LINUX JOURNAL PRESENTS

"In Their Words" Voices from the Community

Joe Beda, Co-Founder and CTO of Heptio, on Becoming a Cloud Native Organization

heptio Unleash the power of Kubernetes with Heptio

Table of Contents

Joe Beda started his career at Microsoft working on Internet Explorer (he was young and naive). Throughout seven years at Microsoft and ten at Google, Joe has worked on GUI frameworks, real-time voice and chat, telephony, machine learning for ads and cloud computing. Most notably, while at Google, Joe started the Google Compute Engine and, along with Brendan Burns and Craig McLuckie, created Kubernetes. Today, Joe Beda is the Co-Founder and CTO of Heptio, which provides IT organizations with what they need to realize the full potential of Kubernetes and transform IT into a business accelerator. Joe proudly calls Seattle home.

Linux Journal Presents: In Their Words—Voices from the Community

About the Sponsor

Heptio

Heptio unleashes the technology-driven enterprise with products and services that help customers realize the full potential of Kubernetes and transform IT into a business accelerator. The company's training, support and professional services speed integration of Kubernetes and related technologies into the fabric of enterprise IT, while its products reduce the cost and complexity of running these systems in production environments. To learn more, visit https://www.heptio.com.

Joe Beda, Co-Founder and CTO of Heptio, on Becoming a Cloud Native Organization

JOE BEDA

Introduction

As Linux has become the mainstay of Enterprise IT, it has become increasingly challenging to install, test and ultimately review properly new products built for large, scalable environments. Although *Linux Journal* publishes substantial, in-depth product reviews, we can't possibly review every worthwhile product, especially in an arena like ours that grows and changes so fast. Increasingly, too, important discussions focus on issues of design process, organization and communication—not just on specific products or tools.

Linux Journal's "In Their Words" series will allow some really smart new product creators to tell about their product and their universe, in great technical depth, in their own words. This series is not intended to sell you anything, but rather to keep you aware, with the technical depth you need, of the latest and greatest advancements and innovations in our community.

Information Week provides this definition of "cloud native" (from 2015):

> At the heart of "cloud native" lies Linux, Linux containers, and the concept of applications assembled as microservices in containers. Indeed, the Linux Foundation launched the Cloud Native Computing Foundation. But cloud native means a lot more than implementing Linux clusters and running containers. It's a term that recognizes that getting software to work in the cloud requires a broad set of components that work together. It also requires an architecture that departs from traditional enterprise application design.

Cloud computing simply used to mean running your software on someone else's servers. It was an incremental adjustment, like moving from a manual to automatic transmission. But as cloud computing, or "cloud native" thinking has evolved, it has become a new development and delivery platform in general. Rather than an incremental step, going cloud native is akin to when we switched from horses to automobiles. Transportation was never the same again, and likewise, neither will the world of development.

In this ebook, Joe Beda goes beyond tools and design, beyond the traditional boundaries of software engineering, with an expansive discussion of cloud native thinking. Then, with sections devoted to cloud native in practice, devops, containers and clusters, microservices and security, he presents a roadmap for organizations to move toward a cloud native model.

As Joe says, "We are still at the beginning of this journey." Jump in and let's see where it leads.

Cloud Native Definition

Since I started my journey toward building Heptio along with CEO Craig McLuckie, I've been doing a lot of thinking around where the industry is going. Craig and I spent quite a bit of time at Google (16 years between the two of us) and have a good understanding of how Google builds and manages systems. But chances are you don't work at Google. So how do all of these evolving new concepts apply to a typical company/developer/operator?

In this ebook, I examine multiple ways to understand and apply cloud native thinking. There's no hard and fast definition, and other terms and ideas overlap. Automation and new architectures are key to realizing the promise of managing complexity and achieving greater velocity. But, at its root, cloud native thinking is as much about teams, people and culture as it is about technology.

One important note: *you don't have to run in the cloud to be cloud native*. The techniques that I describe here can be applied incrementally as appropriate and should help smooth any transition to the cloud or between clouds.

The real value of cloud native thinking goes far beyond the basket of technologies that are closely associated with it. To understand where our industry is really going, we need to examine where and how we can make companies, teams and people more successful.

These techniques have been proven at big, technology-centric, forward-looking companies that have dedicated many resources to the effort—think Google or Netflix or Facebook. Smaller, more flexible companies are also realizing value here. However, *there are very few examples of this philosophy being applied outside technology early adopters.* We are still at the beginning of this journey when viewed across the wider IT world.

Key outcomes based on early experiences include:

- **More efficient and effective teams.** Cloud native tooling lets you break down big problems into smaller components for more focused and nimble teams.

- **Reduced drudgery through automating much of the manual work that causes operations pain and downtime.** Automation can enable *self-healing and self-managing infrastructure*, and teams can expect systems to do more.

- **More reliable infrastructure and applications.** Building automation to handle expected churn often results in better failure modes for unexpected events and failures. For example, if it is a single command or button click to deploy an application for development, testing or production, it can be much easier to manage deployment in a disaster-recovery scenario, either automatically or manually.

- **Auditable, visible and debuggable applications.** Complex applications can be very opaque. The tools used for cloud native applications, by necessity, can provide more insight into what is happening within an application.

- **Deep security.** Many IT systems today have a hard outer shell and a soft gooey center. Modern systems should be secure and "least trust" by default. A cloud native approach lets application developers play an active role in creating securable applications.

- **More efficient usage of resources.** Automated deployment and management of applications and services open up opportunities to apply algorithmic automation. For instance, a cluster scheduler/orchestrator can automate placement of work on machines, instead of an operations team managing a similar assignment in a spreadsheet.

Cloud Native in Practice

Let's look more closely now at what it means to put cloud native thinking into practice in the following areas:

- Integration with existing systems
- DevOps
- Containers and orchestration
- Microservices
- Security

Like any area with active innovation, there is quite a bit of churn in the cloud native world. It isn't always clear how best to apply these ideas. In addition, any project of significance is inevitably too important and too large to re-write from scratch (see Joel Spolsky's classic blog post on this: https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/). Instead, I encourage you to experiment with these new structures for newer projects or for new parts of existing projects. Then, as you tackle improving older parts of the system, take the time to apply new techniques and learnings as appropriate. Look for ways to break out new features or systems as microservices—in other words, break up a large application into smaller pieces that can be developed and managed independently.

There are no hard and fast rules. Every organization is different, and software development practices must be scaled to the team and project at hand. The map is not the territory. Some projects are amenable to experimentation, and others are critical enough that they should be approached much more carefully. There are also situations in the middle where the techniques that were proven out need to be formalized and tested at scale before they are applied to critical systems.

Cloud Native Systems and Tools

A cloud native approach is defined by better tooling and better systems. Without this tooling, *each new service in production will have a high operational cost.* Each new service is a separate thing that has to be monitored, tracked, provisioned and so forth. That overhead is one of the main reasons why sizing of microservices should be done in an appropriate way. *The benefits in development team velocity must be weighed against the costs of running more things in production.*

Similarly, introducing new technologies and languages, although exciting, comes with cost and risk that must be weighed carefully. Charity Majors has a great talk about this issue at https://www.oreilly.com/ideas/a-young-ladys-illustrated-primer-to-technical-decision-making.

Automation is the key to reducing the operational costs associated with building and running new services. Systems like Kubernetes, containers, CI/CD (continuous integration/continuous deployment), monitoring and so on all have the same overarching goal of making application development and operations teams more efficient so they can move faster and build more reliable products.

The newest generation of tools and systems are better set up to deliver on the promise of cloud native over older traditional configuration management tools, as they help to break the problem down so that it can be spread across teams easily.

The newest generation of tools and systems are better set up to deliver on the promise of cloud native over older traditional configuration management tools, as they help to break the problem down so that it can be spread across teams easily. Newer tools generally empower individual development and ops teams to retain ownership and be *more productive through self-service IT.*

DevOps

It is probably most useful to think of DevOps as a *cultural shift* whereby developers must care about how their applications are run in a production environment. In addition, the operations folks are empowered to know how the application works, so they can play an active part in making the application more reliable. Building understanding and empathy between those teams is key.

And if you re-examine the way applications are built and how the operations team is structured, you can improve and deepen this relationship.

For example, Google does not employ traditional operations teams. Instead, Google defines a new type of engineer called the *Site Reliability Engineer*. These are highly trained engineers who not only carry a pager, but also are expected and empowered to play a critical role in pushing applications to be ever more reliable through automation.

When a pager goes off at 2am, anyone who answers does the exact same thing—try to fix things so that they can go back to bed. What defines an SRE is what happens at 10am the next morning. Instead of just accepting that this is the way things are, as traditional siloed operations people might do, SREs work with the development team to ensure that an outage like that never will happen again. *The SRE and development teams have incentives aligned* around making the product as reliable as possible. This approach, combined with blameless post-mortems (https://www.pagerduty.com/blog/blameless-post-mortems-strategies-for-success), can lead to healthy projects that don't collect technical debt.

SREs are some of the most highly valued people at Google. In fact, oftentimes products launch without SREs with the expectation that the *development team will run their product* in production. The process of bringing on SREs often involves the development team proving to the SRE team that the product is ready. It is expected that the development team will have done all of the leg work, including setting up monitoring and alerting, alert playbooks and release processes. The dev team should be able to show that pages are at a minimum and that most problems have been automated away. Other companies don't have a separate SRE/ops role and instead follow the mantra of "You build it, you run it". Regardless of the details, when executed well, this model leads to more and more robust and automated systems over time.

As the role of operations becomes more involved and application-specific, it doesn't make sense for a single team to own the entire operations stack. Instead, we can think in terms of operations specialization. Let's take it from the bottom up:

- **Hardware Ops.** This is already clearly separable. In fact, it is easy to see cloud IaaS as "Hardware Ops as a Service".

- **Operating System Ops.** Someone has to make sure the machines boot and that there is a good kernel. Breaking this out from application dependency management mirrors the trend of minimal OS distributions focused on hosting containers (CoreOS Container Linux, Red Hat Project Atomic, Ubuntu Snappy, Google Container Optimized OS).

- **Cluster Ops.** In a containerized world, a compute cluster becomes a logical infrastructure platform. The cluster system (such as Kubernetes) provides a set of primitives that enables many of the traditional operations tasks to be self-service.

- **App Ops.** Each application can now have a dedicated app ops team as appropriate. As I mentioned previously, the dev team can and should play this role as necessary.

> The ops team members are expected to go deeper on the application because they don't have to be experts in the other layers. For example, at Google, the AdWords front-end SRE team talks to the AdWords front-end development team a lot more than they'll talk to the cluster SRE team. This alignment of incentives can lead to better outcomes.

There probably is room for other specialized SRE teams depending on the needs of the organization. For instance, storage services could be broken out as a separate service with dedicated SREs, or there might be a team responsible for building and validating the base container image that all teams should use as a matter of policy.

Containers and Clusters

It's helpful to try to get to the root of why containers are exciting to so many folks. In my mind, there are three reasons for this excitement:

- Packaging and portability
- Efficiency
- Security

Let's look at each of these in turn.

First, containers provide a **packaging mechanism**. This allows the building of a system to be separated from its deployment. In addition, the artifacts/images that are built

Over time, expect to see much more sophisticated ways to balance the needs of containers cohabitating a single host without noisy neighbor issues.

are much more portable across environments (dev, test, staging, production) than more traditional approaches, such as VM images. Deployments also become more atomic. Traditional configuration management systems (Puppet, Chef, Salt or Ansible) can leave systems in a half-configured state that is hard to debug. It is also easy to have unintended version skew across machines without realizing it. Containers help keep configurations and versions stable.

Second, containers can be lighter weight than full systems, which can lead to more *efficient resource utilization*. This was the main driver when Google introduced cgroups—one of the core kernel technologies underlying containers. By sharing a kernel and allowing for more fluid overcommit, containers can make it easier to "use every part of the buffalo". Over time, expect to see much more sophisticated ways to balance the needs of containers cohabitating a single host without noisy neighbor issues.

Finally, many users view containers as a *security boundary*. Although containers can be more secure than simple UNIX processes, care should be taken before viewing them as a hard security boundary. The security assurances provided by Linux namespaces may be appropriate for "soft" multi-tenancy where the workloads are semi-trusted but not appropriate for "hard" multi-tenancy where workloads are actively antagonistic. This situation continues to evolve as many people are working hard to improve the security situation for containers. Jess Frazelle has a fun container CTF (capture the flag) up at https://contained.af to demonstrate some best practices.

There is ongoing work in multiple quarters to blur the lines between containers and VMs. Early research into systems like unikernels is interesting but won't be ready for wide production for years yet.

Although containers provide an easy way to achieve the goals of portability, efficiency and security, they aren't absolutely necessary. Netflix, for instance, traditionally has run a very modern stack (and is often an AWS example customer) by packaging and using VM images (EC2 AMIs) similar to how others use containers.

Most of the original excitement about containers centered on managing the software on a single node in a more reliable and predictable way. The next step in this evolution manages the software in terms of clusters (also often known as orchestrators). Taking a number of nodes and binding them together with automated systems creates a new *self-service logical infrastructure* for development and operations teams.

Clusters help eliminate ops drudgery. With a container cluster, you make computers take over the job of figuring out which workload should go on which machine. Instead of paging people, clusters also automatically fix things when hardware fails in the middle of the night.

Clusters *enable the operations specialization* that allows application ops to thrive as a separate discipline. With a well defined cluster interface, application teams can concentrate on solving the problems that are immediate to the application itself.

And clusters make it possible to *launch and manage more services*. This allows new architectures that can *unlock velocity* for development teams—specifically, architectures based on microservices.

Microservices

This is a new name for a concept that has been around for a long time. Basically, it is a way to break up a large application into smaller pieces that can be developed and managed independently. Let's look at some of its key attributes:

- **Strong and clear interfaces.** Tight coupling between services must be avoided. Documented and versioned interfaces help solidify the contracts between services and also retain a certain degree of freedom for consumers and producers of these services.

- **Independently deployed and managed.** It should be possible for a single microservice to be updated without having to touch all the other services. It is also desirable to be able to roll back a version of a microservice easily. These requirements mean that both the API and any data schemas for the service must be forward- and backward-compatible. Good cooperation and communication mechanisms between the appropriate ops and dev teams are vital here.

- **Resilience built in.** Microservices should be built and tested to be independently resilient. Code that consumes a service should strive to continue working and do something reasonable if the consumed service is down or misbehaving. Similarly, any service that is offered should be able to handle unanticipated load and bad input gracefully.

Sizing of microservices can be a tricky thing to get right. Avoid services that are too small (pico-services); instead, split services across natural boundaries (languages, async queues, scaling requirements), and keep team sizes reasonable (that is, two-pizza teams).

The application architecture should be allowed to *grow in a practical and organic way.* Instead of starting with 20 services, start with two or three, and split services as complexity in that area grows. Oftentimes the architecture of an application isn't clearly understood until the application is well under development. Remember that applications are rarely "finished" but rather always a work in progress.

Are microservices a new concept? Not really. They represent another type of software componentization. We've always split up code into libraries. In a microservices-driven world, the "linker" becomes a run-time concept instead of a build-time concept. (A recognition of this parallel can be found in the linkerd project. This is a CNCF project based on the Twitter Finagle system.) This is also very similar to the SOA push from several years ago but without all of the XML. Viewed from another angle, the database has almost always been a "microservice", in that it often is implemented and deployed in a way that satisfies the points above.

Constraints can lead to productivity. Although it's tempting to allow each team to choose a different language or framework for each microservice, consider standardizing on a few languages and frameworks. Doing so helps improve knowledge transfer and mobility within the organization. However, be open to making exceptions to policy as necessary. This is a key advantage of this world over a more vertically integrated and structured PaaS. In other words, *constraints should be a matter of policy rather than capability.*

Although most people view microservices as an implementation technique for a large application, there are other types of services that form the *services spectrum*:

- **Shared artifact, private instance.** In this scenario, the development process is shared across many instances of the service. There may be one dev team and many ops teams, or perhaps a unified ops team that works across dedicated instances. Many databases fall into this category where different teams are running private instances of a single MySQL binary.

- **Shared instance.** Here a single team provides a shared service to many applications and teams inside an organization. The service may partition data and actions per user (multi-tenant) or provide a single simple service that is used very widely (for example, serving an HTML UI for a common branding bar, and serving up machine learning models).

- **Big-S service.** Most enterprises don't produce services like this, but they may consume them. This is the typical "hard" multi-tenant service that is built to service a large number of different customers. This type of service requires a level of accounting and hardening that isn't often necessary inside an enterprise. SendGrid or Twilio falls into this category.

As services become not an implementation detail but a common infrastructure in an enterprise, the *service network* morphs from a per-application concept to something that can span the entire company. There is an opportunity and a danger in allowing these types of dependencies.

Security

Security is still a big question in the cloud native world. Old techniques don't apply cleanly, and cloud native approaches may appear to be a step backward initially. But this brave new world also introduces opportunities.

Container Image Security Quite a few tools can help users audit their container images to ensure that they are fully patched. I don't have a strong opinion on the various options.

The real problem is what do you do once you find a vulnerable container image. This is a place where the

NOTE: This section doesn't cover all aspects of security in the new cloud native world. Also, although I'm not a security expert, it is something I've paid attention to throughout my career. Consider these remarks as part of a map of things to consider.

Even if all of the things you are running on your cluster are patched, it doesn't ensure your network is clear of untrusted activity.

market hasn't yet provided a great set of solutions. Once a vulnerable image is found, the problem is no longer technical—it's a process/workflow issue. You need to identify the groups in your organization that are affected, where in your container image tree to fix the problem, and how best to test and push out a new patched version.

CI/CD (continuous integration/continuous deployment) is a critical piece of the puzzle because it enables automated, quick release processes for new images. Integration with orchestration systems lets you identify which users are using which vulnerable images, and allows you to verify that a new fixed version is actually being run in production. Finally, policy in your deployment system can help prevent new containers from being launched with a known bad image. (In the Kubernetes world, this policy is called *admission*.)

Microservice and Network Security Even if all of the things you are running on your cluster are patched, it doesn't ensure your network is clear of untrusted activity.

Traditional *network-based security tools don't work well* in a dynamically scheduled short-lived container world. Short-lived containers may not be around long enough to be scanned by traditional scanning tools. And by the time a report is generated, the container in question may be gone. With dynamic orchestrators, *IP addresses don't have* *long-term meaning* and can be reused automatically.

The solution is to integrate network analysis tools with the orchestrator, so that logical names (and other metadata) can be used in addition to raw IP addresses. This approach can help make alerts more easily actionable.

Many networking technologies *leverage encapsulation to implement an "IP address per container"*. This can create issues for network tracing and inspection tools if such networking systems are deployed in production. Luckily, much of this has been alleviated by the standardization on VXLAN, VLANs, or by using no encapsulation/virtualization.

However, in my opinion, the biggest issues are around microservices. When there are many services running in production, it is necessary to *ensure that only authorized clients are calling any particular service.*

Furthermore, with the reuse of IP addresses, clients need to know that they are communicating with the correct service. As of now, this is largely an unsolved problem. There are two (non-mutually exclusive) ways to approach this problem.

The first approach means more flexible networking systems and the opportunity to implement host-level firewall rules (outside any container) to enable fine-grained access policies for which containers can call other containers. I've been calling this approach *network micro-segmentation*. The challenge here is to configure these policies to work with dynamic scheduling. There is support in Kubernetes to define and apply network policy. In addition, while early yet, multiple companies are working to provide support at the network level, coordination with the orchestrator and higher-level application definitions. One big caveat: micro-segmentation becomes less effective the more widely any specific service is used. If a service has hundreds of callers, simple "access implies authorization" models are no longer effective.

The second approach is for applications to play a larger role in implementing authentication and encryption inside the data center. This approach works as services take on many clients and become "soft multi-tenant" inside a large organization. *A system of identity for production services* is required, however. A little while ago, I started a project called SPIFFE (Secure Production Identity Framework For Everyone) to provide this level of authentication and encryption everywhere. This project is being developed and shepherded by another startup called Scytale. The ideas behind SPIFFE are proven inside companies like Google but haven't been widely deployed elsewhere.

What I've included here is just the tip of the iceberg when discussing cloud security. This is a complex topic that needs careful attention, and details will vary depending on your specific environment. We are constantly exploring the landscape of threats and how to protect against them. Expect more from Heptio on cloud security in the future.

I hope this discussion of multiple ways to think about and apply cloud native application development serves as a jumping off point for your own cloud native thinking. Be sure to see the following Resources section for more information, and please let us know your thoughts on cloud native by reaching out to @jbeda, @cmcluck or @heptio on Twitter.■

Resources

Heptio: https://heptio.com. We simplify and scale Kubernetes for developers and operators, unleashing IT to become an accelerator for the technology-driven enterprise.

Minimal OS Distributions Focused on Hosting Containers:

- CoreOS Container Linux: https://coreos.com/os/docs/latest
- Red Hat Project Atomic: http://www.projectatomic.io
- Ubuntu Snappy: https://developer.ubuntu.com/core
- Google Container Optimized OS: https://cloud.google.com/container-optimized-os/docs

linkerd: https://linkerd.io

Twitter Finagle: https://twitter.github.io/finagle

VXLAN (Virtual Extensible LAN): https://en.wikipedia.org/wiki/Virtual_Extensible_LAN

SPIFFE (Secure Production Identity Framework For Everyone): https://spiffe.io