Why Troubleshooting Dev Cloud-Native Environments Is So Damn Hard

Oh, I miss the good old days when servers and clients roamed freely across local networks, and root cause analysis was as easy as picking barriers from the tree. Nowadays, troubleshooting involves numerous infrastructure layers, servers are nowhere to be seen, and issues get introduced into the codebase faster than you can call the SRE.

How do you investigate a problem that happens once in a blue moon, with no apparent pattern, on a cloud-native Kubernetes dev environment used by 60 developers located across three different timezones? Identifying problems before they reach a production environment has never been trickier.

A root cause analysis canal

It's safe to assume that the application you're building is massively complex, and massively complex applications tend to lack visibility into transactions, containers, and microservices. Cloud-native systems are dynamic and scalable. Your application probably manages service performance and communication between numerous components, databases, and repositories. Add to that the countless pods, namespaces, and deployments integrated into your development and testing workflows, and you end up with one massive headache.

It doesn't get much more distributed than this (but if it can, we'll find a way)

Maintaining a holistic view is mandatory when troubleshooting complex systems, yet it's almost impossible when dealing with microservices. Understanding, not to mention recreating, the state of the system when a problem initially occurred is challenging at the least in distributed systems. When microservices run on different frameworks and nodes, intermittently interacting using asynchronous and sometimes unreliable communications, reproducing problems (especially latency issues) can seem impossible. To complicate things further, Kubernetes lacks breakpoints, rendering troubleshooting even harder.

But wait, you also have to monitor and collect information on CI/CD related issues, such as pipelines or jobs that run unexpectedly (or stop running altogether), as well as

warning and error handling. Has your head exploded yet? Hold on, there's more to come.

The shortcomings of available tools

Observability is a system attribute, and elastic software requires elastic observability. Yet, it's not entirely clear how to establish that with this relatively new technology.

Not only are classic monitoring tools unsuitable for cloud-native dev environments, but they were also designed to help troubleshoot in hindsight, as opposed to in real-time. Most APM, instrumentation tools, dev logging tools, and SRE tools are not container-aware nor track errors on the application level (unless it crashes entirely). They look for unusual performance patterns and for services that stop responding. Yet, distributed systems require more than log metrics and event monitoring. They need insights into how its users consume the application. It is not enough to identify exceptions – they must also be tracked continuously and in context.

Oh, the humanity!

DevOps-centric teams are always looking for valuable ways to increase application observability. But even the ideal troubleshooting software, designed for cloud-native distributed serverless environments, will not be accepted unless it's easily integrated into the workflow. If it causes too much fuss or if it requires additional coding, R&D will simply not use it.

As for the SREs, multiple dashboards can cause cognitive overload, as they continually look for patterns through magnitudes of data. Considering the human factor and organizational culture is a must when choosing a troubleshooting tool. Otherwise, you're doomed to fail.

In search of a single source of truth

Managing problems efficiently during the development stage will help you prepare to handle issues discovered by users in production (who mentioned proactivity? Certainly not I). A simple APM or SRE tool will not do, since troubleshooting complex systems' complex problems requires organizing data into a clear contextual storyline.

In the second part of this series, we'll explore the growing field of solutions and review its leaders. So cheer up, young Padawan, help is on the way!