DOI: 10.2507/34th.daaam.proceedings.xxx

INFERENCING – NEXT STEP OBSERVATION

Zeljko Seremet & Kresimir Rakic

This Publication has to be referred as: Seremet, Z[eljko] & Rakic, K[resimir] (2023). Inferencing – next step observation, Proceedings of the 34th DAAAM International Symposium, pp.xxxx-xxxx, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-08-2, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/34th.daaam.proceedings.xxx

Abstract

A production software system is observable to the extent that new internal system states can be understood without making arbitrary guesses, predicting those failure modes in advance or shipping new code to understand that state. In this way, the concept of observation of management theory is extended to the field of software engineering. But it is with modern distributed systems that the scalpel and searchlight afforded by observability tooling becomes absolutely nonnegotiable.
In distributed systems, the ratio of somewhat predictable failure modes to novel and never-before-seen failure modes is heavily weighted toward the bizarre and unpredictable. Those unpredictable failure modes happen so commonly, and repeat rarely enough, that they outpace the ability for most teams to set up appropriate and relevant enough monitoring dashboards to easily show that state to the engineering teams responsible for ensuring the continuous uptime, reliability, and acceptable performance of their production applications.
In article explored monitoring and observability and how they differ. It looked at how observability is poorly defined today with loose boundaries, which results in uncontrolled data, tool, and spend sprawl. Meanwhile, the latest progress in AI could resolve some of the observability problems we have today with a new class of Inferencing solutions based on AI.

Keywords: observability; AI; software; engineering; montoring

1. Introduction

In the software development industry, the topic of observation has generated a lot of interest and is often on hot new topic lists. Considering its applicability in the field of SRE, as a narrow subfield of software engineering, it is used when debugging at all levels of the developed system. But as things seem to inevitably go when a new hot topic sees a spike in adoption interest, complex ideas become too ripe for misunderstanding without a deeper look at the many nuances contained within a simple topic label. The aim of the article is to clarify the area of observability in detail. Beginning with the mathematical origins of the term "observability" to examining how software development professionals have adapted it to describe the characteristics of production software systems. **Error! Reference source not found. Error! Reference source not found. Error! Reference source not found. Error! Reference source not found.**

Furthermore, explains what “observability” means, how to determine if a software system is observable, why observability is necessary, and how observability is used to find problems in ways that are not possible with other approaches.

It also looks at why observability customization is needed for use in production software systems, and the underlying problem that drives the development of the next phase of inferencing.

2. The Mathematical Description of Observability

The term "observability" was coined by engineer Rudolf E. Kálmán in 1960. **Error! Reference source not found.** It has since grown to mean many different things in different communities. Let's explore the landscape before turning to our own definition for modern software systems.

In his 1960 paper, Kálmán introduced a characterization he called observability to describe mathematical control systems. In control theory, observability is defined as a measure of how well internal states of a system can be inferred from knowledge of its external outputs.

This definition of observability studies observability and controllability as mathematical duals, along with sensors, linear algebra equations, and formal methods. This traditional definition of observability is the realm of mechanical engineers and those who manage physical systems with a specific end state in mind. **Error! Reference source not found.**

In a textbook focused on mathematics and process engineering, observability has a formal meaning in traditional systems engineering terminology. However, when this same concept is adapted for use with narrower virtual software systems, it opens up a radically different way of interacting with and understanding written code.

3. Applying Observability to Software Systems

Kálmán's definition of observability can be applied to modern software systems. When adapting the concept of visibility to software, additional considerations specific to the domain of software engineering must also be considered. For a software application to have visibility, it must be able to do the following:

- Understand the inner workings of your application
- Understand any system state the application may have entered, even a new one that has never been seen before and could not have been predicted
- Understand the internal workings and state of the system solely by observation and testing using external tools
- Understanding the internal state without sending any new custom code to handle it (as this implies necessary prior knowledge to explain)

A good litmus test to determine if these conditions are true is to ask yourself the following questions:

- Can open questions about the inner workings of the applications be continuously answered to explain any anomalies without reaching investigative dead ends (ie the problem may be in a particular group of things but cannot be further disaggregated to confirm)?
- Can one understand what a particular user of the software might experience at any given time?
- Can any cross-section of system performance be quickly seen, from top-level summary views to individual and exact user requests that may be contributing to slowness (and anywhere in between)?
- Can arbitrary groups of user requests be compared in ways that allow one to correctly identify which attributes are commonly shared by all users experiencing unexpected behavior in the application?
- Once suspicious attributes are found within a single user request, can all user requests be searched to identify similar patterns of behavior to confirm or rule out suspicions?
- Can I identify which system user generates the most load (and therefore slows down the application the most), as well as the 2nd, 3rd or 100th user that generates the most load?
- Can it be identified which of those users generating the most load have only recently started to affect performance?
- If the 142nd slowest user complained about performance speed, can their requests be isolated to understand why exactly things are slow for that particular user?
- If users complain about timeouts, but graphs show that 99, 99.9, even 99.99 percent of requests are fast, can the hidden timeouts be found?
- Can questions like the previous ones be answered without first having to anticipate that one day they might have to change them (and therefore set up special monitors in advance to collect the necessary data)?
- Can such application questions be answered even if this problem has never been seen or debugged before?
- Can questions like the previous one be answered quickly, so that a new question can be iteratively asked, and then another, and another, until the real source of the problem is reached, without losing the train of thought (which usually means getting the answer within seconds instead of minutes)?
- Can questions like the above be answered even if the problem has never happened before?
- Can any fault in the system be quickly (within minutes) isolated, no matter how complex, deeply buried, or hidden within the stack?

Meeting all of the previous criteria is a high bar for many software engineering organizations to clear. If that tape can be removed, it will no doubt come to a conclusion as to why observation has become such a popular topic for software engineering teams.

Put simply, the definition of "observability" for software systems is a measure of how well one can understand and explain any state the system may get into, no matter how novel or bizarre. One must be able to comparatively debug that bizarre or novel state across all dimensions of system state data, and combinations of dimensions, in an ad hoc iterative investigation, without being required to define or predict those debugging needs in advance. If any can be understood any bizarre or novel state without needing to ship new code, the ability to observability is achieved.

Adapting the traditional concept of observability for software systems in this way is a unique approach with additional nuances worth exploring. For modern software systems, observability is not about the data types or inputs, nor is it about mathematical equations. It is about how people interact with and try to understand their complex systems. Therefore, observability requires recognizing the interaction between both people and technology to understand how those complex systems work together.

If this definition is accepted, many additional questions emerge that demand answers:

- How does one gather that data and assemble it for inspection?
- What are the technical requirements for processing that data?
- What team capabilities are necessary to benefit from that data?

The application of observability to software systems has much in common with its control theory roots. However, it is far less mathematical and much more practical. In part, that's because software engineering is a much younger and more rapidly evolving discipline than its more mature mechanical engineering predecessor. Production software systems are much less subject to formal proofs. That lack of rigor is, in part, a betrayal from the scars we, as an industry, have earned through operating the software code we write in production.

As engineers attempting to understand how to bridge the gap between theoretical practices encoded in clinical tests and the impacts of what happens when the code runs at scale, there was not looking for a new term, definition, or functionality to describe how it got there. It was the circumstances of managing systems and teams that led to the evolving practices away from concepts, like monitoring, that simply no longer worked. The industry must move beyond the current gaps in tools and terminology to overcome the pain and suffering caused by outages and the lack of more proactive solutions. **Error! Reference source not found. Error! Reference source not found.**

Observability is the solution to that gap. Complex production software systems are a mess for a variety of both technical and social reasons. Therefore, both social and technical solutions are needed to get out of this hole. Observability alone is not the entire solution to all software engineering problems. But it helps to see clearly what's happening in all the obscure corners of the software, where you are otherwise typically stumbling around in the dark and trying to understand things.

4. Why Observability Matters Now

The traditional approach of using metrics and monitoring of software to understand what it's doing falls drastically short. This approach is fundamentally reactive. It may have served the industry well in the past, but modern systems demand a better methodology.

For the past two or three decades, the space between hardware and its human operators has been regulated by a set of tools and conventions most call "monitoring." Practitioners have, by and large, inherited this set of tools and conventions and accepted it as the best approach for understanding that squishy virtual space between the physical and their code. And they have accepted this approach despite the knowledge that, in many cases, its inherent limitations have taken them hostage late into many sleepless nights of troubleshooting. Yet, they still grant it feelings of trust, and maybe even affection, because that captor is the best they have. **Error! Reference source not found.**

With monitoring, software developers can't fully see their systems. They squint at the systems. They try, in vain, to size them up and to predict all the myriad ways they could possibly fail. Then they watch—they monitor—for those known failure modes. They set performance thresholds and arbitrarily pronounce them "good" or "bad." They deploy a small robot army to check and recheck those thresholds on their behalf. They collect their findings into dashboards. They then organize themselves around those robots into teams, rotations, and escalations. When those robots tell them performance is bad, they alert themselves. Then, over time, they tend to those arbitrary thresholds like gardeners: pruning, tweaking, and fussing over the noisy signals they grow. **Error! Reference source not found.**

5. Is This Really the Best Way?

For decades, that's how developers and operators have done it. Monitoring has been the de facto approach for so long that they tend to think of it as the only way of understanding their systems, instead of just one way. Monitoring is such a default practice that it has become mostly invisible. As an industry, we generally don't question whether we should do it, but how.

The practice of monitoring is grounded in many unspoken assumptions about systems. But as systems continue to evolve—as they become more abstract and more complex, and as their underlying components begin to matter less and less—those assumptions become less true. As developers and operators continue to adopt modern approaches to deploying software systems (SaaS dependencies, container orchestration platforms, distributed systems, etc.), the cracks in those assumptions become more evident.

More people, therefore, are finding themselves slamming into the wall of inherent limitations and realizing that monitoring approaches simply do not work for the new modern world. Traditional monitoring practices are catastrophically ineffective for understanding systems. The assumptions of metrics and monitoring are now falling short. To understand why they fail, it helps to examine their history and intended context. **Error! Reference source not found. Error! Reference source not found.**

6. Why Are Metrics and Monitoring Not Enough?

In 1988, by way of the Simple Network Management Protocol (SNMPv1 as defined in RFC 1157), the foundational substrate of monitoring was born: the metric. A metric is a single number, with tags optionally appended for grouping and searching those numbers. Metrics are, by their very nature, disposable and cheap. They have a predictable storage footprint. They're easy to aggregate along regular time-series buckets. And, thus, the metric became the base unit for a generation or two of telemetry—the data collected from remote endpoints for automatic transmission to monitoring systems.

Many sophisticated apparatuses have been built a top the metric: time-series databases (TSDBs), statistical analyses, graphing libraries, fancy dashboards, on-call rotations, ops teams, escalation policies, and a plethora of ways to digest and respond to what a small army of robots is telling.

But an upper bound exists to the complexity of the systems you can understand with metrics and monitoring tools. And once you cross that boundary, the change is abrupt. What worked well enough last month simply does not work anymore. It starts back to low-level commands like strace, tcpdump, and hundreds of print statements to answer questions about how the system is behaving daily. **Error! Reference source not found.**

It's hard to calculate exactly when that tipping point will be reached. Eventually, the sheer number of possible states the system could get itself into will outstrip the team's ability to pattern-match based on prior outages. Too many brand-new, novel states are needing to be understood constantly. The team can no longer guess which dashboards should be created to display the innumerable failure modes.

Monitoring and metrics-based tools were built with certain assumptions about architecture and organization, assumptions that served in practice as a cap on complexity. These assumptions are usually invisible until they are exceeding them, at which point they cease to be hidden and become the bane of the ability to understand what's happening. Some of these assumptions might be as follows:

- The application is a monolith.
- There is one stateful data store ("the database"), that is running.
- Many low-level system metrics are available (e.g., resident memory, CPU load average).
- The application runs on containers, virtual machines (VMs), or bare metal
- System metrics and instrumentation metrics are the primary source of information for debugging code.
- A static and long-running set of nodes, containers, or hosts to monitor.
- Engineers examine systems for problems only after problems occur.
- Dashboards and telemetry exist to serve the needs of operations engineers.
- Monitoring examines "black-box" applications in much the same way as local applications.
- The focus of monitoring is uptime and failure prevention.
- Examination of correlation occurs across a limited (or small) number of dimensions.

When compared to the reality of modern systems, it becomes clear that traditional monitoring approaches fall short in several ways. The reality of modern systems is as follows:

- The application has many services.
- There is polyglot persistence (i.e., many databases and storage systems).
- Infrastructure is extremely dynamic, with capacity flicking in and out of exisence elastically.
- Many far-flung and loosely coupled services are managed.
- Engineers actively check to see how changes to production code behave, to catch tiny issues early, before they create user impact.
- Automatic instrumentation is insufficient for understanding what is happening in complex systems.
- Software engineers own their own code in production and are incentivized to proactively instrument their code and inspect the performance of new changes as they're deployed.
- The focus of reliability is on how to tolerate constant and continuous degradation, while building resiliency to user-impacting failures by utilizing constructs like error budget, quality of service, and user experience.
- Examination of correlation occurs across a virtually unlimited number of dimensions.

The last point is important, because it describes the breakdown that occurs between the limits of correlated knowledge that one human can be reasonably expected to think about and the reality of modern system architectures. So many possible dimensions are involved in discovering the underlying correlations behind performance issues that no human brain, and in fact no schema, can possibly contain them.

With observability, comparing high-dimensionality and high-cardinality data becomes a critical component of being able to discover otherwise hidden issues buried in complex system architectures. **Error! Reference source not found. Error! Reference source not found. Error! Reference source not found. Error! Reference source not found.**

7. The problem with observability

The fundamental problem with observability is — not knowing what information might be needed beforehand. The data required depends on the issue. The nature of production errors is that they are unexpected and long-tail: if they could've been foreseen, they'd have been fixed already.

This is what makes observability fuzzy: there's no clear scope around what and how much to capture. So, observability became "any data that could potentially help us understand what is happening".

Today, the best way to describe observability as it is implemented is — "Everything outside of metrics, plus metrics."

Fig. 1. Observability in practice **Error! Reference source not found.**

A perfectly observable system would record everything that happens in production, with no data gaps. Thankfully, that is impractical and prohibitively expensive, and 99% of the data would be irrelevant anyway, so an average observability platform needs to make complex choices on what and how much telemetry data to capture. Different vendors view this differently, and depending on who you ask, observability seems slightly different.

7.1. Commonly cited descriptions of observability are unhelpful

Common articulations of observability, like "observability is being able to observe internal states of a system through its external outputs", are vague and do not give a clear indication of what it is, nor guide in deciding whether there is sufficient observability for the purpose.

In addition, most of the commonly cited markers that purport to distinguish observability from monitoring are also vague, if not outright misleading. **Error! Reference source not found.Error! Reference source not found.**

8. Inferencing — the next stage after Observability?

To truly understand what the next generation would look like, it starts with the underlying goal of all tools, such as OpenTelemetry, AppDynamics, Datadog, Dynatrace, Grafana, Grafana, and so on. This is to keep production systems healthy and running as expected and, if anything goes wrong, to allow you to quickly understand why and resolve the issue.

That are three distinct levels in how tools can support:

- Level 1: "Tell me when something is off in my system" — monitoring
- Level 2: "Tell me why something is off (and how to fix it)" — let's call this inferencing
- Level 3: "Fix it yourself and tell me what you did" — auto-remediation

Traditional monitoring tools do Level 1 reasonably well and help us detect issues. We have not yet reached Level 2 where a system can automatically tell us why something is breaking. **Error! Reference source not found.**

So we introduced a set of tools called observability that sit somewhere between Level 1 and Level 2, to "*help understand why something is breaking*" by giving us more data. **Error! Reference source not found.**

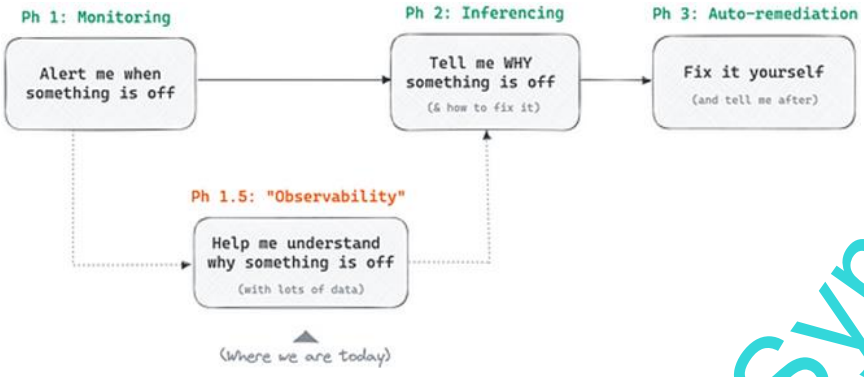

Fig. 2. Evolution of production troubleshooting systems **Error! Reference source not found.**

9. Inferencing — Observability plus AI

Over the last years, rapid advancements in Generative AI have impacted nearly every industry. Likewise in computer science, the next step after observability is inference — where the platform can reasonably explain why an error occurred, so we can fix it. This becomes possible now in 2023 with the rapid evolution of AI models over the last few months.

Imagine a solution that:

- Automatically surfaces just the errors that need immediate developer attention.
- Tells the developer exactly what is causing the issue and where the issue is: this pod, this server, this code path, this line of code, for this type of request.
- Guides the developer on how to fix it.
- Uses the developer's actual actions to improve its recommendations continuously.

There is some early activity in this space, but this is an open space yet and we can expect several new companies to emerge here over the next couple of years.

10. Conclusion

A production software system is observable to the extent that new internal system states can be understood without making arbitrary guesses, predicting those failure modes in advance, or shipping new code to understand that state. In this way, the concept of observation of management theory is extended to the field of software engineering.

In article explored monitoring and observability and how they differ. It looked at how observability is poorly defined today with loose boundaries, which results in uncontrolled data, tool, and spend sprawl. Meanwhile, the latest advances in artificial intelligence could solve some of the observability issues (eg why something is off and how to fix it) with a new class of Inferencing solutions based on AI.

Future work will investigate the shortcomings of Inferencing solutions based on AI on real applications.

11. References

[1] Seremet, Z., & Rakic, K. (2022). Platform Engineering and Site Reliability Engineering: The Path to DevOps Success, DAAAM International Scientific Book 2022, Vol. 21, ISSN 1726-9687, ISBN: 978-3-902734-34-1, Editor: B. Katalinic hard cover, Publisher DAAAM International Vienna, Vienna
[2] Majors, C., Fong-Jones, L. & Miranda, G. (2022). Observability Engineering, Published by O'Reilly Media
[3] Russ, M. (2019). Chaos Engineering Observability, first ed., O'Reilly Media, Inc
[4] Sridharan, C. (2018). Distributed Systems Observability, first ed., O'Reilly Media, Inc
[5] Kálmán, R. E. (1960). On the General Theory of Control Systems, IFAC Proceedings Volumes 1, no. 1: 491–502.
[6] Grobmann, M. & Klug. C. (2017). Monitoring container services at the network edge, in: 29th International Teletraffic Congress, ITC 29, ITC Press, pp. 130–133.
[7] Samyukktha, T. (2022). Inferencing: The AI-Led Future of Observability?, Available from: https://dzone.com/articles/inferencing-the-ai-led-future-of-observability Accessed: 2022-09-20

[8] Sheldon, R. (2023). Top observability tools for 2023, Available from: https://www.techtarget.com/searchitoperations/tip/Top-observability-tools, Accessed: 2023-09-20