The InfoQ eMag / Issue #95 / June 2021

Kubernetes and Cloud Architectures

InfoQ

Kubernetes Is Not Your Platform, It's Just the Foundation

The Evolution of Distributed Systems on Kubernetes

Containers Are Contagious and Often Misused

Kubernetes and Cloud Architectures

IN THIS ISSUE

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITORS Lawrence Nyveen & Susan Conant / DESIGN Dragos Balasoiu
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Manuel Pais

is co-author of the book Team Topologies: organizing business and technology teams for fast flow. Recognized as a DevOps thought leader, Manuel is an independent IT organizational consultant and trainer, focused on team interactions, delivery practices and accelerating flow. Manuel is also an instructor on Continuous Delivery for LinkedIn and on Team Topologies for the Team Topologies Academy. Manuel tweets @manupaisable

Bilgin Ibryam

is a product manager and a former architect at Red Hat, a committer, and a member of the Apache Software Foundation. He is an open-source evangelist, regular blogger, speaker, and author of Kubernetes Patterns and Camel Design Patterns books. Bilgin's current work focuses on distributed systems, event-driven architecture, and repeatable cloud-native application development patterns and practices. Follow him @bibryam for future updates on similar topics.

Holly Cummins

is an innovation leader in IBM Corporate Strategy, and spent several years as a consultant in the IBM Garage. As part of the Garage, she delivers technology-enabled innovation to clients across various industries, from banking to catering to retail to NGOs. Holly is an Oracle Java Champion, IBM Q Ambassador, and JavaOne Rock Star. She co-authored Manning's Enterprise OSGi in Action.

Alaa Tadmori

is a Cloud Solutions Architect at Microsoft. He is an early adopter and enthusiast of the cloud computing model. Alaa believes in responsible IT, the solutions we're building today are shaping our world and will be the world of our children so we have a responsibility and a choice everyday to make our world a better place. When not at work Alaa likes to spend time with his family and read nonfiction books.

A LETTER FROM THE EDITOR

Richard Seroter

is Director of Outbound Product Management at Google Cloud, with a master's degree in Engineering from the University of Colorado. He's also an instructor at Pluralsight, the lead InfoQ.com editor for cloud computing, a frequent public speaker, the author of multiple books on software design and development, and a former 12-time Microsoft MVP for cloud. As Director of Outbound Product Management at Google Cloud, Richard leads a team focused on products and customer success for app modernization (e.g. Anthos). Richard maintains a regularly updated blog on topics of architecture and solution design and can be found on Twitter as @rseroter.

Does it feel to you like the modern application stack is constantly shifting with new technologies and practices emerging at a blistering pace? It does to me. Every week I seem to come across a new web framework, open-source data integration framework, or architectural anti-pattern that used to be a best practice. But then I stop, take a breath, and see some underlying stability.

The fundamentals for the next decade are in place. The public cloud is here to stay. You'll continue to use private infrastructure while counting on software-as-a-service and cloud-based managed services more and more. Kubernetes is mainstream. It's not for everyone, but it will be everywhere. As will higher-level application platforms that keep the focus on the software itself. It's still early days on service mesh technologies, but the indicators are there that this will also be part of the common substrate that our apps depend on.

It also appears that many good practices are cemented in software teams around the world. There's little disagreement that the core ideas behind DevOps—continuously delivering small batches of customer-driven value—are important. And the field of software testing has broadened as teams go beyond unit testing to also look at resilience testing. Microservices can be a good or bad thing depending on how you apply them, but most people agree that distributed systems are generally better for scale, resilience, and release velocity. We're also settling into good security practices, ranging from zero-trust to secure software supply chains.

We've hand-picked a set of articles that highlight where we're at today. With a focus on cloud-native architectures and

Kubernetes, these contributors paint a picture of what's here now, and what's on the horizon.

First up, Manuel Pais resets your expectations on what it means to have a “platform." He makes the case that Kubernetes is a terrific foundation for your team's platform, but more is needed. Pais encourages us to consider the overall developer experience and how to introduce capabilities to your teams. He's a leading thinker about effective team topologies, and you should carefully consider his advice for sustainable success.

Next, Bilgin Ibryam looks further into Kubernetes and the ecosystem around, and on top of it. He starts off by reminding us about platform lifecycle, networking, and storage considerations that often fuel the broader ecosystem around something like Kubernetes. Ibryam then proceeds to consider microservices architectures as a whole, before iterating through a set of modern Kubernetes extensions including service mesh, Knative, Dapr, Envoy, Apache Camel, Cloudstate, and more.

In the third piece, Holly Cummins takes the stance that cloud-native architecture is about much more than microservices and Kubernetes. Cummins outlines some of the important fundamentals, including smart takes on CI/CD. process improvement, and modern operations.

Our last featured piece is from Alaa Tadmori. He discussed that containers are not the only solution for the write-once run-everywhere objective. Tadmori states that you can use cross-platform frameworks to build clean/simple solutions that are also easily deployable between different environments/clouds.

Increasingly, I'm seeing many teams focus heavily on customer outcomes. We're changing the definition of “done" to reflect that usable, valuable software in the hands of our customers is what matters. Regardless of your tech stack or how you deliver it, it's about value. And that gives me hope that we're all heading in the right direction for the decade ahead.

Kubernetes Is Not Your Platform, It's Just the Foundation

by **Manuel Pais**, Team Topologies co-author, organizational consultant, trainer

I read a lot of articles and see presentations on impressive Kubernetes tools, automation, and different ways to use the technology but these often offer little context about the organization other than the name of the teams involved.

We can't really judge the success of technology adoption, particularly Kubernetes, if we don't know who asked for it, who needs it, and who is implementing it in which ways. We need to know how teams are buying into this new offering — or not.

In Team Topologies, which I wrote with Matthew Skelton, we talk about fundamental types of teams, their expected behaviors and purposes, and perhaps more importantly about the interactions among teams. Because that's what's going to drive adoption of technology. If we don't take that into consideration, we might fall into a trap and build technologies or services that no one needs or which have no clear use case.

Team cognitive load has a direct effect on platform success and team interaction patterns can help reduce unnecessary load on product teams. We'll look at how platforms can minimize cognitive load and end with ideas on how to take a team-centric approach to Kubernetes adoption.

Is Kubernetes a platform?

In 2019, Melanie Cebula, an infrastructure engineer at Airbnb, gave an excellent talk at Qcon. As DevOps lead editor for InfoQ, I wrote the story about it. In the first week online, this story had more than 23,000 page views. It was my first proper viral story, and I tried to understand why this one got so much attention. Yes, Airbnb helped, and Kubernetes is all the rage, but I think the main

factor was that the story was about simplifying Kubernetes adoption at a large scale, for thousands of engineers.

The point is that many developers and many engineers find it complicated and hard to adopt Kubernetes and change the way they work with the APIs and the artifacts they need to produce to use it effectively.

The term “platform" has been overloaded with a lot of different meanings. Kubernetes is a platform in the sense that it helps us deal with the complexity of operating microservices. It helps provide better abstractions for deploying and running our services. That's all great, but there's a lot more going on. We need to think about how to size our hosts, how and when to create/destroy clusters, and how to update to new Kubernetes versions and who that will impact. We need to decide on how to isolate different environments or applications, with namespaces, clusters, or whatever it might be. Anyone who has worked with Kubernetes can add to this list: perhaps worrying about security, for example. A lot of things need to happen before we can use Kubernetes as a platform.

One problem is that the boundaries between roles are often unclear in an organization. Who is the provider? Who is the owner responsible for doing all of this? Who is the consumer? What are the teams that will consume the platform? With blurry boundaries, it's complicated to understand who is responsible for what and how our decisions will affect other teams.

Evan Bottcher defines a digital platform as “a foundation of self-service APIs, tools, and services, but also knowledge and support, and everything arranged as a compelling internal product." We know that self-service tools and APIs are important. They›re critical in allowing a lot of teams to be more independent and to work autonomously. But I want to bring attention to his mention of “knowledge and support". That implies that we have teams running the platform that are focused on helping product teams understand and adopt it, besides providing support when problems arise in the platform.

Another key aspect is to understand the platform as “a compelling internal product", not a mandatory platform with shared services that we impose on everyone else. As an industry, we've been doing that for a long time, and it simply doesn't work very well. It often creates more pain than the benefits it provides for the teams forced to use a platform that is supposed to be a silver bullet. And we know silver bullets don't exist.

We have to think about the platform as a product. It's meant for our internal teams, but it's still a product. We're going to see what that implies in practice.

The key idea is that Kubernetes by itself is not a platform. It's a foundation. Yes, it provides all this great functionality — autoscaling, self-healing, service discovery, you name it — but a good product is more than just a set of features. We need to think about how easy it is to adopt, its reliability, and the support for the platform.

A good platform, as Bottcher says, should create a path of least resistance. Essentially, the right thing to do should be the easiest thing to do with the platform. We can't just say that whatever Kubernetes does is the right thing. The right thing depends on your context, on the teams that are going to consume the platform, which services they need the most, what kind of help they need to onboard, and so on.

One of the hard things about platforms is that the needs of the internal teams are going to change, with respect to old and new customers. Teams that are consuming the platform are probably going to have more specific requests and requirements over time. At the same time, the platform must remain understandable and usable for new teams or new engineers adopting it. The needs and the technology ecosystem keep evolving.

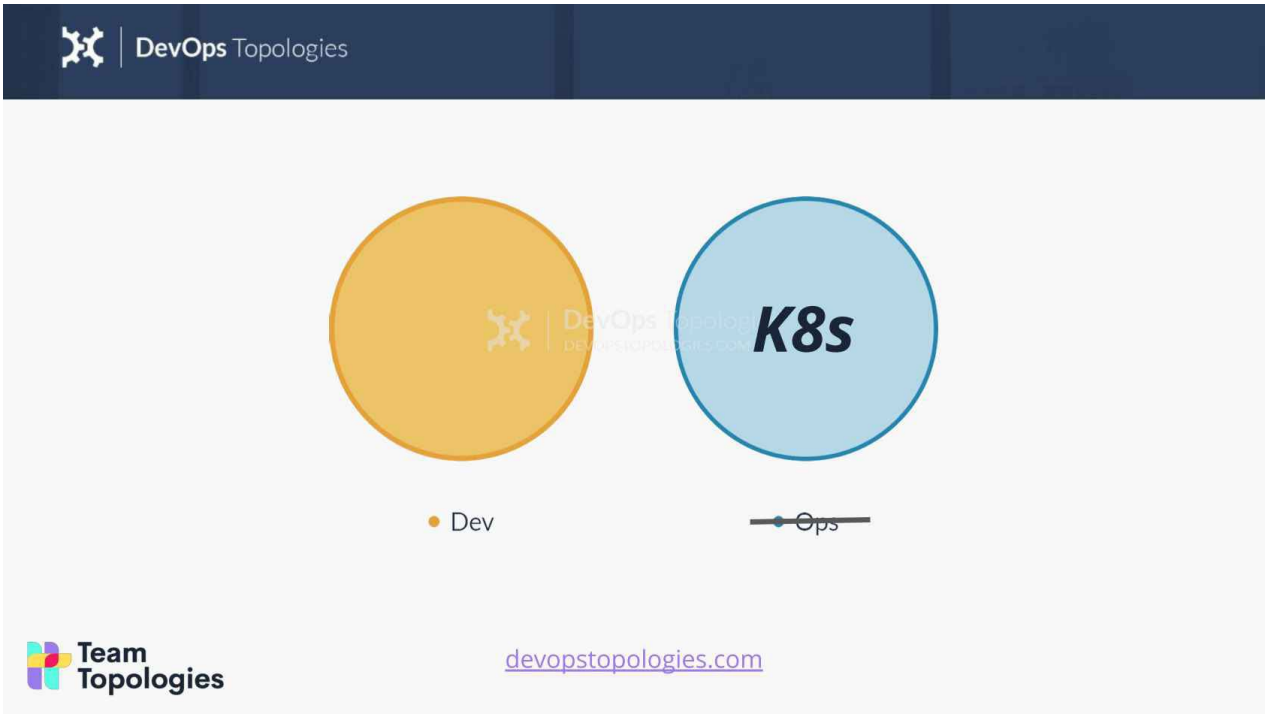


Figure 1: Avoid creating isolated teams when bringing Kubernetes into the organization.

Team cognitive load

Adopting Kubernetes is not a small change. It's more like an elephant charging into a room. We have to adopt this technology in a way that does not cause more pain than the benefits it brings. We don't want to end up with something like Figure 1, that resembles life before the DevOps movement, with isolated groups. It's the Kubernetes team now rather than the operations team, but if we create an isolationist approach where one group makes decisions independent from the other group, we're going to run into the same kinds of problems.

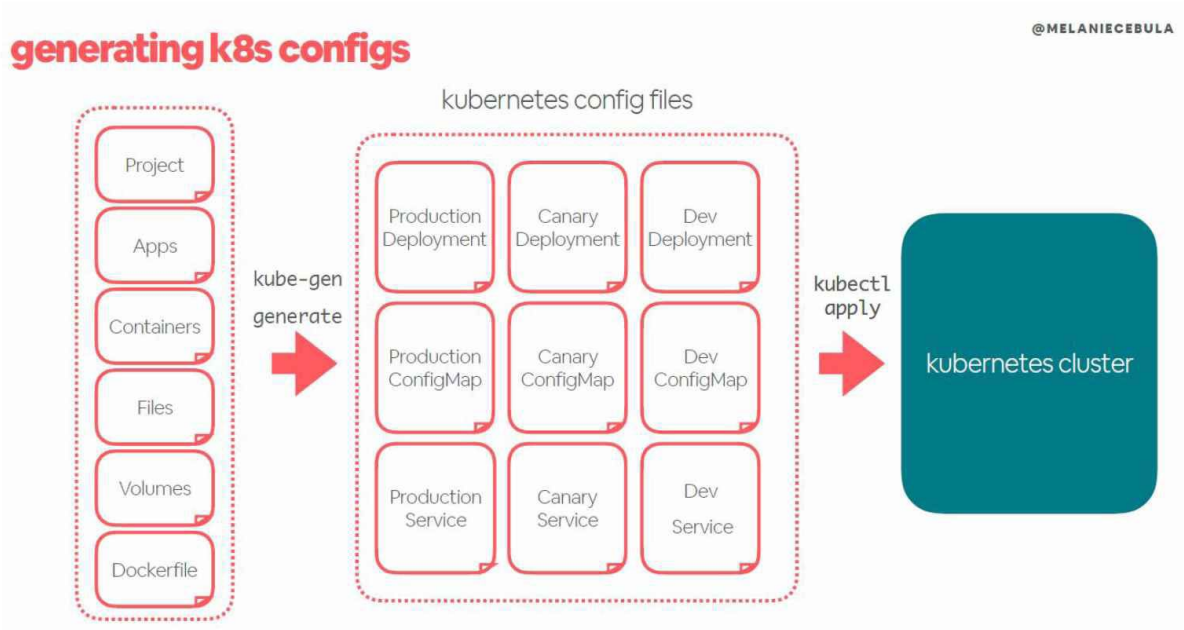
If we make platform decisions without considering the impact on consumers, we're going to increase pain in the form of their cognitive load, the amount of effort our teams must put out to understand and use the platform.

John Sweller formulated cognitive load as "the total amount of mental effort being used in the working memory". We can break this down into three types of cognitive load: intrinsic, extraneous, and germane.

We can map these types of cognitive load in software delivery: intrinsic cognitive load is the skills I need to do my work; extraneous load is the mechanics, things that need to happen to deliver the value; and germane is the domain focus.

Intrinsic cognitive load, if I'm a Java developer, is knowing how to write classes in Java. If I don't know how to do that, this takes effort. I have to Google it or I have to try to remember it. But we know how to minimize intrinsic cognitive load. We can have classical training, pair programming, mentoring, code reviews, and all techniques that help people improve their skills.

Extraneous cognitive load includes any task needed to deliver the work I'm doing to the customers or to production. Having to remember how to deploy this application, how to access a staging environment, or how to clean up test data are all things that are not directly related to the problem to solve but are things that I need to get done. Team topologies and platform teams in particular can minimize extraneous cognitive



48

Figure 2: Airbnb uses kube-gen to simplify their Kubernetes ecosystem.

load. This is what we're going to explore throughout the rest of this article.

Finally, germane cognitive load is knowledge in my business domain or problem space. For example, if I'm working in private banking, I need to know how bank transfers work.

The point of minimizing extraneous cognitive load is to free up as much memory available for focus on the germane cognitive load.

Jo Pearce's "Hacking Your Head" articles and presentations go deeper into this.

The general principle is to be mindful of the impact of your platform choices on your teams' cognitive loads.

Case studies

I mentioned Airbnb engineer Melanie Cebula's excellent talk at Qcon so let's look at how that company reduced the cognitive load of their development teams.

"The best part of my day is when I update 10 different YAML files to deploy a one-line code change," said no one, ever. Airbnb teams were feeling this sort of pain as they embarked on their Kubernetes journey.

To reduce this cognitive load, they created a simple command-line tool, kube-gen, which allows the application teams or service teams to focus on a smaller set of configurations and details, which are specific to their own project or services. A team needs to configure only those settings like files or volumes specifically related to the germane aspect of their application.

The kube-gen tool then generates the boilerplate code configuration for each environment, in their case: production, canary, and development environments. This makes it much easier for development teams to focus on the germane parts of their work.

As Airbnb did, we essentially want to clarify the boundaries of the services provided by the platform and provide good abstractions so we reduce the cognitive load on each service team.

In Team Topologies, we talk about four types of teams, shown in Figure 3. Stream-aligned teams provide the end-customer value, they're the heartbeat that delivers business value. The three other types of teams provide support and help reduce cognitive load. The platform team shields the details of the lower-level services that these teams need to use for deployment, monitoring, CI/CD, and other lifecycle supporting services.

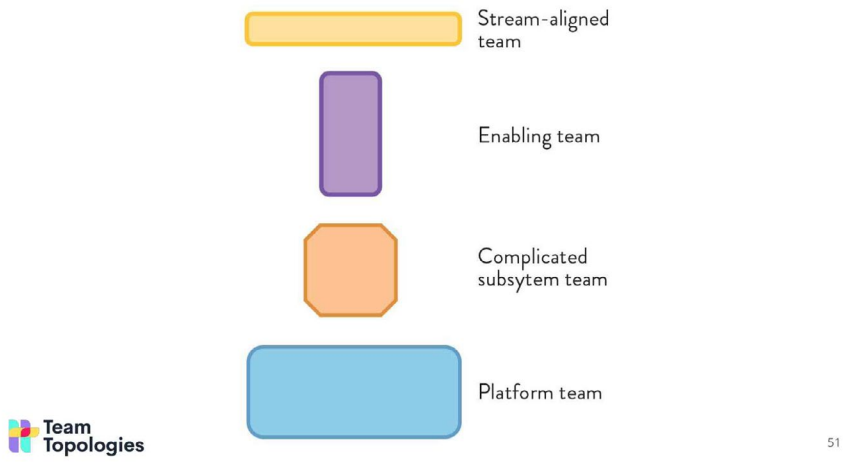


Figure 3: Four types of teams.

The stream-aligned team resembles what organizations variously call a product team, DevOps team, or build-and-run team — these teams have end-to-end ownership of the services that they deliver. They have runtime ownership, and they can take feedback from monitoring or live customer usage for improving the next iteration of their service or application. We call this a "stream-aligned team" for two reasons. First, "product" is an overloaded term; as our systems become more and more complex, the less precise it is to define a standalone product. Second, we want to acknowledge the different types of streams beyond just the business-value streams; it can be compliance, specific user personas, or whatever makes sense for aligning a team with the value it provides.

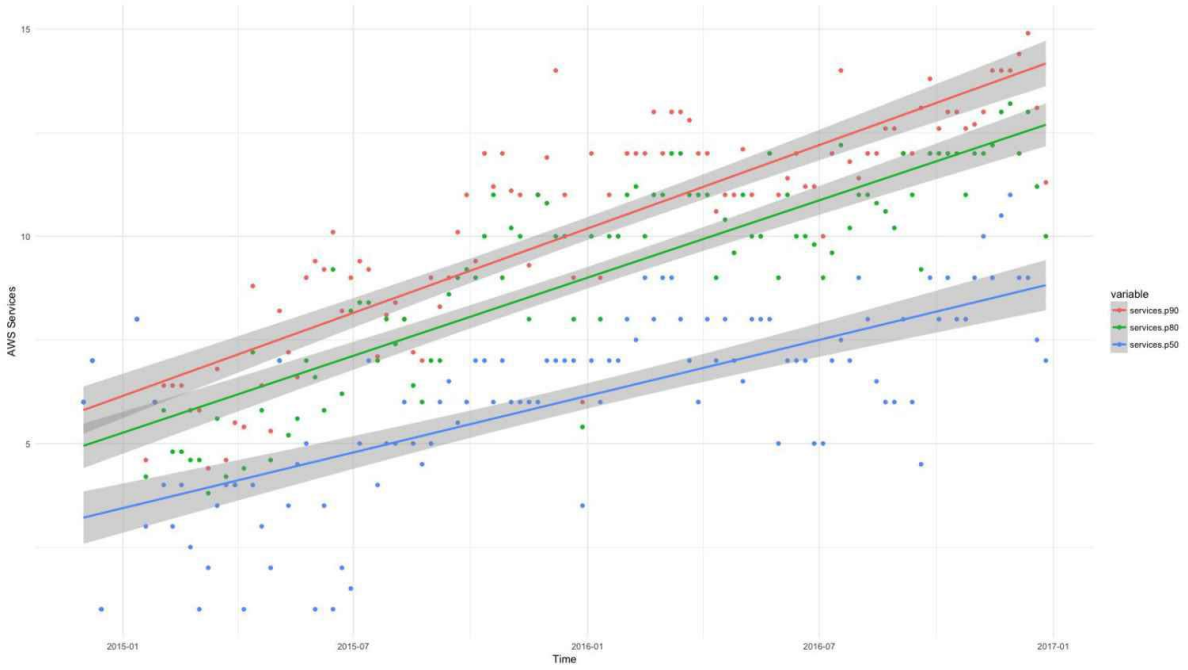
Another case study is uSwitch, which helps users in the UK compare utility providers and home services and makes it easy to switch between them.

A couple of years ago, Paul Ingles, head of engineering at uSwitch, wrote "Convergence to Kubernetes", which brought together the adoption of Kubernetes technology, how it helps or hurts teams and their work, and data for meaningful analysis. Figure 4, from that article, measures all the different teams' low-level AWS service calls at uSwitch.

When uSwitch started, every team was responsible for their own service, and they were as autonomous as possible. Teams were responsible for creating their own AWS accounts, security groups, networking, etc. uSwitch noticed that the number of calls to these services was increasing, correlated with a feeling that teams were getting slower at delivering new features and value for the business.

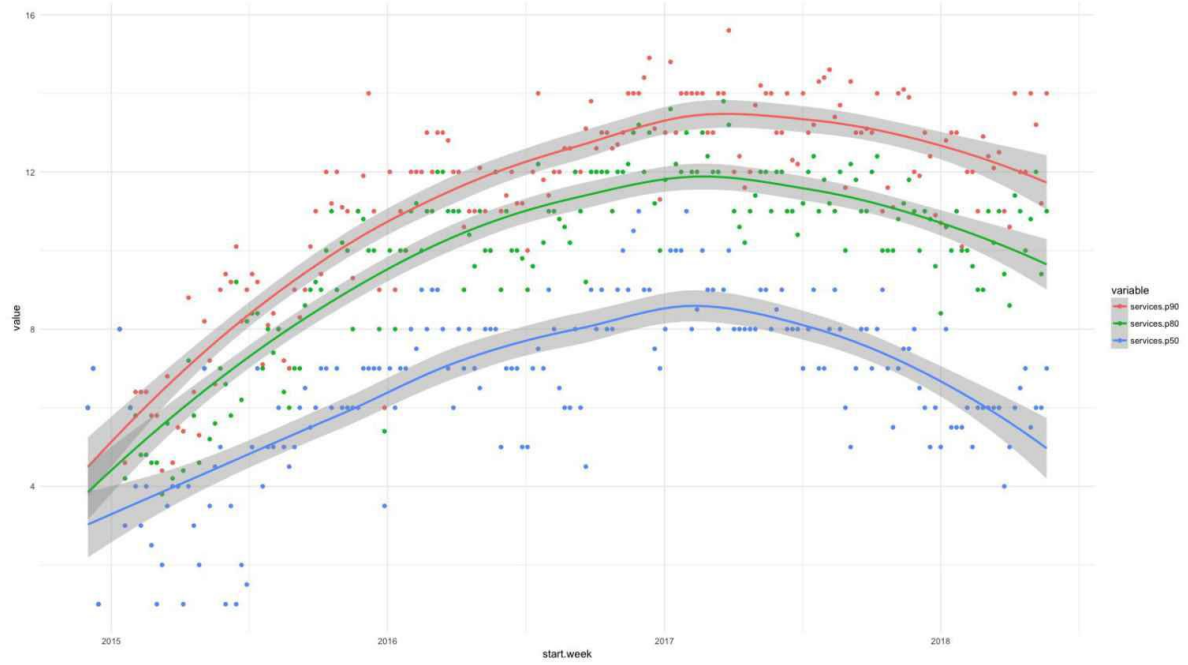
What uSwitch wanted to do by adopting Kubernetes was not only to bring in the technology but also to change the organizational structure and introduce an infrastructure platform team. This would address the increasing cognitive load generated by teams having to understand these different AWS services at a low level.

That was a powerful idea. Once uSwitch introduced the platform, traffic directly through AWS decreased. The curves of calls in Figure 5 is a proxy for the cognitive load on the application teams. This concept aligns with a platform team's purpose: to enable stream-aligned teams to work more autonomously with self-service capabilities and reduced extraneous cognitive load. This is a very different conceptual starting point from



57

Figure 4: Use of low-level AWS services over time.



59

Figure 5: The number of low-level AWS service calls dropped after uSwitch introduced their Kubernetes-based platform.

saying, "Well, we're going to put all shared services in a platform."

Ingles also wrote that they wanted to keep the principles they had in place before around team autonomy and teams working with minimal coordination, by providing a self-service infrastructure platform.

Treat the platform as a product

We talk about treating the platform as a product — an internal product, but still a product. That means that we should think about its reliability, its fit for purpose, and the developer experience (DevEx) while using it.

First, for the platform to be reliable, we need to have on-call support, because now the platform sits in the path of production. If our platform's monitoring services fail or we run out of storage space for logs, for example, the customer-facing teams are going to suffer. They need someone who provides support, tells them what's going on, and estimates the expected time for a fix. It should also be easy to understand the status of the platform services and we should have clear, established communication channels between the platform and the stream teams in order to reduce cognitive load in communications. Finally, there needs to be coordination with potentially affected teams for any planned downtime.

Secondly, having a platform that's fit for purpose means that we use techniques like prototyping and we get regular feedback from the internal customers. We use iterative practices like agile, pair programming, or TDD for faster delivery with higher quality. Importantly, we should focus on having fewer services of higher quality and availability rather than trying to build every service that we can imagine to be potentially useful. We need to focus on what teams really need and make sure those services are of high quality. This means we need very good product management to understand priorities, to establish a clear but flexible roadmap, and so on.

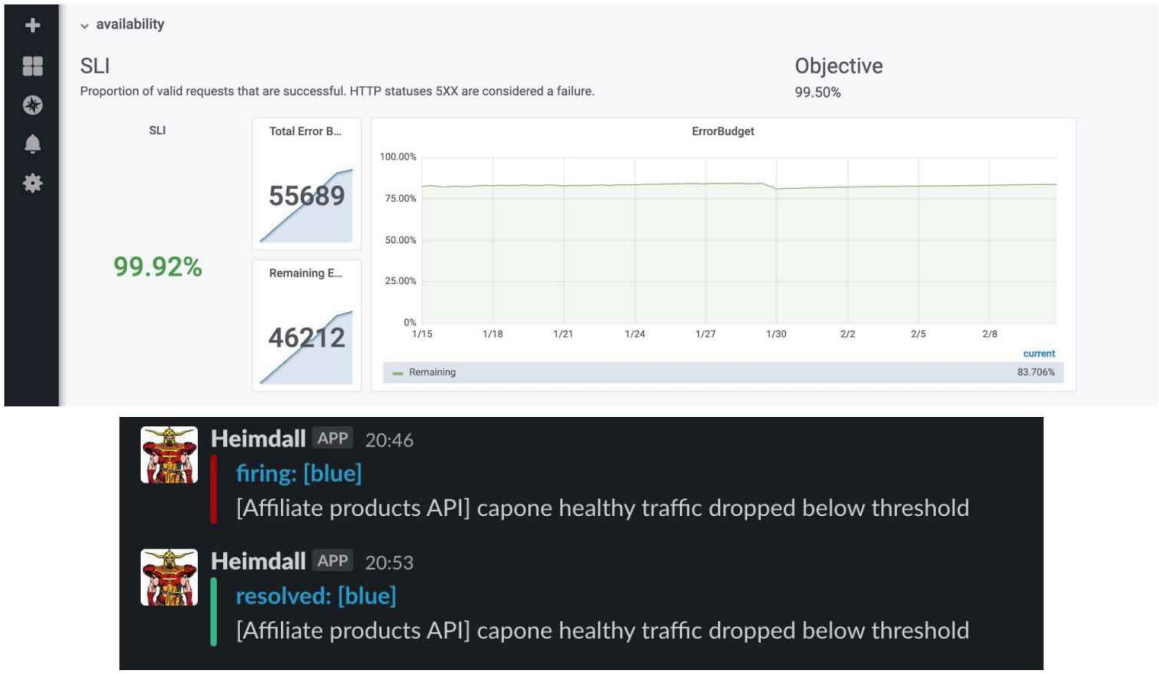
Finally, development teams and the platform should speak the same language in order to maximize the experience and usability. The platform should provide the services in a straightforward way. Sometimes, we might need to compromise. If development teams are not familiar with YAML, we might think of the low cost, low effort, and long-term gain of training all development teams in YAML. But these decisions are not always straightforward and we should never make them without considering the impact on the development teams or the consuming teams. We should provide the right levels of abstraction for our teams today, but the context may change in the future. We might adopt better or higher levels of abstraction, but we always should look at what makes sense given the current maturity and engineering practices of our teams.

Kubernetes helped uSwitch establish these more application-focused abstractions for things like services, deployments, and ingress rather than the lower-level service abstractions that they were using before with AWS. It also helped them minimize coordination, which was another of the key principles.

I spoke with Ingles and with Tom Booth, at the time infrastructure lead at uSwitch, about what they did and how they did it.

Some of the things that the platform team helped the service teams with were providing dynamic database credentials and multi-cluster load balancing. They also made it easier for service teams to get alerts for their customer-facing services, define service-level objectives (SLOs), and make all that more visible. The platform make it easy for teams to configure and monitor their SLOs with dashboards — if an indicator drops below a threshold, notifications go out in Slack, as shown in Figure 6.

Teams found it easy to adopt these new practices. uSwitch teams are familiar with YAML



72

Figure 6: An example of a SLO threshold notification in Slack at uSwitch.

and can quickly configure these services and benefit from them.

Achievements beyond the technical aspects

uSwitch's journey is fascinating beyond the technical achievements. They started this infrastructure adoption in 2018 with only a few services. They identified their first customer to be one team that was struggling without any centralized logging, metrics, or autoscaling. They recognized that growing services around these aspects would be a successful beginning.

In time, the platform team started to define their own SLAs and SLOs for the Kubernetes platform, serving as an example for the rest of the company and highlighting the improvements in performance, latency, and reliability. Other teams could observe and make an informed decision to adopt the platform services or not. Remember, it was never mandated but always optional.

Booth told me that uSwitch saw traffic increasing through the Kubernetes platform versus what was going directly through AWS and this gave them some idea of how much adoption was taking place. Later, the team addressed some critical cross-functional gaps in security, GDPR data privacy and handling of alerts and SLOs.

One team, one of the more advanced in both engineering terms and revenue generation, was already doing everything that the Kubernetes platform could provide. They had no significant motivation to adopt the platform — until they were sure that it provided the same functionality with the increased levels of reliability, performance, and so on. It no longer made sense for them to take care of all these infrastructure aspects on their own. The team switched to use the Kubernetes platform and increased its capacity to focus on the business aspects of the service. That was the "ultimate" prize for the platform team, to gain the adoption from the most advanced engineering team in the organization.

Four Key Metrics

Having metrics can be quite useful. As the platform should be considered a product, we can look at product metrics — and the categories in Figure 7 come from

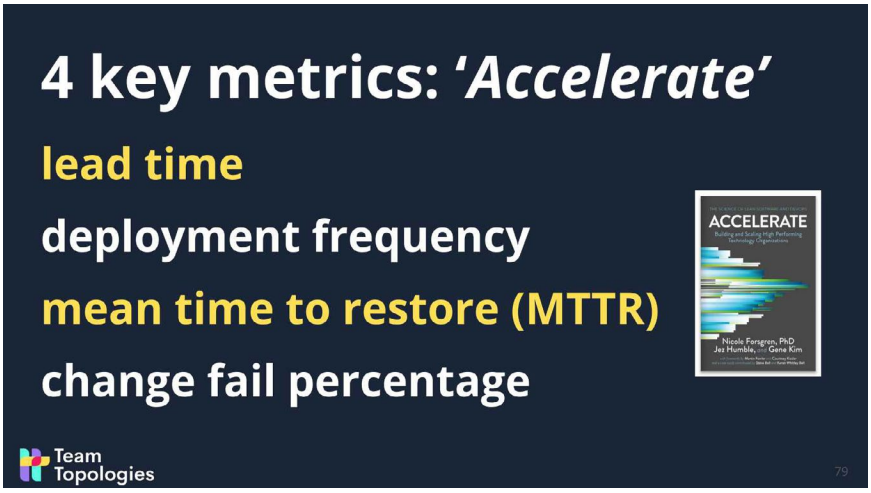


Figure 7: Four key metrics from the Accelerate book.

Figure 8: The survey that Twiliio's platform team sends out to engineering teams.

the book Accelerate by Nicole Forsgren, Jez Humble, and Gene Kim.

They write that high-performing teams do very well at these key metrics around lead time, deployment frequency, MTTR, and change failure rate. We can use this to help guide our own platform services delivery and operations.

Besides the Accelerate metrics, user satisfaction is another useful and important measurement. If we're creating a product for users, we want to make sure it helps them do their job, that they're happy with it, and that they recommend it to others. There's a simple example from Twilio.

Every quarter or so, their platform team surveys the engineering teams with some questions (Figure 8) on how well the platform helps them build, deliver, and run their service and how compelling it is to use.

With this simple questionnaire, you can look at overall user satisfaction over time and see trends. It's not just about the general level of satisfaction with the technical services, but also with the support from the platform team. Dissatisfaction may arise because the platform team was too busy to listen to feedback for a period of time, for example. It's not just about the technology. It's also about interactions among teams.

Yet another important area to measure is around platform adoption and engagement. In the end, for the platform to be successful, it must be adopted. That means it's serving its purpose. At the most basic level, we can look at how many teams in the organization are using the platform versus how many teams are not.

We can also look at adoption per platform service, or even adoption of a particular service functionality. That will help understand the success of a service or feature. If we have a service that we expected to be easily adopted but many teams are lagging behind, we can look for what may have caused that.

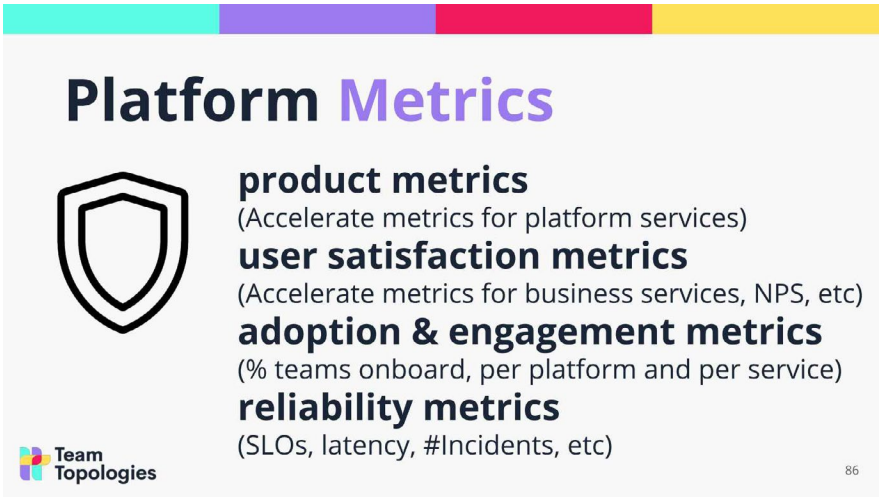


Figure 9: Useful metrics for your platform as a product.

Finally, measuring the reliability of the platform itself, as uSwitch did, is important as well. They had their own SLOs for the platform and this was available to all teams. Making sure we provide that information is quite important.

Figure 9 shows some examples of the metrics categories. Each organization with its own context might have different specific metrics, but the types and categories that we should be looking at should be more or less the same.

Team interactions

The success of the platform team is the success of the stream-aligned teams. These two things go together. It's the same for other types of supporting teams. Team interactions are critical because the definition of success is no longer just about making technology available, it's about helping consumers of that technology get the expected benefits in terms of speed, quality, and operability.

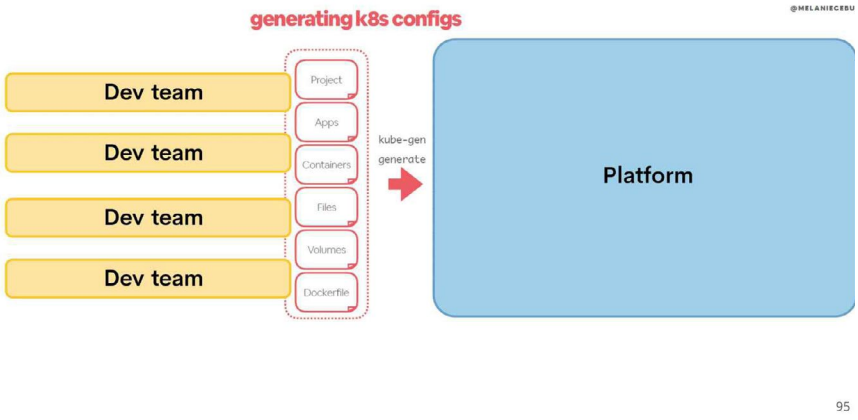


Figure 10: Airbnb's platform team provides an abstraction of the underlying Kubernetes architecture.

Airbnb effectively had a platform team (although internally called Infrastructure team) to abstract a lot of the underlying details of the Kubernetes platform, as shown in Figure 10. This clarifies the platform's boundaries for the development teams and exposes those teams to a much smaller cognitive load than telling them to use Kubernetes and to read the official documentation to understand how it works. That's a huge task that requires a lot of effort. This platform-team approach reduces cognitive load by providing more tightly focused services that meet our teams' needs.

To accomplish this, we need adequate behaviors and interactions between teams. When we start a new platform service or change an existing one, then we expect strong collaboration between the platform team and the first stream-aligned teams that will use the new/changed service. Right from the beginning of the discovery period, a platform team should understand a streaming team's needs, looking for the simplest solutions and interfaces that meet those needs. Once the stream-aligned team starta using the service,

then the platform team's focus should move on to supporting the service and providing up-to-date, easy to follow documentation for onboarding new users. The teams no longer have to collaborate as much and the platform team is focused on providing a good service. We call this interaction mode X-as-a-Service.

Note that this doesn't mean that the platform hides everything and the development teams are not allowed to understand what's going on behind the scenes. That's not the point. Everyone knows that it's a Kubernetes-based platform and we should not forbid teams from offering feedback or suggesting new tools or methods. We should actually promote that kind of engagement and discussion between stream-aligned teams and platform teams.

For example, troubleshooting services in Kubernetes can be quite complicated. Figure 11 shows only the top half of a flow chart for diagnosing a deployment issue in Kubernetes. This is not something we want our engineering teams to have to go through every time there's a problem. Neither did Airbnb.

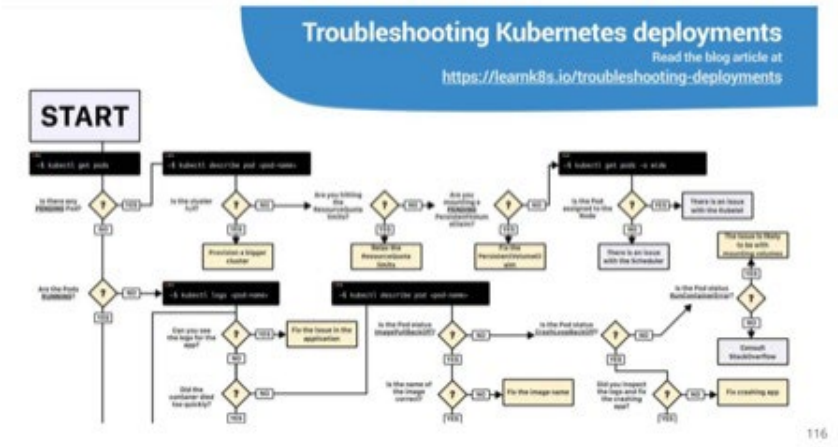


Figure 11: Half of a Kubernetes deployment troubleshooting flow chart.

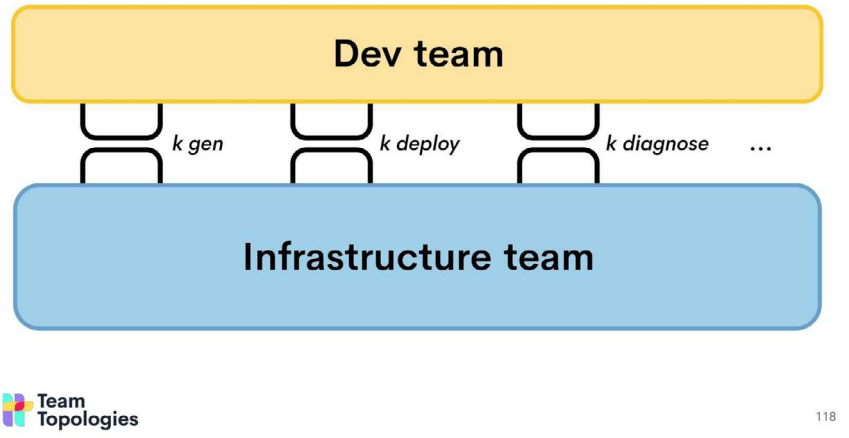


Figure 12: Services provided by the infrastructure (platform) team at Airbnb.

Airbnb initiated a discovery period during which teams closely collaborated to understand what kind of information they needed in order to diagnose a problem and what kinds of problems regularly occurred. This led to an agreement on what a troubleshooting service should look like. Eventually, the service became clear and stable enough to be consumed by all the stream-aligned teams.

Airbnb already provided these two services in their platform: kube-gen and kube-deploy. The new troubleshooting service, kube-diagnose, collected all the relevant logs, as well as all sorts of status checks and other useful data to simplify the lives of their development teams. Diagnosing got a lot easier, with teams focused on potential causes for problems rather than remembering where all the data was or which steps to get them.

Figure 13 shows just how broad the cloud-native landscape is. We don't want stream-aligned teams to have to deal with that on their own. Part of the role of a platform team is to follow the

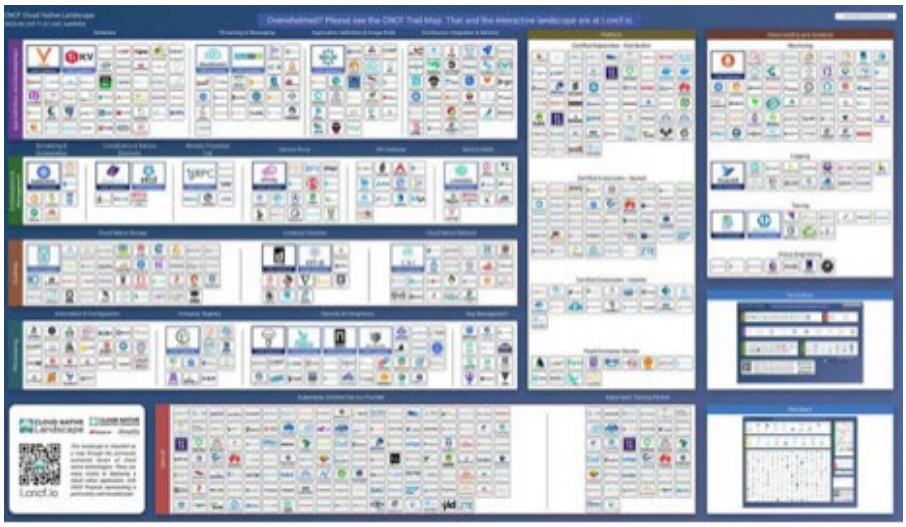


Figure 13: The cloud-native landscape.

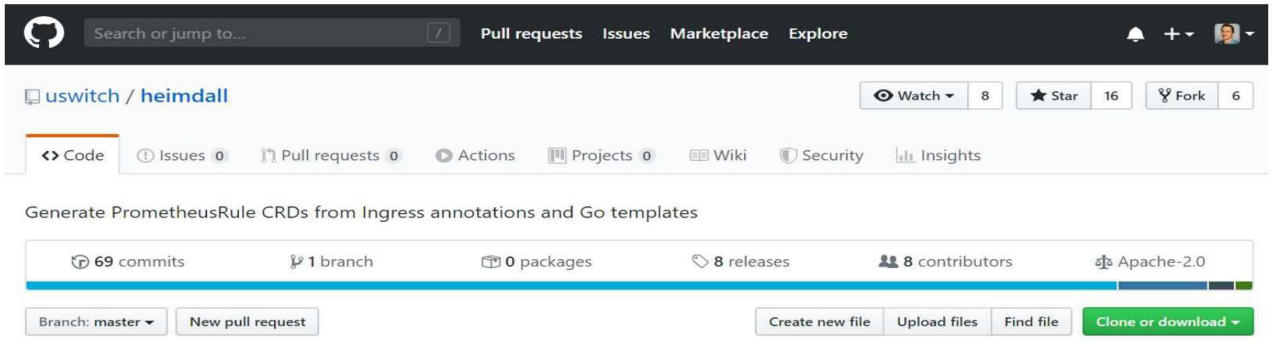
technology lifecycle. We know how important that is. Let's imagine there's a CNCF tool that just graduated which would help reduce the amount of custom code (or get rid of an older, less reliable tool) in our internal platform today. If we can adopt this new tool in a transparent fashion that doesn't leak into the platform service usage by stream-aligned teams, then we have an easier path forward in terms of keeping up with this ever-evolving landscape.

If, on the contrary, a new technology we'd like to adopt in the platform implies a change in one or more service interfaces that means we need to consult the stream-aligned teams to understand the effort required of them and evaluate the trade-offs at play. Are we getting more benefit in the long run than the pain of migration/adaptation today?

In any case, the internal platform approach helps us make visible the evolution of the technology inside our platform.

The same goes for adopting open source solutions. For example, uSwitch open sourced the Heimdall application that provides the chat tool integration with the SLOs dashboard service. And there are many more open-source tools. Zalando, for example, has really cool stuff around cluster lifecycle management.

The point being that if a piece of open-source technology makes sense for us, we can adopt it more easily if they sit under a certain level of abstraction in the platform. And if it is not a transparent change, we can always collaborate with stream-aligned teams to identify what would need to change in terms of their usage of the affected service(s).



123

Figure 14: uSwitch open-sourced Heimdall tool that their platform team created for internal use initially.

Starting with team-centric approach for Kubernetes adoption

There are three keys to a team-focused approach to Kubernetes adoption.

We start by assessing the cognitive load on our development teams or stream-aligned teams. Let's determine how easy it is for them to understand the current platform based on Kubernetes. What abstractions do they need to know about? Is this easy for them or are they struggling? It's not an exact science, but by asking these questions we can get a feel for what problems they're facing and need help with. The Airbnb case study is important because it made it clear that a tool-based approach to Kubernetes adoption brings with it difficulties and anxiety if people have to use this all-new platform without proper support, DevEx, or collaboration to understand their real needs.

Next, we should clarify our platform. This often is simple, but we don't always do it. List exactly all services we have in the platform, who is responsible for each, and all the other aspects of a digital platform that I've mentioned like responsibility for on-call support, communication mechanisms, etc. All this should be clear. We can start immediately by looking at the gaps between our actual Kubernetes implementation and an ideal digital platform, and addressing those.

Finally, clarify the team interactions. Be more intentional about when we should collaborate and when should we expect to consume a service independently. Determine how we develop new services and who needs to be involved and for how long. We shouldn't just say that we're going to collaborate and leave it as an open-ended interaction. Establish an expected duration for the collaboration, for example two weeks to understand what teams need from a new platform service and how the service interface should look like, before we're actually building out such service.

Do the necessary discovery first, then focus on functionality and reliability. At some point, the service can become “generally available” and interactions evolve to “X as a service".

There are many good platform examples to look at from like Zalando, Twilio, Adidas, Mercedes, etc.

The common thread among them isa digital platform approach consisting not only oftechnical services but good support, on-call, high quality documentation and all these things that make the platform easy for their teams to use and accelerate their capacity to deliver and operate their software more autonomously. I also wrote an article for TechBeacon that goes a bit deeper into these ideas.

TL;DR

- Kubernetes itself is not a platform but only the foundational element of an ecosystem not only of tools, and services, but also offering support as part of a compelling internal product.
- Platform teams should provide useful abstractions of Kubernetes complexities to reduce cognitive load on stream teams.
- The needs of platform users change and a platform team needs to ease their journey forward.
- Changes to the platform should meet the needs of stream teams, prompting a collaborative discovery phase with the platform team followed by stabilizing the new features (or service) before they can be consumed by other stream teams in a self-service fashion.
- A team-focused Kubernetes adoption requires an assessment of cognitive load and tradeoffs, clear platform and service definitions, and defined team interactions.

SPONSORED ARTICLE

The Cause and Effects of Cluster Sprawl

Kubernetes gives organizations the ability to run Kubernetes clusters at scale across different cloud infrastructures and distributions. Unfortunately, this is where many of the challenges begin. As the number of clusters and workloads grow, they are being managed independently with very little consistency. The result is a chaotic environment of cluster and workload sprawl, creating redundant efforts and wasted resources, a myriad of governance challenges, and a software environment that is difficult if not impossible for the organization to support.

Below are four ways that cluster sprawl impacts your multi-cluster operations.

Provisioning

In adopting Kubernetes, organizations need to maintain granular control over how and where new clusters are used, as well as who is able to engage in policy and operational needs of those services. However, various teams are provisioning and using clusters across a wide variety of projects with very little consistency, unified management, and visibility to empower divisions of labor across roles in the organization.

As an organization grows in its use of infrastructure as a service, its ability to monitor, manage, and optimize those resources in a cost-effective manner often fails to keep pace. And this problem only grows in complexity as new clusters are added, new users on-board, off-board, or change teams, and projects multiply.

Configuration

As your organization expands its usage of Kubernetes, clusters will exist in different pockets each with differing policies, roles, and configurations in their usage.

Individual teams need to keep their environments secure, well patched up, and up-to-date. If you only have one or two ways that things are configured that means your staff is more likely to do the right thing. Conversely, when there are infinite levels of variation on a hundred different ways of configuring and using Kubernetes, it makes it incredibly challenging to simplify and create consistency across organizational clusters. Teams can't configure clusters and services based on intended architectures from the beginning or define best practice architectures for their Kubernetes environments. They also lose the flexibility to configure access and delegate responsibilities as a user's role within the organization changes.

Without a consistent way to configure new clusters, it can have a serious impact on the speed and efficiency of the deployment process.

Deployment

Deploying each service is time-consuming and requires significant engineering effort. And if these technologies aren't configured correctly or standardized, the results can be detrimental to the business.

Services are deployed repeatedly and independently within and across clusters. In addition, all configuration and policy management, such as roles and secrets, are repeated, wasting time and creating the opportunity for mistakes.

Please read the full-length version of this article here.

The Evolution of Distributed Systems on Kubernetes

by **Bilgin Ibryam**, Product Manager at Red Hat

At the QCon in March, I gave a talk on the evolution of distributed systems with Kubernetes. First, I want to start with a question, what comes after microservices? I'm sure you all have an answer to that, and I have mine too. You'll find out at the end what I think that will be. To get there, I suggest we look at what are the needs of the distributed systems. And how those needs have been evolving over the years, starting with monolithic applications to Kubernetes and with more recent projects such as Dapr, Istio, Knative, and how they are changing the way we do distributed systems. We will try to make some predictions about the future.

Modern Distributed Applications

To set a little more context on this talk, when I say distributed systems, what I have in mind is systems composed of multiple components, hundreds of those. These components can be stateful, stateless, or serverless. Moreover, these components can be created in different languages running on hybrid environments and developing open-source technologies, open standards, and interoperability. I'm sure you can make such systems using closed source software or create them on AWS and other places. For this talk specifically, I'm looking at the Kubernetes ecosystem and how you can create such a system on the Kubernetes platform.

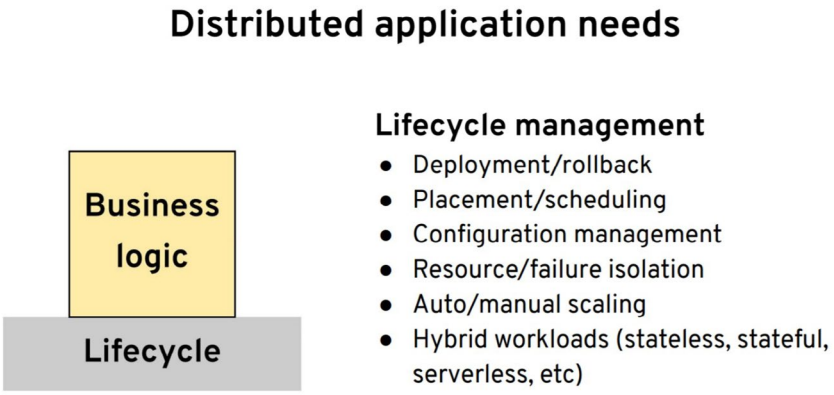
Let's start with the needs of distributed systems. What I have in mind is we want to create an application or service and write some business logic. What else do we need from the platform from our runtime to build distributed systems? At the foundation, at the beginning is we want some lifecycle capabilities.

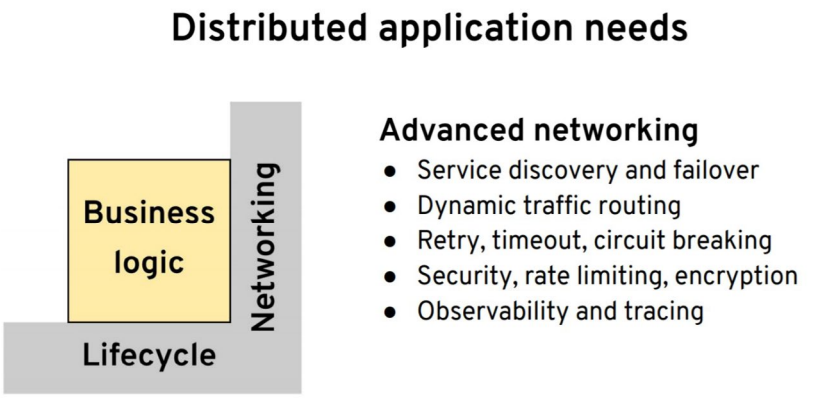
When you write your application in any language, then we want to have the ability to package and deploy that application reliably, to do rollbacks, health checks. And be able to place the application on different nodes and do resource isolation, scaling, configuration management, and all of these things. These are the very first things you would need to create a distributed application.

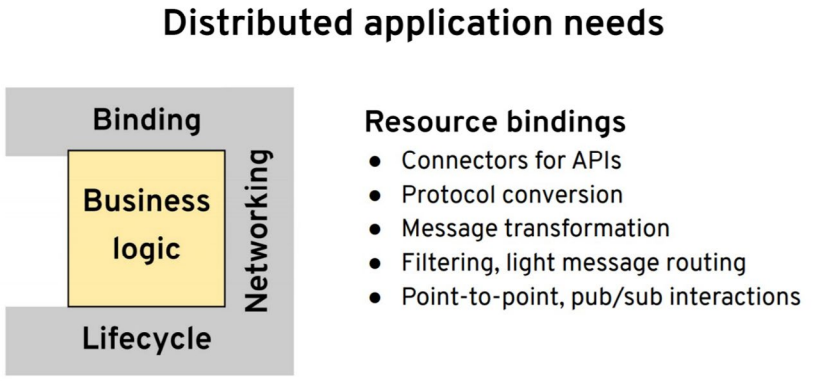
The second pillar is around networking. Once we have an application, we want it to reliably connect to other services, whether within the cluster or in the outside world. We want to have abilities such as service discovery, load balancing.
We want to do traffic shifting, whether for different release strategies or some other reasons.

Then we want to have an ability to do resilient communication with other systems, whether that is through retries, timeouts, circuit breakers, of course.
Have security in place, and get adequate monitoring, tracing, observability, and all that.

Once we have networking, the next thing is we want to have the ability to talk to different APIs and endpoints, i.e., resource bindings - to talk to other protocols and different data formats. Maybe even be able to transform from one data format to another one. I would also include here things such as light filtering, that is, when we subscribe to a topic, maybe we are interested only in certain events.







What do you think is the last category? It is state. When I say state and stateful abstractions, I'm not talking about the actual state management, such as what a database does or a file system. I'm talking more about developer abstractions that behind the scenes rely on the state. Probably, you need to have the ability to do workflow management.

Maybe you want to manage long-running processes or do temporal scheduling or some cron jobs to run your service periodically. Perhaps you also want to do distributed caching, have idempotence, or be able to do rollbacks. All of these are developer-level primitives, but behind the scenes, they rely on having some state. You want to have these abstractions at your disposal to create sound distributed systems.

We will use this framework of distributed system primitives to evaluate how these have been changing on Kubernetes and other projects.

Monolithic Architectures - Traditional Middleware Capabilities

Suppose we start with the monolithic architectures and how we get those capabilities. In that case, the first thing is when I say monolith, and what I have in mind, in the context of distributed applications, is the ESB. ESBs are pretty powerful, and when we check our list of needs, we would say that ESBs had excellent support for all stateful abstractions.

With an ESB, you could do the orchestration of long-running processes, do distributed transactions, rollbacks, and idempotence. Furthermore, ESB's also provide outstanding resource binding capabilities and have hundreds of connectors,

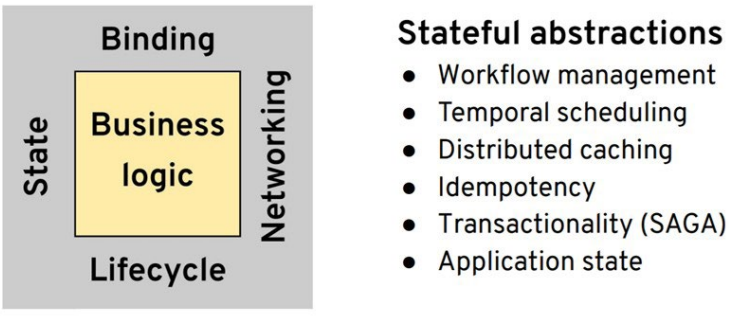
support transformation, orchestration, and even have networking capabilities. And lastly, an ESB can even do service discovery and load balancing.

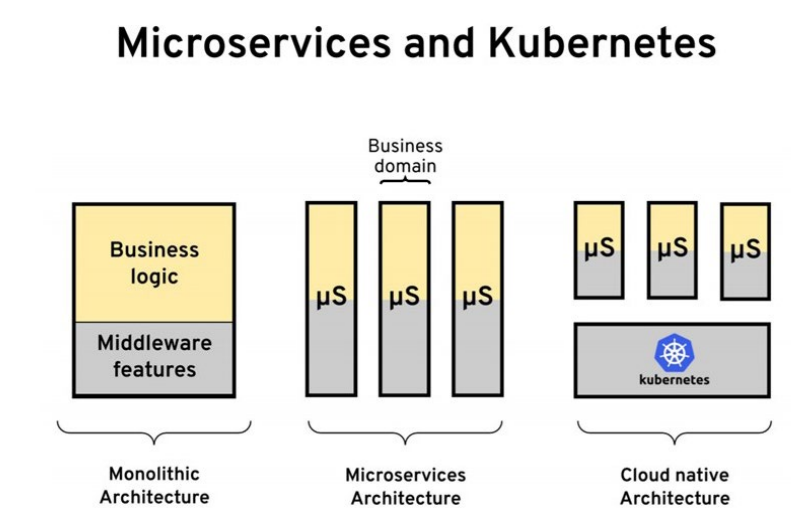
It has all things around the resiliency of the networking connection so that it can do retries. Probably, because by nature, an ESB is not very distributed, it doesn't need very advanced networking and releases capabilities. Where ESB lacks is primarily lifecycle management. Because it's a single runtime, the first thing is you are limited to using a single language. That's typically the language that the actual runtime is created in, Java, or .NET, or something else. Then, because it's a single runtime, we cannot easily do declarative deployments or do an automatic placement. The deployments are pretty big, quite heavy, so it usually involves human interaction. And another difficulty with such a monolithic architecture is scaling: "We cannot scale individual components."

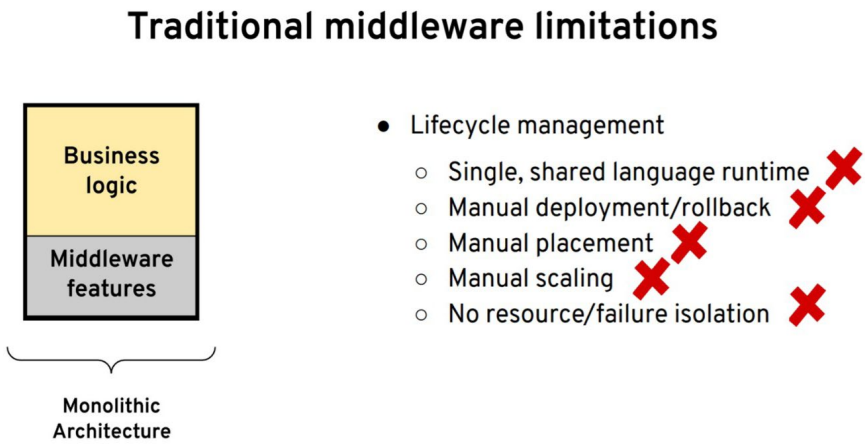
Last but not least, around isolation, whether that's resource isolation or fault isolation. None of these can be done with monolithic architectures. From our needs' framework point of view, the ESB's monolithic architectures don't qualify.

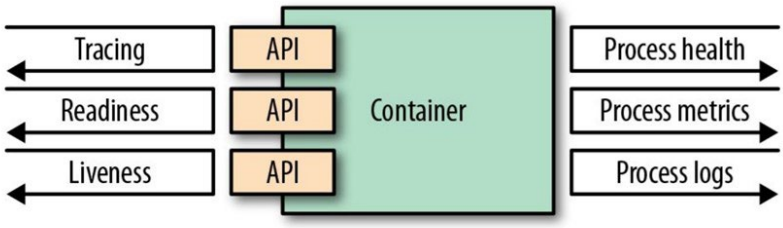
Cloud-native Architectures - Micrososervices and Kubernetes

Next, I suggest we look at cloud-native architectures and how those needs have been changing.









If we look at a very high level, how those architectures have been changing, cloud-native probably started with the microservices movement.

Microservices allow us to split a monolithic application by business domain. It turned out that containers and Kubernetes are actually a good platform for managing those microservices. Let's see some of the concrete features and capabilities that Kubernetes becomes particularly attractive for microservices.

In the very beginning, the ability to do health probes is what made Kubernetes popular. In practice, it means when you deploy your container in a pod, Kubernetes will check the health of your process. Typically, that process model is not good enough. You still may have a process that's up and running, but it's not healthy. That's why there is also the option of using readiness and liveness checks. Kubernetes will do a readiness check to decide when your application is ready to accept traffic during startup. It will do a liveness check to check the health of your service continuously. Before Kubernetes, this wasn't very popular, but today almost all languages, all frameworks, all runtimes have health checking capabilities where you can quickly start an endpoint.

The next thing that Kubernetes introduced is around the managed lifecycle of your application - what I mean is that you are no longer in control of when your service will start up and when it will shut down. You trust the platform to do that. Kubernetes can start up your application; it can shut it down, move it around on the different nodes. For that to work, you have to properly implement the events

that the platform is telling you during startup and shutdown.

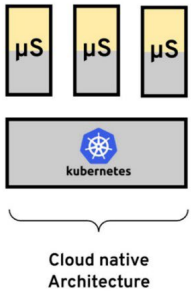
Another thing that Kubernetes made popular is around deployments and having those declaratively. That means you don't have to start the service anymore; check the logs whether it has started. You don't have to upgrade instances manually - Kubernetes with declarative deployments can do that for you. Depending on the strategy you chose, it can stop old instances and start new ones. Moreover, if something goes wrong, it can do a rollback.

Another thing is around declaring your resource demands. When you create a service, you containerize it. It is a good practice to tell the platform how much CPU and memory that service will require. Kubernetes uses that knowledge to find the best node for your workloads. Before Kubernetes, we had to manually place an instance to a node based on our criteria. Now we can guide Kubernetes with our preferences, and it will make the best decision for us.

Nowadays, on Kubernetes, you can do polyglot configuration management. You don't need in your application runtime anything to do configuration lookup. Kubernetes will make sure that the configurations end up on the same node where your workload is. The configurations are mapped as a volume or environment variable ready for your application to use.

It turns out those specific capabilities I just spoke about are also related. For example, if you want to do an automatic placement, you have to tell Kubernetes the resource requirements of your service. Then you have to tell it what deployment strategy to use. For the strategy to work correctly, your application has to implement the events coming from the environment. It has to implement health checks. Once you put all of these best practices in place and use all of these capabilities, your application becomes an excellent cloud-native citizen, and it's ready for automation on Kubernetes (this represents the foundational patterns for running workloads on Kubernetes).
And lastly, there are other patterns around structuring the containers in a pod, configuration management, and behavior.The next topic I want to cover briefly is around workloads. From the lifecycle point of view, we want to be able to run different workloads. We can do that on Kubernetes, too. Running Twelve-Factor Apps and stateless microservices is pretty straightforward. Kubernetes can do that. That's not the only workload you will have. Probably you will also have stateful workloads, and you can do that on Kubernetes using a stateful set.

Another workload you may have is a singleton. Maybe you want an instance of an app to be the only one instance of your app throughout the whole cluster - you want it to be a reliable singleton. When it fails, it should be started again. Hence, you can choose between stateful sets and replica sets depending on your needs and whether you want the singleton to have at least one or at most one semantic. Another workload you may have is around jobs and cron jobs - with Kubernetes, you can do those as well.



Lifecycle capabilities

- Deployment/rollback ✔
- Placement/scheduling ✔
- Configuration management ✔
- Resource/failure isolation ✔
- Auto/manual scaling ✔
- Hybrid workloads: stateless, stateful, batch jobs, ~~serverless~~ ✔

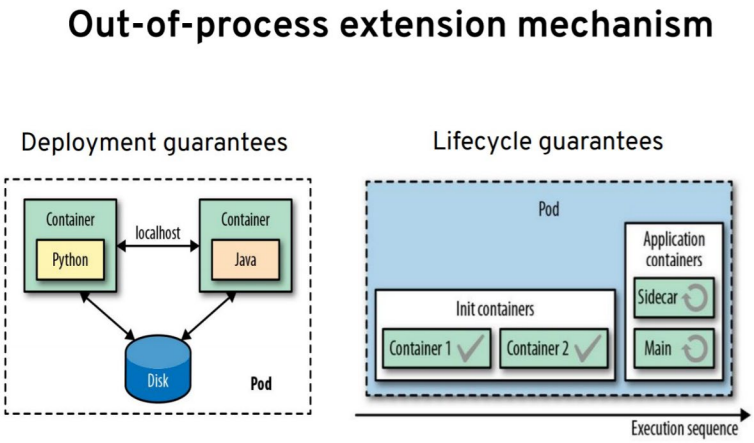
If we map all of these Kubernetes features to our needs, Kubernetes satisfies the life cycle needs. The list of requirements I usually create is primarily driven by what Kubernetes provides us today. These are expected capabilities from any platform, and what Kubernetes can do for your deployment is configuration management, resource isolation, and failure isolation. Furthermore, it supports different workloads except serverless on its own.

Then, if that's all Kubernetes gives for developers, how do we extend Kubernetes? And how can we make it give us more features? Therefore, I want to describe the two common ways that are used today.

Out-of-process Extension Mechanism

The first thing is the concept of a pod, an abstraction used to deploy containers on nodes. Moreover, a pod gives us two sets of guarantees:

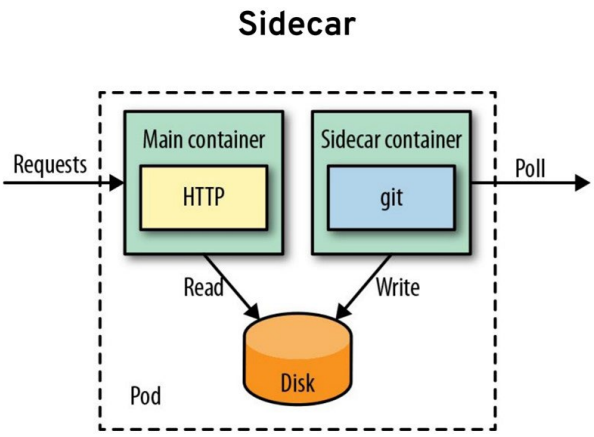
- The first set is a deployment guarantee - all containers in a pod always end up on the same node. That means they can communicate with each other over localhost or asynchronously using the file system or through some other IPC mechanism.
- The other set of guarantees a pod gives us is around lifecycle. Not all containers within a pod are equal.



Depending on if you're using init containers or application containers, you get different guarantees. For example, init containers are run at the beginning; when a pod starts, it runs sequentially, one after another. They run only if the previous container has been completed successfully. They help implement workflow-like logic driven by containers.

Application containers, on the other hand, run in parallel. They run throughout the lifecycle of the pod, and this is the foundation for the sidecar pattern. A sidecar can run multiple containers that cooperate and jointly provide value to the user. That's one of the primary mechanisms we see nowadays for extending Kubernetes with additional capabilities.

To explain the following capability, I have to tell you how Kubernetes works internally briefly. It is based on the reconciliation loop. The idea of the reconciliation loop is to drive the desired state to the actual state. Within Kubernetes, many bits rely on that. For example, when you say I want two pod instances, this is the desired



state of your system. There is a control loop that continually runs and checks if there are two instances of your pod. If two instances are not there, it will calculate the difference if there is one or more than two. It will make sure that there are two instances.

There are many examples of this. Some are replica sets or stateful sets. The resource definition maps to what the controller is, and there is a controller for each resource definition. This controller makes sure that the real world matches the desired one, and you can even write your own custom controller.

When running an application in a pod and you cannot load any configuration file changes at runtimes. However, you can write a custom controller that detects config map changes, restart your pod and application - and thus pick up the configuration changes.

It turns out that even though Kubernetes has a good collection of resources, that they are not enough for all the different needs you may have. Kubernetes introduced the concept of custom resource definitions. That means you can go and model your requirements and define an API that lives within Kubernetes. It lives next to other Kubernetes native resources. You can write your own controller in any language that understands your model. You can design a ConfigWatcher implemented in Java that describes what we explained earlier. That is what the operator pattern is, a controller that works with the custom resource definitions. Today, we see lots of operators coming up, and that's the second way for extending Kubernetes with additional capabilities.

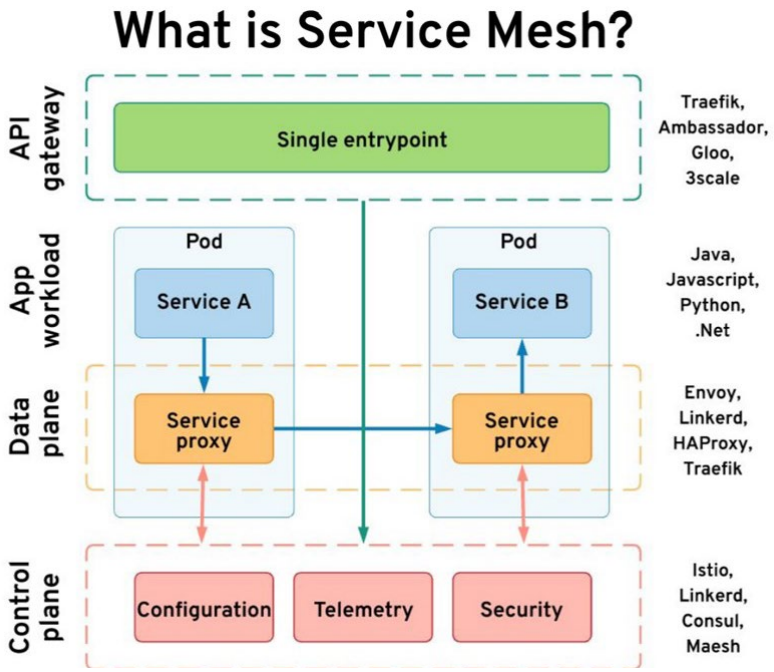
Next, I want to briefly go over a few platforms built on top of Kubernetes, which heavily use sidecars and operators to give developers additional capabilities.

What is a Service Mesh?

Let's start with the service mesh, and what is a service mesh?

We have two services, service A that wants to call service B, and it can be in any language. Consider that this is our application workload. A service mesh uses sidecar controllers and injects a proxy next to our service. You will end up with two containers in the pod. The proxy is a transparent one, and your application is completely unaware that there is a proxy - that is intercepting all incoming and outgoing traffic. Furthermore, the proxy also acts as a data firewall.

The collection of these service proxies represents your data plane and are small and stateless. To get all the state and configuration, they rely on the control plane. The control plane is the stateful part that keeps all the configurations, gathers metrics, takes decisions, and interacts with the data plane. Moreover, they are the right choice for different control planes and data planes. And as it turns out, we need one more component - an API gateway



to get data into our cluster. Some service meshes have their own API gateway, and some use a third party. All of these components, if you look into those, provide the capabilities we need.

An API gateway is primarily focused on abstracting the implementation of our services. It hides the details and provides borderline capabilities. Service mesh does the opposite. In a way, it enhances the visibility and reliability within the services. Jointly, we can say that API gateway and service mesh provide all the networking needs. To get networking capabilities on top of Kubernetes, using just the services is not enough: "You need some service mesh."

What is Knative?

The next topic I like to discuss is Knative - a project started by Google a few years ago. It is a layer on top of Kubernetes that gives you serverless capabilities and has two main modules:

- Knative Serving - focused around request-reply interactions, and
- Knative Eventing - more for event-driven interactions.

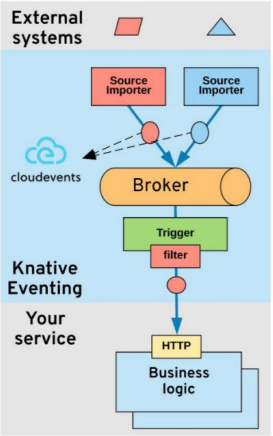
Just to give you a feel, what Knative Serving is? With Knative Serving, you define a service, but that is different from a Kubernetes service. This is a Knative service. Once you define a workload with a Knative service, you get a deployment but with serverless characteristics. You don't need to have an instance up and running. It can be started from zero when a request arrives. You get serverless capabilities; it can scale up rapidly and scale down to zero.

Knative Eventing gives us a fully declarative event management system. Let's assume we have some external systems we want to integrate with, some external event producers. At the bottom, we have our application in a container that has an HTTP endpoint. With Knative Eventing, we can start a broker, which can trigger a broker that Kafka maps, or it can be in memory or some cloud service. Furthermore, we can start importers that connect to the external system and import events into our broker. Those importers can be, for example, based on Apache Camel, which has hundreds of connectors.

Once we have our events going to the broker, then declaratively with the YAML file, we can subscribe our container to those events. In our container, we don't need any messaging clients - for example, a Kafka client. Our container would get events through HTTP POST using cloud events. This is a fully platform-managed messaging infrastructure. As a developer, you have to write your business code in a container and don't deal with any messaging logic.

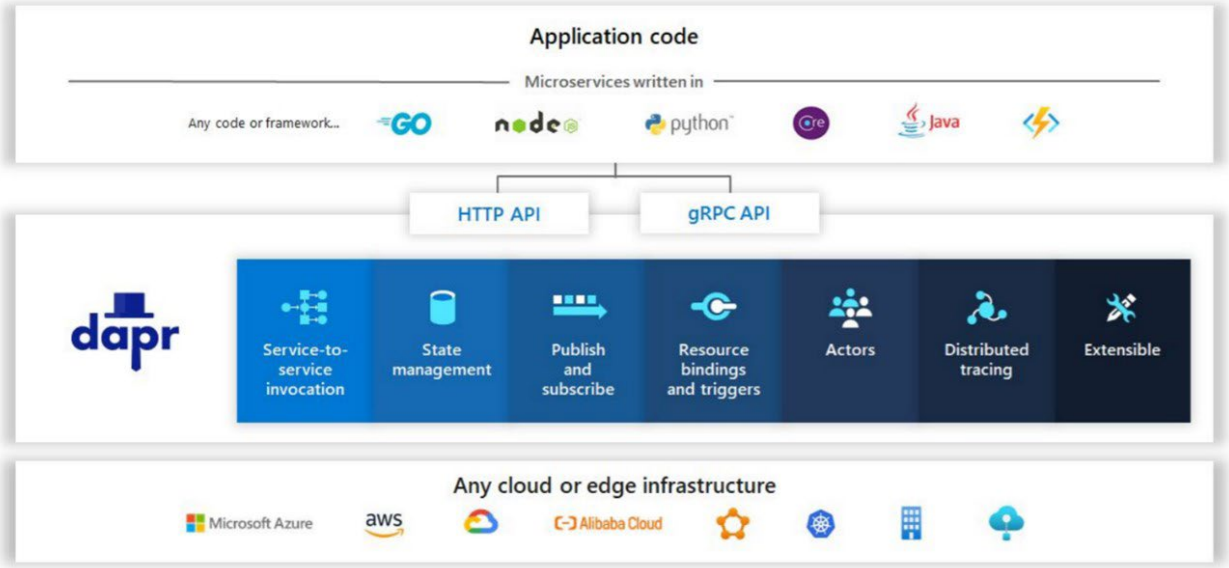
From our needs' point of view, Knative satisfies a few of those. From a lifecycle point of view, it gives our workloads serverless capabilities, so the ability to scale to zero, and activate from zero and go up. From a networking point of view, if there is some overlap with the service mesh, Knative can also do traffic shifting. From a binding point of view, it has pretty good support for binding using Knative importers. It can give us Pub/Sub, or point-to-point interaction, or even some sequencing. It

Knative Eventing concepts



- Sources (Kafka, CronJob, Apache Camel 200+, etc)
- Broker implementations (In-memory, Kafka, etc)
- CloudEvents data format
- Trigger with filters
- Sequence: chaining multiple steps composed of containers

Dapr architecture



Source: https://github.com/dapr/docs

satisfies the needs in a few categories.

What is Dapr?

Another project using sidecars and operators is Dapr, which was started by Microsoft only a few months ago - and is rapidly getting popular. Moreover, version 1.0 is considered to be production-ready. It is a distributed systems toolkit as a sidecar - everything in Dapr is provided as a sidecar and has a set of what they call building blocks or a set of capabilities.

What are those capabilities? The first set of capabilities is around networking. Dapr can do service discovery and point-to-point integration between services. Similarly, it can also do the tracing, reliable communications, retries, and recovery to service mesh. The second set of capabilities is around resource binding:

- It has lots of connectors to cloud APIs, different systems, and
- also can do messaging publish/subscribe and other logic.

Interestingly, Dapr also introduces the notion of state management. In addition to what Knative and service mesh gives you, Dapr also has abstraction on top of the state store. Furthermore, you can have key-value-based interaction with Dapr backed by a storage mechanism.

At a high level, the architecture is you have your application at the top, which can be in any language. You can use the client libraries provided by Dapr, but you don't have to. You can use the language features to do HTTP and gRPC called the sidecar. The difference to service mesh is that here the Dapr sidecar is not a transparent proxy. It is an explicit proxy that you have to call from your application and interact with over HTTP or gRPC. Depending on what capabilities you need, Dapr can talk to other systems, such as cloud services.

On Kubernetes, Dapr is deployed as a sidecar and can work outside of Kubernetes (it's not only Kubernetes). Furthermore, it also has an operator - and sidecars and operators are the primary extension mechanism. A few other components manage certificates, deal with actor-based modeling, and inject the sidecars. Your workload interacts with the sidecar and does all the magic to talk to other services, giving you some interoperability

with different cloud providers. It also gives you additional distributed system capabilities.

If I were to sum up, what these projects are giving you, we could say that ESB is the early incarnation of distributed systems where we had the centralized control plane and data plane - yet it didn't scale well. There is still a centralized control plane with cloud-native, but the data plane is decentralized - and is highly scalable with sound isolation.

We always would need Kubernetes to do good lifecycle management, and on top of that, you would probably need one or more add-ons. You may need Istio to do advanced networking. You may use Knative to do serverless workloads or Dapr to do the integration. Those frameworks play nicely with Istio and Envoy. From a Dapr and Knative point of view, you probably have to pick one. Jointly, they are providing what we used to have on an ESB in a cloud-native way.

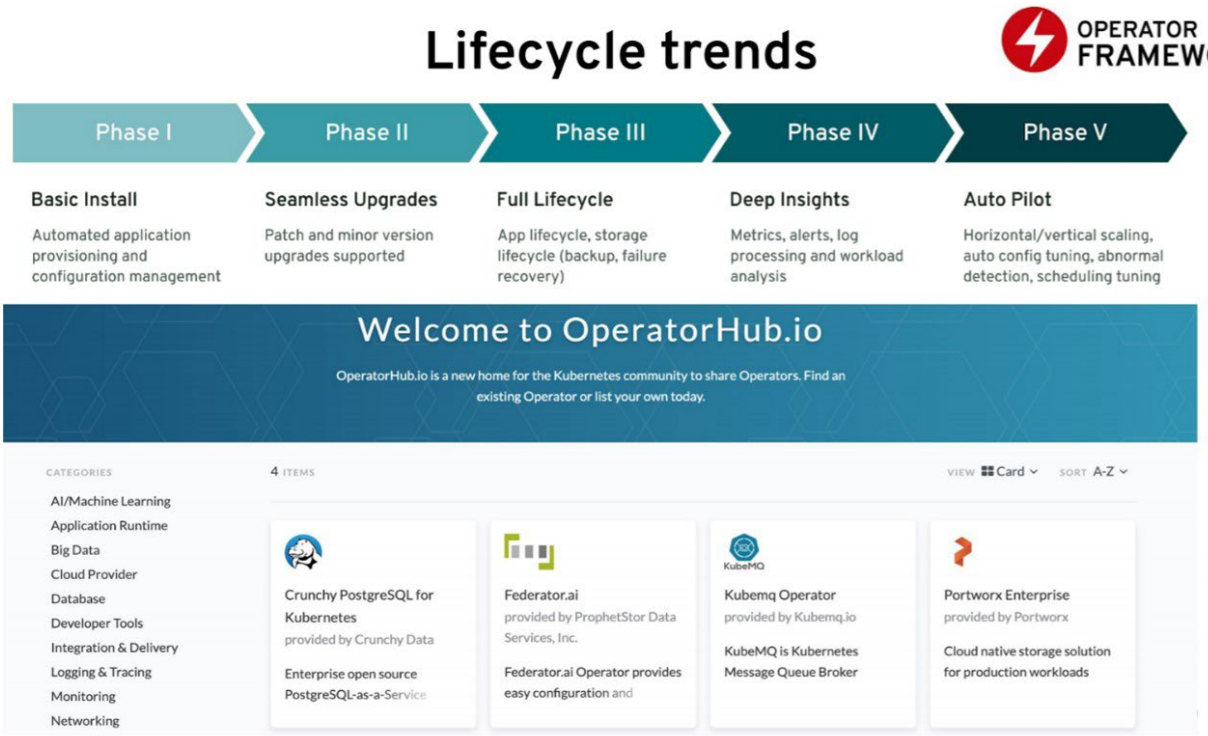
Future Cloud-Native Trends - Lifecycle Trends

For the next part, I have made an opinionated list of a few projects where I think exciting developments are happening in these areas.

I want to start with the lifecycle. With Kubernetes, we can do a useful lifecycle of your application, which might not be enough for more complex lifecycle management. For example, you may have scenarios where the deployment primitive in Kubernetes is not enough for your application if you have a more complex stateful application.

In these scenarios, you can use the operator pattern. You can use an operator that does deployment and upgrade where it also backs up maybe the storage of your service to S3. Furthermore, you may also find out that the actual health checking mechanism in Kubernetes is not good enough. Suppose the liveness check and readiness check are not good enough. In that case, you can use an operator to do a more intelligent liveness and readiness check of your application, and based on that, perform recovery.

A third area would be auto-scaling and tuning. You can



Source: https://operatorhub.io

have an operator understanding your application better and do auto-tune on the platform. Today, there are primarily two frameworks for writing operators, the Kubebuilder from Kubernetes special interest group and the Operator SDK, which is part of the operator framework created by Red Hat. It has a few things:

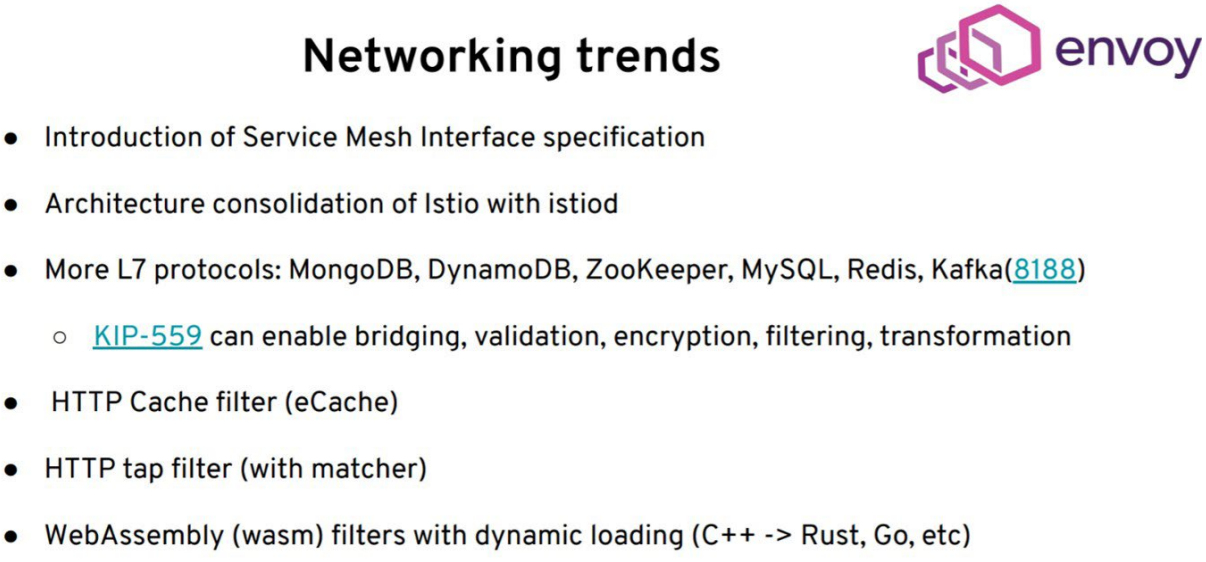
The Operator SDK lets you write operators - an Operator Lifecycle Manager, about managing the operator's lifecycle and OperatorHub, where you can publish your operator. You will see over 100 operators manage databases, message queues, and monitoring tools if you go there today. From lifecycle space, probably operators are the area where most active development is happening in the Kubernetes ecosystem.

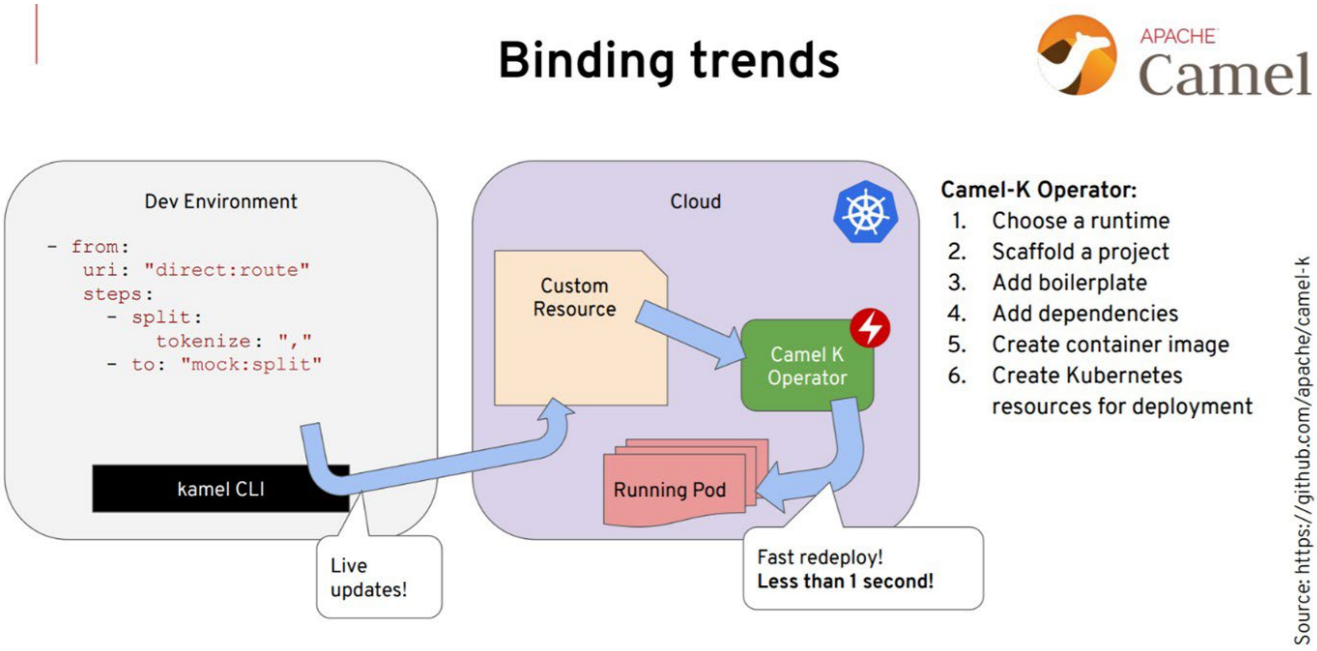
Networking Trends - Envoy

Another project I picked is Envoy. The introduction of service mesh interfaces specification will make it easier for you to switch different service mesh implementations. There has been some consolidation on Istio architecture in the deployment. You don't have to deploy seven pods for the control plane; now, you can just deploy once. More interestingly is what's happening at the data plane in the Envoy project. We see that more and more Layer 7 protocols are added to Envoy.

Service mesh adds support for more protocols such as MongoDB, ZooKeeper, MySQL, Redis, and the most recent one is Kafka. I see that the Kafka community is now further improving their protocol to make it friendlier for service meshes. We can expect that there will be even more tight integration, more capabilities. Most likely, there will be some bridging capability. You can do an HTTP call locally from your application in your service, and the proxy will, behind the scene, use Kafka. You can do transformation and encryption outside of your application in a sidecar for the Kafka protocol.

Another exciting development has been the introduction of HTTP caching. Now Envoy can do HTTP caching. You don't have to use caching clients within your applications. All of that is done transparently in a sidecar. There are tap filters, so you can tap the traffic and get a copy of the traffic. Most recently, the introduction of WebAssembly, means if you want to write some custom filter for Envoy, you don't have to write it in C++ and compile the whole Envoy runtime. You can write your filter in WebAssembly, and deploy that at runtime. Most of these are still in progress. They are not there, indicating that the data plane and service mesh have no intention of stopping, just supporting HTTP and gRPC. They are interested in supporting more application-layer protocols to offer you more, to enable more use cases.





Source: https://github.com/apache/camel-k

Mostly, with the introduction of WebAssembly, you can now write your custom logic in the sidecar. That's fine as long as you're not putting there some business logic.

Binding Trends - Apache Camel

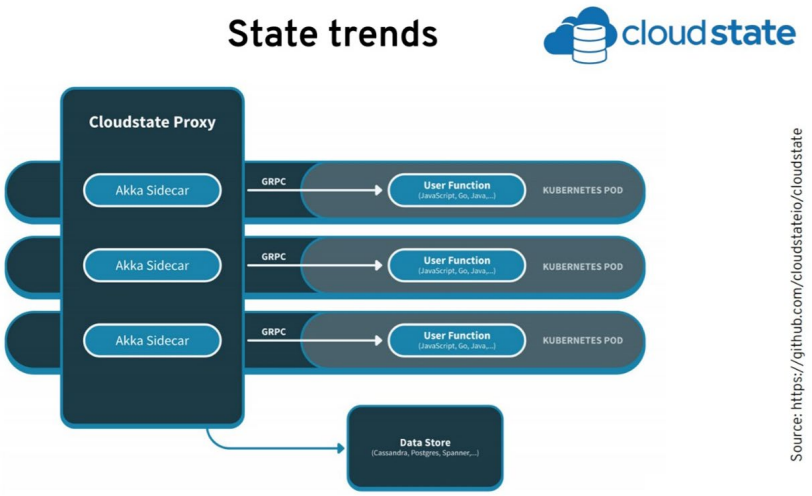
Apache Camel is a project for doing integrations, and it has lots of connectors to different systems using enterprise integration patterns. Camel version 3, for instance, is deeply integrated into Kubernetes and uses the same primitives we spoke about so far, such as operators.

You can write your integration logic in Camel in languages such as Java, JavaScript, or YAML. The latest version has introduced a Camel operator that runs in Kubernetes and understands your integration. When you write your Camel application, deploy it to a custom resource, the operator then knows how to build the container or find dependencies.

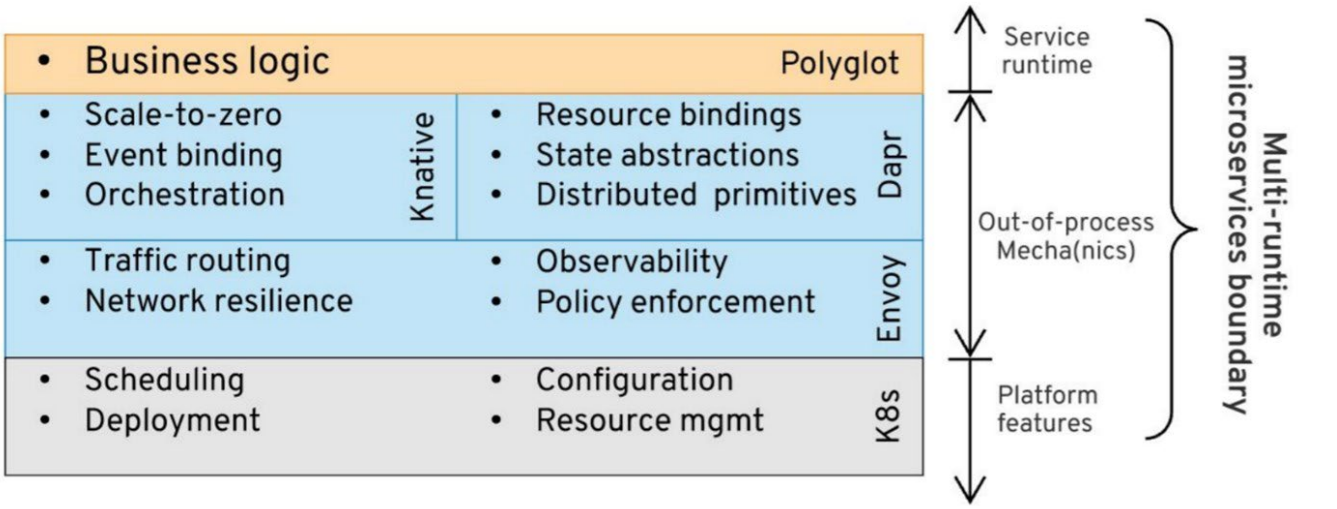
Depending on the platform's capabilities, whether that's Kubernetes only, whether that's Kubernetes with Knative, it can decide what services to use and how to materialize your integration. There is quite a lot of intelligence going outside of your runtime - but into the operator - and all of that happens pretty fast. Why would I say it's a binding trend? Mainly because of the capabilities of Apache Camel with all the connectors it provides. The interesting point here is how it integrates deeply with Kubernetes.

State Trends - Cloudstate

Another project I like to discuss is Cloudstate and around state-related trends. And Cloudstate is a project by Lightbend and primarily focused on serverless and function-driven development. With their latest



Source: https://github.com/cloudstateio/cloudstate



releases, they are integrating deeply with Kubernetes using sidecars and operators.

The idea is, when you write your function, all you have to do in your function is use gRPC to get state, to interact with state. The whole state management happens in a sidecar that is clustered with other sidecars. It enables you to do event sourcing, CQRS, key-value lookups, messaging.

From your application point of view, you are not aware of all these complexities. All you do is a call to a local sidecar, and the sidecar handles the complexity. It can use, behind the scenes, two different data sources. And it has all the stateful abstractions you would need as a developer.

So far, we have seen the current state of the art in the cloud-native ecosystem and some of the recent developments that are still in progress. How do we make sense of all that?

Multi-runtime Microservices Are Here

If you look at how microservice looks on Kubernetes, you will need to use some platform functionality. Moreover, you will need to use Kubernetes features for lifecycle management primarily. And then, most likely, transparently, your service will use some service mesh, something like an Envoy, to get enhanced networking

capabilities, whether that's traffic routing, resilience, enhanced security, or even if it is for a monitoring purpose. On top of that, depending on your use case, you may need Dapr or Knative, depending on your workloads. All of these represent your out-of-process, additional capabilities. What's left to you is to write your business logic, not on top, but as a separate runtime. Most likely, future microservices will be this multi-runtime composed of multiple containers. Some of those are transparent, and some of those are very explicit that you use.

Smart Sidecars and Dumb Pipes

If I look a little bit deeper, how that might look like, you write your business logic in some high-level language. It doesn't matter what it is; it doesn't have to be Java only as you can use any other language and develop your custom logic in-house.

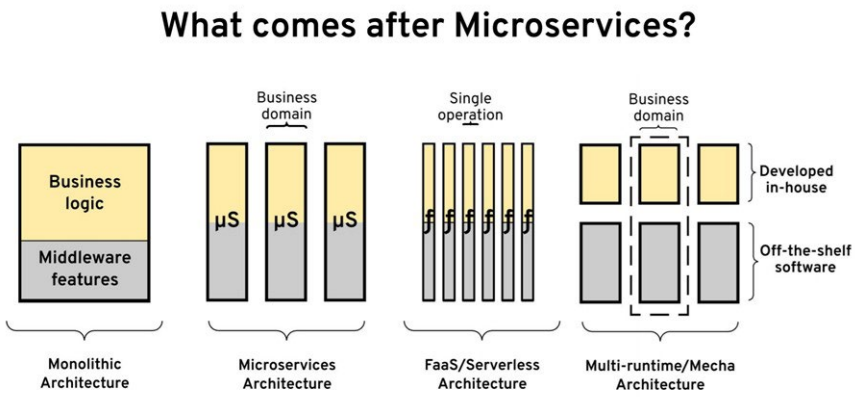
All the interactions of your business logic with the external world happen through the sidecar, integrating with the platform and does the lifecycle management. It does the networking abstractions for the external system and gives you advanced binding capabilities and state abstraction.

The sidecar is something you don't develop. You get it off the shelf. You configure it with a little bit of YAML or JSON, and you use it. That means you can update sidecars easily because it's not embedded anymore into your runtime. It makes patching, updating easier. It enables polyglot runtime for our business logic.

What Comes After Microservices?

That brings me to the original question, what comes after microservices?

I would argue that maybe FaaS is not the best model - as functions are not the best model for implementing reasonably complex services where you want multiple operations to reside together when they have to interact with the same dataset. Probably, multi-runtime, I call it Mecha architecture, where you have your business logic in one container, and you have all the infrastructure-related concerns as a separate container. They jointly represent a multi-runtime microservice. Maybe that's a more suitable model because it has better properties.You get all the benefits of microservice. You still have all your domain, all the bounded contexts in one place. You have all the infrastructure and distributed application needs in a separate container, and you combine them at runtime. Probably, the closest thing that's getting to that right now is Dapr. They are following that model. If you're only interested from a networking aspect, probably using Envoy is also getting close to this model.



If we see how the architectures have been evolving, application architectures at a very high level started with monolithic applications. Yet Microservices gives us the guiding principles on how to split a monolithic application into separate business domains. After that came serverless and Function-as-a-Service (FaaS), where we said we could split those further by operations, giving us extreme scaling - because we can scale each operation individually.

TL;DR

- Modern distributed applications have needs around lifecycle, networking, binding, and state management that cloud-native platforms must provide.
- Kubernetes has great support around lifecycle management but relies on other platforms using the sidecar and operator concepts to satisfy the networking, binding, and state management primitives.
- Future distributed systems on Kubernetes will be composed of multiple runtimes where the business logic forms the core of the application, and sidecar "mecha" components offer powerful out-of-the-box distributed primitives.
- This decoupled mecha architecture offers the benefits of cohesive units of business logic and improves day-2 operations, such as patching, upgrades, and long-term maintainability.

Cloud Native is About Culture, not Containers

by **Holly Cummins**, Innovation leader in IBM Corporate Strategy

At the QCon London last year, I provided a Cloud-Native session about Culture, not Containers. What had started me thinking about the role of culture in cloud native was a great InfoQ article from Bilgin Ibryam. One of the things Bilgin did was define a cloud native architecture as lots of microservices, connected by smart pipes. I looked at that and thought it looked totally unlike applications I wrote, even though I thought I was writing cloud native applications. I'm part of the IBM Garage, helping clients get cloud native,, and yet I rarely used microservices in my apps. The apps I create mostly looked nothing like Bilgin's diagram. Does that mean I'm doing it wrong, or is maybe the definition of cloud native a bit complicated?

I don't want to single Bilgin out, since Bilgin's article was called "Microservices in the post-Kubernetes Era," so it would be a bit ridiculous if he weren't talking about microservices a lot in that article. It's also the case that almost all definitions of cloud native equate it to microservices. Everywhere I looked, I kept seeing the assumption that microservices equals native and Cloud-native equals microservices. Even the Cloud Native Computing Foundation used to define cloud native as all about microservices, and all about containers, with a bit of dynamic orchestration in there. Saying cloud native doesn't always involve microservices, which puts me in this peculiar position because not only am I saying Bilgin is wrong, I'm saying the Cloud Native Computing Foundation is wrong - what did they ever know about Cloud-native? I'm sure I know way more than them, right?

Well, obviously I don't. I'm on the wrong side of history on this one. I will admit that. (Although I'm on

the wrong side of history, I notice that the CNCF have updated their definition of cloud native and, while microservices and containers are still there, they don't seem quite as mandatory as they used to be, so a little bit of history might be on my side!). Right or wrong, I'm still going to die on my little hill; that Cloud-native is about something much bigger than microservices. Microservices are one way of doing it. They're not the only way of doing it.

In fact, you do see a range of definitions within our community. If you ask a bunch of people what Cloud-native means, some people will say "born on the cloud". This was very much the original definition of Cloud-native back before microservices were even a thing. Some people will say it's microservices.

Some people will say, "oh no, it's not just microservices, it's microservices on Kubernetes, and that's how you get Cloud-native". This one I don't like, because to me, Cloud-native shouldn't be about a technology choice. Sometimes I see Cloud-native used as a synonym for DevOps, because a lot of the cloud native principles and practices are similar to what devops teaches.

Sometimes I see Cloud-native used just as a way of saying "we're developing modern software": "We're going to use best practices; it's going to be observable; it's going to be robust; we're going to release often and automate everything; in short, we're going to take everything we've learned over the last 20 years and develop software that way, and that's what makes it Cloud-native".

In this definition, cloud is just a given - of course it's on cloud, because we're developing this in 2021.

Sometimes I see Cloud-native used just to mean Cloud. We got so used to hearing Cloud-native that every time we talk about Cloud, we just feel like we have to tack a '-native' on afterwards, but we're really just talking about Cloud. Finally, when people say Cloud-native, sometimes what they mean is idempotent. The problem with this is if you say Cloud-native means idempotent, everybody else goes, "What? What we really mean by "idempotent" is rerunnable? If I take it, shut it down, and then start it up again, there's no harm done. That's a fundamental requirement for services on the Cloud.

With all of these different definitions, is it any wonder we're not entirely sure what we're trying to do when we do Cloud-native?

Why?

"What are we actually trying to achieve?" is an incredibly important question. When we're thinking about technology choices and technology styles, we want to be stepping back just from "I'm doing Cloud-native because that's what everybody else is doing" to thinking "what problem am I actually trying to solve?" To be fair to the CNCF, they had this "why" right on the front of their definition of Cloud-native. They said, "Cloud-native is about using microservices to build great products faster." We're not just using microservices because we want to; we're using microservices because they help us build great products faster.

This is where we step back to make sure we understand the problem we're solving. Why couldn't we build great products faster before? It's easy to skip this step, and I think all of us are guilty of this sometimes. Sometimes the problem that we're actually trying to solve is that everybody else is doing it, so we fear missing out unless we start doing it. Once we put it like that, FOMO isn't a great decision criteria. Even worse, "my CV looks dull" definitely isn't the right reason to choose a technology.

Why Cloud?

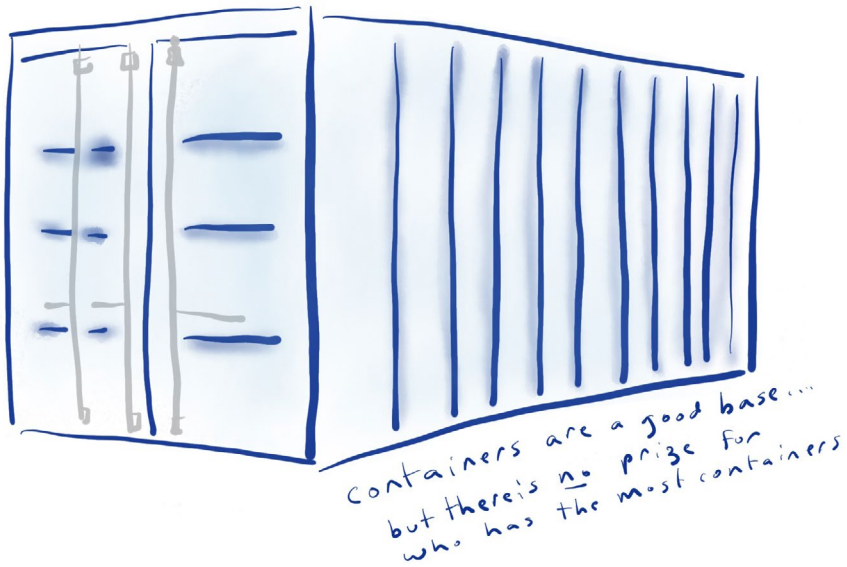
I think to get to why we should be doing things in a Cloud-native way; we want to step back and say, "Why were we even doing things on the Cloud?" Here are the reasons:

- **Cost**: Back when we first started putting things on the Cloud, price was the primary motivator. We said, "I've got this data center, I have to pay for the electricity, I have to pay people to maintain it. And I have to buy all the hardware. Why would I do that when I could use someone else's data center?" What creates a cost-saving between your own data center and someone else's data center is that your own data center has to stock up enough hardware for the maximum demand. That's potentially a lot of capacity which is unused most of the time. If it's someone else's data center, you can pool resources. When demand is low, you won't pay for the extra capacity.
- **Elasticity**: The reason Cloud saves you money is because of that elasticity. You can scale up; you can scale down. Of course, that's old news now. We all take elasticity for granted.
- **Speed**: The reason we're interested in Cloud now is because of the speed. Not necessarily the speed of the hardware, although some cloud hardware can be dazzlingly fast. The cloud is an excellent way to use GPUs, and it's more or less the only way to use quantum computers. More generally, though, we can get something to market way, way faster via the cloud than we could when we had to print software onto CD-Roms and mail them out to people, or even when we had to stand instances up in our own data center.

12 Factors

Cost savings, elasticity, and delivery speed are great, but we get all of that just by being on the Cloud. Why do we need Cloud-native? The reason we need Cloud-native is that a lot of companies found they tried to go to the Cloud and they got electrocuted.

It turns out things need to be written differently and managed differently on the cloud.



Articulating these differences led to the 12 factors. The 12 factors were a set of mandates for how you should write your Cloud application so that you didn't get electrocuted.

You could say the 12 factors described how to write a cloud native application - but the 12 factors had absolutely nothing to do with microservices. They were all about how you managed your state. They were about how you managed your logs. The 12 factors helped applications become idempotent, but "the 12 factors" is catchier than "the idempotency factors".

The 12 factors were published two years before Docker got to market. Docker containers revolutionised how the cloud was used. Containers are so good, it's hard to overstate their importance. They solve many problems and create new architectural possibilities.

Because containers are so it's easy, it's possible to distribute an application across many containers. Some companies are running single applications across 100, 200, 300, 400, or 500 distinct containers. Compared to that kind of engineering prowess, an application which is spread across a mere six containers seems a bit inadequate. In the face of so little complexity, it's easy to think "I must be doing it really wrong. I'm not as good a developer as them over there".

Actually, no. It's not a competition to see how many containers you can have. Containers are great, but the number of containers you have should be tuned to your needs.

Speed

Let's try and remember - what were your needs again? When we think about Cloud, we usually want to be thinking about that speed. The reason we want lots of containers is that we want to get new things to market faster. If we have lots of containers and we're either shipping the exact same things to market or we're getting to market at the same speed, then all of a sudden, those containers are only a cost. They're not helping us, and we're burning cycles managing the complexity that comes with scattering an application in tiny pieces all over the infrastructure. If we have this amazing architecture that allows us to respond to the market but we're not responding, then that's a waste. If we have this architecture, that means we can go fast, but we're not going fast, then that's a waste as well.

How to fail at Cloud-Native

Which brings me to how to fail at Cloud-native. For context, I'm a consultant. I'm a full-stack developer in the IBM Garage. We work with startups and with large companies, helping them get to the cloud and get the most out of cloud. As part of that, we help them solve interesting, tough, problems, and we help them do software faster than they've been able to to do it before. To make sure we're really getting the most out of the cloud, we do a lean startup, extreme programming, design thinking, DevOps; and cloud native. Because I'm a consultant, I see a lot of customers who are on the journey to the Cloud. Sometimes that goes well, and sometimes there are these pitfalls. Here are

some of the traps that I've seen smart clients fall into. So, What is Cloud-Native?

One of the earliest traps is the magic morphing meaning. If I say Cloud-native and I mean one thing and you say Cloud-native and mean another thing, we're going to have a problem communicating...

Sometimes that doesn't really matter, but sometimes it makes a big difference. If one person thinks the goal is microservices and then the other person feels the goal is to have an idempotent system, uh oh. Or if part of an organisation wants to go to the Cloud because they think it's going to allow them to get to market faster, but another part is only going to the Cloud to deliver the exact same speed as before, but more cost-effectively, then we might have some conflict down the road.

Microservices Envy

Often one of the things that drives some of this confusion about goals is because we have a natural tendency to look at other people, doing fantastic things, and want to emulate them. We want to do those fantastic things ourselves without really thinking about our context and whether they're appropriate for us. One of our IBM Fellows has a heuristic when he goes in to talk to a client about microservices. He says, "If they start talking about Netflix and they just keep talking about Netflix, and they never mention coherence, and they never mention coupling, then probably they're not really doing it for the right reasons."

Sometimes we talk to clients, and they say, "Right, I want to modernize to microservices." Well, microservices are not a goal. No customer will look at your website and say, "Oh, microservices. That's nice." Customers are going to look at your website and judge it on whether it serves their needs, whether it's easy and delightful, and, all of these other things. Microservices can be an excellent means to that end, but they're not a goal in themselves. I should also say: microservices are a means. They're not necessarily the only means to that goal.

A colleague of mine in the IBM Garage had some conversations with a bank in Asia-Pacific. The bank was having problems responding to their customers, because their software was all old and heavy and calcified. They were also having a people problem because all of their COBOL developers were old and leaving the workforce. So the bank knew they had to modernise. The main driver in this case wasn't the aging workforce, it was really competitiveness and agility. They were getting beaten by their competitors because they had this big estate of COBOL code and every change was expensive and slow. They said, "Well, to solve this problem, we need to get rid of all of our COBOL, and we need to switch to a modern microservices architecture."

So far, so good. We were just gearing up to jump in with some cloud native goodness, when the bank added that their release board only met twice a year. At this point, we wound back. It didn't matter how many microservices the bank's shiny new architecture would have; those microservices were all going to be assembled up into a big monolith release package and deployed twice a year. That's taking the overhead of microservices without the benefit. Since it's not a competition to see how many containers you have, lots of containers and slow releases would be a stack in which absolutely no one won.

Not only would lots of microservices locked into a sluggish release cadence not be a win, it could be a bad loss. When organisations attempt microservices, they don't always end up with a beautiful decoupled microservices architecture like the ones in the pictures. Instead, they end up with a distributed monolith. This is like a normal monolith, but far worse. The reason that this is extra-scary bad is because a normal, non-distributed, monolith has things like compile-

time checking for types and synchronous, guaranteed, internal communication. Running in a single process is going to hurt your scalability, but it means that you can't get bitten by the distributed computing fallacies. If you take that same application and then just smear it across the internet and don't put in any type checking or invest in error handling for network issues, you're not going to have a better customer experience; you're going to have a worse customer experience.

There's a lot of contexts in which microservices are the wrong answer. If you're a small team, you don't need to have lots of autonomous teams because each independent team would be about a quarter of a person. Suppose you don't have any plans or any desire to release part of your application independently, then you won't benefit from microservices's independence.

In order to give security and reliable communication and discoverability between all of these components of your application that you've just smeared across a part of the Cloud, you're going to need something like a service mesh. You might be either quite advanced on the tech curve or a little bit new to that tech curve. You either don't know what a service mesh is, or you say, "I know all about what a service mesh is. So complicated, so overhyped. I don't need a service mesh. I'm just going to roll my own service mesh instead." This will not necessarily give you the outcome you hoped for. You will still end up with a service mesh, but you have to maintain it, because you wrote it!

Another good reason not to do microservices is sometimes the domain model just doesn't have those natural fracture points that allow you to get nice neat microservices. In that case, it is totally reasonable to say, "You know what? I'm just going to leave it."

Cloud-native spaghetti

If you don't step away from the blob, then you end up with the next problem, which is Cloud-native spaghetti. I always feel slightly panicked when I look at the communication diagram for the Netflix microservices. I'm sure they know what they're doing, and they've got it figured out, but to my eyes, it looks exactly like spaghetti. Making that work needs a lot of really solid engineering and specialised skills. If you don't have that specialisation, then you end up in a messy situation.

I was brought in to do some firefighting with a client who was struggling. They were developing a greenfield application, and so of course they'd chosen microservices, to be as modern as possible. One of the first things they said to me was "any time we change any code at all, something else breaks." This isn't what's supposed to happen with microservices. In fact, it's the exact opposite of what we've all been told happens if we implement microservices. The dream of microservices is that they are decoupled. Sadly, decoupling doesn't come for free. It certainly doesn't magically happen just because you distributed things. All that happens when you distribute things is that you have two problems instead of one.

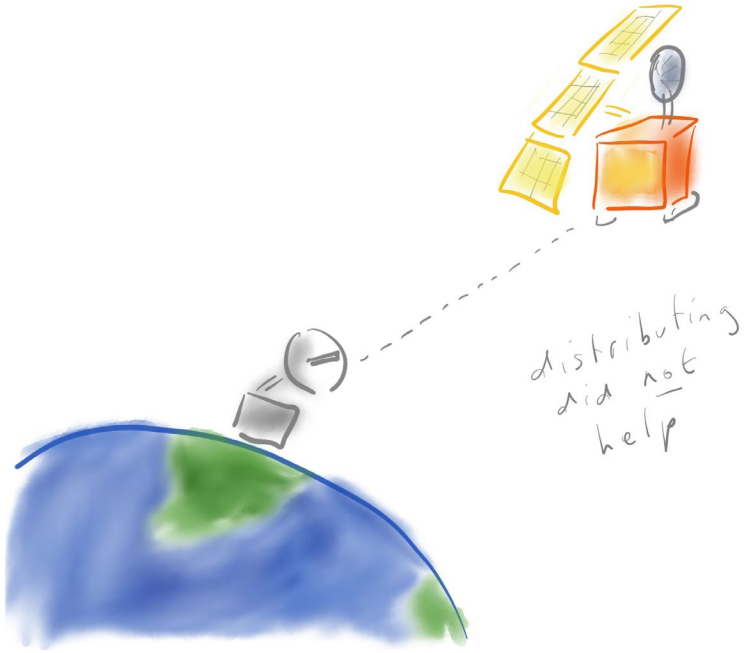
Cloud Native Spaghetti is still spaghetti.

One of the reasons my client's code was so brittle and connected was that they had quite a complex object model, with around 20 classes and 70 fields in some of the classes. Handling that kind of complex object model in a microservices system is tough. In this case, they looked at their complex object model, and decided, "We know it's really bad to

have common code between our microservices because then we're not decoupled. Instead, we're going to copy and paste this common object model across all of our six microservices. Because we cut and paste it rather than linking to it, we're decoupled." Well, no, you're not decoupled. If things break when one thing changes, whether the code is linked or copied, there's coupling.

What was the 'right' thing to do in this case? In the ideal case, each microservice maps neatly to a domain, and they're quite distinct. If you have one big domain and lots of tiny microservices, then there's going to be a problem. The solution is to either decide the domain really is big and merge the microservices, or to do deeper domain modelling to try and untangle the object model into distinct bounded contexts.

Even with the cleanest domain separation, in any system, there will always be some touch points between components - that's what makes it a system. These touch points are easy to get wrong, even if they're minimal, and especially if they're hidden. Do you remember the Mars Climate Orbiter? Unlike the Perseverance, it was designed to orbit Mars from a safe distance, rather than land on it. Sadly, it strayed too close to Mars, got pulled in by Mars's gravity, and crashed. The loss of the probe was sad, and the underlying reason was properly tragic.



The moral of the story is that distributing the system did not help. Part of the system was on Mars, and part of the system was on Earth, and you can't get more distributed than that.

The Orbiter was controlled by two modules, one the probe, and one on earth. The probe module was semiautonomous, since the Orbiter was not visible from earth most of the time. About every three days the planets would align, it would come into view, and the team on earth would fine-tune its trajectory. I imagine the instructions were along the lines of "Oh, I think you need to shift a bit left and oh you're going to miss Mars if you don't go a bit right," except in numbers.

The numbers were what led to the problem. The earth module and probe module were two different systems built by two significant point of coupling between them. Every time the ground team transmitted instructions, what they sent was interpreted in a way that no one expected.

Microservices need consumer-driven contact tests

In this case, the solution, the correct thing to do is be really clear about what the points of coupling are and what each side's expectations are. A great way of doing this is consumer contract-driven tests. Contract tests aren't yet widely used in our industry, despite being a clean solution to a big problem. I think part of the problem is that they can be a bit tricky to learn, which has slowed adoption. Cross-team negotiations about the

tests can also be complicated - although if negotiation about a test is too hard, negotiation about the actual interaction parameters will be even harder. If you're thinking of exploring contract testing, Spring Contract or Pact are good starting points. Which one is right for you depends on your context. Spring Contract is nicely integrated into the Spring ecosystem, whereas Pact is framework-agnostic and supports a huge range of languages, including Java and Javascript.

Contract tests go well beyond what OpenAPI validation does, because it checks the semantics of the APIs, rather than just the syntax. It's a much more helpful check than "well, the fields on each side have the same name, so we're good." It allows you to check, "is my behavior when I get these inputs the expected behavior? Are the assumptions I'm naming about that API over there still valid?" Those are things you need to check, because if they're not true, things are going to get really bad.

Many companies are aware of this risk and are aware that there's an instability in the system when they're doing microservices. To have confidence that these things work together, they impose a UAT phase before releasing them. Before any microservice can be released, someone needs to spend several weeks testing it works properly in the broader system. With that kind of overhead, releasing isn't going to be happening often. Then that leads us to the classic anti-pattern, which is not-actually-continuous continuous integration and continuous deployment, or I/D.

Why Continuous Integration and Deployment isn't

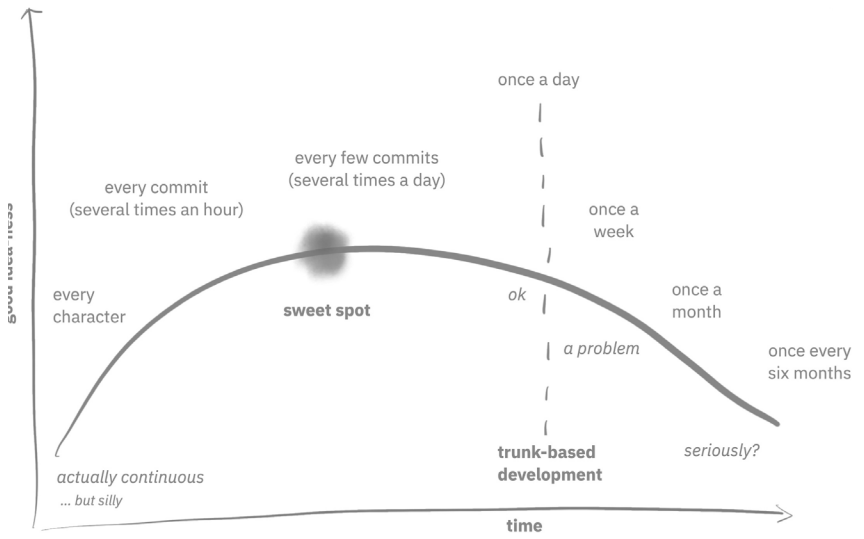
I talk to a lot of customers, and they'll say, "We have a CI/CD." The 'a' sets off alarm bells, because CI/CD, should not be a tool you buy, put on a server, and admire, saying "there's CI/CD." CD/CD is something that you have to be doing. The letters stand for continuous integration and continuous deployment or delivery. Continuous in this context means "integrating really often" and "deploying really often,", and if you're not doing that, then it's simply not continuous.

Sometimes I'll overhear comments like "I'll merge my branch into our CI system next week". This completely misses the point of the "C" in "CI", which stands for continuous. If you merge once a week, that's not continuous. That's almost the opposite of continuous.

The "D" part can be even more of a struggle. If software is only deployed every six months, the CI/CD server may be useful, but no one is doing CD. There may be "D", but everyone has forgotten the "C" part.

How often is it actually reasonable to be pushing to main? How continuous is continuous have to be? Even I will admit that some strict definitions of continuous would be a ridiculous way to write software in a team. If you pushed to main every character, that is technically continuous, but it is going to cause mayhem in a team. If you integrate every commit and aim to commit several times an hour, that's probably a pretty good cadence. If you commit often and integrate every few commits, you're pushing several times a day, so that's also pretty good. If you're doing test-driven development, then integrating when you get a passing test is an excellent pattern. I'm a big advocate of trunk-based development. TBD has many benefits in terms of debugging, enabling opportunistic refactoring, and avoiding big surprises for colleagues. The technical definition of trunk-based development is that you need to be integrating at least once a day to count. I sometimes hear "once a day" described as the bar between "ok" and "just not continuous". Once a week is getting really problematic.

Once you get into one every month, it's terrible. When I joined IBM we used a build system and a code repository called CMVC.



For context, this was about twenty years ago, and our whole industry was younger and more foolish. My first job in IBM was helping build the WebSphere Application Server. We had a big multi-site build, and the team met six days a week, including Saturdays, to discuss any build failures. That call had a lot of focus, and you did not want to be called up on the WebSphere build call. I'd just left university and knew nothing about software development in a team, so some of the senior developers took me under their wings. One piece of advice I still remember was that the way to avoid being on the WebSphere build call was to save up all of your changes on your local machine for six months and then push them all in a batch.

At the item, I was little, and I thought, ok, that doesn't seem like quite the right advice, but I guess you know best. With hindsight, I realize the WebSphere build broke badly because people were saving their changes for six months before then trying to integrate with their colleagues. Obviously, that didn't work, and we changed how we did things.

How often should you integrate?

The next question, which is even harder, is how often you should release? Like with integration, there's a spectrum of reasonable options. You could release every push. Many tech companies do this. If you're deploying it once an iteration, you're still in good company. Once a quarter is a bit sad. You could release once every two years. It seems absurdly slow now, but in the bad old days, this was the standard model in our industry.

How often should you deploy to production?

What makes deploying to production every push possible that deploying is not the same as releasing. If our new code is too incomplete or too scary to actually show to users, we can still deploy it, but keep it hidden. We can have the code actually in the production code base, but nothing is wired to it. That's pretty safe. If we're already a bit too entangled for that, we can use feature flags to flip function on and off. If we're feeling more adventurous, we can do A/B or friends and family testing so only a tiny portion of users see our scary code. Canary deploys are another variation for pre-detecting nightmares, before they hit mainstream usage.

Not releasing has two bad consequences. It lengthens

feedback cycles, which can impact decision making and makes engineers feel sad. Economically, it also means there's inventory (the working software) sat on the shelf, rather than getting out to customers. Lean principles tell us that having inventory sat around, not generating returns, is waste.

Then the conversation is, why can't we release this? What's stopping more frequent deployments? Many organisations fear their microservices, and they want to do integration testing of the whole assembly, usually manual integration testing. One customer, with about 60 microservices, wanted to ensure that there was no possibility that some bright spark of an engineer could release one microservice without releasing the other 59 microservices. To enforce this, they had one single pipeline for all of the microservices in a big batch. This obviously is not the value proposition of microservices, which is that they are independently deployable. Sadly, it was the way that they felt safest to do it.

We also see a reluctance actually to deliver because of concerns about quality and completeness. Of course, these aren't ridiculous. You don't want to anger your customers. On the other hand, as Reid Hoffman said, if you're not embarrassed by your first release, it was too late.

There is a value in continuous improvement, and there is value in getting things being used.

If releases are infrequent and monolithic, you've got these beautiful microservices architecture that allows you to go faster, and yet you're going really slow. This is bad business, and it's bad engineering.

Let's assume you go for the frequent deploys. All of the things which protect your users from half-baked features, like the automated testing, the feature flags, the A/B testing, the SRE, need substantial automation. Often when I start working with a customer, we have a question about testing, and they say, "Oh, our tests aren't automated." What that really means is that they don't actually know if the code works at any particular point. They hope it works, and it might have worked last time they checked, but we don't have any way of knowing whether it works right now without running manual tests.

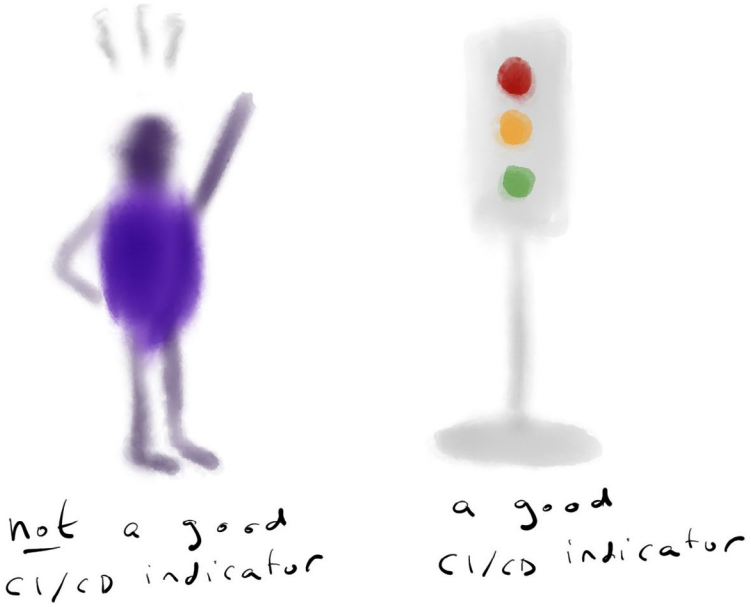
The thing is, regressions happen. Even if all the engineers are the most perfect engineers, there's an outside world which is less perfect. Systems they depend on might behave unexpectedly. If a dependency update changes behavior, something will break even if nobody did anything wrong. That brings us back to "we can't ship because we don't have confidence in the quality".

Well, let's fix the confidence in the quality, and then we can ship.

I talked about contract testing. That is cheap and easy and can be done at a unit test level, but of course, you do also need automated integration tests. You don't want to be relying on manual integration tests or they become a bottleneck.

"CI/CD" seems to have replaced "build" in our vocabularies, but in both cases, it is one of the most valuable things that you have as an engineering organization. It should be your friend, and it should be this pervasive presence everywhere. Sometimes the way the build works is that it's off on a Jenkins system somewhere. Someone who is a bit diligent goes and checks the web page every now and then and notices it's red and goes and tells their colleagues, and then eventually someone fixes the issue. What's much better is just a passive build indicator that everybody can see without opening up a separate page for. If the monitor goes red, it's really obvious, that something changed, and easy to look at the most recent change. A traffic light works if you have one project. If you've got microservices, you're probably going to need something like a set of tiles. Even if you don't have microservices, you're probably going to have several projects, so you need something a bit more

complete than a traffic light, even though the traffic lights are cute.



"We don't know when the build is broken"

If you invest in your build monitoring, then you end up with the broken window situation. I've arrived at customers, and the first thing I've done is I've looked at the build, and I said, "Oh, this build seems to be broken." They've said, "Yeah, it's been broken for a few weeks." At that point, I knew I had a lot of work to do!

Why is a perma-broken build bad? It means you can't do the automated integration testing because nothing is making it out of the build.

In fact, you can't even do manual integration testing, so inter-service compatibility could be deteriorating and no one would know.

New regressions go undetected, because the build is already red.

Perhaps worst of all, it creates a culture so that when one of the other builds goes red, people aren't that worried, because it's more of the same: "Now we've got two red. Perhaps we could get the whole set, and then it would match if we got them all red." Well, no, that's not how it should be.

The locked-down totally rigid, inflexible uncloudy Cloud

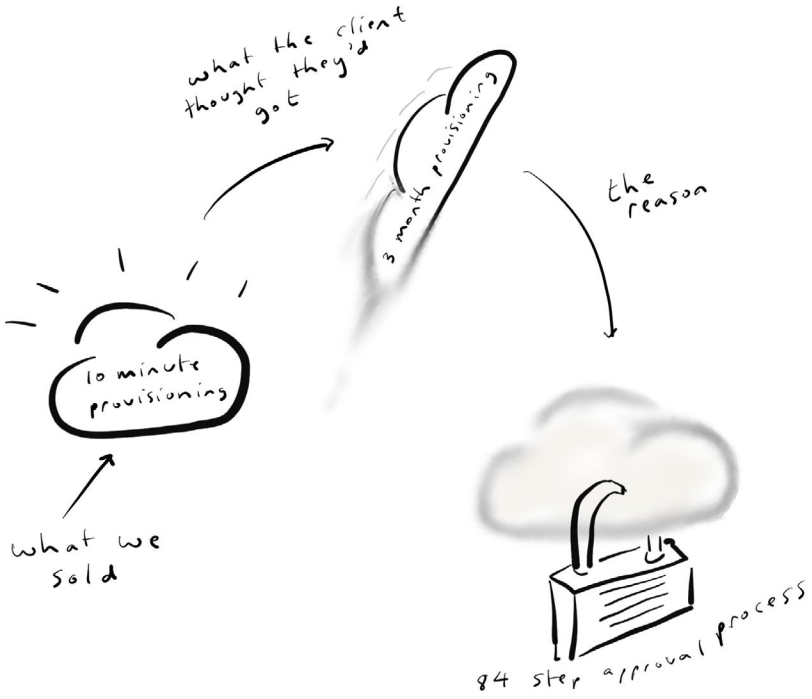
These are all challenges which happen at the team level. They're about how we as engineers manage ourselves and our code. But of course, particularly once you get to an organization of a certain size, you end up with another set of challenges, which is what the organization does with the Cloud. I have noticed that some organisations love to take the Cloud, and turn it into a locked down, totally rigid, flexible, un-cloudy Cloud.

How do you make a Cloud un-cloudy? You say, "Well, I know you could go fast, and I know all of your automation support is going fast, but we have a process. We have an architecture review board, and it meets rather infrequently." It will meet a month after the project is ready to ship, or in the worst case, it will meet a month after the project has shipped. We're going through the motions even though the thing has shipped. The architecture will be reviewed on paper after it's already been validated in the field, which is silly.

Someone told me a story once. A client came to them with a complaint that some provisioning software IBM had sold them didn't work. What had happened was we'd promised that our nifty provisioning software would allow them to create virtual machines in ten minutes. This was several years ago, when "a VM in ten minutes" was advanced and cool. We promised them it would be wonderful.

When the client got it installed and started using it, they did not find it wonderful. They'd thought they were going to get a 10-minute provision time, but what they were seeing is that it took them three months to provision a Cloud instance. They came back to us, and they said,

"your software is totally broken. You mis-sold it. Look, it's taking three months." We were puzzled by this, so we went in and did some investigation. It turns out what had happened was they had put an 84-step pre-approval process in place to get one of those instances.



"This provisioning software is broken"

The technology was there, but the culture wasn't there, so the technology didn't work. This is sad. We take this cloud, it's a beautiful cloud, it has all these fantastic properties, it makes everything really easy, and then another part of the organization says, "Oh, that's a bit scary. We wouldn't want people actually to be able to do things. Let's put it in a cage!" That old-style paperwork-heavy governance is just not going to work – as well as being really annoying to everyone. It's not going to give the results. What's worse, it's not actually going to make things more secure. It's probably going to make them less secure. It's definitely going to make things slower and cost money. We shouldn't be doing it. I talked to another client, a large automotive company, and they were having a real problem with their Cloud provisioning. It was taking a really long time to get instances. They thought, "The way we're going to fix this is we're going to move from Provider A to Provider B." That might have worked, except the slowness was actually with their internal procurement. Switching providers would bypass their established procurement processes, so it might speed things up for a while, but eventually, their governance team were going to notice that the new provider, and impose controls. Once that happened, they would put the regulation in place, and the status quo would be restored. They would have had all the cost of changing but not actually any of the benefits. It's a bit like— I'm sorry to say I have sometimes been tempted to do this— if you're looking at your stove, and you decide, "Oh, that oven is filthy. Cleaning it will be hard, so I'm going to move house, so I don't have to clean the oven." But then, of course, the same thing happens in the other house, and the new oven gets dirty. You need a more sustainable process than just switching providers to try to outfox your own procurement.

If the developers are the only ones changing, if the developers are the only ones going Cloud-native, then it's just not going to work. That doesn't mean a developer-driven free-for-all is the right model. If there isn't some governance around it, then Cloud can become a mystery money pit. Many of us have had the problem of looking at a cloud bill and thinking "Hmm. Yeah, that large, and I don't understand where it's all going or who's doing it."

It is so easy to provision hardware with the Cloud, but that doesn't mean the hardware is free. Someone still has to pay for it. Hardware being easy to

provision also doesn't guarantee the hardware is useful.

When I was first learning Kubernetes, I tried it out, of course. I created a cluster, but then I got side-tracked, because I had too much work in progress. After two months, I came back to my cluster and discovered this cluster was about £1000 a month … and it was completely value-free. That's so wasteful I still cringe thinking about it.

A lot of what our technology allows us to do is to make things efficient. Peter Drucker, the great management consultant, said "There is nothing so useless as doing efficiently that which should not be done at all." Efficiently creating Kubernetes clusters with no value, that's not good. As well as being expensive, there's an ecological impact. Having a Kubernetes cluster consuming £1000 worth of electricity to do nothing is not very good for the planet.

For many of the problems I've described, what initially seems like a technology problem is actually a people problem. I think this one is a little bit different, because this one seems like a people problem and is actually a technology problem. This is an area where tooling actually can help. For example, tools can help us manage waste by detecting unused servers and helping us trace servers back to originators. The tooling for this isn't there yet, but it's getting more mature.

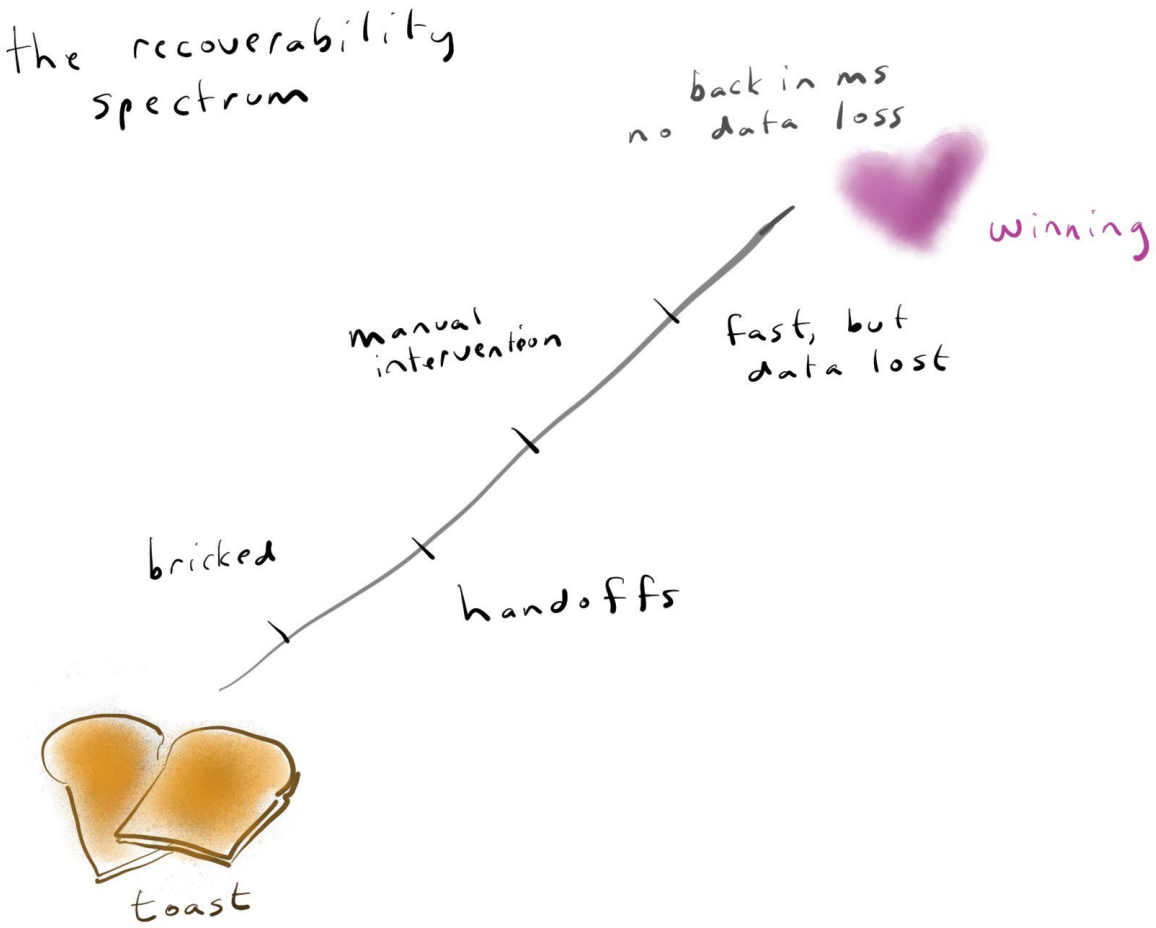
Cloud to manage your Cloud

This cloud-management tooling ends up being on the Cloud, so you end up in the recursion situation to have some Cloud to manage your clouds. My company has a multi-cloud manager that will look at your workloads, figure out the shape of the workload, what the most optimum provider you could have it on is financially, and then make that move automatically. I expect we'll probably start to see more and more software like this where it's looking at it and saying, "By the way, I can tell that there's actually no traffic to his Kubernetes cluster that's been sat there for two months. Why don't you go have some words with Holly?"

Microservices Ops Mayhem

Managing cloud costs is getting more complex, and this reflects a more general thing, which is that cloud ops is getting more complex. We're using more and more cloud providers. There are more and more Cloud instances springing up. We've got clusters everywhere, so how on earth do we do ops for this? This is where SRE (Site Reliability Engineering) comes in.

Site reliability engineering aims to make ops more reproducible and less tedious, in order to make services more reliable. One of the ways it does this is by automating everything, which I think is an admirable goal. The more we automate things like releases, the more we can do them, which is good for both engineers and consumers. The



ultimate goal should be that releases aren't an event; they're business as usual.

What enables that boring-ness is that we have confidence in the recoverability, and it's the SRE who gives us confidence in the recoverability.

I've got another sad space story, this time from the Soviet Union. In the 80s, an engineer wanted to make an update to the code on a Soviet space probe called Phobos. At this time, it was machine code, all 0s and 1s, and all written by hand.

Obviously, you don't want to do a live update of a spacecraft hurtling around the earth with hand-written machine code, without some checks. Before any push, code would be passed through a validator, which was the equivalent of a linter for the machine code.

This worked well, until the automated checker was broken, when changes needed to be made. An engineer said, “Oh, but I really want to do this change. I'll just bypass the automated checks and just do the push of my code to the space probe because, of course, my code is perfect." So they did a live update of a spacecraft hurtling around the earth with hand-written machine code, without checks. What could possibly go wrong?

What happened was a very subtle bug. Things seemed to be working fine. Unfortunately, the engineer had forgotten one zero on one of the instructions. This changed the instruction from the intended instruction to one which stopped the probe's charging fins rotating. The Phobos had fins which turned to orient towards the sun so that it could collect

solar power, no matter which way it was facing. Everything worked great for about two days, until the battery went flat. Once the probe ran out of power, there was nothing that they could do to revive it because the entire thing was dead.

That is an example of a system that is completely unrecoverable. Once it is dead, you are never getting that back. You can't just do something and recover it to a clean copy of the space probe code, because it's up in space.

Systems like this are truly unrecoverable. Many of us believe that all of our systems are almost as unrecoverable as the space probe, but in fact, very few systems are.

Where we really want to be is at the top end of this spectrum, where we can be back in milliseconds, with no data loss. If anything goes wrong, it's just, "ping, it's fixed". That's really hard to get to, but there are a whole bunch of intermediate points that are realistic goals.

If we're fast in recovering, but data is lost, that's not so good, but we can live with that. If we have handoffs and manual intervention, then that will be a lot slower for the recovery. When we're thinking about deploying frequently and deploying with great boredom - we want to be confident that we're at that upper end. The way we get there, handoffs bad, automation, good.

Ways to succeed at Cloud Native

This article has included a whole bunch of miserable stories about things that I've seen that can go wrong. I don't want to leave you with an impression that everything goes wrong all the time because a lot of the time, things do go really right. Cloud native is a wonderful way of developing software, which can feel better for teams, lower costs, and make happier users. As engineers, we can spend less time on the toil and the drudgery and more time on the things that we actually want to be doing… and we can get to market faster.

To get to that happy state, we have to have alignment across the organization. We can't have one group that says microservices, one group saying fast, and one group saying old-style governance. That's almost certainly not going to work, and there will be a lot of grumpy engineers and aggrieved finance officers. Instead, an organisation should agree, at a holistic level, what it's trying to achieve. Once that goal is agreed, it should optimize for feedback, ensuring that feedback loops as short as possible, because that's sound engineering.

TL;DR

- It is possible to be very cloud-native without a single microservice
- Before embarking on a cloud-native transformation, it's important to be clear on what cloud-native means to your team and what the true problem being solved is
- The benefits of a microservices architecture will not be realised if releases involve heavy ceremony, are infrequent, and if all microservices have to be released at the same time.
- Continuous integration and deployment is something you do, not a tool you buy
- Excessive governance chokes the speed out of cloud, but if you don't pay enough attention to what is being consumed, there can be serious waste

Containers Are Contagious and Often Misused

by **Alaa Tadmori,** Cloud Solutions Architect at Microsoft

[My writings are personal opinions and don't represent my company]

Let's get something straight right at the beginning — this article is not to argue that containers are bad, containers are certainly one of many great options developers have in their hands today. This article is also not scoped at the pros/cons of containers, my intent is just to present the developers and dev leads with some considerations around containers that I believe should be on the table when containers are discussed.

Before we dive in let me take your temperature (just kidding, I hate COVID), let's just make sure you live in 2021:

- .Net 5 is released already and (like its predecessor .Net Core) is cross-platform (Windows, Linux, and macOS).
- OpenAPI, OpenID, OAuth2 are dominant open standards (among many others). E.g. the OpenID Foundation mentions that there are 'over one billion OpenID enabled user accounts and over 50,000 websites accepting OpenID for logins'.
- Most major clouds (including Azure) natively support both Linux and Windows operating systems, and natively supports Java as much as they support C# (among other languages). In fact, Java is so popular inside Microsoft that it got its own open-source Java SDK –OpenJDK.

There are clean solutions today to do one infrastructure-as-code for multi-cloud.

If these points are old news to you that is great, otherwise I'm afraid you're still living in 2010 when containers were the only viable solution for many issues and long-term symptoms didn't show up yet. If this is the case it's fine, I trust you have it in you to be open-minded to what is coming.

Containers are not the only solution for the 'write-once run-everywhere' objective

Ten years ago it was the dream of every developer to pull a code from a repo and hit F5 to run it without running into some dependency issue that broke the build; deploying the code to new environments was cumbersome and error-prone. containers are excellent at resolving these challenges but are not the only option in 2021.

(I know this discussion is for developers not for business leads but let's be real, just five years ago if the business folks tell you they need an 'exit-strategy' they were pretty much telling you to containerize your apps)

As the world moves towards cross-platform frameworks and open standards, code deployment between different environments/clouds is becoming easier by day — your app no longer needs to live in a box to get the benefits of container deployments. Take advantage of these cross-platform technologies and build upon open standards. Here are few examples:

Rebus allows you to write against an emulated queue during development and later deploy the code against an actual queue hosted in any supported platform (they support multiple services like Azure Service Bus, RabbitMQ, Amazon SQS, etc.).

Like Rebus, Storage.net abstracts away the storage, it supports Azure Blob, AWS S3 among many others.

GitHub Actions allows you to deploy your same codebase directly into an Azure and/or AWS web app or serverless offerings if you use a language supported by both (like C#, Java, Node.js..etc).

If/when you can't use a modern approach and the requirements ask for containers consider using a low(er)-level language (C++/Rust...etc.) – why bother with an additional abstraction layer (i.e. runtime) and possibly compromise performance if portability is a no issue? Am I saying we should forget C# and Java and start writing Rust? It is actually not a bad idea when optimal performance is required, the application requirements should inform your decision but suffice to say that stacking layers of abstractions even when it doesn't cost much in terms of performance it is certainly not free.

Containers are contagious

Imagine you have a scenario where containers make perfect sense (there are many use cases where this is still the case today), you containerized your app, now imagine you get a second app you want to write, do you go back to the drawing board and check what would be the best solution for this second app or you use your already existing skill set of hosting/running containers and leverage all the knowledge you gained from the first app?

Now imagine that you have 10 apps, are you going to go to the drawing board each time? It is a very compelling argument to use the existing skill set/tools the team has grown building their first containers-solution but that only holds true for the first few apps and only when containers are good solutions for these apps in the first place — it is only when you get to the 'one thousand and one' container when you realize you're living in a containers orchestration/management nightmare.

I have seen this scenario multiple times with some variations with previous clients – the ability to containerize a legacy app and easily moving it to the cloud is indeed charming, few months into the project and you will wake

up from the spell of containers only to find that Kubernetes is now the ruler of the Seven Kingdoms, what Kubernetes handles well goes well but areas that it doesn't handle well will force you to come up with all kinds of ugly workarounds - Kubernetes limits now becomes your own limits. E.g. I had a client who spent some significant time containerizing an application, testing in the sandbox environment was promising, deploying to production was a centimeter away when we found out that the containers don't hold their MAC addresses between restarts (i.e. Kubernetes assigns the MAC when the container starts and these addresses don't persist), it is not a big deal for most apps but in this case, there was a licensing server that depends on MAC addresses for issuing licenses. The project was eventually dropped off before we were able to find a workaround. If you're curious, the closest we came to resolving this was to us Azure Container Networking Interface (CNI) although it was still in the preview that time, it might be available by the time you're reading this, however, this is just an example to demonstrate that containers limits can be hard to workaround.

My advice is to go back to the drawing board and cater to the hosting needs of each app. Consider breaking your app to microservices to build long-term maintainable solutions, use PaaS and Serverless whenever possible. There are many good articles online that discuss the use cases and benefits of using PaaS and Serverless, if there is one thing to mention here it would be the simple fact that the less time you spend on the lower layers (e.g. the webserver, the orchestration engine, the OS… etc.) the more time you get to focus on developing your own applications.

Simplicity is not discussed often enough

We (developers) love complexity – if you don't believe me try it in your own dev team, ask developers to create 'a fork and a knife' application, give them few days, and watch them come back with a 'swiss army knife' application – Complex is often cool to implement and it might be exciting to add these amazing features that the requirements don't call for…..until you deploy your solution and an issue appears, debugging an issue in a complex solution is hard and can waste a lot of valuable time.

Complexity is bad for too many obvious reasons, and it only gets worse over time – think of the 2nd law of thermodynamics, complex solutions will ever grow more complex unless you make Simplicity a primary objective and invest time and effort maintaining it throughout the lifecycle of the solution. We don't discuss simplicity often enough and maybe we should start thinking of it as a requirement alongside the other non-functional requirements so we don't compromise on it. In all cases, any effort of maintaining simplicity will require us to define what it is exactly that we mean by Simple.

(I was tempted to refer to 'Simplicity' as 'Elegance' because I feel simplicity is sometimes mistaken with being 'basic' or 'incapable'. The solution requirements can be demanding yet still be met with simple/elegant solutions. If you tend to correlate simple solutions with incapable solutions please feel free to use the word 'Elegant' instead of 'Simple').

What exactly is Simplicity?

You're probably thinking that simplicity in its general sense is relevant and varies from one solution to another – I agree, let's define it in a way that makes it measurable so we can decide if containers (and everything else) are good or bad in terms of simplicity.

In my view, a simple solution is a solution that every piece of it has a clear purpose that directly serves one or more of the solution requirements & nothing beyond them. The purpose (or in Aristotle's terms the "final cause") of each component (and each piece of code and each and everything else in the solution) must be clear and easily identifiable (intuitive).

For example, let's say we have a requirement to have a web page where users can see their shopping history list, if you read this requirement you would intuitively expect some kind of ViewModel that reflects this list; the 'final cause' of this ViewModel is to be the shopping history list on that web page.

Here is the thing, the opposite should also be true – even if you didn't know the requirements, if you clone the repo and see this ViewModel it is intuitive for you to know its purpose, the minute you see it you will intuitively infer there must be a requirement for users to see their shopping history list. This ViewModel is therefore simple, it doesnt add complexity to the solution because it has that clear requirement-driven purpose attached to it.

Let's apply this to containers, if you clone a solution written in C++, the next thing you would be looking for is a Docker file because abstracting away the dependencies is essential in making this solution portable – using containers here has an intuitive purpose that helps in keeping the solution simple.

If on the other hand the solution is built on top of a modern/ cross-platform technology stack, boxing it in a container is a layer of unintuitive and unjustifiable complexity.

Convenience is the enemy of simplicity.

Conclusion

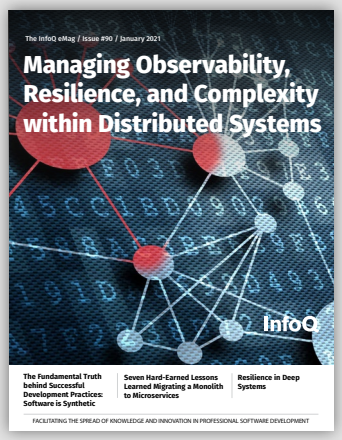
Containers have many use cases where they can bring a lot of power and simplicity to your solutions but they also can add a lot of complexity. Analyze the requirements first then decide on what technology stack is the most appropriate.

Avoid the common fallacy of using containers just because you invested time and effort building your first containerized solution. Containers, like any other technology, have their limits and the abstraction layer they add is not free in terms of performance. If you can do without containers you should absolutely aim for that and resort to using containers only when there is a clear purpose for going this route.

TL;DR

- Containers are not the only solution for the write-once run-everywhere objective, you can use cross-platform frameworks to build clean/ simple solutions that are also easily deployable between different environments/clouds.
- The initial effort of learning how to develop, deploy, and run containerized applications creates a sunk cost effect that makes it hard to move out of containers when needed (the Concorde fallacy).
- Using a modern technology stack can liberate you from Kubernetes. Using PaaS and Serverless gives you more time to focus on your application needs.
- Stacking layers of abstractions affects performance. Using low-level programming language with containers can give you performance gains that can be crucial for some applications.
- Simplicity is vital in maintaining solutions with ever more complex and more demanding requirements. Containers can simplify some solutions but are often an extra layer of unintuitive complexity

Read recent issues

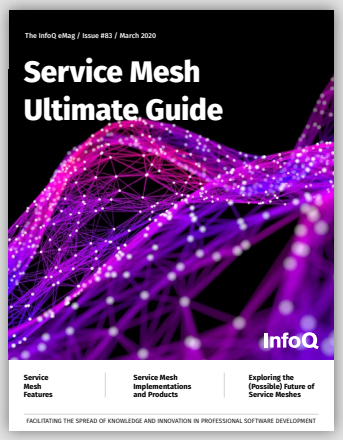


Managing Observability, Resilience, and Complexity within Distributed Systems

This eMag helps you reflect on the subject of reducing complexity within modern applications and distributed systems, and provides you with different perspectives and learned lessons from people who have already had to deal with challenges from the real world.

Re-Examining Microservices after the First Decade

We have prepared this eMag for you with content created by professional software developers who have been working with microservices for quite some time. If you are considering migrating to a microservices approach, be ready to take some notes about the lessons learned, mistakes made, and recommendations from those experts.



Service Mesh: Ultimate Guide

This eMag aims to answer pertinent questions for software architects and technical leaders, such as: what is a service mesh?, do I need a service mesh?, and how do I evaluate the different service mesh offerings?

InfoQ

InfoQ @InfoQ InfoQ InfoQ