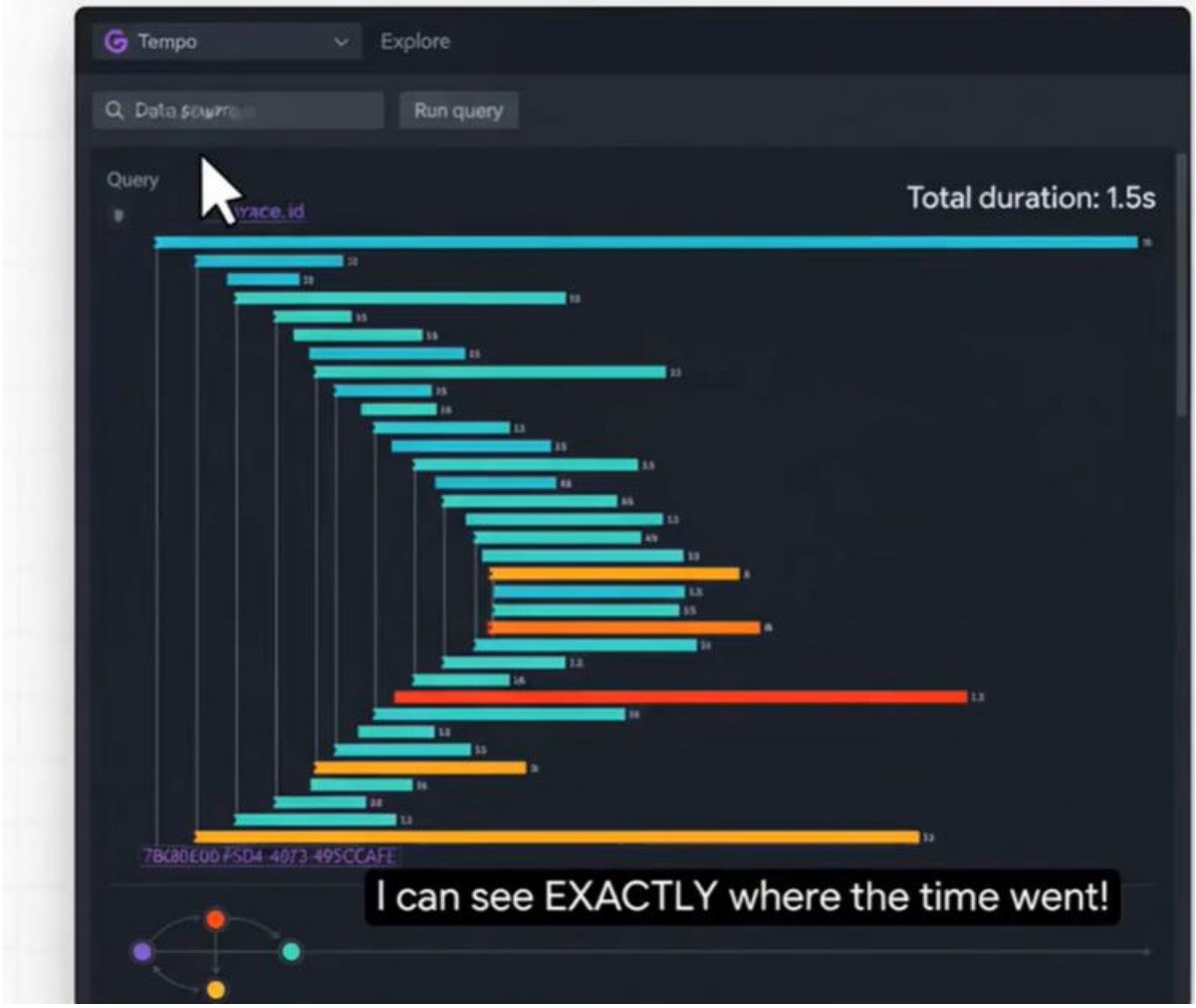
An observability framework to generate and collect application telemetry data like traces, metrics, and logs.

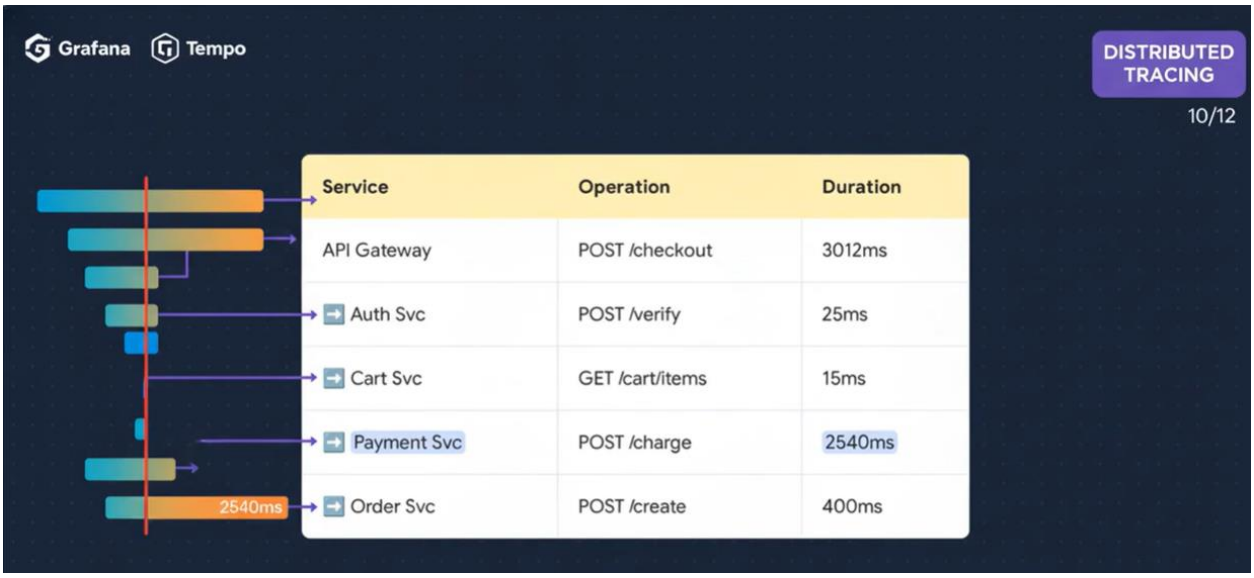



3
The 'Aha!' Moment
Solving the Case
CASE SOLVED!


Promehous
Loki
DISTRIBUTED TRACING
Video 10/12
1. Alert Fires
Prometheus exemplars link metrics to traces.
2. Jump to Trace
Click from a metric directly to a Tempo trace.
3. Find Slow Span
The waterfall view visually identifies bottlenecks.
I can see EXACTALY where the time went!
Tempo
Slow Operation
5. Root Cause Found
Correlate root causes and resolve issues faster.
4. Pivot to Logs
Jump from a trace to related logs in Loki.


DISTRIBUTED TRACING
Tempo
Spoortoll
Desiletall
Retiuge
Groutatt
Soutttamtors
trace_id=a1b2c34e5f6
key to unlocking the full request journey


Tempo
Explore
Run query
Query
Total duration: 1.5s
I can see EXACTLY where the time went!


Grafana
Tempo
DISTRIBUTED TRACING
10/12
Service
Operation
Duration
API Gateway
POST /checkout
3012ms
Auth Svc
POST /verify
25ms
Cart Svc
GET /cart/items
15ms
Payment Svc
POST /charge
2540ms
Order Svc
POST /create
400ms
2540ms


DISTRIBUTED TRACING
10/12
<60s
Critical Path


4
Best Practices
Tracing Like a Pro

DO

Trace process boundaries: Client, Server, Producer, and Database spans.

I can see EXACTLY where the time went!

DON'T

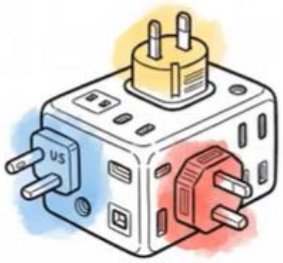
DON'T trace everything. Use Internal spans for key operations, not tight loops.

Your Takeaway

- Tracing shows the full request journey.
- Spans pinpoint individual operations.
- OpenTelemetry is the instrumentation standard.
- Grafana Tempo is the high-scale backend.
- Grafana provides integrated visualization.

What hidden performance mysteries will you uncover?

OpenTelemetry Explainer

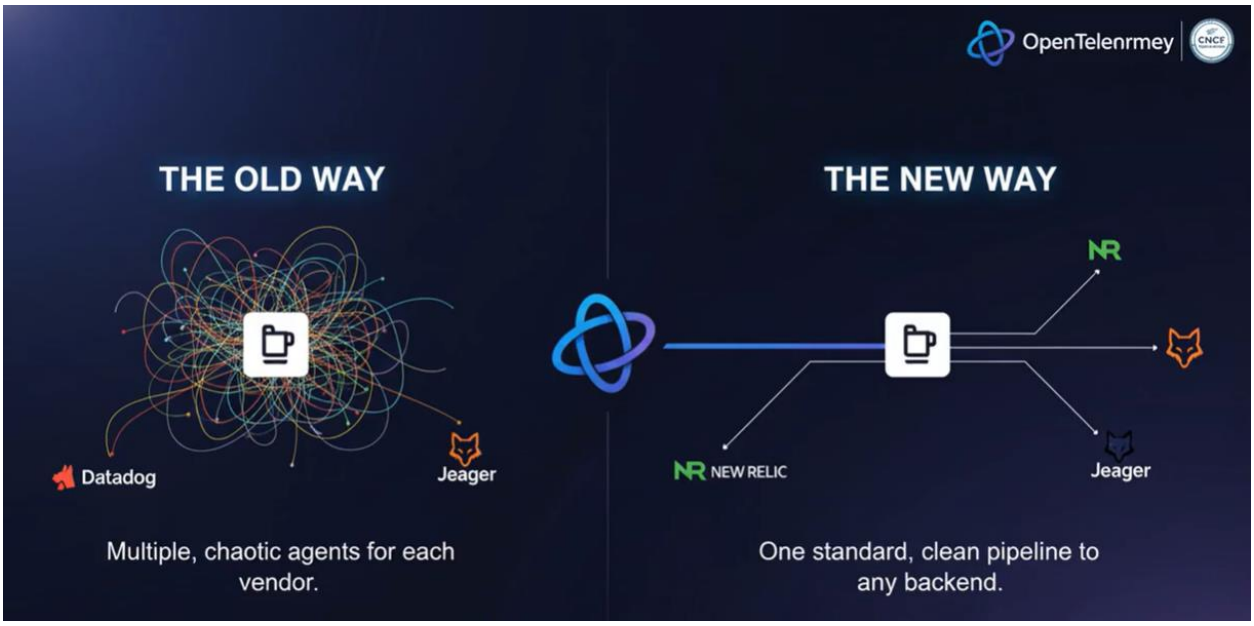


Your application is slow. Users are complaining. But which microservice is the problem?

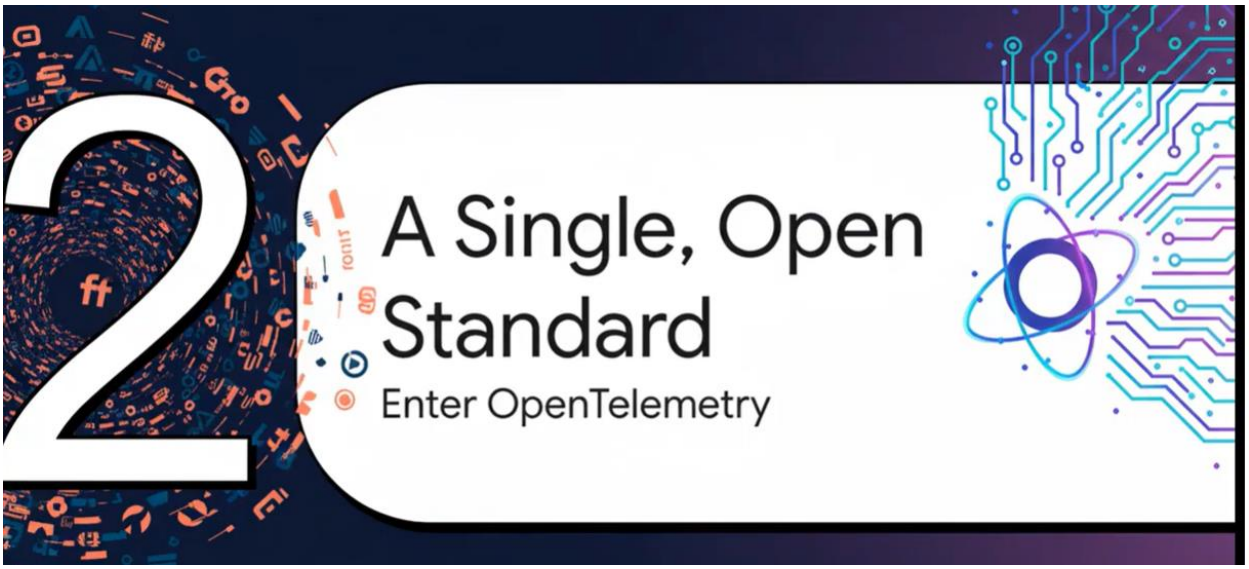
1

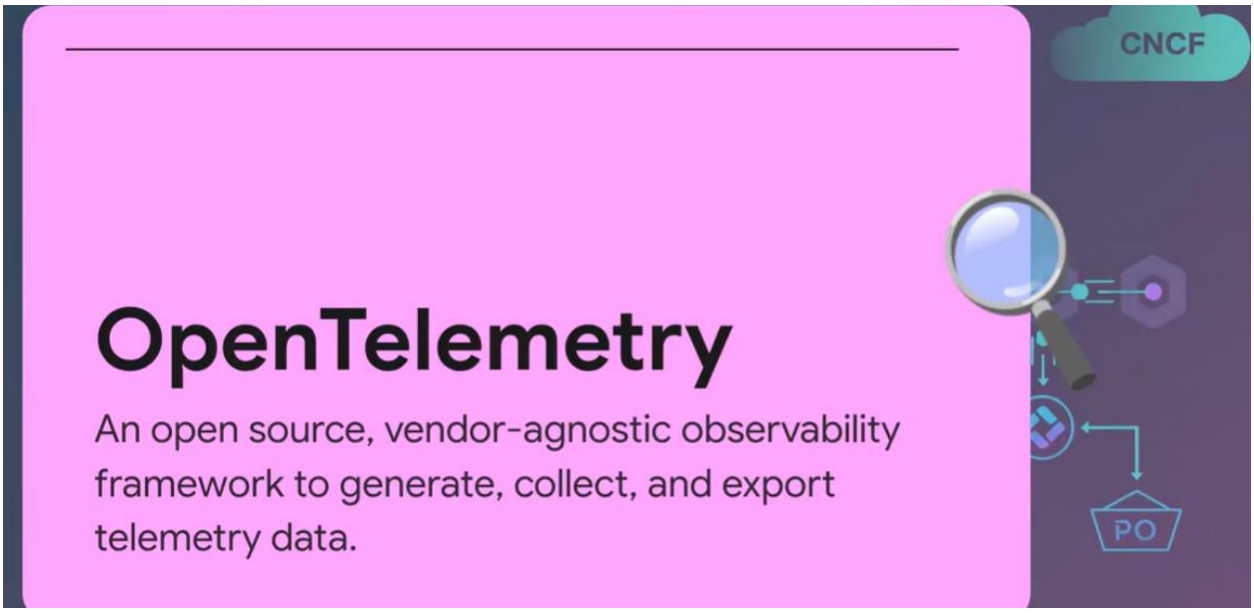
Instrumentation Chaos

Before OpenTelemetry


OpenTelenrmey
CNCF
THE OLD WAY
THE NEW WAY
Datadog
Jeager
NR
NEW RELIC
Jeager
Multiple, chaotic agents for each vendor.
One standard, clean pipeline to any backend.


The Pain of Chaos
aws
Google Cl
datadog
aws
azure
Java
Python
.NET
VENDOR LOCK-IN
HIGH COST
INCOSISTENT QUALITY





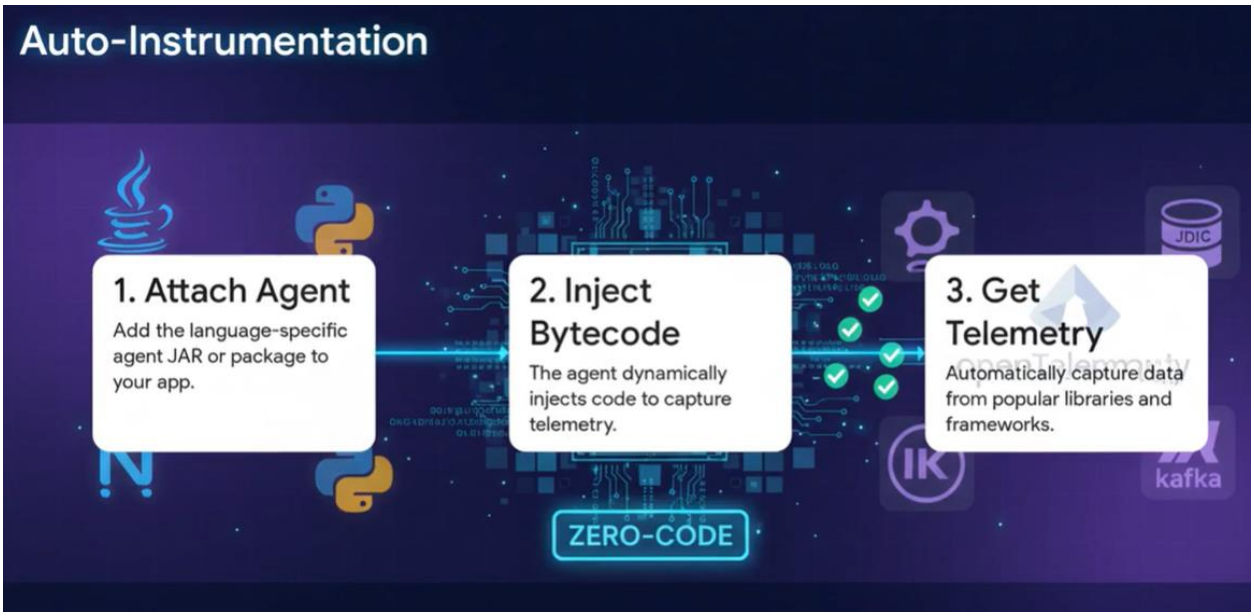
The Three Signals

- Traces: The path of a request through your application.
- Metrics: A measurement captured at runtime.
- Logs: A recording of an event.


CNCF
A graduated project of the Cloud Native Computing Foundation (CNCF)
T
C
OpenTeleemety
merging OpenTracing and OpenCensus to create one powerful standard.


3
Instrumentation in Action
How It Works


Traces
Get telemetry for your app in less than 5 minutes?
Logs


Auto-Instrumentation
1. Attach Agent
Add the language-specific agent JAR or package to your app.
2. Inject Bytecode
The agent dynamically injects code to capture telemetry.
3. Get Telemetry
Automatically capture data from popular libraries and frameworks.
ZERO-CODE
JDIC
kafka

For more detail, use the API to add custom business context like a customer ID or order value, creating richer telemetry.

Collector Pipeline

1. Receivers
Get data in various formats like OTLP, Jaeger, or Prometheus.

→

2. Processors
Modify data through batching, filtering, or sampling.

→

3. Exporters
Send processed data to one or more backend systems.

Why Use a Collector?

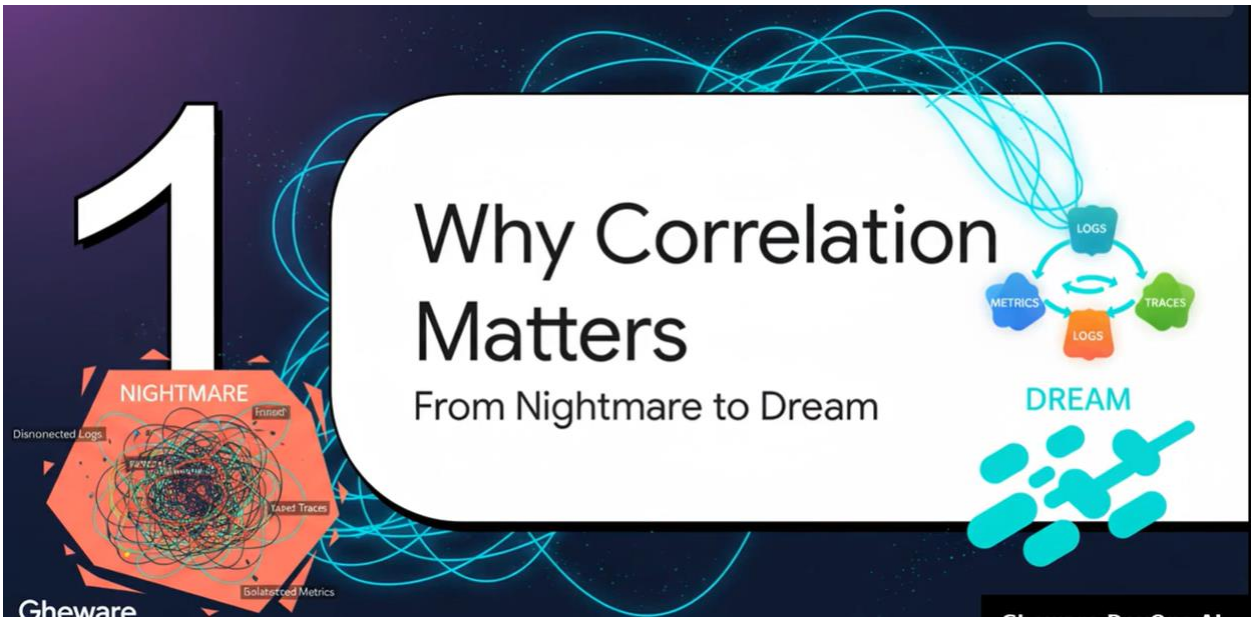
- Cost Control: Sample data to reduce backend costs.
- Reliability: Buffer data and retry sends to prevent data loss.
- Flexibility: Route data to multiple backends simultaneously.

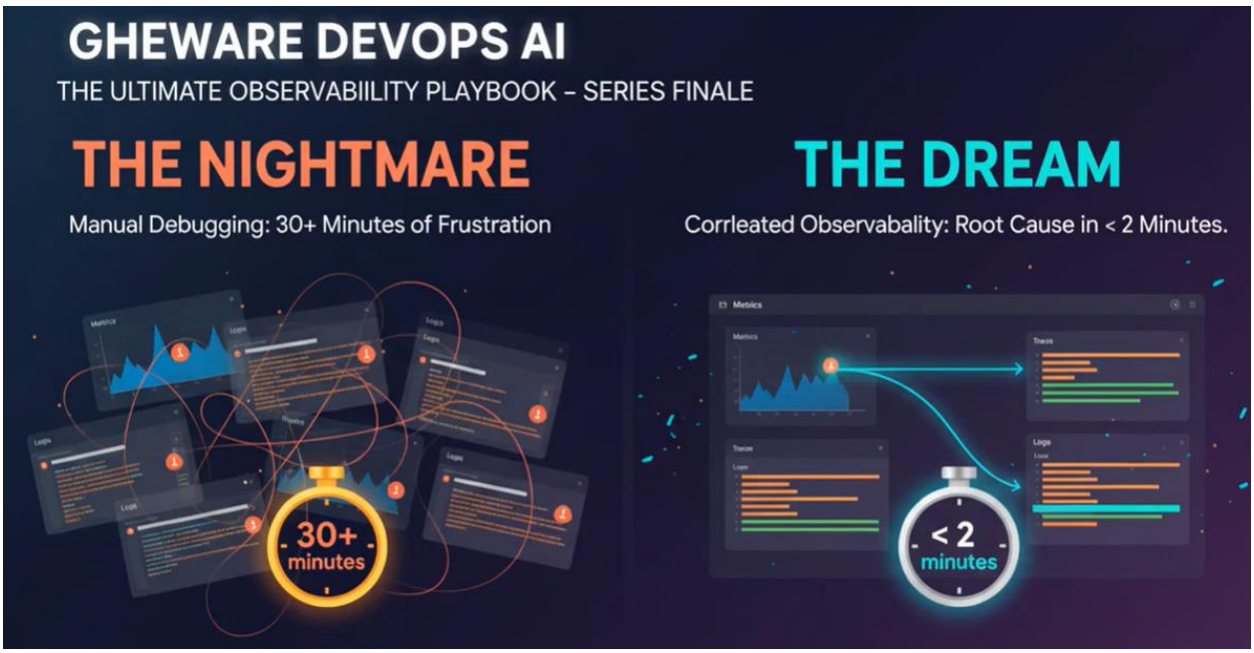
One standard for all services. One format for all your data (traces, metrics, logs). One flexible pipeline to manage it all.

Correlating Metrics, Logs, Traces

Ghewa

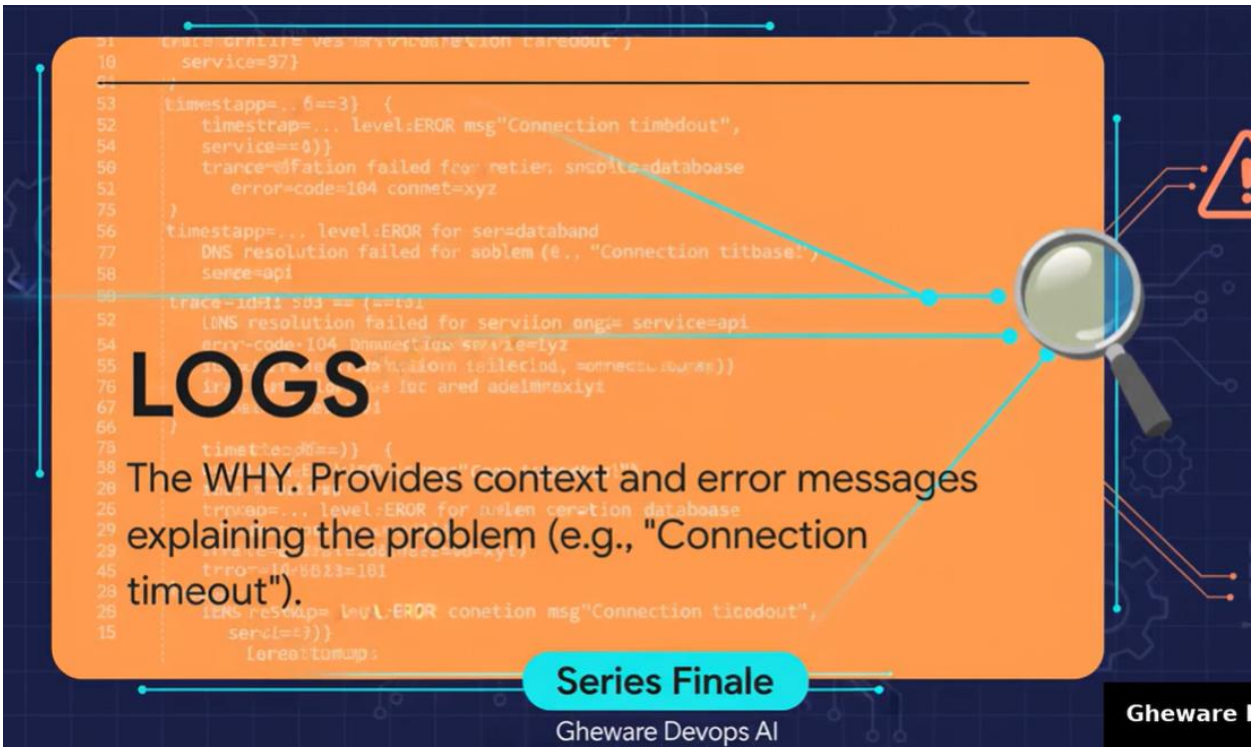
An alert fires. WHERE is the problem? WHICH request failed?

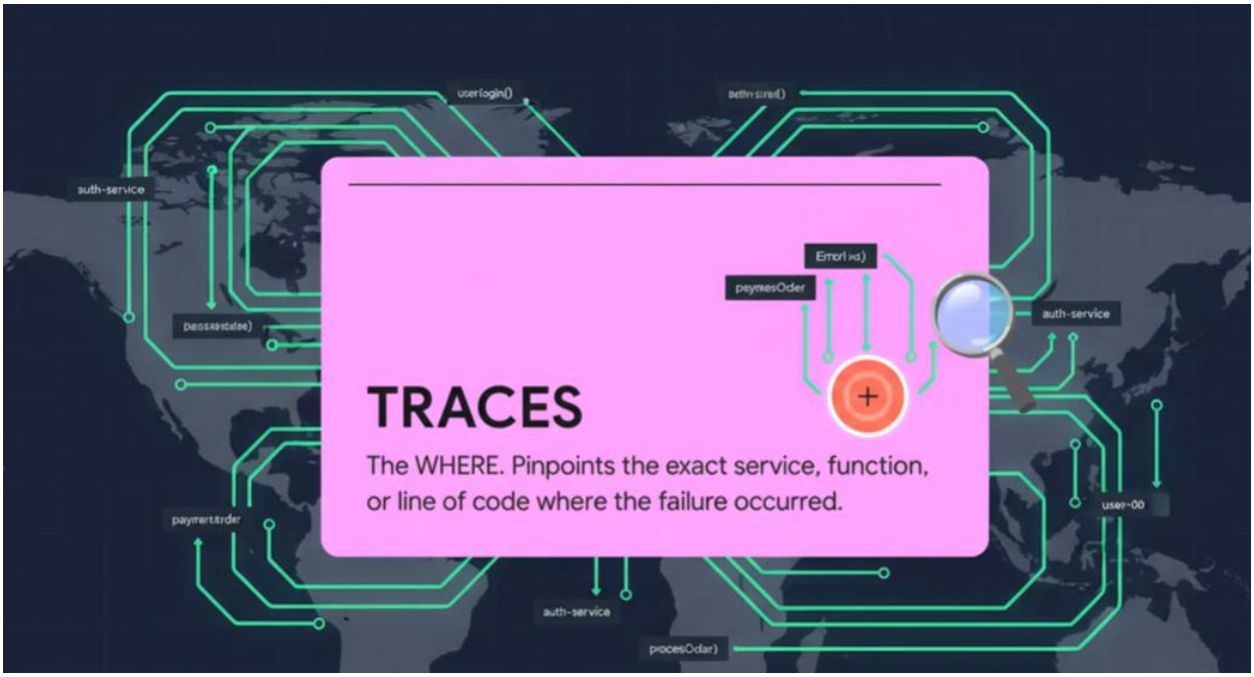

1
Why Correlation Matters
From Nightmare to Dream
NIGHTMARE
Disnonected Logs
Isolated Traces
Isolated Metrics
LOGS
METRICS
TRACES
LOGS
DREAM

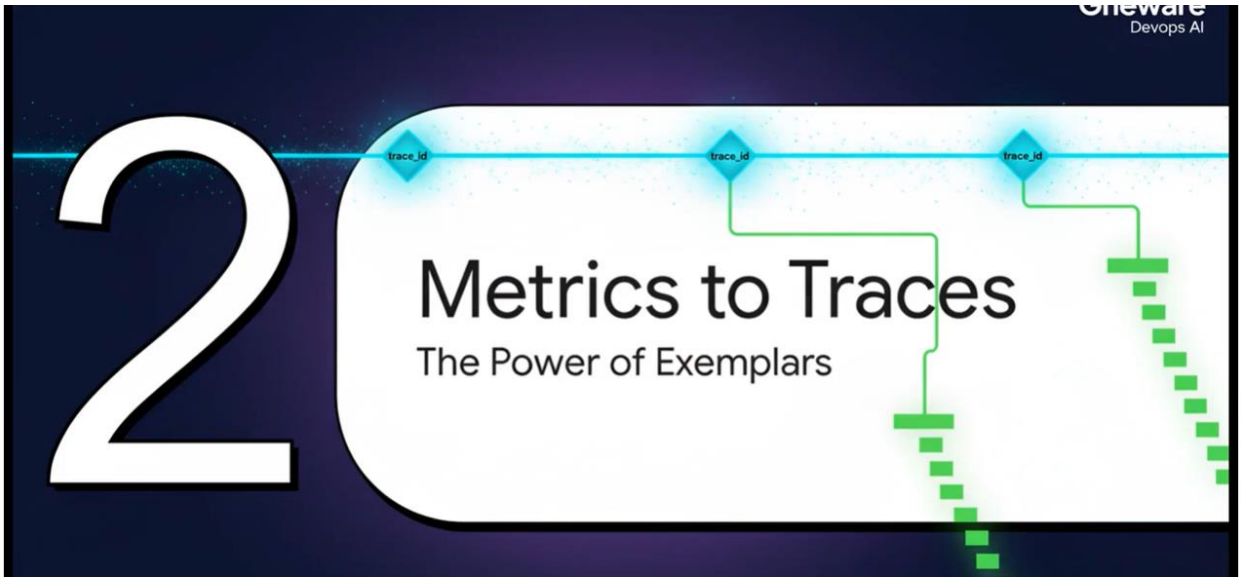

GHEWARE DEVOPS AI
THE ULTIMATE OBSERVABIILITY PLAYBOOK – SERIES FINALE
THE NIGHTMARE
Manual Debugging: 30+ Minutes of Frustration
30+ minutes
THE DREAM
Corrleated Observabality: Root Cause in < 2 Minutes.
< 2 minutes

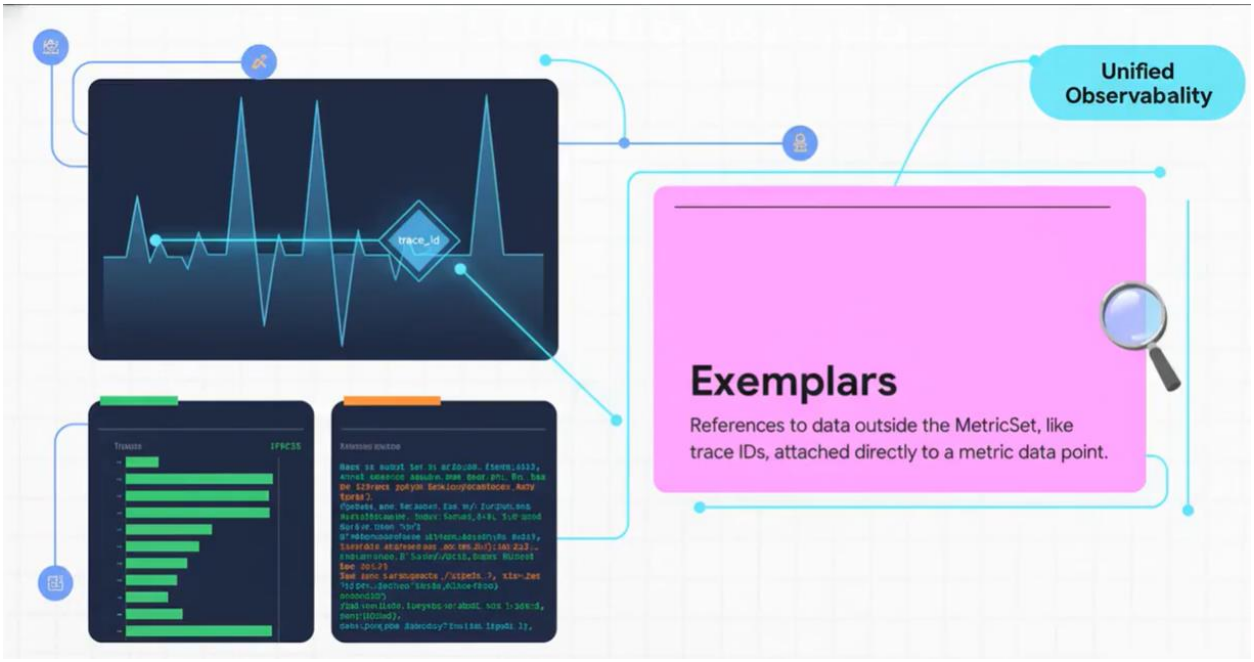

Metrics, logs, and traces are no longer siloed. They are linked together, telling a single cohesive story.
Logs
Complete Picture
Metrics
Traces


METRICS
The WHAT. Tells you that something is wrong (e.g., error rates, latency spikes).

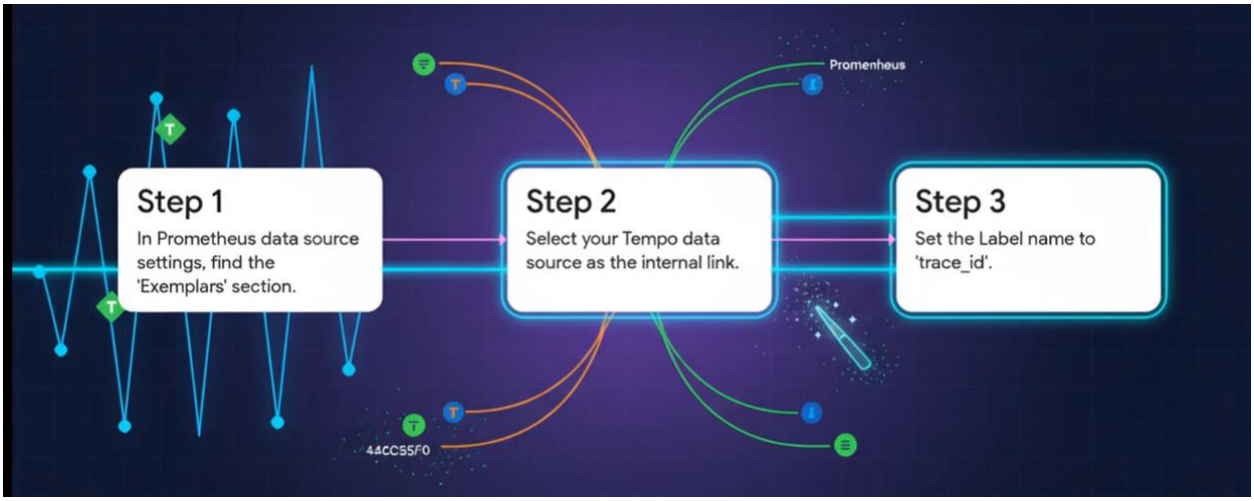

LOGS
The WHY. Provides context and error messages explaining the problem (e.g., "Connection timeout").
Series Finale
Gheware Devops AI
Gheware

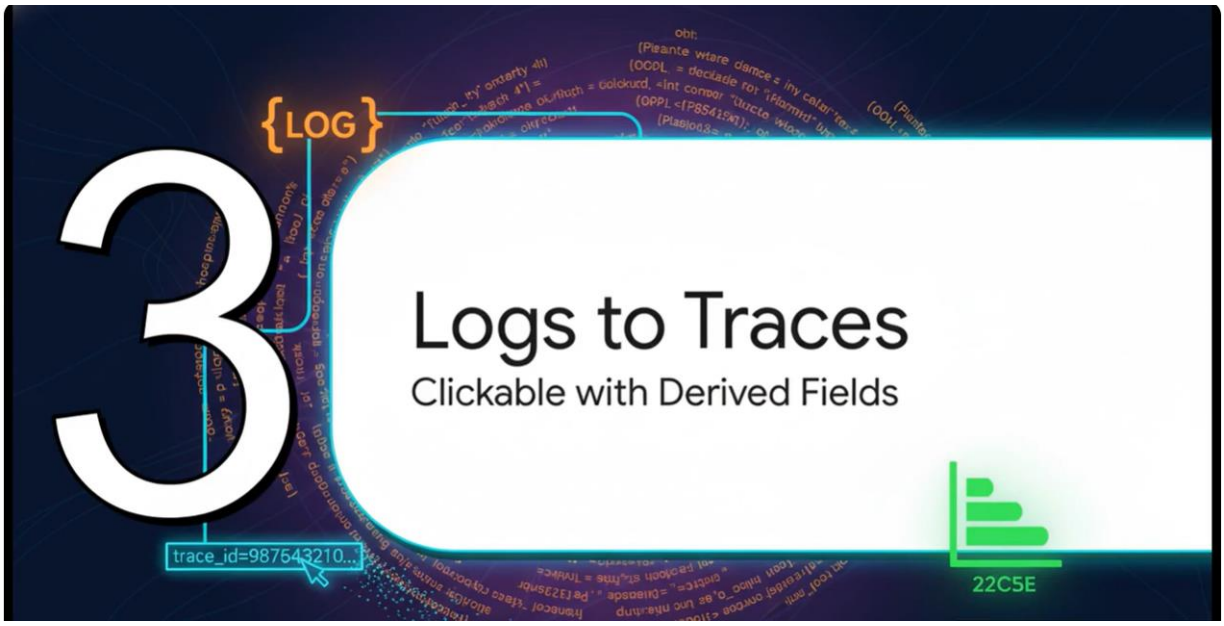

auth-service
auth-service
auth-service
TRACES
The WHERE. Pinpoints the exact service, function, or line of code where the failure occurred.

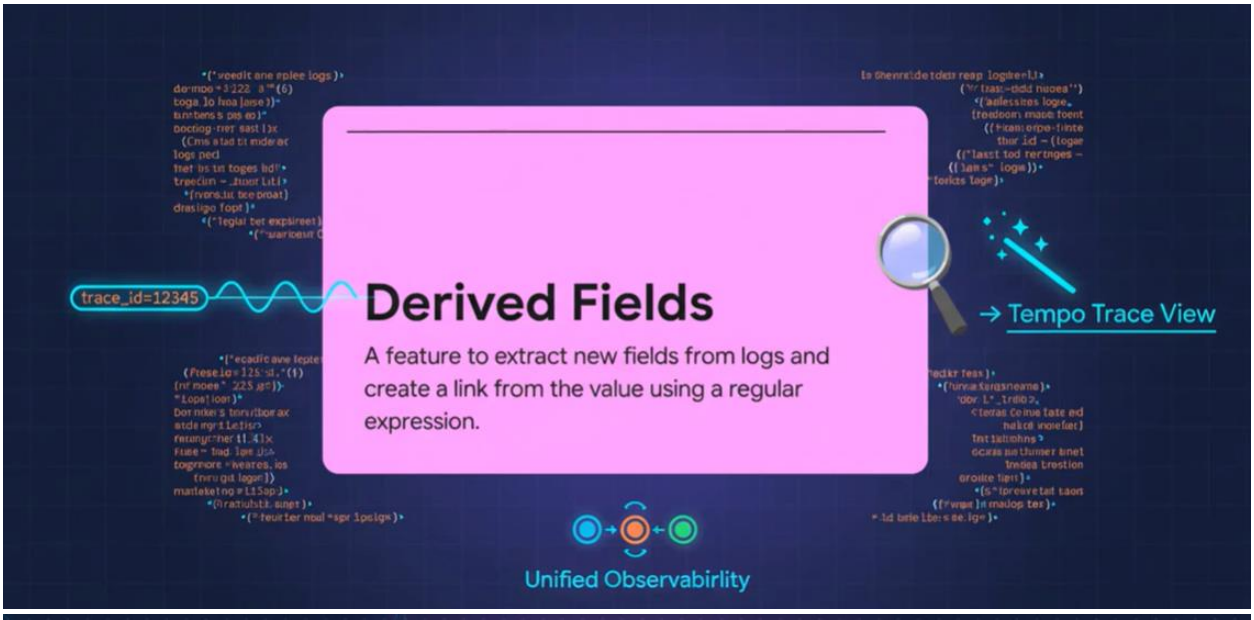

Devops AI
2
trace_id
trace_id
trace_id
Metrics to Traces
The Power of Exemplars

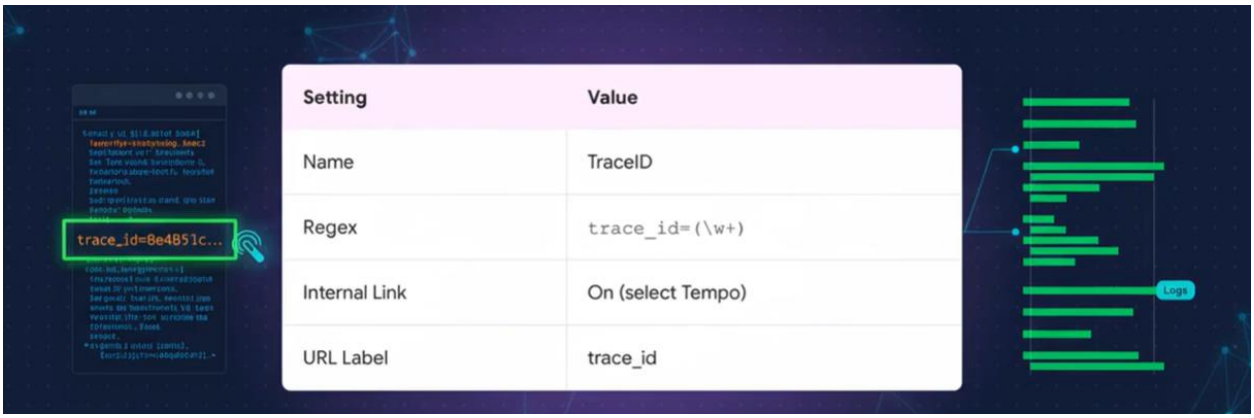

trace_id
Unified Observability
Exemplars
References to data outside the MetricSet, like trace IDs, attached directly to a metric data point.

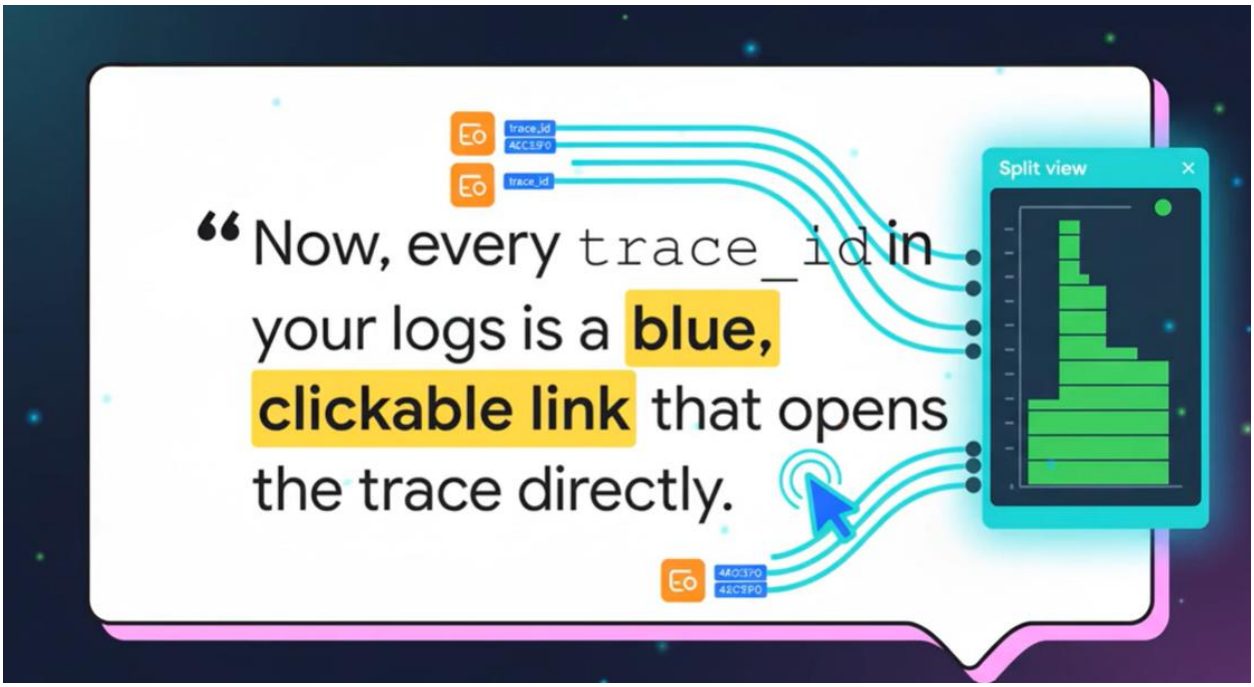
Enable this feature by adding the `--enable-feature=exemplar-storage` flag when starting your Prometheus server.




{LOG}
3
Logs to Traces
Clickable with Derived Fields
trace_id=987643210...
22C5E

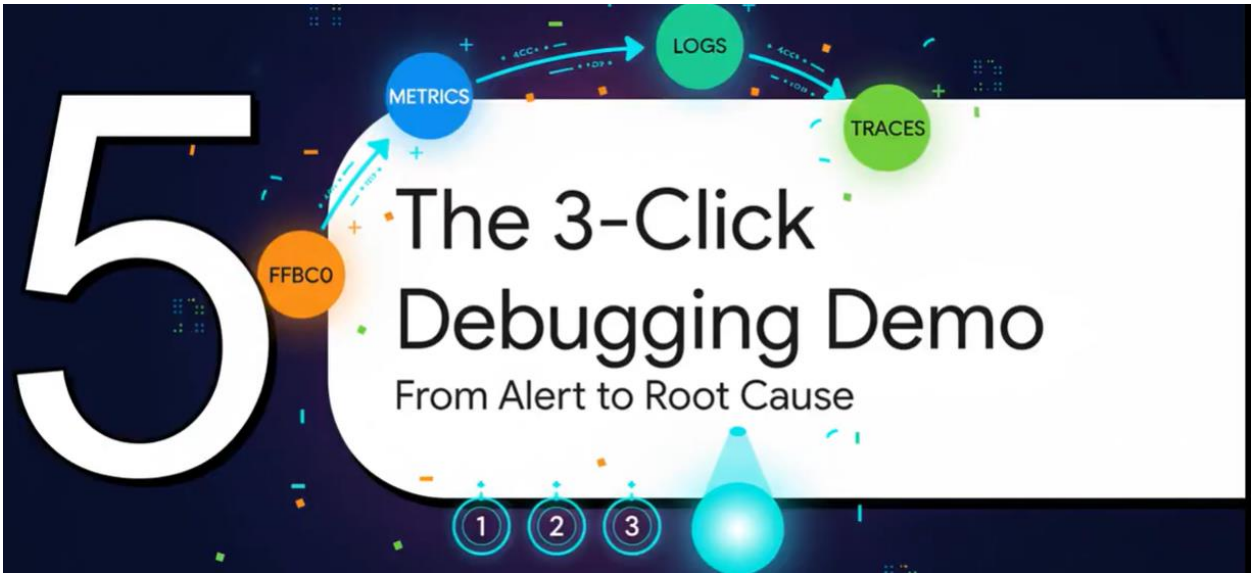

trace_id=12345
Derived Fields
A feature to extract new fields from logs and create a link from the value using a regular expression.
→ Tempo Trace View
Unified Observabirlity


trace_id=8e4851c...
Setting
Value
Name
TraceID
Regex
trace_id=(\w+)
Internal Link
On (select Tempo)
URL Label
trace_id
Logs


trace_id
trace_id
Split view
“Now, every trace_id in your logs is a blue, clickable link that opens the trace directly.


4
Trace
Traces to Logs
Completing the Circle
Logs
Completing the Circle


Tempo to Loki Config
Step 1
In Tempo data source settings, find the 'Trace to logs' section.
Step 2
Select your Loki data source as the target.
Step 3
Map tags to align trace attributes with log labels (e.g., service.name).
Step 4
Ensure 'Filter by trace ID' is toggled on.
42CC5F0


5
METRICS
LOGS
TRACES
FFBC0
The 3-Click Debugging Demo
From Alert to Root Cause
1
2
3

From Alert to Root Cause

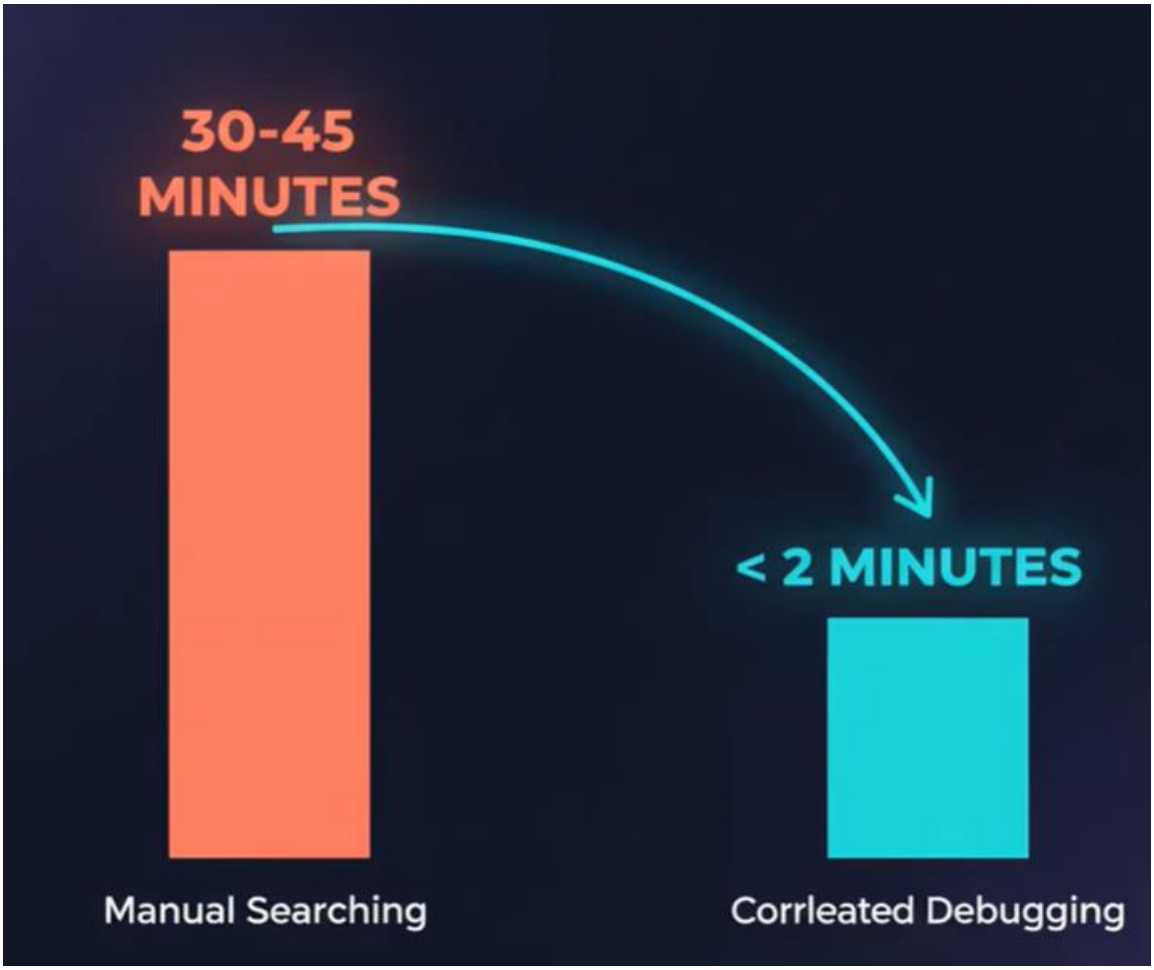
CLICK 1: Exemplar
Click the exemplar diamond on the metric spike to see the slow trace.

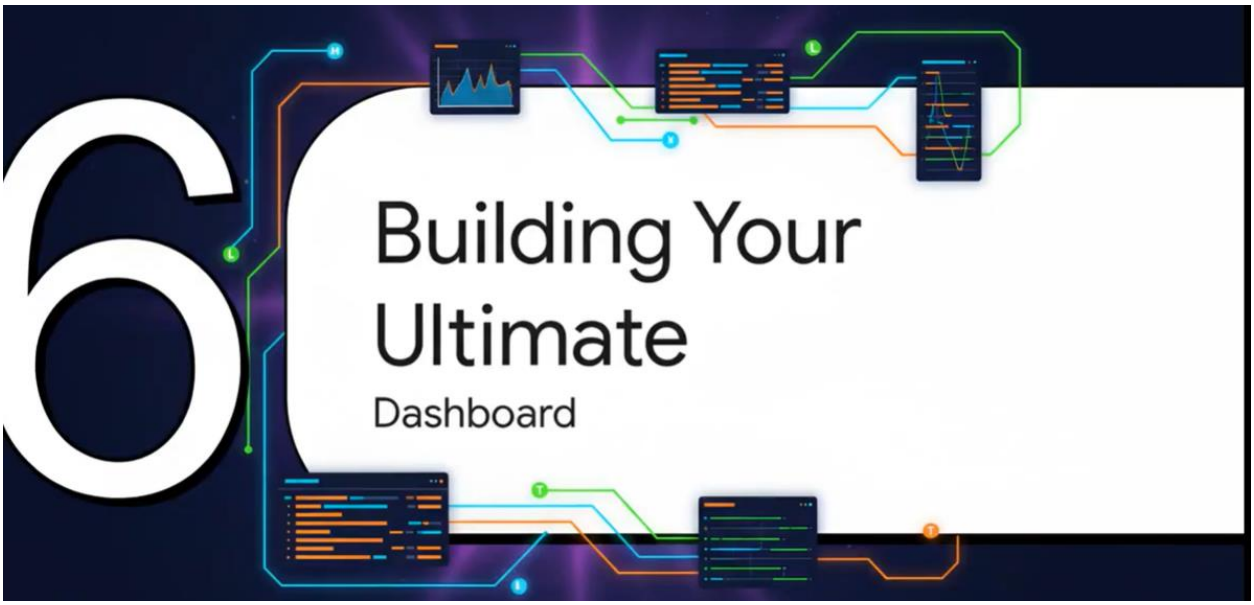
→

CLICK 2: Trace
The trace opens, showing a timeout in the Payment service span.

→

CLICK 3: Logs
Click 'Logs for this span' to see the exact error: "Connection timeout".




6
Building Your Ultimate
Dashboard


Metrics
Traces
service: checkout
Use consistent labels across all signals (metrics, logs, traces).
-00f5d0
Ensure trace_id is included in ALL application logs.
Dashboard Best Practices
Enable exemplars on key golden signal histograms.
Test your correlations before an incident happens.

Journey Complete!

- ✅ Prometheus for metrics
- ✅ Loki for logs
- ✅ Tempo for traces
- ✅ OpenTelemetry for instrumentation
- ✅ Correlation for instant debugging!