KloudMate

TECH @ KLOUDMATE / ENGINEERING NOTE

The Three Hardest Problems We've Had To Solve At KloudMate

Instrumentation without code changes. The cause behind alert bursts. PromQL queries in ClickHouse.

01 Instrumentation

02 Alert correlation

03 Metric queries

By Amitava Saha, CTO · September 29, 2026

KloudMate collects logs, metrics and traces from our customers' systems and tells them when something is broken. A lot of that is well understood engineering. Three problems were not.

They were not hard because the ideas were exotic. They were hard because the details were unforgiving, the edge cases were expensive, and the wrong answer could still look plausible. That is a nasty combination in observability, where a wrong chart often fails quietly.

The three problems sit at different points in the same pipeline:

- Getting data out of a customer's machine without turning setup into a project.
- Turning a burst of alerts into one explainable incident.
- Running PromQL correctly on ClickHouse without dragging raw data into the application.

The common thread: get the small decisions right, every time, at production scale.

//

Instrumenting apps without code changes

The first problem starts before we can show any value. If a customer's data does not reach us, nothing else matters.

On virtual machines, setup used to be painful. Each service needed tracing added by hand, language by language. Someone had to decide which process mattered, figure out how that runtime loads instrumentation, then edit collector YAML before seeing a single trace. For a team already trying to understand its own systems, that is a bad first experience.

The hard part was not "add an agent." It was that there is no single switch for a mixed estate.

Every language hooks in differently. Every host exposes different signals. Databases add another boundary. OpenTelemetry has no zero-code support for PHP 7, so those customers simply could not trace with us through the usual path. eBPF is powerful, but it needs Linux kernel 4.14 or newer and extra privileges. Many production hosts do not have both.

And because our agent runs on customer machines, the bar is higher than "works on our cluster." It must not crash the app. It must not eat CPU. It must not pretend a capability exists when the host cannot support it.

So we built km-agent, one agent per machine.

It discovers services and databases on the host, then immediately sends host metrics and logs. Where eBPF is available, it builds a service map showing which services talk to each other. Customers can turn on zero-code instrumentation per service from the UI. For PHP 7, the agent installs an open-source tracer that still supports it, and our collector converts that data into OpenTelemetry.

The important design choice was to make the best available path useful instead of making the ideal path mandatory. If eBPF is there, use it. If zero-code instrumentation is possible, offer it. If a legacy runtime needs a different tracer, support it and normalize the data. Customers should not need to know which of those paths they are on.

The result is simple: data starts flowing in minutes, and tracing becomes a product decision instead of a setup marathon.

Turning tracing on in the UI is not the same as having it active inside a running process; it starts the next time that service restarts. That restart may have to wait for a suitable production window. A host without the kernel or privileges for eBPF still gets the signals it can safely provide. We would rather show those steps and limits clearly than imply that a click can instrument a running process or unlock an unsupported feature.

//

Finding the cause behind alert bursts

One failed database can trigger dozens of alerts. The database complains. Every service that depends on it complains. Latency alerts fire. Error alerts fire. Saturation alerts fire. The on-call engineer gets a wall of noise and has to manually answer two questions: which alerts belong together, and where did the trouble start?

That sounds like a grouping problem. It is really an identity, topology and restraint problem.

Two alerts firing at the same time are not always related. Timing alone is not enough. To group alerts properly, we need to know what runs where, what owns what and what calls what. No single source gives us that.

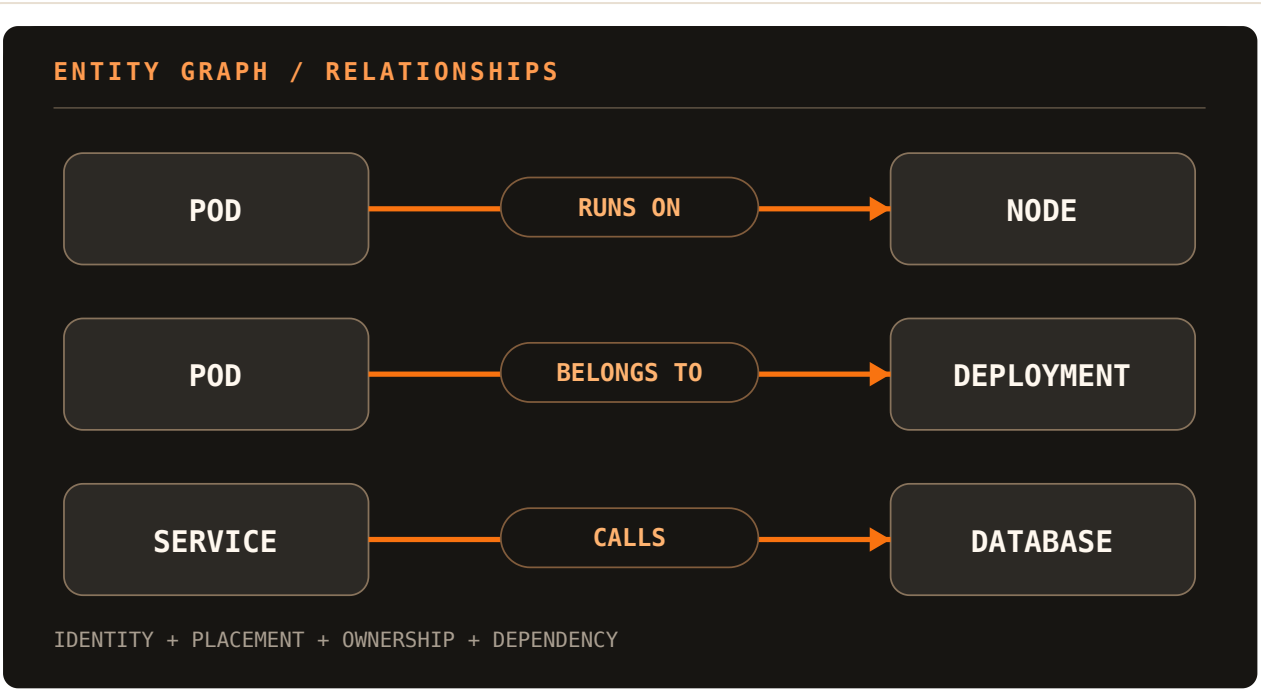
Cloud providers, operating systems and Kubernetes all name the same machine differently. If those records stay separate, dependencies look broken. If we merge records that should not be merged, the graph invents relationships. Both failures are bad. One hides context. The other creates false confidence.

Then there is the second trap: proximity is not causation. Two pods can share a node without sharing an incident. A host running hundreds of pods cannot become a reason to glue unrelated alerts together. If we get that wrong, we do not reduce noise. We hide a second incident inside the first one.

There is also no clean answer key. Customers do not label historical alerts by incident. New data sources show up every week, each with its own naming habits. So we could not train once and declare victory.

We built an entity graph and a correlation engine that reads it.

Entity resolution matches the same resource across sources. A cloud instance, its hostname and its Kubernetes node become one entity. The graph links entities by placement, ownership and dependency: a pod runs on a node, a pod belongs to a deployment, a service calls a database.



Entity resolution joins names for the same resource; edges keep placement, ownership and dependency separate. Sharing a node alone is not enough to join two alerts.

When the graph sees a new kind of data, an AI model reads its labels once and writes a rule. After that, ordinary code applies the rule. We backtest each new rule against past alerts and ship it only if it improves grouping. That keeps the model cost bounded and keeps the runtime behavior deterministic.

The outcome matters more than the mechanism. Related alerts arrive as one incident. Each alert records the signal that placed it there, so the grouping can be explained. A rule firing across 500 hosts stays one group instead of becoming 500 pages. Known hubs are ignored so shared infrastructure does not pull unrelated failures into the same story.

The correlation engine tries the strongest signals first: the same alert rule, a shared identity label, co-firing history, causal priors (known cause-and-effect patterns between kinds of alerts), and finally the topology. An AI model steps in only for alerts that no other signal explains. Shared timing or a busy node alone does not justify merging incidents.

The goal is not to be clever. It is to give the engineer one sane place to start.

//

Running PromQL correctly in ClickHouse

Our metric queries used to copy raw data points into the application and evaluate PromQL there. That worked until it did not. Big queries over many series or long time ranges became slow. Some failed.

The obvious answer was “push the query down to the database.” The hard answer was “push it down without changing what the query means.”

Prometheus has many small rules that are easy to miss and expensive to get wrong. Staleness decides when a series counts as gone. `rate()` extrapolates at the edges of a time window. If one of those decisions changes, the chart can still look smooth and still be wrong. A plausible chart is not a correctness test.

ClickHouse adds another constraint: it does not think in time series. Every PromQL step has to become ordinary SQL. And that SQL cannot solve the problem by materializing billions of intermediate rows. That only moves the bottleneck from the application to the database.

So we built a compiler that turns PromQL into ClickHouse SQL and pushes the whole query down. We also changed the metric storage layout so those queries read less data.

Neither half works alone. A fast translation with different answers is wrong. A faithful translation that explodes into too much data is useless for the queries that forced the work in the first place. The compiler has to preserve Prometheus semantics. The storage layout has to make that work practical near the data.

The customer-facing contract stays boring, which is exactly the point. Customers keep writing the PromQL they already know. The database does the work next to the data. Large intermediate results stop travelling to the application. The answers remain the answers Prometheus would have given.

That is the real obligation when you replace the engine under a familiar language: users should not have to rewrite expressions because your internals changed.

//

What these problems taught us

The hardest observability problems are rarely about inventing a new algorithm. They are about respecting boundaries.

Setup has to respect the customer's environment, not our preferred architecture. Correlation has to respect the difference between shared infrastructure and shared failure. Query execution has to respect the semantics of a language users already trust.

> In all three cases, the tempting shortcut was to make the system look smarter than it was. The better path was narrower: be explicit about what is known, be conservative about what is inferred, and make every important decision explainable.

We keep improving the KloudMate platform. km-agent is getting deeper support for Windows and Docker, the entity graph is adding network flow data, and metric queries are getting faster.
