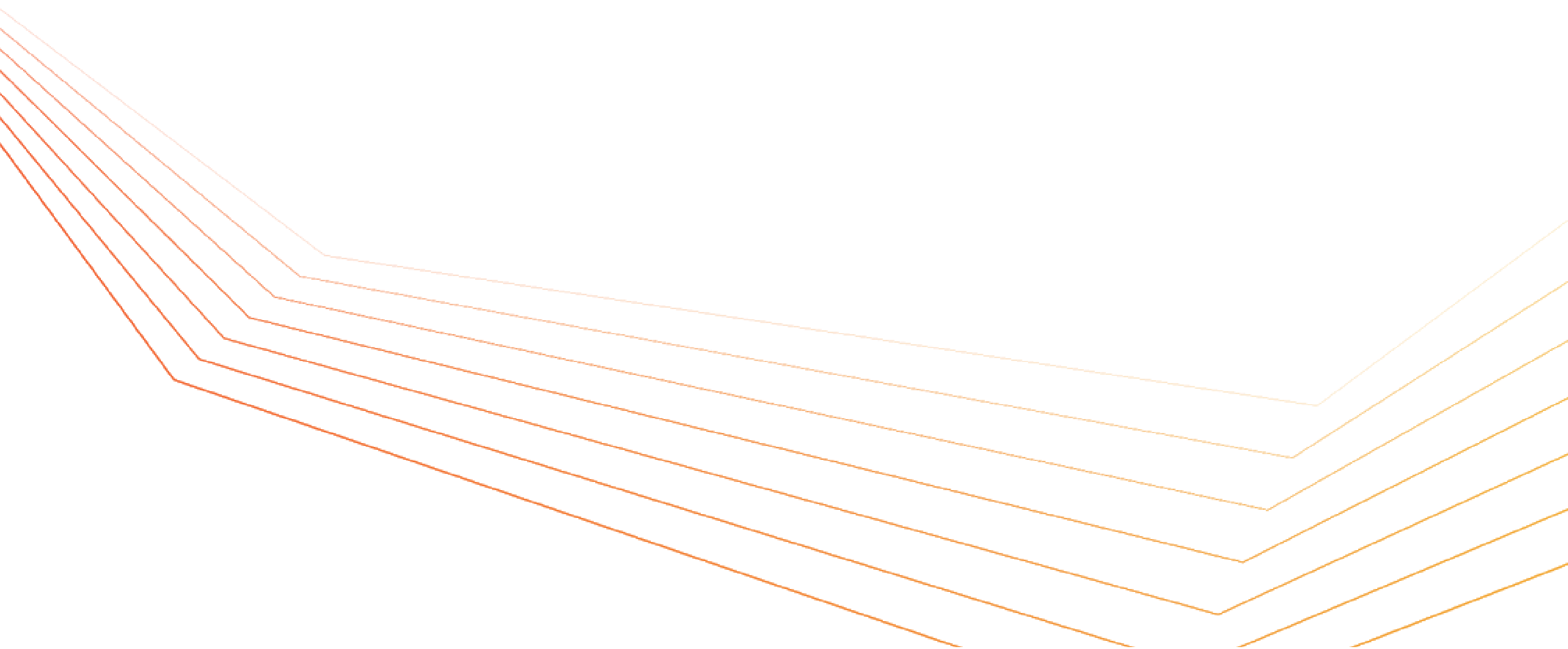
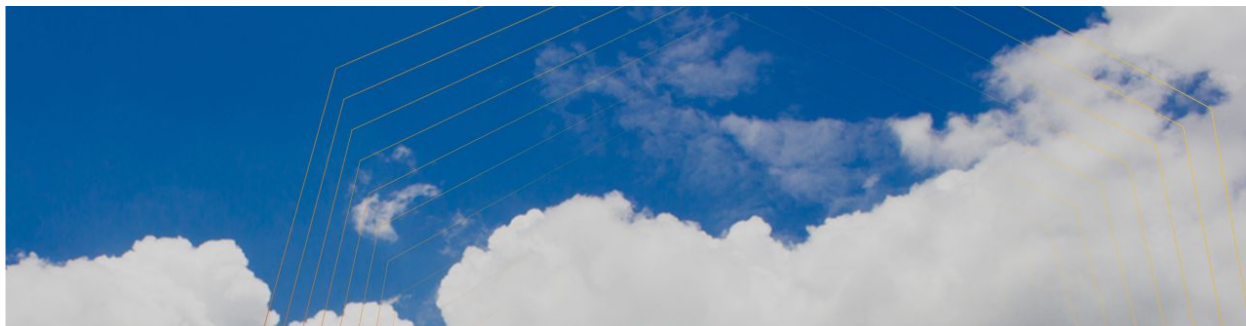
How to Think Cloud Native

6 bite-sized thought pieces on the definition and development of true cloud native capabilities

Joe Beda, CTO and co-founder

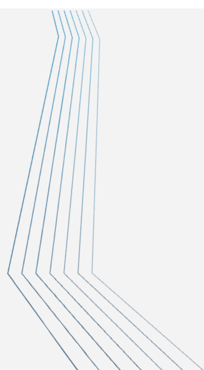




01. Definitions

There is no hard and fast definition for what Cloud Native means. In fact there are other overlapping terms and ideologies. At its root, Cloud Native is structuring teams, culture and technology to utilize automation and architectures to manage complexity and unlock velocity. Operating in this mode is as much a way to scale the people side of the equation as much as the infrastructure side.

One important note: you don't have to run in the cloud to be "Cloud Native". These techniques can be applied incrementally as appropriate and should help smooth any transition to the cloud.



> Cloud Native is structuring teams, culture and technology to utilize automation and architectures to manage complexity and unlock velocity.

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

At this point, these techniques have been proven at technology centric, forward looking companies that have dedicated large amounts of 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 of technology early adopters. We are still at the beginning of this journey when viewed across the wider IT world.

With some of the early experiences being proven out and shared, what themes are emerging?

- **More efficient and happier teams.** Cloud Native tooling allows for big problems to be broken down into smaller pieces for more focused and nimble teams.

- **Drudgery is reduced** through automating much of the manual work that causes operations pain and downtime. This takes the form of self healing and self managing infrastructure. 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. 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 automate deployment in a disaster recovery scenario (either automatically or manually).

- **Auditable, Visible and Debuggable.** Complex applications can be very opaque. The tools used for Cloud Native applications, by necessity, usually provide much 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. Cloud Native enables application developers to play an active role in creating securable applications.

- **More efficient usage of resources.** Automated “cloud like” ways of deploying and managing applications and services opens up opportunities to apply algorithmic automation. For instance, a cluster scheduler/orchestrator can automate placement of work on machines vs. having an ops team manage a similar assignment in a spreadsheet.

02. In Practice

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 the ideas laid out in the previous part.** In addition, any project of significance will be too important and too large for a from-scratch rewrite. Instead, I encourage you to experiment with these new structures for newer projects or for new parts of an existing project. As older parts of the system are improved, take the time to apply new techniques and learnings as appropriate. Look for ways to break out new features or systems as microservices.

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 while 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 being applied to critical systems.

Cloud Native is defined by better tooling and systems. Without this tooling, **each new service in production will have a high operational cost**. It is a separate thing that has to be monitored, tracked, provisioned, etc. 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, while exciting, comes with cost and risk that must be weighed carefully.

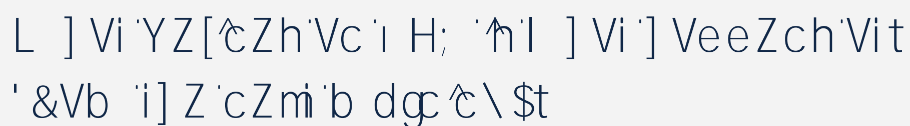
Automation is the key to reducing the operational costs associated with building and running new services. Systems like Kubernetes, containers, CI/CD, monitoring, etc all

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 easily be spread across teams. Newer tools generally empower individual development and ops teams to retain ownership and be **more productive through self service IT.**

03. 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 aware and empowered to know how the application works so that they can actively play a part in making the application more reliable. Building an understanding and empathy between these teams is key.



But this can go further. If we reexamine the way that applications are built and how the operations team is structured, we can improve and deepen this relationship.
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 (that are compensated at the same level as other engineers) that not only carry a pager but are expected and empowered to play a critical role in pushing applications to be ever more reliable through automation.

When the pager goes off at 2am, anyone answering that page does the exact same thing — try to figure out what is going on so that he/she can go back to bed. What defines an SRE is what happens at 10am the next morning. Do the operations people

just complain or do they work with the development team to ensure that a page like that will never happen again? **The SRE and development teams have incentives aligned** around making the product as reliable as possible. That, combined with blameless post-mortems, can lead to healthy projects that don't collect technical debt.

SREs are some of the most highly valued people at Google. In fact, often times 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 play books 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.

As the role of the operations becomes much more involved and application specific, it doesn't make as much sense for a single team to own the entire operations stack. This leads to the idea of Operations Specialization. In some ways this is a type of "anti-devops". 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".

- **OS 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, Red Hat Project Atomic, Ubuntu Snappy, Rancher OS, VMWare Photon, Google Container Optimized OS).

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

- **App Ops**. Each application now can have a dedicated apps team as appropriate. As above, the dev team can and should play this role as necessary. This ops team is expected to go deeper on the application as they don't have to be experts in the other layers. For example, at Google, the AdWords Frontend SRE team will talk to the AdWords Frontend development team a lot more than they'll talk to the cluster SRE (borg-sre) team. This alignment of incentives can lead to better outcomes.

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

04. Containers and Clusters

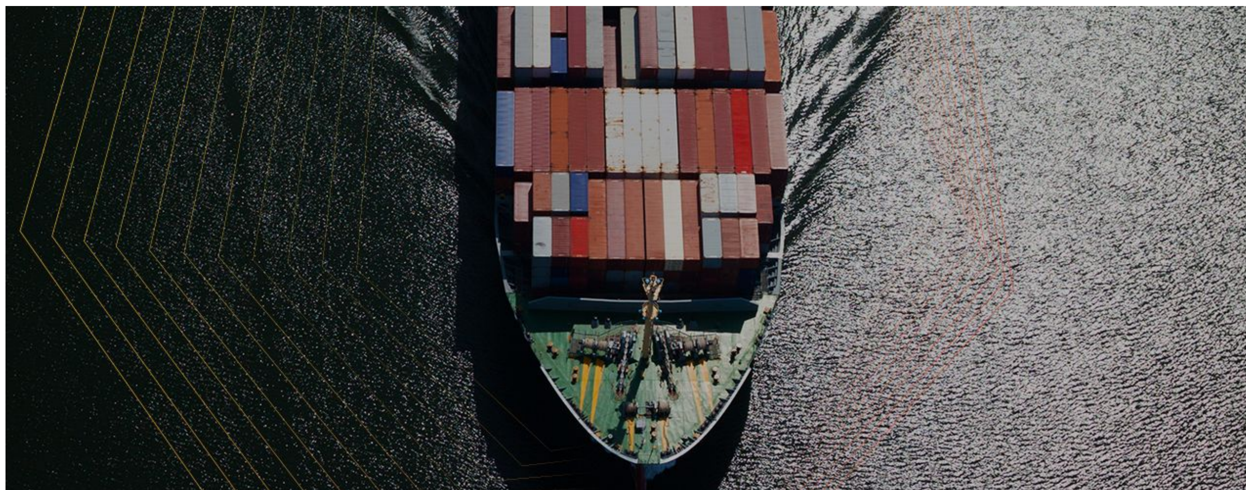
There is quite a bit of excitement around containers. It is helpful to try to get to the root of why containers are exciting to so many folks. In my mind, there are three different reasons for this excitement:

1. Packaging and portability
2. Efficiency
3. 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 the deployment of those systems. In addition, the artifacts/images that are built are much more portable across environments (dev, test, staging, prod) than more traditional approaches such as VM images. Finally, deployments become more atomic. Traditional configuration management systems (puppet, chef, salt, ansible) can easily 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.

Second, containers can be lighter weight than full systems leading to **increased 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 much more fluid overcommit, containers can make it easier to "use every part of the cow." 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. While 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.

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.

While containers provide an easy way to achieve the goals above, they aren't absolutely necessary. Netflix, for instance, has traditionally run a very modern stack (and is the AWS poster child) by packaging and using VM images similar to how others use containers.

t t

9ąj hiZgh] Zæ Zab ^cViZ deh Ygj Y\Zgn$

While most of the original push around containers centered around managing the software on a single node in a more reliable and predictable way, the next step of this evolution is around clusters (also often known as orchestrators). Taking a number of nodes and binding them together with automated systems creates a new self service set of logical infrastructure for development and operations teams.

With a container cluster we make computers take over the job of figuring out what workload should go on which machine. Clusters also silently fix things up when hardware fails in the middle of the night instead of paging someone.

The first thing that clusters do is **enable the operations specialization** (as described above) that allows application ops to thrive as a separate discipline. By having a well defined cluster interface, application teams can concentrate on solving the problems that are immediate to the application itself.

The second benefit of clusters is that it makes it possible to **launch and manage more services**. This allows new architectures (via microservices described in the next installment of this series) that can unlock velocity for development teams.

05. Microservices

Microservices are a new name for a concept that has been around for a very long time. Basically, it is a way to break up a large application into smaller pieces so that they can be developed and managed independently. Let's look at some of the key aspects here:

- **Strong and clear interfaces.** Tight coupling between services must be avoided. Documented and versioned interfaces help to solidify that contract and retain a

certain degree of freedom for both the consumers and producers of these services.

- **Independently deployed and managed.** It should be possible for a single microservice to be updated without synchronizing with all of the other services. It is also desirable to be able to roll back a version of a microservice easily. This means the binaries that are deployed must be forward and backward compatible both in terms of API and any data schemas. This can test the cooperation and communication mechanisms between the appropriate ops and dev teams.

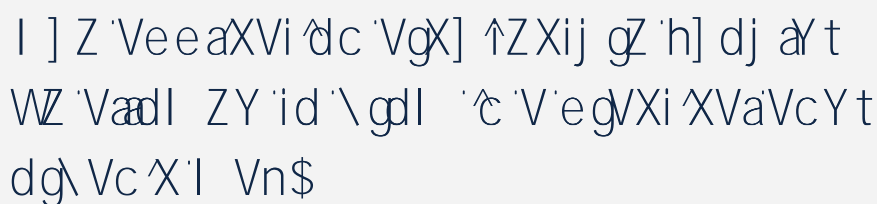
- **Resilience built in.** Microservices should be built and tested to be independently resilient. Code that consumes a service should strive to continue working and do something reasonable in the event that the consumed service is down or misbehaving. Similarly, any service that is offered should have some defenses with respect to unanticipated load and bad input.

Sizing of microservices can be a tricky thing to get right. I'd say to avoid services that are too small (pico-services) and instead aim to split services across natural boundaries (languages, async queues, scaling requirements) and to keep team sizes reasonable (i.e. 2 pizza teams).

Instead of starting with 20 services start with 2–3 and split services as complexity in that area grows. Oftentimes the architecture of an application isn't well understood until the application is well under development. This also acknowledges that applications are rarely "finished" but rather always a work in progress.

Are microservices a new concept? Not really. This is really another type of software componentization. We've always split code up into libraries. This is just moving the "linker" from being a build time concept to a run time concept. 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 is often implemented and deployed in a way that satisfies the points above.

Constraints can lead to productivity. While it is tempting to allow each team to pick a different language or framework for each microservice, consider instead standardizing on a few languages and frameworks. Doing so will 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.**



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

1. **Service as implementation detail**. As described above, this is useful for breaking down a large application team into smaller teams that stretch from development to operations.

2. **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 many teams are running private instances of a single MySQL binary.

3. **Shared instance.** Here a single team provides a shared service to many applications and teams inside of an organization. The service may partition data and actions per user (multi-tenant) or provide a single simple service that is use

very widely (serving HTML UI for a common branding bar, serving up machine learning models, etc).

4. **Big-S Service.** Most enterprises won't produce a service like this but may consume them. This is the typical "hard" multi-tenant service that is built to service a large number of very disparate customers. This type of service requires a level of accounting and hardening that isn't often necessary inside of an enterprise. Something like SendGrid or Twilio would fall into this category.

As services shift from being an implementation detail to a common infrastructure offered up within an enterprise the **service network** morphs from being 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.

06. Security

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

Container Image Security

There are quite a few tools that help users to audit their container images to ensure that they are fully patched. I don't have a strong opinion on the various options there.

The real problem: what do you do once you find a vulnerable container image? This is a place where the market hasn't provided a great set of solutions. You will want to identify which groups within your organization are impacted, 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 as it will enable automated and quick release processes for the new images. Furthermore, integration with orchestration systems will enable you to identify which

users are using which vulnerable images. It will also allow 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).

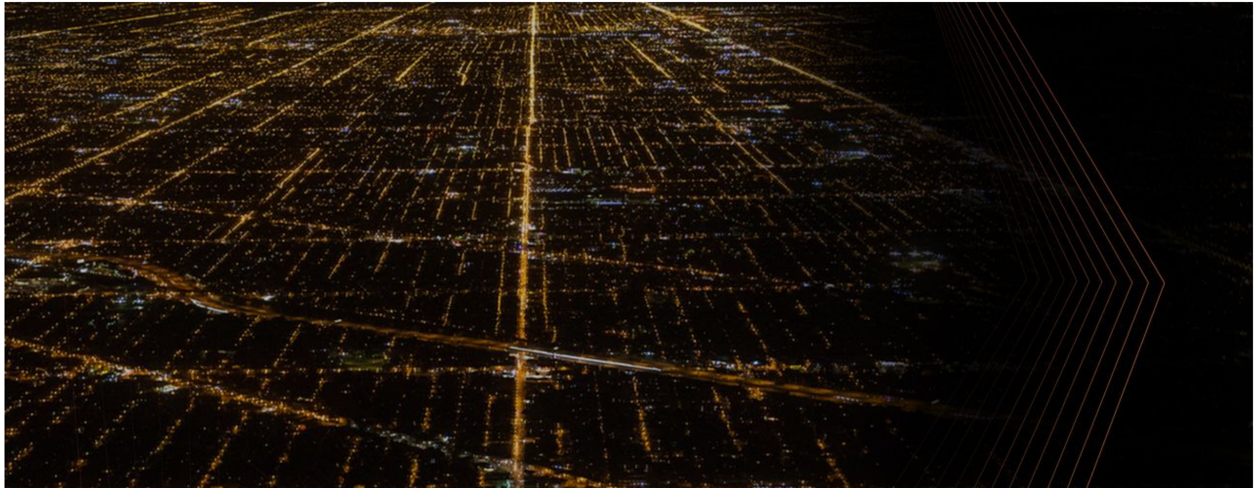
Microservice and Network Security

But even if all of the things you are running on your cluster are patched, it doesn't ensure that there all activity on your network is trusted.

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, **IPs 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 will likely make alerts more easily actionable.

Many of the networking technologies **leverage encapsulation to implement an "IP per container"**. This can create issues for network tracing and inspection tools. They will have to be adapted if such networking systems are deployed in production. Luckily, much of this has standardized on VXLAN, VLANs or no encapsulation/virtualization so support can be leveraged across many such systems.



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

First, the 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 which other containers. I've been calling this approach **network micro-segmentation**. The challenge here is one of configuring such policy in the face of dynamic scheduling. While early yet, there are multiple companies working to make this easier through support in the network, 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 100s 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 datacenter. This works as services take on many clients and become "soft multi-tenant" inside a large organization. This requires a **system of identity for production services**. As a side project, I've started a project called SPIFFE (Secure Production Identity Framework For Everyone). These ideas are

proven inside of companies such as Google but haven't been widely deployed elsewhere.

Security is a deep topic and I'm sure that there are threats and considerations not listed here. This will have to be an ongoing discussion.

There's a start on how to think cloud native. If you're keen to continue the discussion, then please reach out to us via:

Heptio.com | @heptio | @jbeda