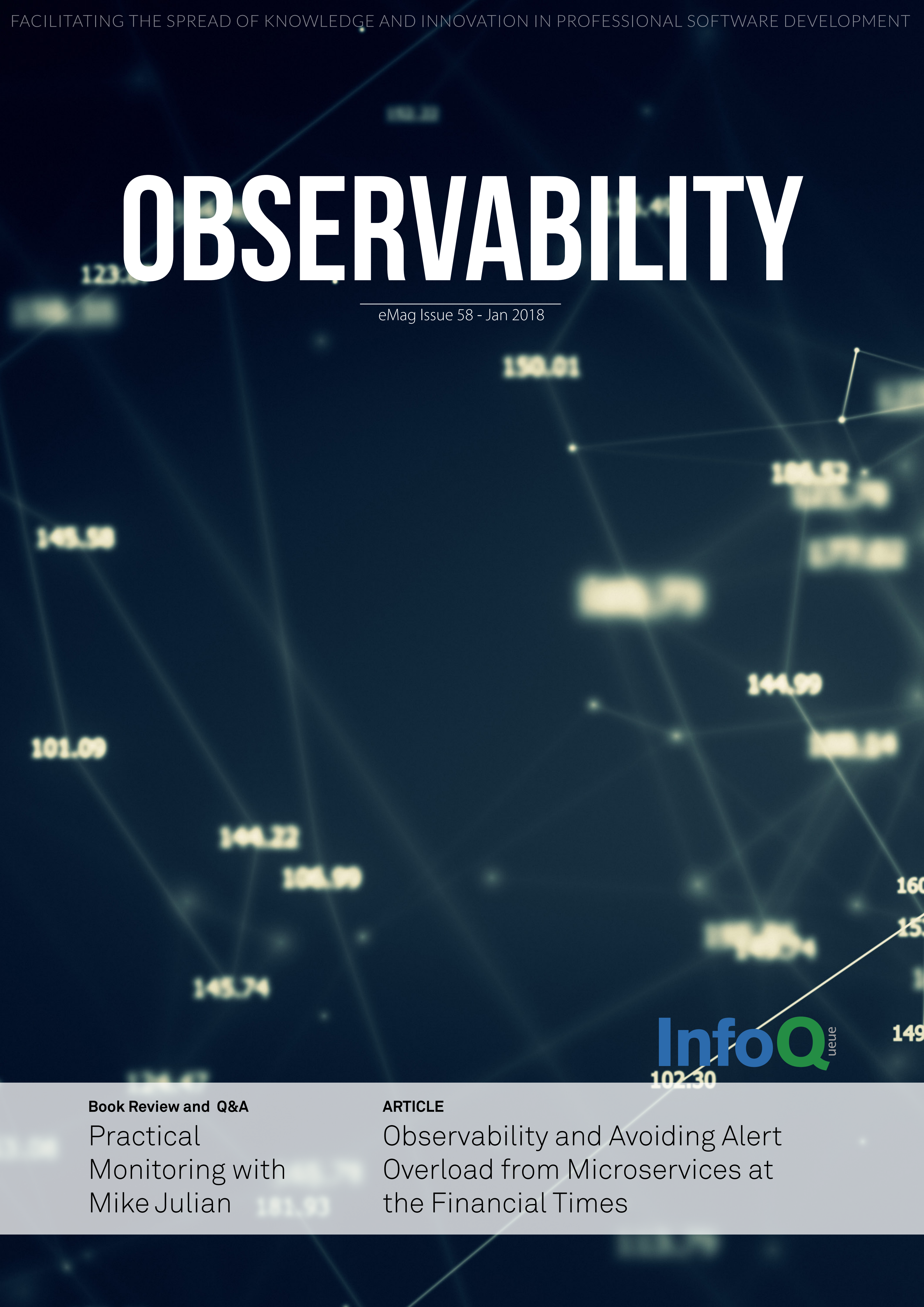
FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

OBSERVABILITY

eMag Issue 58 - Jan 2018



Book Review and Q&A

Practical Monitoring with Mike Julian

ARTICLE

Observability and Avoiding Alert Overload from Microservices at the Financial Times

IN THIS ISSUE

FOLLOW US

facebook.com /InfoQ

@InfoQ

google.com /+InfoQ

linkedin.com company/infoq

CONTACT US

GENERAL FEEDBACK feedback@infoq.com
ADVERTISING sales@infoq.com
EDITORIAL editors@infoq.com

A LETTER FROM THE EDITOR

Daniel Bryant

The topic of "observability" has been getting much attention recently, particularly in relation to building and operating "cloud native" systems. Several thought-leaders within this space like Cindy Sridharan have mused that observability could simply be a re-packaging of the age-old topic of monitoring (and argued that no amount of "observability" or "monitoring" tooling can ever be a substitute to good engineering intuition and instincts). Others, like Charity Majors have looked back at the roots of the term, which was taken from control theory and corresponds to a measure of how well internal states of a system can be inferred from knowledge of its external outputs. Both Sridharan and Majors discuss that the implementation of an observable systems should enable engineers to ask ad hoc (or following an incident, post hoc) questions about how the software works during execution. This emag explores the topic of observability in-depth, covering the role of the "three pillars of observability" -- monitoring, logging, and distributed tracing -- and relates these topics to designing and operating software systems based around modern architectural styles like microservices and serverless.

The first article, "Practical Monitoring: Book Review and Q&A with Mike Julian", provides a foundational introduction to the topic of monitoring, and presents an overview of core principles, monitoring antipatterns, and monitoring design patterns. The next article, a Q&A with Kresten Krab Thorup, explores "The Value of Logging within Cloud Native Applications", and argues that aggregating logs from diverse components or services that make up a running system provides an excellent way to monitor, debug and understand modern software systems. The third article completes the exploration of the three pillars of observability by examining the past, present and future of distributed tracing. InfoQ sat down with Ben Sigelman, co-author of the original Google Dapper tracing paper and co-founder of LightStep, and discussed distributed tracing benefits -- the identification of performance bottlenecks and the ability to "drill-down" into specific requests -- and challenges -- making sense of the trace data, and the processing of extremely high volumes of generated trace data.

The second half of this emag focuses on practical use cases, and explores the impact of observability for a modern software architect. First, Charity Majors discusses "Observability and Understanding the Operational Ramifications of a System", and argues that the health of the system no longer matters -- we have entered an era where the health of each individual event, or each individual user's experience, is what truly matters. The next article summarises a thought-provoking talk from Sarah Wells at QCon London last year, "Observability and Avoiding Alert Overload from Microservices at the Financial Times". In order to adapt to the challenges of monitoring a microservices-based application, Wells suggested a three-pronged approach: build a system that can be supported; concentrate on "stuff that matters"; and cultivate alerts and the information they contain. The final two articles, featuring Uwe Friedrichsen and Idit Levine, focus on designing and debugging modern architectures using the principles of observability -- monitoring and logging systems must evolve in ways that reflect current software architecture, and this has an impact that is both technical and cognitive.

The topic of observability is rapidly evolving, and so this emag aims to generate discussion and exploration. At InfoQ we are always keen to encourage the submission of articles that further the conversation: editors@infoq.com

CONTRIBUTORS

Daniel Bryant

is leading change within organisations and technology. His current work includes enabling agility within organisations by introducing better requirement gathering and planning techniques, focusing on the relevance of architecture within agile development, and facilitating continuous integration/delivery. Daniel's current technical expertise focuses on 'DevOps' tooling, cloud/container platforms and microservice implementations.

Mike Julian

is a consultant who helps companies build better monitoring for their applications and infrastructure. He is editor of Monitoring Weekly, an online publication about all things monitoring. Julian has previously worked as an operations/DevOps engineer for Taos Consulting, Peak Hosting, Oak Ridge National Lab, and others. He is originally from Knoxville, Tenn. and currently resides in San Francisco. Outside of work, he spends his time driving mountain roads in a classic BMW, reading, and traveling. You can find him at Mike Julian, Aster Labs, and Monitoring Weekly.

Kresten Krab

provides technical leadership and vision at Humio. In his previous role as CTO of Trifork, Kresten was responsible for technical strategy and provided consulting advice to teams on a variety of technologies to include distributed systems and databases, Erlang, Java, and mobile-application development. Kresten has been a contributor to several open-source projects, including GCC, GNU Objective-C, GNU Compiled Java, Emacs, and Apache Geronimo/Yoko. Prior to Trifork, Kresten worked at NeXT Software (now acquired by Apple), where he was responsible for the development of the Objective-C tool chain, the debugger, and the runtime system. Kresten has a Ph.D. in computer science from University of Aarhus.

Idit Levine

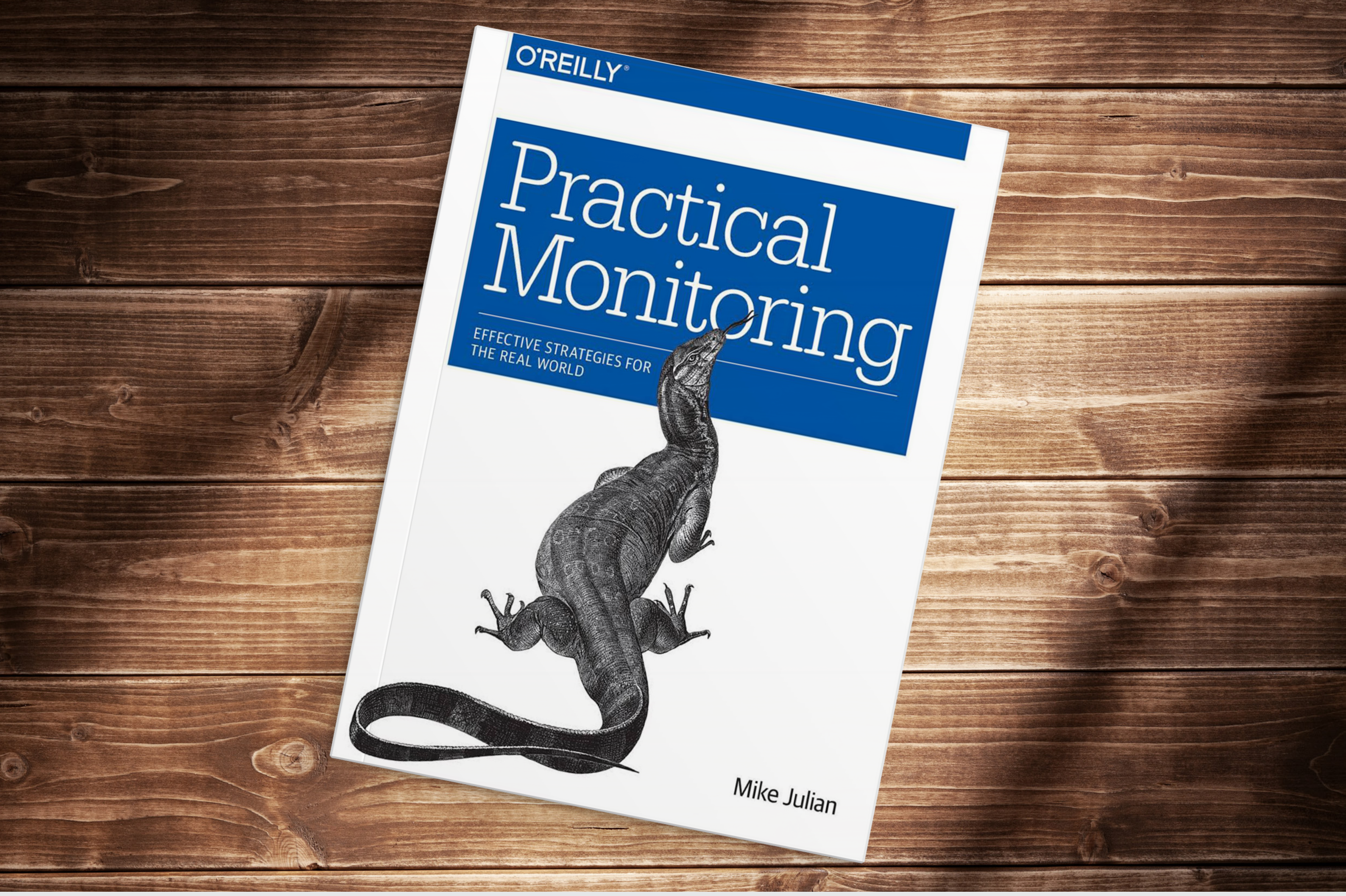
Founder/Leader/Contributor on a variety of Cloud open source Projects. Expert in cluster management like: Kubernetes, Mesos & DockerSwam. Hybrid cloud: AWS, Google Cloud, OpenStack, Xen & vSphere Comfortable with Cloud Foundry and a laundry list of other frameworks and tools.

Charity Majors

is a cofounder and engineer at Honeycomb, a startup that blends the speed of time series with the raw power of rich events to give you interactive, iterative debugging of complex systems. She has worked at companies like Facebook, Parse, and Linden Lab, as a systems engineer and engineering manager, but always seems to end up responsible for the databases too. She loves free speech, free software, and a nice peaty single malt.

Uwe Friedrichsen

is CTO of Codecentric AG, where he focuses on resilience, scalability, and the IT of (the day after) tomorrow. He has traveled the IT world for many years and is always in search of innovative ideas and concepts. Often, you can find him sharing ideas at conferences or in his many articles, blog posts, and tweets.



Read online on InfoQ

KEY TAKEAWAYS

Mike Julian's new book, Practical Monitoring, provides a foundational introduction to the topic of monitoring, and presents an overview of core principles, monitoring antipatterns, and monitoring design patterns.

Monitoring is an action -- a thing you do -- while observability is an attribute of a system that enables monitoring. The more observable a system is, the better you can monitor it, the better you can reason about it, the better you can simply understand how it works.

The most common antipattern, and most insidious one, is constantly looking for the next hot tool that's going to solve all your problems.

When it comes to business metrics it is crucial that everyone at least understands what these metrics are, why they matter, and how the app/infrastructure makes them available.

Practical Monitoring Book Review and Q&A with Mike Julian

by **Daniel Bryant**

Mike Julian's recently published Practical Monitoring (O'Reilly) aims to provide readers with a foundational introduction to the topic of monitoring as well as practical guidelines on how to monitor service-based applications and cloud infrastructure.

Julian discusses in the preface that the monitoring landscape of today is vastly different than it was only a few years ago. With the widespread popularity of ephemeral cloud infrastructure and architectural approaches like microservices came new problems for monitoring and created new ways to solve old problems. The book aims to address these issues, and also answer common monitoring questions such as: Do you have a nagging feeling that your monitoring needs improvement but you're just not sure where to start or how to do it? Are you plagued by constant, meaningless alerts? Does your monitoring system routinely miss real problems?

Practical Monitoring is focused on readers who seek a foundational understanding of monitoring. The preface states that it is suitable for junior staff as well as non-technical staff looking to learn about monitoring and warns "if you already have a great grasp on monitoring, this probably is not the book for you". Although Julian introduces and discusses many modern monitoring tools such as StatsD, InfluxDB, Prometheus, and Sensu, he does not inspect specific tools but instead focuses on practical, real-world examples on how such tools should be deployed within a holistic approach to monitoring.

The book begins with an overview of monitoring principles, and looks at monitoring anti-patterns as well as current good practices in monitoring design patterns. The anti-patterns of tool obsession, monitoring as a job, and checkbox monitoring reinforce Julian's argument that any approach to monitoring should be holistic. The best-practice patterns he presents — such as composable monitoring, monitor from the user perspective, and continual improvement — also demonstrate the influence of modern software-engineering approaches, such as a focus on modularity, cultivating a user-centric approach, and principles from lean. Julian also covers how to create effective alerting, and discusses the associated people and organizational challenges of being on call and managing incidents. This section of the book concludes with a basic primer of the use of statistics within monitoring, and covers the use of mean, average, and median, as well as quantiles and standard deviation.

The remainder of *Practical Monitoring* covers monitoring tactics, and includes a discussion of monitoring from the perspective of both the business and technology. The chapter on monitoring the business discusses concepts such as key performance indicators (KPIs), and provides techniques to identify and capture these. The book presents real-world use cases that help the reader understand the explanations and guidance. Monitoring tactics from the technology perspective is provided within a chapter for each of following: front end, application (back end), server, network, and security.

The concluding chapter examines how to conduct a monitoring assessment, and revisits key concepts from the rest of the book with a focus on how to identify associated current strengths and weaknesses within an organization. Julian suggests how to prioritize monitoring efforts, and leaves a clear message for the reader that "monitoring is never done, since the business, application, and infrastructure will continue to evolve over time".

Practical Monitoring is available to purchase via the companion website and also Safari books. InfoQ recently sat down with the author to find out more about his motivations for writing the book.

InfoQ: Could you introduce yourself and say a little about your motivation for writing *Practical Monitoring?*

Mike Julian: I'm a former operations engineer turned business owner. I run a consulting company called Aster Labs where I focus on helping companies improve their monitoring. I'm the editor of Monitoring Weekly, a weekly e-mail newsletter about all things monitoring. I'm also involved in a few other projects, which you can find on my personal site, mikejulian.com.

Every time I was at an event, pub, or coffee shop and someone would find out that I'm "the monitoring guy," the very next question would be something along the lines of, "My monitoring stinks. What should I do?" or my personal favorite, "What's the best monitoring tool these days?" After the bajillionth time answering the same questions with the same answers, I decided to just write a book about it all. Specifically, I wanted a book that wasn't oriented around how to use this tool or that tool and instead talked about the principals behind monitoring. And thus, Practical Monitoring was born.

InfoQ: Your book makes good use of design patterns, which many developers can relate to. Can we ask why you chose this approach?

Julian: You know, it was actually accidental. I begin the book with what I think is the most important topic: anti-patterns and things not to do. After writing that chapter, I realized I should probably tell people what they should do, too, making Chapter 2: "Monitoring Design Patterns" a thing. I think it worked out quite well. Ultimately, it fits very well with my tool-agnostic approach to monitoring in that you should focus on good patterns, avoid bad ones, and everything else will fall into place.

InfoQ: Can you explain a little about how operational and infrastructure monitoring has evolved over the last five years? How have cloud, containers, new data-store technologies and new language runtimes impacted monitoring?

Julian: The rise of ephemeral infrastructure (e.g., containers, short-lived cloud instances, serverless) and distributed architectures has drastically changed how we do monitoring. Even five years ago, Graphite and StatsD were still cutting-edge tools, and emitting metrics from inside the app was a novel idea for many teams. Nowadays, not only is such a setup commonplace, but many teams are finding it insufficient.

Specifically, we're now talking about how to handle millions of metrics, how to monitor code that exists for fractions of seconds, how to effectively trace requests through hundreds of microservices, and more. I think these problems transcend languages and storage back ends, and speak more directly to how we reason about the systems we build. This is certainly a much harder (and more interesting!) problem than monitoring, say, the latest NoSQL data store.

InfoQ: What role do QA/testers have in relation to monitoring and observability of a system, both from a business and operational perspective?

Julian: I think it's actually a mixed bag for QA teams: as applications and systems become more observable and capable of checking and reporting on their own health/functionality, the role of QA diminishes significantly.

On the other hand, QA is in a great position to work with engineering on what metrics and health checks the app needs to make the QA team's job easier and more automated. Certainly, there are some aspects of a system that can't easily be automated for testing, but those that can be automated should be. QA is in the best position to say how that should look.

InfoQ: How important it is for engineers to understand statistics in relation to monitoring? Can you recommend key things to learn?

Julian: You can get surprisingly far with very basic statistics. A cursory understanding of the use and limitations of averages, median, and percentiles really solves a lot of the use cases the typical engineer is likely to encounter. For example, one of the most misunderstood statistical concepts is that of the percentile and the limitations on it. If you record the 90th percentile of a dataset every week over 12 weeks and then average those 12 data points together, the answer is inaccurate (because a percentile is intentionally losing data). In order to calculate the 90th percentile for a 12-week period, you'd need to have the full 12 weeks of data.

If you want a book about stats that's more approachable than your college textbook, I recommend Naked Statistics by Charles Wheelan, which I really enjoyed reading during my research.

InfoQ: What is the most common monitoring anti-pattern you see? Can you recommend an approach to avoid this?

Julian: The most common one, and most insidious one, is constantly looking for the next hot tool that's going to solve all problems (Chapter 1's "Anti-pattern #1: Tool Obsession"). You can read more about it in the book, but the quick version is that there is no magic here and teams have done quite well with awful tools. I've seen plenty of teams using the latest tools and miserably fail to build any effective monitoring. As the old saying goes, a craftsperson doesn't blame their tools for bad work.

The solution is to recognize that your tools probably aren't the problem, and that you need to look much deeper at what you're actually doing, how you're monitoring apps and infrastructure, and why you think your monitoring isn't very good. There's a 99% chance that your tools are fine, and in fact your strategy is to blame.

InfoQ: There has been some great discussion recently about monitoring versus observability from engineers like Cindy Sridharan and Charity Majors. What are your thoughts on this?

Julian: I think Cindy is brilliant and totally on point about all of it. If I really had to sum it up, I'd put it this way: monitoring is an action — a thing you do — while observability is an attribute of a system that enables monitoring (credit to Baron Schwartz for that take on it).

Many of you have no doubt been in the situation where you're trying to monitor some homegrown application only to realize it's a black box with no logs or metrics — that's an unobservable system. The more observable a system is, the better you can monitor it, the better you can reason about it, the better you can simply understand how it works. Really, improving observability is a matter of improving the application.

I don't talk about observability much in the book and instead conflate it with monitoring. That was intentional. Observability versus monitoring is a nuanced topic and not one that really matters when you're just getting started with monitoring. I imagine that once your monitoring matures, the concept of observability will begin to matter a lot more to you and your team.

InfoQ: You talk about business metrics and KPIs in the book. Who do you believe is most responsible for ensuring these are implemented: product owners, developers, or operators? Or is it a team effort, and if so, how should everyone work together?

Julian: It's really a team effort, though everyone has a different role to play. For example, let's take the example of user growth over time on a SaaS app. Product owners/managers define that this is something they care about, and developers write the code to make reporting on that data easy. In a more complex scenario, technical operations/system administrators will have a role: the cost to service a user. Calculating how much your infrastructure costs per user is a great way to eventually increase profit margins, but is also helpful to understand if your current infrastructure is tenable or not. For example, if the cost to provide service to a customer outpaces the revenue from a customer, you've got a bit of a problem on your hands, and this kind of data is something that system administrators will have (or can calculate) and which the business is often just guessing at.

No matter who does what, when it comes to business metrics, I think it's crucial that everyone at least understands what these metrics are, why they matter, and how the app/infrastructure makes them available.

InfoQ: Can you share any tactics for an engineer that wants to understand and implement KPIs for the business? Where is the best source of KPIs, and how should engineers present results to the business?

Julian: Sit down with a product manager or your nearest VP and ask them a few questions: How does the business make money? How do we know if we're doing well or doing poorly? What are the targets for those metrics?

You'll get a great sense of how the business actually functions and what matters. You can follow it up with one last question: What data that you don't currently have would make decisions easier? Sometimes you can help with that problem, sometimes you can't.

Either way, having a better understanding of how the business works and what data is used to judge the health of the company is always valuable.

InfoQ: Thanks for taking the time to sit down with us today. Is there anything else you would like to share with the InfoQ readers?

Julian: Thank you! It's been a pleasure. The last thing I want to say is this: improving monitoring is a journey, and often a long one. Improve a small amount every day and you'll do fine, but don't expect a major overhaul overnight or even by next month.

Further information on the book can be found on this website, and also also on Safari.

```
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif _operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

    #selection at the end -add back the deselected mirror modifier object
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier ob is the active ob
    #mirror_ob.select = 0
```

Read online on InfoQ

The Value of Logging Within Cloud-Native Applications

Q&A with Kresten Krab

by **Daniel Bryant**

InfoQ recently sat down with Kresten Krab, CTO at Humio, to discuss the role of logging within the overall topic of system observability.

Krab began by stating that cloud and container technology provide a lot of advantages, but at the cost of understandability — which is potentially the best way to view the term "observability". The discussion covered many topics, but a key theme is that aggregating logs from diverse components or services that make up a running system provides an excellent way to monitor, debug, and understand modern software systems.

KEY TAKEAWAYS

Aggregating logs from diverse components or services that make up a running system provides an excellent way to monitor, debug and understand modern software systems.

For debugging or incident response, you need a system that makes it easy to do ad-hoc queries; it is important to have a logging solution that does not impose a schema on what you log.

Logs naturally evolve from a more verbose level, to a more structured and better information-to-noise ratio level. Neglecting to cultivate this evolution this is an anti-pattern.

The future impact of Artificial Intelligence (AI) and Machine Learning (ML) in the logging space will likely be big. For now, focus on getting logs at the fingertips of developers to let them interact with them and employ the human intelligence to provide interpretation.

InfoQ: Could you introduce yourself and say a little about your current work at Humio, please?

Kresten Krab: For the last two years, I've been CTO at Humio, a startup that we launched to provide a better way for DevOps teams to understand their systems. Twenty years ago, I co-founded Trifork, which is now a bespoke software-solution provider with 400+ employees, where our mission has always been to help other companies succeed with new technology.

I've been involved with teams implementing new technology, training, and building conferences to spread the knowledge. As part of this, we've seen the ever-increasing complexity of software systems being built all the way from the first web-enabling projects in the late '90s to today's complex cloud solutions, and I've seen the struggle in teams trying to understand, debug, and monitor their systems in production.

So we observed that aggregating logs from diverse components or services that make up a running system provides an excellent way to monitor, debug, and understand these systems. At Humio, we refer to this as the ability to "feel the hum of your system". Logs are a great "lowest common denominator" point of integration for understanding a system because logs are already there. You don't need to augment existing systems to make them generate logs: they are already generating logs and you just need to gather them and put them on a shared timeline of events.

We have found that existing providers of log-management tools require you to limit your logging — whether it is costs, quota limitations, complexity, or performance — and we thought we could do better. So you can say we're on a mission to democratize logging. Humio is the product we're building from the ground up to let everyone share this insight.

InfoQ: There has been some discussion recently about monitoring versus observability. How does logging relate?

Krab: We welcome this discussion very much. The term "observability" fits well with our mantra of "feeling the hum of your system". I don't think there is a "versus" discussion here; the term "observability" covers a spectrum of information being emitted by a system. The spectrum goes from detailed logs of events or traces (being the most verbose and rich in information) to traditional monitoring events or aggregated stats. You can derive the stats from the events, but not the other way around. An excellent read on this spectrum of information is also Cindy Sridharan's blog post on "Monitoring in the Time of Cloud Native".

So, if you're only gathering metrics in the traditional way of monitoring, then you're throwing information away. This works well if the system being monitored is mature and well understood but that is usually not the case when you're building a system in the first place. In other words, you often don't know what will be causing calamities, so having a richer base to search is a huge advantage. It enables you to go back in time and search for patterns that you only now realize are important.

Our vantage point is that all these sources of information fit well into a time-series text data storage with a rich query capability, and that it is a huge advantage to be able to process both event-style information and metrics-style information in a shared tool without having to limit yourself to debug and understand a system in terms of what you thought was important. If you've ever had an "I wish I indexed that" moment then you know what I mean.

InfoQ: Can you explain a little about how operational and infrastructure logging has evolved over the last five years? How have cloud, containers, and new language runtimes impacted monitoring and logging?

Krab: Software nowadays is no longer a single body of code you can build and test in isolation. Cloud, containers, and all this tech obviously provide a lot of advantages, but at the cost of "understandability" (which is maybe the best way to view the term "observability"). The system components are increasingly scattered and remote, and less likely to be under your direct control. This evolution goes hand in hand with the DevOps movement, which has changed the way people think about software. There are a lot of teams now who have a system they "care about", as opposed to just building a piece of software and "throwing it over the wall to ops".

So, understanding the behavior of your software system is now largely only possible in the wild. Most software systems are a composition of other systems that are out of your control. Think of your software system as an autonomous car; it has to be put on the road to be tested and improved, but in many ways we're still building software as if we could test it in the lab. This is the honeycomb

```
{
  "time": "2017-06-14T20:44:04+00:00",
  "Location": {
    "Region": "us-east-1",
    "Zone": "us-east-1a",
    "Host": "10.0.0.4",
    "Service": "dogpics",
    "Version": "1.0.0",
    "Endpoint": "/dogs/search"
    "Params": "good dogs"
  },
  "User": {
    "User_id": "some_guid",
    "Originating_Host": "94.100.180.199",
    "Remote_Host": "10.0.0.35",
    "User_Agent": "dogsview"
  }
  "Response Code": 503,
  "Latency": 5046,
  "Message": "Deadline exceeded fetching results from
search store 10.0.0.10",
  "Suspect" "10.0.0.10"
}
```

"When investigating a problem, I want to know if it's caused by a *particular* customer.

Being able to break down metrics on higher-cardinality fields gives very actionable insights."

Try it FREE FOR 30 DAYS at

https://honeycomb.io/signup

impact of cloud and containers on our software systems, and I think we have to come to terms with this to deal with it head on.

InfoQ: How have new architectural styles such as microservices and function as a service (FaaS), which are in effect distributed systems, impacted logging?

Krab: In terms of exposing the resource consumption of the individual components, and being able to improve parts of your system individually, I think these are wins. But in terms of understanding your system as a whole — in particular, if you don't capture and centralize your logging — these mostly contribute adversely to the big picture because information is scattered across diverse platforms and components.

It is daunting to speak for logging in general, but I can say what we do at Humio in this space. For platforms such as DC/OS, Mesos, Kubernetes, Heroku, Cloud Foundry, AWS, etc., we provide integrations that make it simple to grab all the logs and put them in one place. On each of these platforms, logging is more or less done in an uniform way, and that lets a logging infrastructure capture a wide range of logs without a lot of configuration. So, with these architectural styles, where you have your system in a shared infrastructure, you can now get the logs as a side effect of using the platform, which simplifies getting access to them.

InfoQ: What types of query are engineers typically making of logging systems, and how are modern logging platforms adapting to this?

Krab: We see our users going through an evolution. At first, they make free text searches, using the logging platform as a search engine for their logs. But then, quickly the focus changes to extracting information from text and building aggregations over that extracted data.

As you get to know your logs, engineers will develop metrics — aggregate queries — that are important for the health of the system. These are the queries that make up dashboards and the input data for alerting. For a majority of systems, you can live with these aggregate stats being computed from the logs as opposed to be built into the system itself as a monitoring metric.

For debugging or incident response, you need a system that makes it easy to do ad hoc queries, which makes it important to have a logging solution that does not impose a schema on what you log. In these situations, we generally see engineers asking questions about things that they did not think about up front.

An interesting thing is the feedback loop that happens when developers realize that they can interact with the logs. You see that logs evolve incrementally and becoming more structured as you try to debug or improve your system. The word "incremental" is important here, because you cannot build the perfect set of logs for your system from day one. So, you end up refactoring your logs: new subsystems log more verbosely, and as a subsystem matures, you tend to reduce the information-to-noise ratio in the logs. So, your logging platform should be able to cope with this diversity.

InfoQ: What is the most common logging anti-pattern you see? Can you recommend an approach to avoid it?

Krab: Well, the thing that hurts my heart is to hear stories of someone unable to log (or, god forbid, being unable to access logs) because of quota, cost, or company policy. We see customers who reduce their logging by sampling data at ingest. This can be necessary at scale, but should be avoided as far as possible.

I mentioned the refactoring of logs before. Logs naturally evolve from a more verbose level to a more structured level with a better information-to-noise ratio. This process is like weeding your garden, and I'd consider it an anti-pattern to neglect doing this.

InfoQ: What role do you think QA/testers have in relation to the observability of a system, particularly in relation to logging?

Krab: Logs are super useful for testing and QA. As part of our own automated tests and CI setup, we capture logs from the builds and run queries over these as part of acceptance, as well as reporting and alerting for the builds. In this way, you can use log aggregation to construct integration tests as well as a means to improve performance and general quality of your tests.

InfoQ: What will the impact of AI/ML be on logging, both in regards to implementing effective logging and also providing insight into issues (or potential issues)?

Krab: I think this will likely be big. For now, we focus on getting logs at the fingertips of developers to let them interact with them and employ the human intelligence to provide interpretation. AI/ML generally requires a baseline to allow you to identify outliers, and as a system that generates logs stabilizes, the logging platform will be able to provide this baseline. The richness of logging and the high entropy in logs do provide a challenge for both AI and ML, as they tend to do better in a low-dimensionality setting.

I think that believing that a logging system can auto-magically detect outliers in arbitrary logs is an impossibility. You need some sort of interaction where users of the system extract and generalize information in the logging flow that is deemed interesting for outlier detection. For particular kinds of logs, this may be achieved more or less automatically, but in the general case, you will need some kind of data scientist's capacity to choose what to look out for.

InfoQ: Thanks for taking the time to sit down with us today. Is there anything else you would like to share with the InfoQ readers?

Krab: Thank you too. Feel free to swing by humio.com, and try our SaaS solution or ask us to how to run Humio on your own gear. We are always keen to discuss these ideas about logging in more depth!

Distributed Tracing: Exploring the Past, Present and Future with Dapper, Zipkin and LightStep [x]PM

Distributing tracing is increasingly seen as an essential component for observing distributed systems and microservice applications.

KEY TAKEAWAYS

Distributing tracing is increasingly seen as an essential component for observing distributed systems and microservice applications. There are several popular open source standards and frameworks like the OpenTracing API and OpenZipkin

The basic idea behind distributed tracing is relatively straightforward -- specific request inflexion points must be identified within a system and instrumented. All of the trace data must be coordinated and collated to provide a meaningful view of a request

Request tracing is similar in concept to Application Performance Management (APM). An emerging challenge is processing the volume of the data generated from increasingly large-scale systems

Google overcame this issue when implementing their Dapper distributed tracing system by sampling traces, typically 1 in 1000, but modern commercial tracing products claim to be able to analyse 100% of requests.

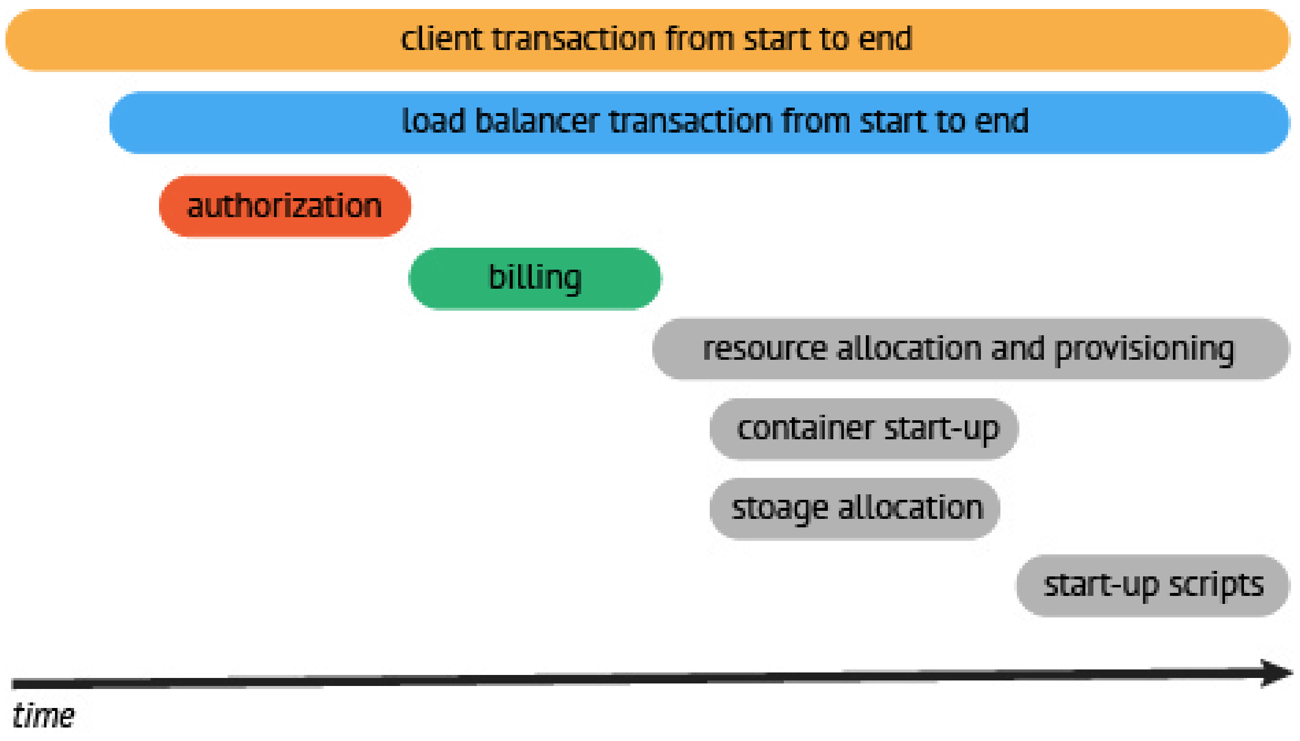

client transaction from start to end
load balancer transaction from start to end
authorization
billing
resource allocation and provisioning
container start-up
stoage allocation
start-up scripts
time

Figure 1. Visualising a basic trace with a series of spans over the lifetime of a request (image taken from the OpenTracing documentation)

This article provides an introduction to and overview of this technique, starting with an exploration of Google's Dapper request tracing paper -- which in turn led to the creation of the Zipkin and OpenTracing projects -- and ending with a discussion of the future of tracing with Ben Sigelman, creator of the new LightStep [x]PM tracing platform.

As stated in the original Dapper paper, modern Internet services are often implemented as complex, large-scale distributed systems -- for example, using the popular microservice architectural style. These applications are assembled from collections of services that may be developed by different teams, and perhaps using different programming languages. At Google-scale these application span thousands of machines across multiple facilities, but even for relatively small cloud computing use cases it is now recommended practice to run multiple versions of a service spread across geographic "availability zones" and "regions". Tools that aid in understanding system behaviour, help with debugging, and enable reasoning about performance issues are invaluable in such a complex system and environment.

The basic idea behind request tracing is relatively straightforward: specific inflexion points must be identified within a system, application, network, and middleware -- or indeed any point on a path of a (typically user-initiated) request -- and instrumented. These points are of particular interest as they typically represent forks in execution flow, such as the parallelization of processing using multiple threads, a computation being made asynchronously, or an out-of-process network call being made. All of the independently generated trace data must be collected, coordinated and collated to provide a meaningful view of a request's flow through the system. Cindy Sridharan has provided a very useful guide that explores the fundamentals of request tracing, and also places this technique in the context of the other two pillars of modern monitoring and "observability": logging and metrics collection.

Decomposing a Trace

As defined by the Cloud Native Computing Foundation (CNCF) OpenTracing API project, a trace tells the story of a transaction or workflow as it propagates through a system. In OpenTracing and Dapper, a trace is a directed acyclic graph (DAG) of "spans", which are also called segments within some tools, such as AWS X-Ray. Spans are named and timed operations that represent a contiguous segment of work in that trace. Additional contextual annotations (metadata, or "baggage") can be added to a span by a component being instrumented -- for example, an application developer may use a tracing SDK to add arbitrary key-value items to a current span. It should be noted that adding annotation data is inherently intrusive: the component making the annotations must be aware of the presence of a tracing framework.

Trace data is typically collected "out of band" by pulling locally written data files (generated via an agent or daemon) via a separate network process to a centralised store, in much the same fashion as currently occurs with log and metrics collection. Trace data is not added to the request itself, because this allows the size and semantics of the request to be left unchanged, and locally stored data can be pulled when it is convenient.

When a request is initiated a "parent" span is generated, which in turn can have causal and temporal relationships with "child" spans. Figure 1, taken from the OpenTracing documentation, shows a common visualisation of a series of spans and their relationship within a request flow. This type of visualisation adds the context of time, the hierarchy of the services involved, and the serial or parallel nature of the process/task execution. This view helps to highlight the system's critical path, and can provide a starting point for identifying bottlenecks or areas to improve. Many distributed tracing systems also provide an API or UI to allow further "drill down" into the details of each span.

The Challenges of Implementing Distributed Tracing

Historically it has been challenging to implement request tracing with a heterogeneous distributed system. For example, a microservices architecture implemented using multiple programming languages may not share a common point of instrumentation. Both Google and Twitter were able to implement tracing by creating Dapper and Zipkin (respectively) with relative ease because the majority of their inter-process (inter-service) communication occurred via a homogenous RPC framework -- Google had created Stubby (a variant of which has been released as the open source gRPC) and Twitter had created Finagle.

The Dapper paper makes clear that the value of tracing is only realised through (1) ubiquitous deployment -- i.e. no parts of the system under observation are not instrumented, or "dark" -- and (2) continuous monitoring -- i.e. the system must be monitoring constantly, as unusual events of interest are often difficult to reproduce.

The rise in popularity of "service mesh" network proxies like Envoy, Linkerd, and Conduit (and associated control planes like Istio) may facilitate the adoption of tracing within heterogeneous distributed systems, as they can provide the missing common point of instrumentation. Sridharan discusses this concept further in her Medium post discussing observability:

"Lyft famously got tracing support for all of their applications without changing a single line of code by adopting the service mesh pattern [using their Envoy proxy]. Service meshes help with the DRYing of observability by implementing tracing and stats collections at the mesh level, which allows one to treat individual services as blackboxes but still get incredible observability onto the mesh as a whole."

The Need For Speed: Request Tracing and APM

Web page load speed can dramatically affect user behaviour and conversion. Google ran a latency experiment using its search engine and discovered that by adding 100 to 400 ms delay to the display of the results page resulted in a measurable impact on the number of searches a user ran. Greg Linden commented in 2006 that experiments ran by Amazon.com demonstrated a significant drop in revenue was experienced when 100ms delay to page load was added. Although understanding the flow of a web request through a system can be challenging, there can be significant commercial gains if performance bottlenecks are identified and eliminated.

Request tracing is similar in concept to Application Performance Management (APM) -- both are related to the monitoring and management of performance and availability of software applications. APM aims to detect and diagnose complex application performance problems to maintain an expected Service Level Agreement (SLA). As modern software architectures have become increasingly distributed, APM tooling has adapted to monitor (and visualise) this. Figure 2 shows a visualisation from the open source Pinpoint APM tool, and similar views can be found in commercial tooling like Dynatrace APM and New Relic APM.

An emerging challenge within the request tracing and APM space is processing the volume of the data generated from increasingly large-scale systems. As stated by Adrian Cockcroft, VP of Cloud Architecture Strategy at AWS, public cloud may have democratised access to powerful and scalable infrastructure and services, but monitoring systems must be more available (and more scalable) than the systems that they are monitoring. Google overcame this issue when implementing Dapper by sampling traces, typically 1 in 1000, and still found that meaningful insight could be generated with this rate. Many engineers and thought leaders working within the space -- including Charity Majors, CEO of Honeycomb, an observability platform -- believe that samping of monitoring data is essential:

"It's this simple: if you don't sample, you don't scale.

If you think this is even a controversial statement, you have never dealt with observability at scale OR you have done it wastefully and poorly."

InfoQ recently attended the CNCF CloudNativeCon in Austin, USA, and sat down with Ben Sigelman, one of the authors of the original Dapper paper and CEO and co-founder of LightStep, who has recently announced a new commercial tracing platform, "LightStep [x]PM". Sigelman discussed that LightStep's unconventional architecture (which utilises machine learning techniques within locally installed agents) allows the analysis of 100.0% of transaction data rather than 0.01% that was implemented with Dapper:

"What we built was (and is still) essential for long-term performance analysis, but in order to contend with the scale of the systems being monitored, Dapper only centrally recorded 0.01% of the performance data; this meant that it was challenging to apply to certain use cases, such as real-time incident response (i.e., "most firefighting").

LightStep have worked with a number of customers over the

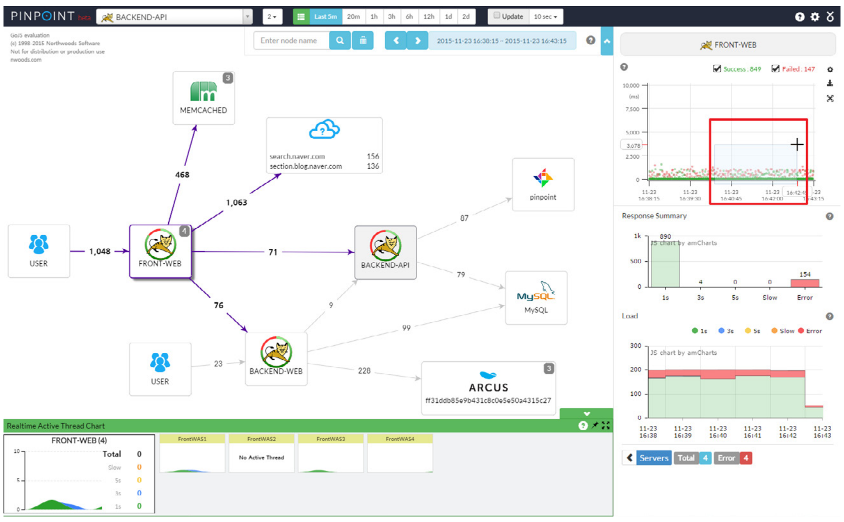


Figure 2. Request tracing within modern APM tooling (image taken from the Pinpoint APM GitHub repository)

past 18 months -- including Lyft (utilising the Envoy proxy as an integration point), Twilio, GitHub, and DigitalOcean -- and have demonstrated that their solution is capable of handling high volumes of data:

"Lyft sends us a vast amount of data – LightStep analyzes 100,000,000,000 microservice calls every day. At first glance, that data is all noise and no signal: overwhelming and uncorrelated. Yet by considering the entirety of it, LightStep can measure how performance affects different aspects of Lyft's business, then explain issues and anomalies using end-to-end traces that extend from their mobile apps to the bottom of their microservices stack."

LightStep [x]PM is currently available as a SaaS platform, and Sigelman was keen to stress that although 100% of requests can be analysed, not all of this data is exfiltrated from the locally installed agents to the centralised platform. Sigelman sees this product as a "new age APM" tool, which will provide value to customers looking for performance monitoring and automated root cause analysis of complex distributed applications.

Conclusion

Response latency within distributed systems can have significant commercial impact, but understanding the flow of a request through a complex system and identifying bottlenecks can be challenging. The use of distributed tracing -- in combination with other techniques like logging and monitoring metrics -- can provide insight into distributed applications like those created using the microservices architecture pattern. Open standards and tooling are emerging within the space of distributed tracing -- like the OpenTracing API and OpenZipkin -- and commercial tooling is also emerging which potentially competes with existing APM offerings. There are several challenges with implementing distributed tracing for modern Internet services, such as processing the high volume of trace data and generating meaningful insight, but both the open source ecosystem and vendors are rising to the challenge.

Read online on InfoQ

KEY TAKEAWAYS

The current best-practice approaches to developing software -- microservices, containers, cloud native -- are all ways of coping with massively more complex systems. However, our approach to monitoring has not kept pace.

Majors argues that the health of the system no longer matters. We've entered an era where what matters is the health of each individual event, or each individual user's experience (or other high cardinality dimensions).

Don't attempt to "monitor everything". In the chaotic future we're all hurtling toward, you actually have to have the discipline to have radically fewer paging alerts -- not more.

Many of us don't have the problems of large distributed systems. If you can get away with a monolith and a LAMP stack and a handful of monitoring checks, Majors suggests that you should absolutely do that.

Charity Majors on Observability and Understanding the Operational Ramifications of a System

by **Daniel Bryant**

InfoQ recently sat down with Charity Majors, CEO of Honeycomb and co-author (with Laine Campbell) of *Database Reliability Engineering*, to discuss the topics of observability and monitoring.

InfoQ: Could you introduce yourself and share a little about your experience of monitoring systems, especially data-store technologies?

Charity Majors: I'm an operations engineer, co-founder, and (wholly accidentally) CEO of Honeycomb. I've been on call for various corners of the Internet ever since I was 17 years old: university, Second Life, Parse, Facebook. I've always gravitated towards operations because I love chaos and data because I have a god complex. I do my best work when the material is critical, unpredictable, and dangerously high stakes. Actually, when I put it that way, maybe it was inevitable for me to end up as CEO of a startup….

One thing I have never loved, though, is monitoring. I've always avoided that side of the room. I will prototype and build v1 of a system, or I will deep dive and debug or put right a system, but I steer away from the stodgy areas of building out metrics and dashboards and curating monitoring checks. It doesn't help that I'm not so capable when it comes to visualization and graphs.

InfoQ: How have operational and infrastructure monitoring evolved over the last five years? How have cloud, containers, and new (old) modular architectures impacted monitoring?

Majors: Oh man. There's a tidal wave of technological change that's been gaining momentum over the past five years. Microservices, containers, cloud native, schedulers, serverless... all these movements are ways of coping with massively more complex systems (driven by Moore's law, the mobile diaspora, and the platform-ization of technical products). The center of gravity is moving relentlessly to the generalist software engineer, who now sits in the middle of all these APIs for in-house services and third-party services. And their one job is to craft a usable piece of software out of the center of this storm.

What's interesting is that monitoring hasn't really changed. Not in the past... 20 years.

You've still got metrics, dashboards, and logs. You've got much better ones! But monitoring is a very stable set of tools and techniques, with well-known edge cases and best practices, all geared around monitoring and making sure the system is still in a known good state.

However, I would argue that the health of the system no longer matters. We've entered an era where what matters is the health of each individual event or each individual user's experience or each shopping cart's experience (or other high cardinality dimensions). With distributed systems, you don't care about the health of the system, you care about the health of the event or the slice.

This is why you're seeing people talk about observability instead of monitoring, about unknown unknowns instead of known unknowns, and about distributed tracing, honeycomb, and other event-level tools aimed at describing the internal state of the system to external observers.

InfoQ: How has the approach to monitoring data-store technologies changed over the last few years?

Majors: Databases and networks were the last two priesthoods of system specialists. They had their own special tooling, inside language, and specialists, and they didn't really belong to the engineering org. That time is over. Now, you have roles like database-reliability engineer (DBRE), which acknowledges the deep specialist knowledge while also wrapping them into the fold of continuous integration/continuous deployment, code review, and infrastructure automation.

This goes for monitoring and observability tooling as well. Tools create silos. If you want your engineering org to be cross-functionally literate, if you want a shared on-call rotation, you have to use the same tools to debug and understand your databases as you do the rest of your stack. That's why Honeycomb and other next-generation services focus on providing an software-agnostic interface for ingesting data. Anything you can turn into a data structure, we can help you debug and explore. This is such a powerful leap forward for engineering teams.

InfoQ: With the rise in popularity of DBaaS technologies like AWS RDS and Google Spanner, has the importance of monitoring database technologies risen or fallen? And what has been the impact for the end users/operators?

Majors: Monitoring isn't really the point. I outsource most of my monitoring to AWS, and it's terrific! We use RDS and Aurora at Honeycomb despite being quite good at databases ourselves, because it isn't our core competency. If AWS goes down, let them get paged.

Where that doesn't let me off the hook is observability, instrumentation, and architecture. We have architected our system to be resilient to as many problems as possible, including an AWS Availability Zone going down. We have instrumented our code and we slurp lots of internal performance information out of MySQL, so that we can ask any arbitrary question of our stack — including databases. This rich ecosystem of introspection and instrumentation is not particularly biased towards the traditional monitoring stack's concerns of actionable alerts and outages.

It will always be the engineer's responsibility to understand the operational ramifications and failure models of what we're building, auto-remediate the ones we can, fail gracefully where we can't, and shift as much operational load to the providers whose core competency it is as humanly possible. But honestly, databases are just another piece of software. In the future, you want to treat databases as much like stateless services as possible (while recognizing that, operably speaking, they aren't) and as much like the rest of your stack as possible.

InfoQ: What role do QA/testers have in relation to monitoring and observability of a system, both from a business and operational perspective? Should the QA team be involved with the definition of SLOs and SLAs?

Majors: I've never worked with QA or testers. I kind of feel like QA lost the boat a decade ago and failed to move with the times. I deeply love the operations-engineering profession, and I'm trying to make sure the same doesn't happen to ops. There will always, always be a place for operational experts... but we are increasingly a niche role, and for most people, we will live on the other side of an API. Developers will own and operate their own services, and this is a good thing! Our roles as operational experts are to empower and educate and be force amplifiers. And to build the massive world-class platforms they can use to build composable infrastructure stacks and pipelines, like AWS and Honeycomb.

InfoQ: What is the most common monitoring anti-pattern you see, both from the perspective of the data store and application? Can you recommend any approaches to avoid these?

Majors: "Monitor everything." Dude, you can't. You *can't*. People waste so much time doing this that they lose track of the critical path, and their important alerts drown in fluff and cruft. In the chaotic future we're all hurtling toward, you actually have to have the discipline to have radically *fewer* paging alerts, not more. Request rate, latency, error rate, saturation. Maybe some end-to-end checks that stress critical key-performance-indicator (KPI) code paths.

People are over-paging themselves because their observability blows and they don't trust their tools to let them reliably debug and diagnose the problem. So they lean heavily on over-paging themselves with clusters of tens or hundreds of alerts, which they pattern-match for clues about what the root cause might be. They're flying blind for the most part; they can't just explore what's happening in production and casually sate their curiosity. I remember living that way too, and that's why we wrote Honeycomb. So we would never have to go back.

InfoQ: Thanks for taking the time to sit down with us today. Is there anything else you would like to share with the InfoQ readers?

Majors: Nothing I say should be taken as gospel. Lots of people don't have the problems of large distributed systems, and if you don't have those problems, you shouldn't take any of my advice. If you can get away with a monolith and a LAMP stack and a handful of monitoring checks, you should absolutely do that. Someday, you may reach a tipping point where it becomes harder and more complicated to achieve your goals without microservices and explorable event-driven observability, but you should do your best to put that day off. Live and build as simply as you possibly can.

Getting Started in Observability with Structured Logging

You might be think observability is a lot of work, but a quick path to success is structured logging. You don't need fancy libraries or agents and you can make incremental changes to your existing logging setup.

Structured logging means having a logging API to help you provide *consistent context* in events. An unstructured logger accepts strings. A structured logger accepts a map, hash, or dictionary describing all the attributes you can think of for an event:

- The function name and log line number
- The server's host name
- The application's build ID or git SHA
- Information about the client issuing a request
- Timing information

The format and transport details — whether you choose JSON or something else, whether you log to a file or stdout or straight to a network API — are less important!

Let's write a structured logging library!
Structured logging basically means you make a map and print it out or shove it in a queue:

```
def log(**data):
   print json.dumps(data)
```

Maybe we're not ready for open-source fame yet, but there are two nice things about this "library": 1) it doesn't let you pass a bare string message; you *have* to pass a dictionary of key-value pairs, and 2) it produces _structured, self-describing_ output that can be consumed by humans and machines alike.

For example, this log line is not self-describing:

```
127.0.0.1 - - [12/Oct/2017 17:36:36]
"GET / HTTP/1.1" 200 -
```

This seems obvious, but if we start adding more data, it will be hard to remember which dash means what. In contrast, write code like:

```
log(upstream_address="127.0.0.1",
    hostname="my-awesome-appserver",
    date=datetime.now().isoformat(),
    request_method="GET",
    request_path="/",
    status=200)
```

That will produce output that's comprehensible to both machines and humans.

To include the same context at different places in the code, wrap the logger in a class to bind context to:

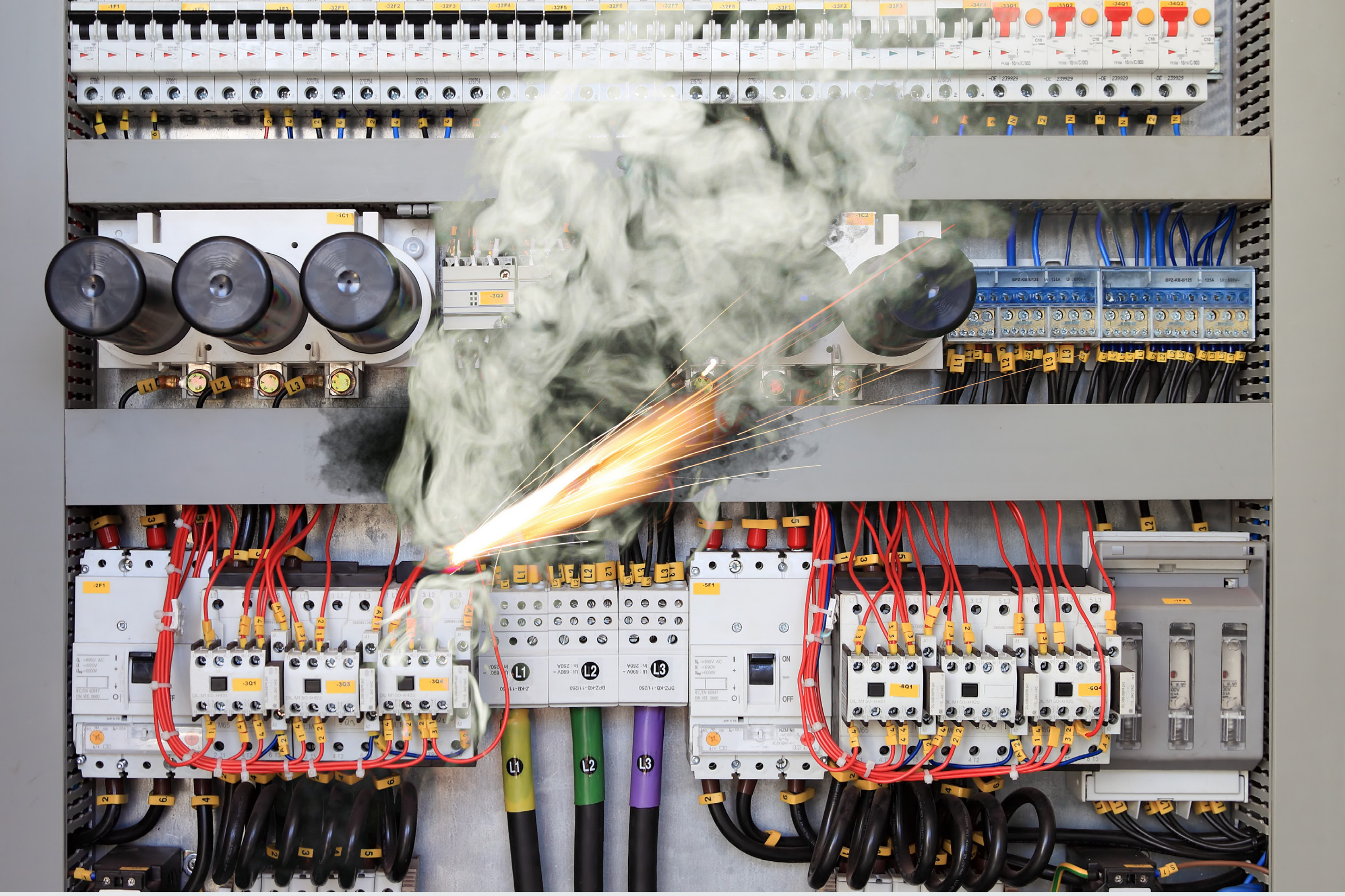
```
class Logger(object):
   def __init__(self):
       self.context = {}

   def log(self, **data):
       data.update(self.context)
       print json.dumps(data)

   def bind(self, key, value):
       self.context[key] = value
```

Now you can write

```
logger = Logger()
logger.bind("hostname", "my-awesome-
appserver")
logger.bind("build_id", 2309)
```

And all calls to `logger.log` automatically include `host name` and `build_id`.

Click here to read the full article



Read online on InfoQ

Observability and Avoiding Alert Overload from Microservices at the Financial Times

by **Daniel Bryant**

KEY TAKEAWAYS

Any microservice-based application is a distributed systems, and accordingly, services do not run independently. If something fails, it can often lead to cascade failures, which complicates monitoring and alerting.

In order to adapt to the challenges of monitoring a microservices-based application, Wells suggested a three-pronged approach: build a system that can be supported; concentrate on “stuff that matters”; and cultivate alerts and the information they contain.

Creating alerts should be part of the normal development workflow: “code, test, alerts”. In order to ensure that the development team know if an alert stops working, tests should be added to validate the alert.

A microservices architecture lets you move fast, but there is an associated operational cost, particularly around monitoring and observability. Make sure it’s a cost you’re willing to pay.

At QCon London 2017, Sarah Wells presented “Avoiding Alerts Overload from Microservices” in which she cautioned that developers and operators must fundamentally change the way they think about monitoring when building a distributed microservice-based system.

Wells, a principal engineer at the Financial Times (FT), began the talk by stating that knowing when there is a problem is not enough: an alert must only be triggered when an action by a human is required. A microservices architecture may allow the development team to move fast but there is an operational cost, and the number (and complexity) of alerts generated by a microservice-based system can be overwhelming.

“A microservices architecture lets you move fast, but there is an associated operational cost. Make sure it’s a cost you’re willing to pay,” she noted.

FT’s FT.com website is powered by a microservice back end that primarily uses the Java and Go programming languages, packaged and deployed with Docker and CoreOS on the Amazon Web Services (AWS) platform. FT stores data within MongoDB, Elastic, Neo4j, and Apache Kafka.

There are 99 functional services, with 350 running instances at any given time, and 52 nonfunctional services with 218 running instances. Wells stated that if each of the 568 service instances were checked every minute, this would result in 817,920 checks per day.

Running containers on shared virtual machines (VMs) requires 92,160 system-level checks, for a total of 910,080 checks per day. In addition, any microservice-based application is a distributed system and, accordingly, services do not run independently.

If something fails, it can often lead to cascade failures, which further complicates monitoring and alerting. (see Figure 1)

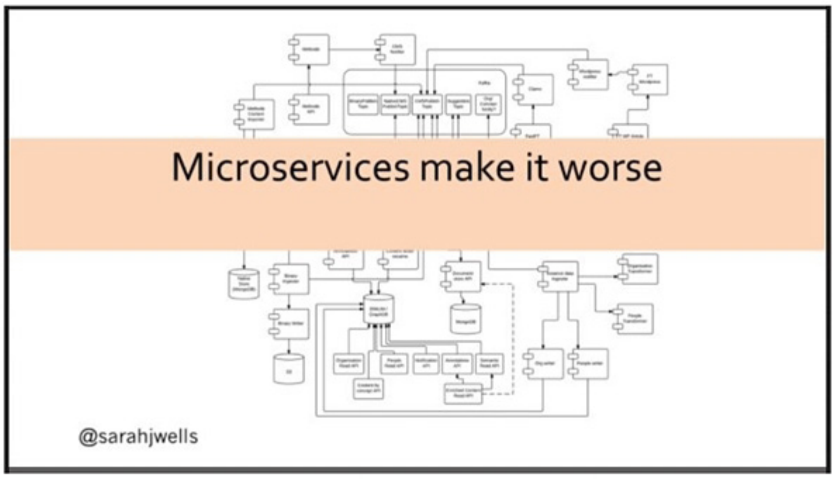


Figure 1: Wells stated that a microservices-based application makes the challenges of monitoring worse.

In order to adapt to the challenges of monitoring a microservices-based application, Wells suggested a three-pronged approach: build a system that can be supported, concentrate on “stuff that matters”, and cultivate alerts and the information they contain.

In order to build a system that can be supported, log aggregation and monitoring are essential. Log aggregation is required due to the volume of services and potential latency introduced via communication over a network, which means that logs may go missing or get increasingly delayed. This in turn means that log-based alerts may miss issues, particularly time-sensitive issues. Effective log aggregation requires a method that finds all related logs, and accordingly the FT team uses transaction ID for correlation.

Traditional monitoring tooling like Nagios is often limited, as it does not provide a service-level view, and the default (infrastructure) checks include things that cannot be fixed. In a microservices-based system, monitoring should take place at the service and VM level. Monitoring needs

Figure 2: The FT.com technical team’s SAWS aggregated monitoring.

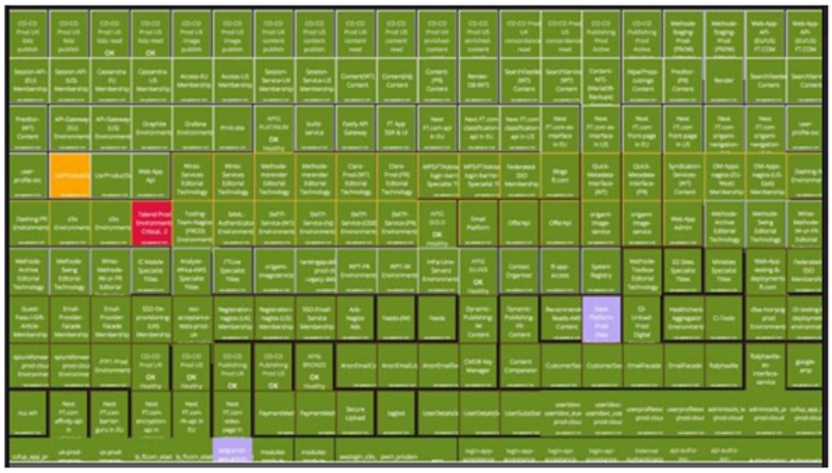
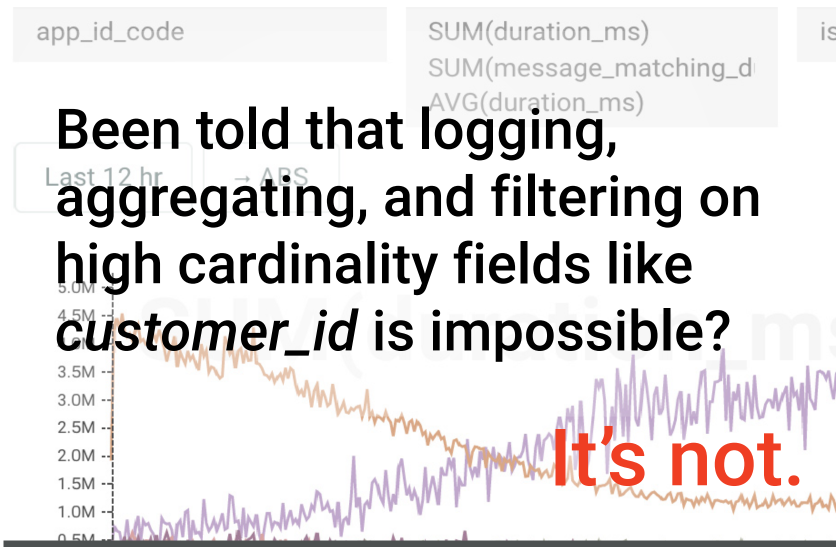


Figure 3: The FT.com microservices alert dashboard, which is powered by the Dashing framework.

to be aggregated and made visual, and the FT technical team uses a custom framework named SAWS (built by Silvano Dossan, shown in Figure 2) and Dashing. They also extensively use graphing via Graphite and Grafana.

When developing polyglot services, logging and monitoring integration must be made easy for any language that is used. The expectations, or operational contract, must be specified, and each service owner is responsible for implementing functionality to meet this requirement. For example, the FT health-check standard requires that every service expose a health-check endpoint over HTTP (http://service/__health) that returns a 200 if the service can run the health check and a JSON document containing multiple checks that can contain additional information but must return `“ok”:true` or `“ok”:false`. (see Figure 3)

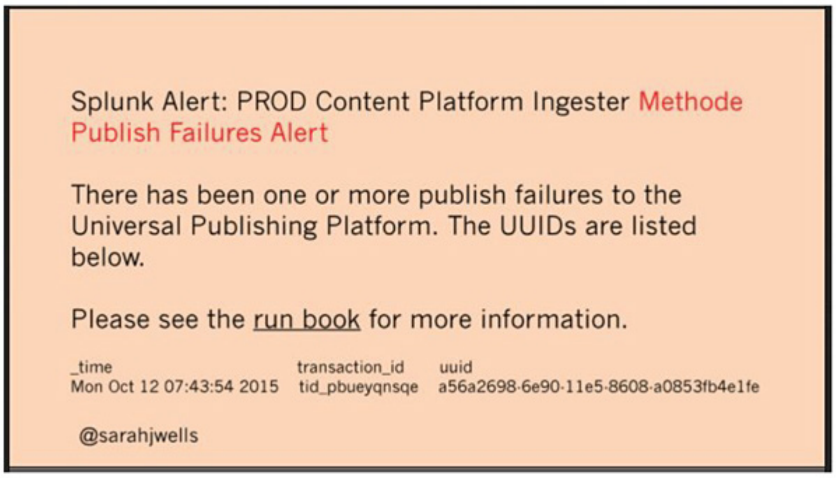
A core goal of monitoring and alerting is to know about problems before clients do, and accordingly the practice of running synthetic requests that mimic user functionality behavior is vital. If functionality relating to a key user journey is broken — for example, an FT editor cannot publish a new article — it must be fixed immediately. Wells stated that engineers must learn to prioritize and “concentrate on the stuff that matters”. The FT technical team has also created dashboards that show core client statistics, such as number of errors and response latency, but Wells stressed that it is “the end-to-end [business functionality] that matters” and “if you just want information, create a dashboard or report”.



Been told that logging, aggregating, and filtering on high cardinality fields like *customer_id* is impossible?

It’s not.

Try it FREE FOR 30 DAYS at
https://honeycomb.io/signup


Splunk Alert: PROD Content Platform Ingester Methode Publish Failures Alert

There has been one or more publish failures to the Universal Publishing Platform. The UUIDs are listed below.

Please see the run book for more information.

```
_time                      transaction_id    uuid
Mon Oct 12 07:43:54 2015   tid_pbueyqnsqe    a56a2698-6e90-11e5-8608-a0853fb4e1fe
```

@sarahjwells

Figure 4: A FT.com alert example with information that includes the issue, the impact, transaction IDs, and a link to the associated runbook.

Alerts must continually be cultivated, and if an alert is received that doesn't make sense or does not require human interaction, it must be corrected or removed. If an issue occurs but it triggered no alert, then one should be added as part of the fix. Key information must be included within each alert: for example, an overview of the business impact, the associated runbook location, and corresponding transaction IDs that triggered the issue.

The FT team uses dedicated "Ops Cops" (on-call members of the development team, rotated regularly) to watch for issues with monitoring, and has integrated alerting within the team's Slack messaging system. The team uses a predefined list of emojis (with a clear, stated purpose for each) to indicate when and how an issue is being managed and resolved.

Concluding the talk, Wells suggested that creating alerts should be part of the normal development "code, test, alerts" workflow. In order to ensure that the development team knows if an alert stops working, tests should be added to validate the alert. The FT technical team subscribes to the philosophy of chaos testing and, inspired by Netflix's Simian Army and Chaos Monkey, has created a "Chaos Snail" (which is "smaller than a monkey, and written in Bash shell!"). Wells cautioned that proactivity is required when maintaining and dealing with alerts in a non-trivial system, and out-of-date information can be worse than none at all. Automate updates wherever possible, and find ways to share what is changing.

The slides for Wells's "Avoiding Alerts Overload From Microservices" talk can be found on Speaker Deck, and the video can be found on InfoQ.

Read online on InfoQ

Uwe Friedrichsen on Functional Service Design and Observability

by **Daniel Bryant**

KEY TAKEAWAYS

The key ideas associated with microservices are the properties that support independence of the rest of the application landscape and quick evolvability. This is often ignored by implementers.

Monitoring and logging systems need to evolve in ways that reflect current software architecture, and this has an impact that is both technical and cognitive.

Many things learned in a classic software engineering education about how to slice functionality (such as "Don't Repeat Yourself") does not work for distributed systems, like microservices.

The term "microservices" itself will probably disappear in the future, but the new architectural style of functional decomposition is here to stay.

At the microXchg 2017 conference in Berlin, Uwe Friedrichsen discussed the core concepts of "Resilient Functional Service Design" and how to create observable systems.

Friedrichsen believes that in order to develop effective systems, microservice developers must learn about fault-tolerant design patterns and caching but not use them to mitigate fundamentally bad (overly coupled) system design, understand domain-driven design (DDD) and modularity,

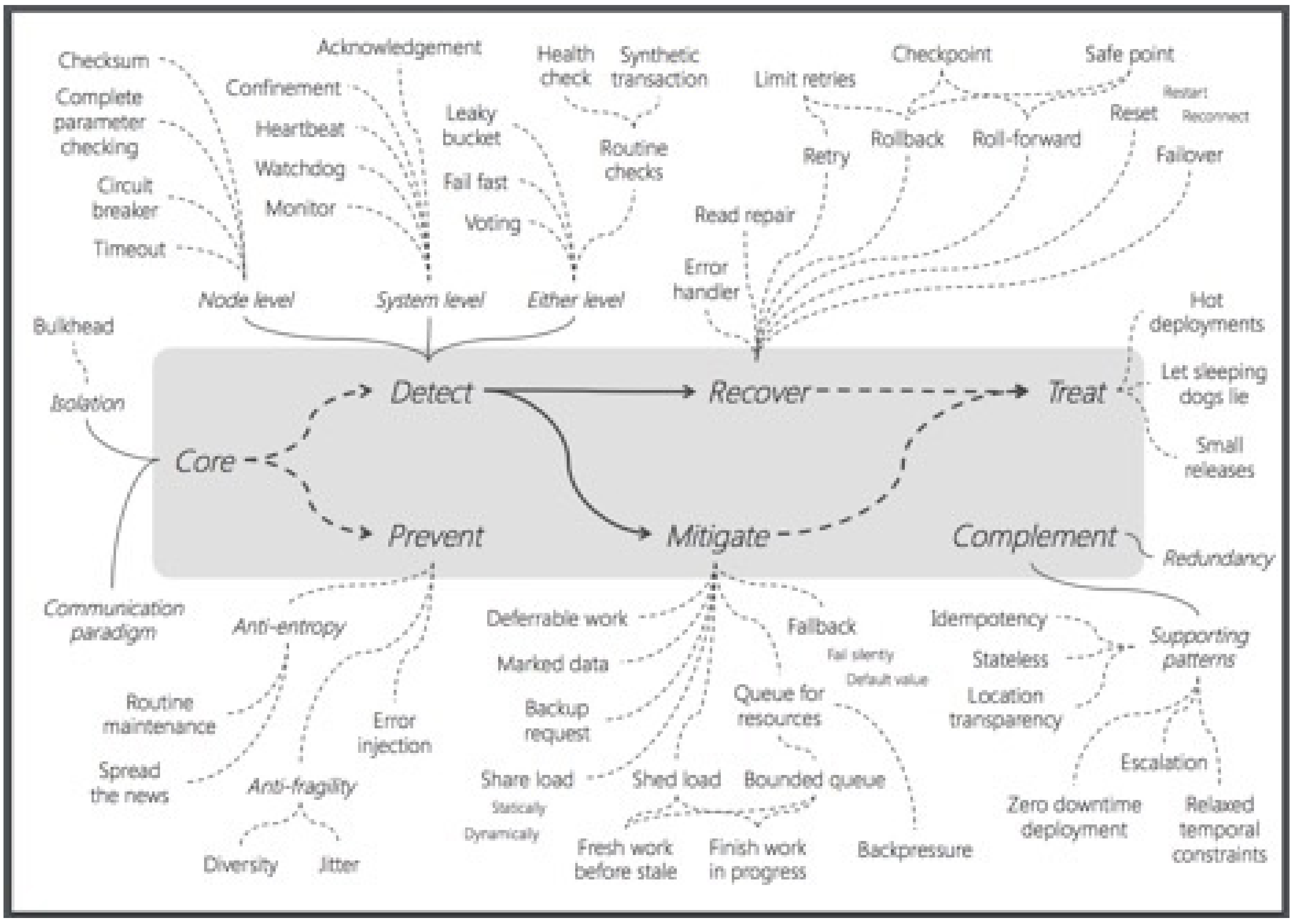


Figure 1: The fundamental concepts and patterns of system resilience.

and aim to design for replaceable components rather than reuse.

Friedrichsen, CTO at Codecentric, began the presentation by stating that the goal of software development is to deliver business value, and for this to be realized, the software must be run in production and be highly available. Modern architectural styles, such as microservices, mean that everything is now distributed, most likely spanning at least a local area network, and therefore failures within the system are normal and not predictable. Developers must learn about the fundamentals of resilience, shown in Figure 1.

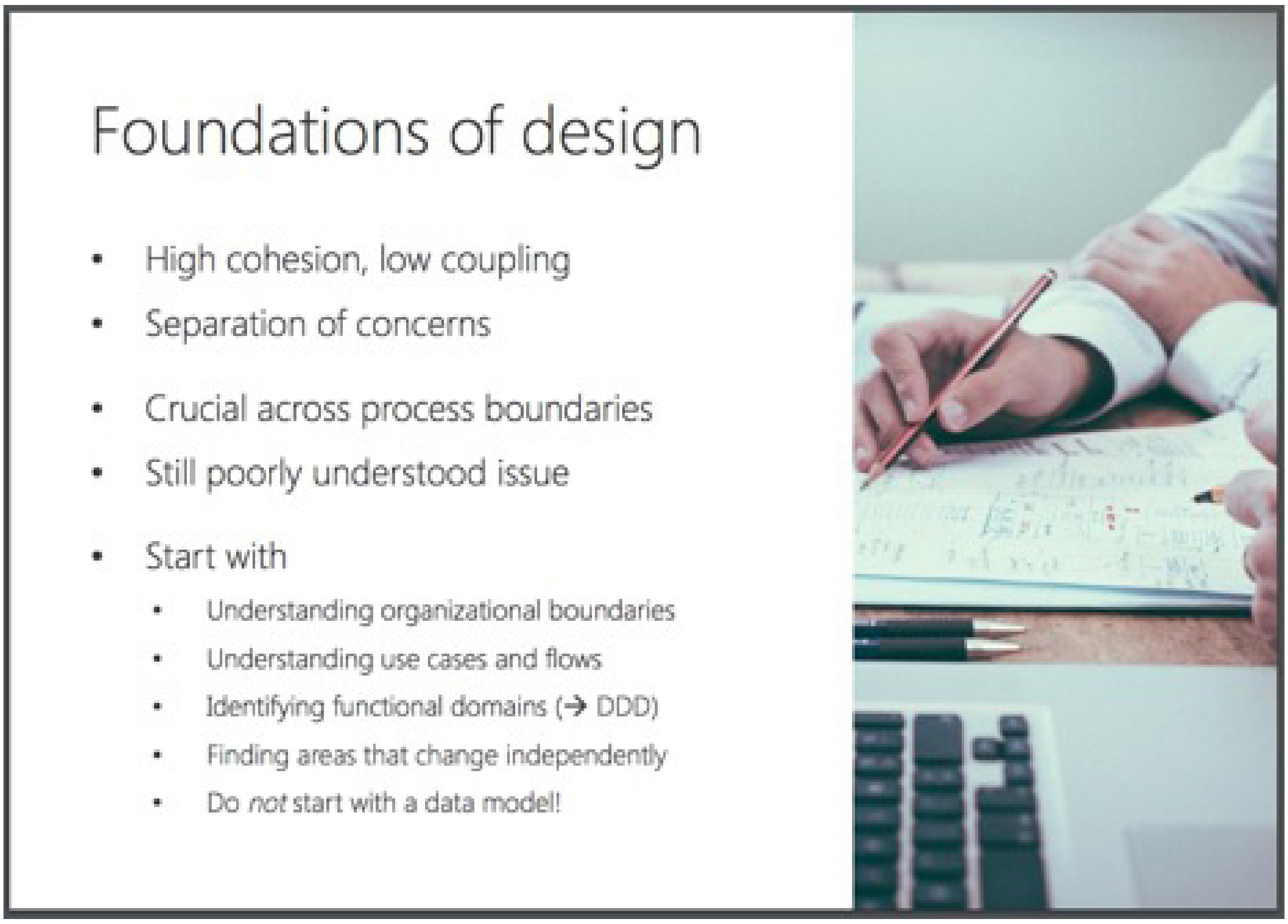
Developers should familiarize themselves with fault-tolerant design patterns, such as circuit breakers, bulkheads, timeouts, and retries, which have been popularized by Michael Nygard's *Release It!* book. Caching, although useful, should be deployed with care and not used simply to overcome bad system design, such as a long activation path involving many dependent services.

Friedrichsen presented a series of foundations of design for microservices (pictured in Figure 2), which included a series of design principles focusing on high cohesion, low coupling, and separation of concerns. These principles are especially crucial across system boundaries, and even though the theory was well documented in the '70s by David Parnas (PDF link), it is still often misunderstood.

DDD is a useful tool, but many developers overly focus on the static context model of domain, something Friedrichsen calls "entity DDD". The dynamic behavior of the system is often more illustrative of the business activities, domain events, and flow of data.

Quoting Fred Brooks, Friedrichsen discussed the promise of software reuse that developers have spent many years chasing. Brooks suggested that the effort required to create a reusable component (over one that fits a purpose for a single case) is typically multiplied by three, meaning that any return on investment for reusability is only seen when a component has been used without modification at least four times.

Monitoring and logging systems need to evolve in ways that reflect current software architecture, and this has an impact that is both technical and cognitive.



Foundations of design

- High cohesion, low coupling
- Separation of concerns
- Crucial across process boundaries
- Still poorly understood issue
- Start with
  - Understanding organizational boundaries
  - Understanding use cases and flows
  - Identifying functional domains (→ DDD)
  - Finding areas that change independently
  - Do *not* start with a data model!

Figure 2: Uwe Friedrichsen's foundations of service design.

The communication paradigm used within a microservices system also greatly influences the functional service design, and Friedrichsen suggested that care should be taken with up-front architectural choices that may limit future modification and extensibility.

The concluding message and core takeaway from the talk was that developers and architects need to relearn functional service design when implementing distributed systems like those being created by microservice architectures, as the properties of these systems expose and multiply the effects of design issues we have known about for many years.

InfoQ sat down with Friedrichsen to further discuss the challenges of designing resilient and observable services.

InfoQ: What is the most challenging issue for the current batch of organizations implementing microservices?

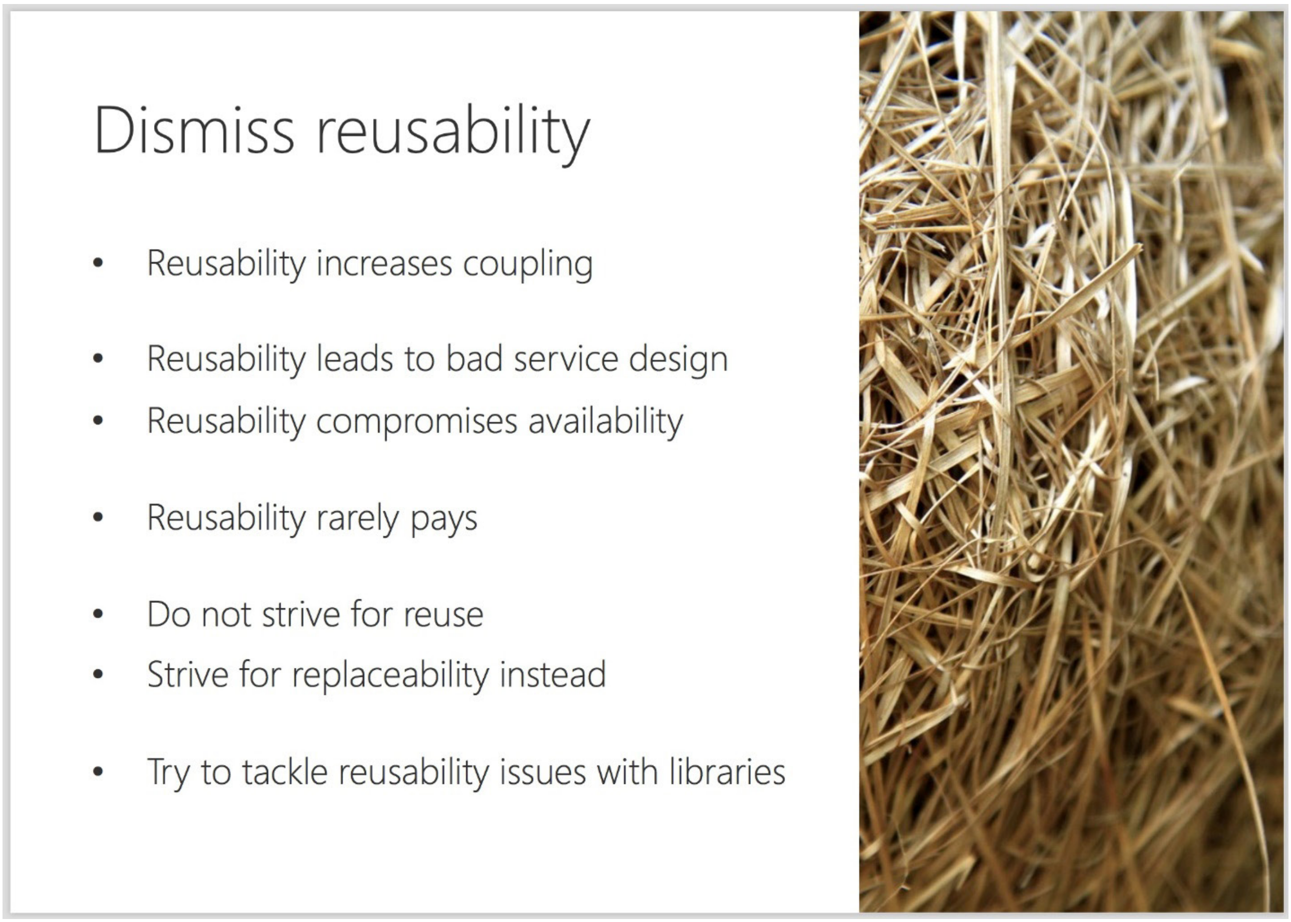
Uwe Friedrichsen: That they are doing microservices in the first place.

"Microservices" has become a popular, mainstream term and everybody who uses Spring Boot or the like claims to do microservices. Do not get me wrong: there is nothing wrong about Spring Boot, but writing a standalone application that exposes some HTTP interface does not mean that you write a microservice.

The key ideas with microservices are the properties that support independence of the rest of the application landscape and quick evolvability. Unfortunately, based on what I can observe, people put too little effort into those properties that define a microservice.

Microservices is an architectural style that helps you to move fast. You need to move fast in IT if your company lives in a fast-moving market and if the IT needs to support business to move fast. But even then, a lot more than just an architectural style is needed to move fast as IT.

The problem of those hyped trends like microservices is that people often try to pick them up even if they are not an adequate solution for their situation. If your IT does not seriously try to be-



Dismiss reusability

- Reusability increases coupling
- Reusability leads to bad service design
- Reusability compromises availability
- Reusability rarely pays
- Do not strive for reuse
- Strive for replaceability instead
- Try to tackle reusability issues with libraries

Figure 3: Friedrichsen suggested developers should dismiss reusability and instead strive for replaceability.

come faster, you probably do not need microservices. You can still use Spring Boot (it is fun, after all), but you should not call it microservices.

InfoQ: Moving now towards development patterns, can you recommend a technique for encouraging developers and architects to think about functional service design?

Friedrichsen: If I knew one, I would sell it for a lot of money and be real rich…. But again, let us be fair. I see three factors that make it hard to get developers and architects to think about functional service design to the extent they should:

1. It is hard, really hard, and even after 40+ years of software architecture and design it is poorly understood.

A lot of people then mention DDD, and DDD indeed is a good starting point. Still, only knowing the patterns is completely different from being able to create a sound design for your current business problem using those patterns — especially if you have your product manager or product owner sitting over you all the time, urging you to be more productive.

Also, everything we learned in our software education about how to slice functionality — e.g., functional decomposition, DRY (don't repeat yourself) or creating re-usable functionality — does not work for distributed systems, like microservices are. If you use those design best practices, you will end up with an extremely poor design that will haunt you badly in production. Basically, we have to relearn design for distributed systems and, based on my personal observations, we still have to learn a lot about how to do that right.

2. The real issues of IT are swamped by more attractive sideshows.

New frameworks, programming languages, endless debates about how to do this and how to do that, tons of opinionated people who try to tell you if you don't do this or that, you are doing it all wrong… — we drown in shiny new stuff and opinions. Go to a conference, read an IT magazine, or just look at your Twitter timeline and you know what I mean. And all of this promises a lot more fun than trying to learn how to design well.

3. We lose our collective memory every five years.

Based on my observations, we face a new generation of developers coming from university (or wherever else) about every five years. From a different perspective, this means that we lose our collective memory also every five years. These people do not (yet) know the talk or article that was an eye-opener for you several years ago. They have to relearn all that on their own from scratch — every single person who starts in IT.

What makes things harder with respect to "timeless" topics like functional design is the fact that in IT, new is considered valuable and old is considered worthless. We are a fast-moving business, aren't we? What value could knowledge that is five, 10, or even more than 20 years old possibly have? And even if some people eventually understand that not all old knowledge is worthless, that we keep telling and forgetting the same stories over and over again, it mostly remains unheard of by the vast numbers of new developers joining IT every year.

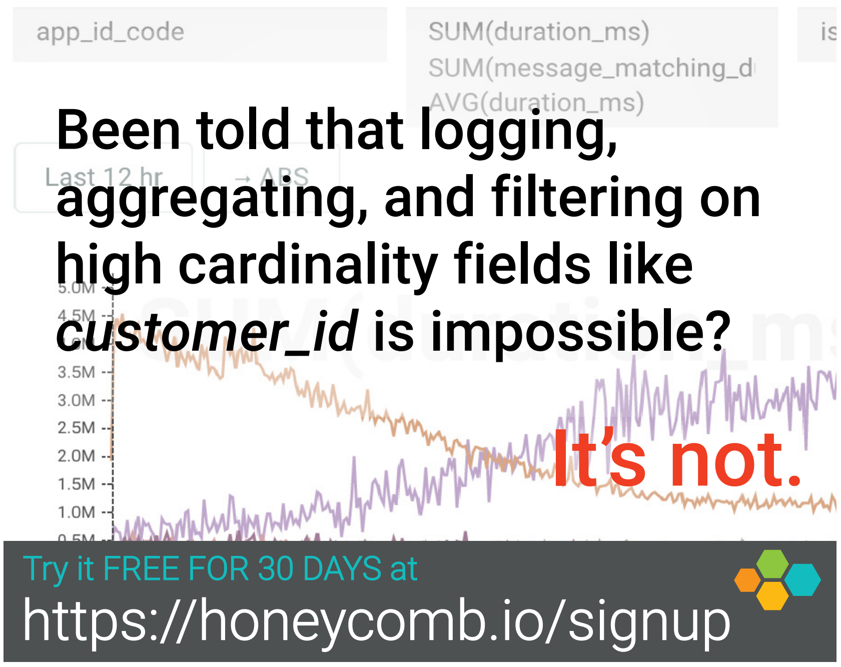
InfoQ: You mention the rapid change of people within the IT industry. What does the future hold for the microservices architectural style itself?

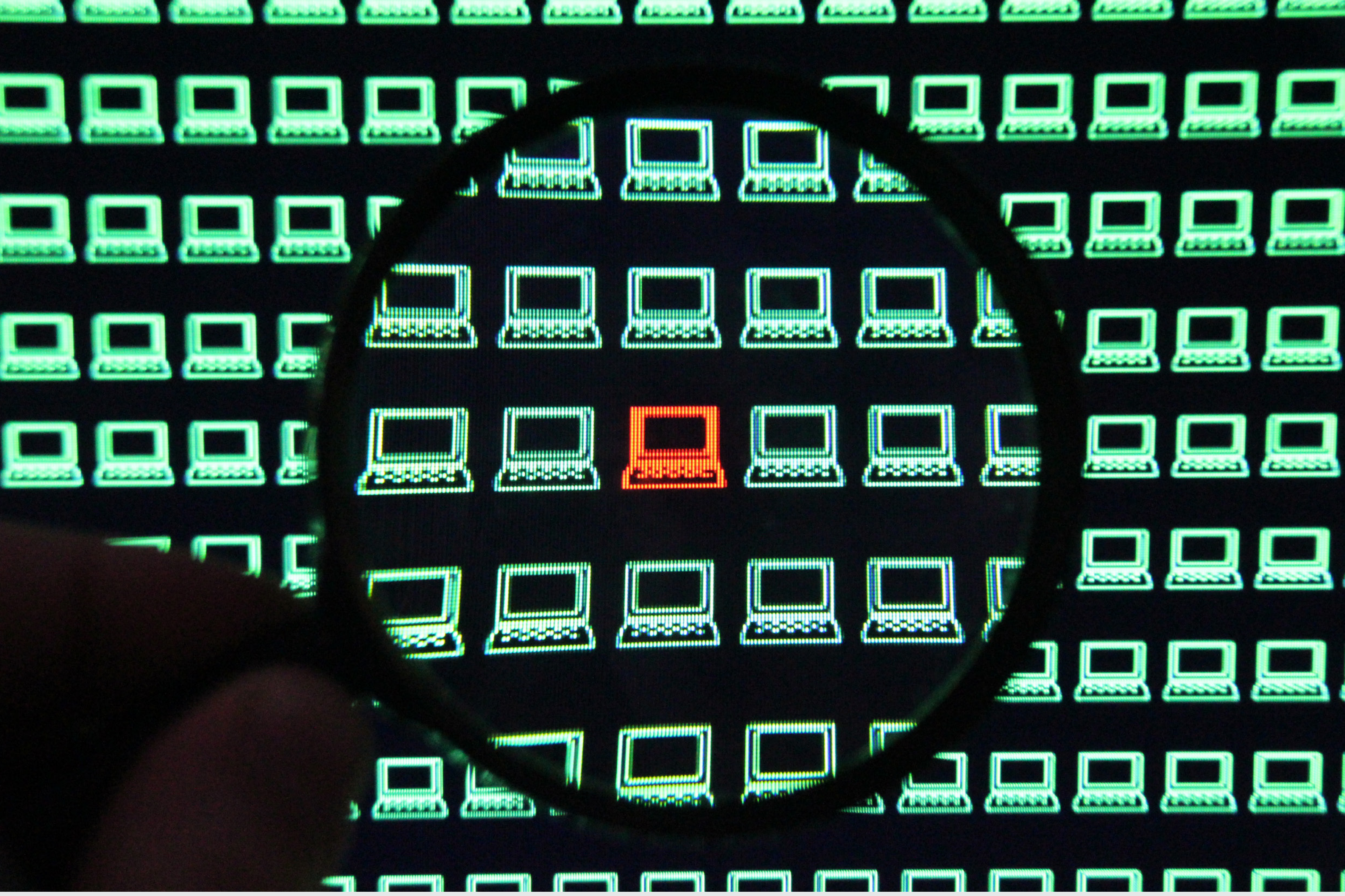
Friedrichsen: If I am really frank: I have no idea. The term "microservices" itself will probably eventually be burnt — as all hype terms become after a majority of vendors, consultants, and people who just want to adorn themselves with the new cool thing have picked them up. The architectural style, on the other hand, is here to stay. Actually, the style was not new when we started to call it "microservices"; it existed for many years. It just was updated based on the advances in IT and then called "microservices". And probably after the next update, the style will be called something different.

In the near future, we maybe will see some more differentiation of the microservices style. Not everybody needs all properties of a pure microservices style. Many people might get along with just a subset of the microservices-style properties in order to satisfy the needs they face.

But to be honest: in the end, I have no idea.

The full video for Friedrichsen's talk, "Resilient Functional Service Design", can be found on InfoQ.


Been told that logging, aggregating, and filtering on high cardinality fields like *customer_id* is impossible?
It's not.
Try it FREE FOR 30 DAYS at
https://honeycomb.io/signup



Debugging Distributed Systems

Q&A with the “Squash” Microservice Debugger Creator Idit Levine

KEY TAKEAWAYS

The ability to monitor and debug an application is important during development and in production. Debugging a microservice-based application is more challenging than debugging a monolithic application, as it is difficult to attach a native debugger to multiple processes that communicate across a network.

Squash in an open source microservice debugging tool that orchestrates run-time debuggers attached to microservices and provides familiar features like setting breakpoints, stepping through the code, viewing and modifying variables etc

We should aspire to provide distributed applications the same level of observability and control that is available for monolithic applications. A service mesh may be the future best point of integration for such observation, for example, logging, tracing and in-process debugging

InfoQ recently sat down with Idit Levine, CEO of solo.io and creator of the new open source “Squash” microservices debugger, and discussed the challenges of observing and debugging distributed systems and applications.

InfoQ: Hi Idit, and welcome to InfoQ! Could you introduce yourself, and discuss a little about your latest venture solo.io **please?**

Levine: Hi Daniel, thank you for having me. I am the founder and CEO of solo.io, whose general mission is to streamline the cloud stack. I've been in the cloud management space for 12 years, since I've joined DynamicOps (the developer of vCAC, later acquired by VMware) as one of its first employees.

Most recently I was the CTO of the cloud-management division at EMC. There I led, designed and implemented project unik, an open source platform for automating Unikernels compilation and deployment, and project layer-x, an open source framework for cross-cluster scheduling.

Solo.io is currently in stealth mode, but my commitment to the open source community is as strong as ever. That's why we recently released Squash, an open source platform for debugging microservices applications. We plan to enhance Squash and bring other valuable tools to the community in the near future.

InfoQ: Can you explain a little about how operational and infrastructure monitoring has evolved over the last five years? How have cloud, containers, and new architectural styles like microservices impacted monitoring and debugging?

Levine: Monitoring the state of an application is important during development and in production. With a monolithic application, this is rather straightforward, since one can attach a native debugger to the process and have the ability to get a complete picture of the state of the application and its evolution.

Monitoring a microservice-based application poses a greater challenge, particularly when the application is composed of tens or hundreds of microservices. Due to the fact that any request may involve being processed by many microservices running multiple times -- potentially on different servers -- it is exceptionally difficult to follow the "story" of the application and identify the causes of problems when they arise.

Currently, the main methodology relies on obtaining a trace of all transactions and dependencies using tools that, for example, implement the OpenTracing standard. These tools capture timing, events, and tags, and collect this data out-of-band (asynchronously). OpenTracing allows users to perform critical path analysis and monitor request latency, perform topological analysis and identify bottlenecks due to shared resources. Users can also log what they think could be useful data, like the values of different variables, error messages etc.

InfoQ: We've been keenly watching the evolution of Squash -- an open source tool that allows the debugging of microservices application running on container orchestration from IDE -- and would be keen to hear the goals of the project and rationale for creating this?

Levine: OpenTracing tools are very powerful, but they have limitations and gaps. Since logging the state of the application during runtime can be expensive and result in performance overhead, one needs to limit the amount of collected information. One way to do this is to follow only a subset of the transactions, and not all of them. Tuning the size of this sample represents a tradeoff between the amount of information collected on one hand, and the price in performance and costs on the other.

One consequence is that once a problem is identified, it is possible that some needed information is missing. Obtaining this information requires running the application again, and waiting for the data to be collected. Moreover, OpenTracing is not a runtime debugger and does not allow changing variables during runtime to explore potential solutions to a problem. Any attempt to fix a problem requires wrapping the code, running the application, and waiting for the data again. Solving a problem may necessitate several such iterations, which can be both daunting and expansive.

Our vision for Squash is to complement the OpenTracing tools and close these gaps. The main goal of Squash is to provide an efficient tool for debugging microservices applications. Squash orchestrates run-time debuggers attached to microservices, providing familiar features like setting breakpoints, stepping through the code, viewing and modifying variables etc. Importantly, Squash allows the developer to seamlessly follow the application and skip between microservices. Squash takes care of all the necessary piping, allowing developers to focus on their own code and solve the issues they actually care about. To make Squash accessible and easy to adopt, it integrates with existing popular IDEs.

Squash is designed to provide essential capabilities for monitoring the life cycle of an application both in the development phase, Debugging a microservice-based application is more challenging than debugging a monolithic application, as it is difficult to attach a native debugger to multiple processes that communicate [exclusively] across a network.

allowing development of robust code, as well as during production, allowing fast adaptation of the code when new difficulties arise.

InfoQ: What are the future plans for Squash?

Levine: We recognized that Squash can leverage a service mesh (like Istio) and proxy (like Envoy) to let users debug application that run in the mesh without pausing the entire service. Accordingly, we've just officially pushed Squash http envoy filter to Envoy upstream. Next, we will work with the Istio team to configure this project to use it.

We have received community requests to integrate Squash with more platforms, like Mesos and Docker Swarm, and we hope to also integrate it with Cloud Foundry. We have also added support for more debuggers, like Java, Node js and Python. Lastly, we are looking forward to support more IDEs, including IntelliJ IDEA and Eclipse.

In addition, we are talking with the OpenTracing-community leader, with the aim to integrate OpenTracing with Squash. The vision is that users would be able to identify latency between two services via OpenTracing, and zoom-in to resolve the problem with Squash.

InfoQ: We've seen you talk about Unikernels, and would be keen to get your opinion on the role this technology will play in the future? Bryan Cantrill has famously stated that Unikernels are unfit for production**, and are also entirely undebuggable. What do you think about this?**

Levine: I believe that Unikernels will play a significant role in the future, mainly in the IoT space. The benefits of Unikernels – their "slim" footprint, security, performance – are a great fit to IoT devices where the storage is limited and one prefers to include minimal code rather than a full-blown OS.

I believe unik is a fantastic orchestration tool to build and run a Unikernel, and it seems that the community agrees based on the traffic and clones on the GitHub repository. I am very happy that people are using unik. Next, I hope to extend unik to be more than a Unikernel tool, by supporting Kata Containers, LinuxKit, FreeRTOS and other IoT embedded device software.

Bryan is absolutely right that Unikernels can only be production ready when monitoring and debugging tools for Unikernels become available. Currently, such tools do not exist.

When we built unik we had to debug the Unikernel, and we did that using the gdb debugger. I can therefore testify that debugging Unikernels is indeed possible, but can be extremely hard.

I think that the community, which recognizes the huge potential of Unikernels, should invest in creating new tools that will automate this process and make it easier. Squash, for example, is already leveraging debuggers like gdb, so potentially it could be expanded to help debugging Unikernels.

InfoQ: "Serverless" technology is also getting increasingly popular, and would a tools like Squash also be useful for debugging applications/ functions deployed here?

Levine: Definitely! Actually, we originally thought of Squash as a tool for debugging serverless applications. However, most people who run serverless apps today use the public cloud FaaS platforms -- and for good reasons, as this is currently the most mature offering. Such platforms take the complexity away from the user, but also take away the control and flexibility.

Users do not have any control or access to the environment that the functions run on. This really limit the ability of the community to innovate in the serverless space, and forces it to come up with hacks and "creative" solutions to overcome its limitations. I am not a fan of "hacks", and therefore when we built Squash we gave priority to platforms that provide us with the hooks to plug into.

InfoQ: What other tools do you think future developers will need to understand and debug large-scale, rapidly evolving container-based applications?

Levine: As a community, we should aspire to provide distributed applications the same level of observability and control that is available for monolithic applications. A combination of existing tools already points us in the right direction. Log collection can be done by OpenTracing tools, metrics collected by Prometheus, and debugging by Squash. All of these methods should plugin to a service mesh to achieve full efficiency.

InfoQ: What role do you think QA/Testers have in relation to observability and debuggability of a system?

Levine: In one possible mode of action, I would expect the QA and testers to focus on the logs and provide context. With container-based applications, this should be done using OpenTracing. The developer will then be able to reproduce the bug and use Squash to attach a debugger, step through the code, and resolve the issue.

InfoQ: Thanks once again for taking the time to sit down with us today. Is there anything else you would like to share with the InfoQ readers?

Levine: We at solo are working hard of building more open source tools to facilitate microservices development and operation. In particular, we are focused on innovative and helpful tools to accelerate adoption of microservices in the enterprise. We are super excited about our plans for 2018 -- please stay tuned!

Additional information on solo.io can be found at the company website, and the open source Squash microservices debugger can be found on GitHub.

PREVIOUS ISSUES

57

Streaming Architecture

This InfoQ emag aims to introduce you to core stream processing concepts like the log, the dataflow model, and implementing fault-tolerant streaming systems.

Faster, Smarter DevOps

56

This DevOps eMag has a broader setting than previous editions. You might, rightfully, ask “what does faster, smarter DevOps mean?”. Put simply, any and all approaches to DevOps adoption that uncover important mechanisms or thought processes that might otherwise get submerged by the more straightforward (but equally important) automation and tooling aspects.

Cloud Native

55

In this eMag, the InfoQ team pulled together stories that best help you understand this cloud-native revolution, and what it takes to jump in. It features interviews with industry experts, and articles on key topics like migration, data, and security.

Reactive JavaScript

54

This eMag is meant to give an easy-going, yet varied introduction to reactive programming with JavaScript. Modern web frameworks and numerous libraries have all embraced reactive programming. The rise in immutability and functional reactive programming have added to the discussion. It’s important for modern JavaScript developers to know what’s going on, even if they’re not using it themselves.