FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

KUBERNETES

PAST, PRESENT AND FUTURE

eMag Issue 69 - Mar 2019

InfoQ

ARTICLE
The Kubernetes Effect

ARTICLE
Who's Running My Kubernetes Pod?

VIRTUAL PANEL
Kubernetes and the Challenges of Multi-Cloud

IN THIS ISSUE

FOLLOW US


facebook.com /InfoQ


@InfoQ


linkedin.com company/infoq

youtube.com /MarakanaTechTV

CONTACT US

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

A LETTER FROM THE EDITOR

Daniel Bryant

Over the last four years the Kubernetes container orchestration framework has taken the software engineering world by storm. With backing from the Cloud Native Computing Foundation (CNCF) and a diverse and strong open source community, it is now vying to become the de facto cloud-agnostic compute abstraction -- indeed, all major public cloud vendors now offer some form of Kubernetes-as-a-Service.

This emag explores how Kubernetes is moving from a simple orchestration framework to a fundamental cloud-native API and paradigm that has implications in multiple dimensions, from operations to software architecture. Topics covered include container runtime options, how to design applications that run effectively on Kubernetes, stateful microservice design considerations, and how to run scalable work loads.

We begin the emag with a look at "The Kubernetes Effect" with Bilgin Ibryam, a principal architect at Red Hat. Here he argues that Kubernetes adds a "completely new dimension to programming language-based building blocks" by offering a new set of distributed primitives and runtimes for creating distributed systems that spread across multiple processes and nodes. Ibryam also presents a series of best practices and patterns for building and running containerised applications on Kubernetes.

Jonas Bonér, founder and CTO of Lightbend, provides the second article, "Stateful Service Design Considerations for the Kubernetes Stack", and here he explores why most developers building applications on top of Kubernetes are still mainly relying on stateless protocols and design. He convincingly argues that focusing exclusively on a stateless design ignores the hardest part in distributed systems: managing state— the data.

The third article offers a deep-dive into container runtime choices for Kubernetes, from Phil Estes, a distinguished engineer & CTO, container and Linux OS architecture strategy for the IBM Watson and Cloud Platform division. The Open Container Initiative (OCI) has successfully standardized the concept of a container and container image in order to guarantee interoperability between runtimes, and Kubernetes has added a Container Runtime Interface (CRI) to allow container runtimes to be pluggable underneath the Kubernetes orchestration layer. Estes argues that between the OCI and the CRI, interoperability and choice is becoming a reality in the container runtime and orchestration ecosystem.

The next article, from Joel Speed, a DevOps engineer working at Pusher, returns to the concept of managing stateful workloads. In the article, titled "Six Tips for Running Scalable Workloads on Kubernetes", he provides practical guidance on topics such as setting resource requests for pods; using affinities to spread applications across nodes and availability zones; and adding pod disruption budgets to allow cluster administrators to maintain clusters without breaking applications.

In one of InfoQ's most popular articles from the past six months, "Microservices in a Post-Kubernetes Era", Bilgin Ibryam argues that although the microservice architecture is still the most popular architectural style for distributed systems, Kubernetes and the cloud native movement has redefined certain aspects of application design and development at scale. He continues by stating that "on a cloud native platform, observability of services is not enough", and suggests that a fundamental prerequisite of deploying and managing applications on this platform is to make microservices automatable. He concludes by discussing that microservices must now be designed for "recovery" by implementing idempotency from multiple dimensions, and that modern developers must be fluent in a programming language to implement the business functionality, and equally fluent in cloud native technologies to address the non-functional infrastructure level requirements.

The emag concludes with a virtual panel discussion led by InfoQ editor Rags Srinivas, titled "Kubernetes and the Challenges of Multi-Cloud". In this article we hear from industry luminaries such as Janet Kuo, Sheng Liang, Marco Palladino, and Lew Tucker, and explore that although Kubernetes is already established in multiple clouds, "multi-cloud" means more than that. The discussion also touches on how the Kubernetes community is coming together to address the challenges related to multi-cloud, and how the adoption of Kubernetes may be going some way in providing vendor lock-in and enabling some portability across cloud platform offerings.

Although there is much talk about Kubernetes, in reality the adoption of the framework is still very much in the early stages, and therefore there is much more learning to be done and discussions to be had. InfoQ welcomes constructive feedback on this emag, and we also encourage readers to submit articles and share their experiences with the community.

QCon.ai by InfoQ

April 15-17, 2019
San Francisco

APPLIED AI AND ML CONFERENCE

for software engineers

*rather than data scientists

Why should I attend?

Learn how software innovators are applying AI & machine learning in **use-case oriented sessions**

Hear about the **tools and techniques** of AI & machine learning

Go in-depth on key topics in our **1 day of workshops**

Meet with AI and ML leaders from innovator and early adopter companies

Gain valuable insights and ideas to shape your AI and machine learning projects

Learn how innovator companies are **applying AI & machine learning in their businesses**

Find out more!

CONTRIBUTORS

Daniel Bryant

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

Bilgin Ibryam

(@bibryam) is a principal architect at Red Hat, committer and member of ASF. He is an open-source evangelist, a blogger, and the author of *Camel Design Patterns* and *Kubernetes Patterns*. In his day-to-day job, Ibryam enjoys mentoring, coding, and leading developers to success with building cloud-native solutions. His current work focuses on application integration, distributed systems, messaging, microservices, DevOps, and cloud-native challenges in general. You can find him on Twitter, Linkedin, or his blog.

Phil Estes

is a distinguished engineer and CTO, Container and Linux OS Architecture Strategy for the IBM Watson and Cloud Platform division. He is currently an OSS maintainer in the Docker (now Moby) engine project and the CNCF containerd project, and is a member of both the Open Container Initiative (OCI) Technical Oversight Board and the Moby Technical Steering Committee. Estes is a member of the Docker Captains program and has broad experience in open source and the container ecosystem. He speaks worldwide at industry and developer conferences as well as meetups on topics related to open source, Docker, and Linux container technology. He regularly blogs on these topics and can be found on Twitter as @estesp.

Joel Speed

is a DevOps engineer who has worked with Kubernetes for the last year. He has been working in software development for over three years and is currently helping Pusher build their internal Kubernetes platform. Recently, he has been focusing on projects to improve autoscaling, resilience, authentication, and authorization within Kubernetes as well as building a ChatOps bot, Marvin, for Pusher's engineering team. While studying, he was heavily involved in the Warwick Student Cinema, containerizing their infrastructure as well as regularly projecting films.

Charles Humble

took over as editor-in-chief at InfoQ.com in March 2014, guiding our content creation including news, articles, books, video presentations and interviews. Prior to taking on the full-time role at InfoQ, Charles led our Java coverage, and was CTO for PRPi Consulting, a renumeration research firm that was acquired by PwC in July 2012. He has worked in enterprise software for around 20 years as a developer, architect and development manager.

Raghavan "Rags" Srinivas

(@ragss) works as an Architect/Developer Evangelist at Microsoft goaled with helping developers build highly scalable and available systems. As an OpenStack advocate and solutions architect at Rackspace he was constantly challenged from low level infrastructure to high level application issues. His general focus area is in distributed systems, with a specialization in Cloud Computing and Big Data. He worked on Hadoop, HBase and NoSQL during its early stages. Rags brings with him over 25 years of hands-on software development and over 15 years of architecture and technology evangelism experience.

Read online on InfoQ

KEY TAKEAWAYS

The way we should look at Kubernetes is more like a fundamental paradigm that has implications in multiple dimensions, rather than as an API to interact with.

Kubernetes adds a completely new dimension to language-based building blocks by offering a new set of distributed primitives and runtime for creating distributed systems that spread across multiple processes and nodes.

Container design patterns are focused on structuring the containers and other distributed primitives in the best possible way for solving the challenges at hand. Patterns include sidecar, ambassador, adapter, initializer, work queue, custom controller, and self-awareness.

THE KUBERNETES EFFECT

by **Bilgin Ibryam**

Kubernetes (sometimes called “k8s”) has come a long way in a very short time. Only two years ago, it had to compete and prove to be better than CoreOS’s Fleet, Docker Swarm, Cloud Foundry Diego, HashiCorp’s Nomad, Kontena, Rancher’s Cattle, Apache Mesos, Amazon ECS, etc.

Today, it is a completely different landscape. Some of these projects openly shut down in favor of joining efforts with Kubernetes. Others didn't openly accept defeat but tactically implemented either partial support or a parallel full integration with Kubernetes, which meant a quiet and slow death for their container orchestrator. In any case, Kubernetes is the last platform standing. More and more big names have joined the Kubernetes ecosystem, not only as users or platinum sponsors, but by fully betting their container business on the success of Kubernetes. Google's Kubernetes Engine, Red Hat's OpenShift, Microsoft's Azure Container Service, IBM's Cloud Container Service, Oracle's Container Engine, and others come to mind.

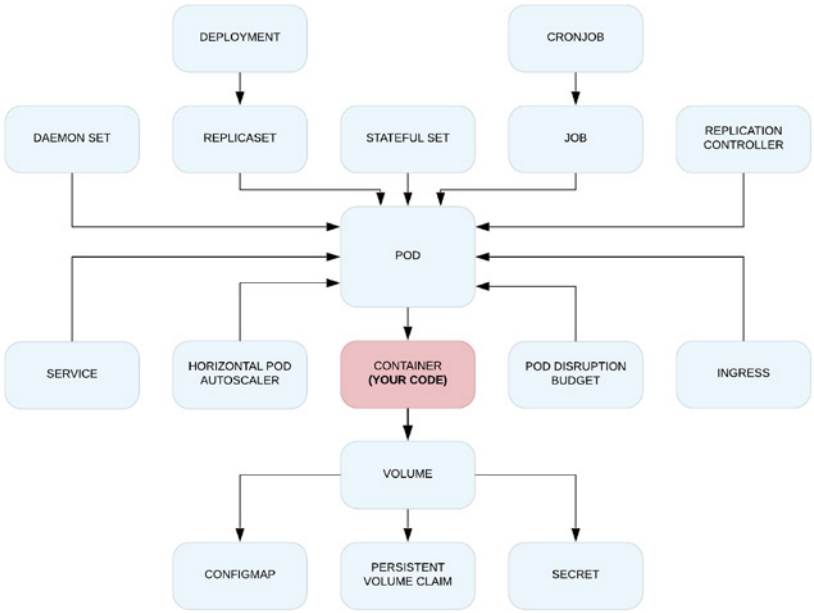


Figure 1: This diagram of a containerized service demonstrates how much it depends on Kubernetes.

This means that developers have to master only a single container orchestration platform to be relevant for 90% of the jobs in the container-related job market. That's a good reason to invest the time to learn Kubernetes well. It also means that we are all knee deep in Kubernetes. Kubernetes is like Amazon, but for containers, if we don't want lock-in, we lock in with Kubernetes. Designing for, implementing, and running applications on Kubernetes gives us the freedom to move our applications between the different cloud providers, Kubernetes distributions, and service providers. It allows us to find certified Kubernetes developers to kick off a project and support personnel to continue running it afterward. It is not the VM, it is not the JVM, it is Kubernetes that is the new application portability layer. It is the common denominator among everything and everybody.

Notice that I didn't write that Kubernetes is the application portability API, but that it is a layer. Figure 1 shows only the Kubernetes objects that we have to explicitly create — which can be called the Kubernetes API. In reality, we are much more coupled with the platform. Kubernetes offers a full set of distributed primitives (such as pods, services, and controllers) that addresses the requirements and drives the design of applications. These new primitives and the platform capabilities dictate the guiding design principles and design patterns we use to implement all future services. They, in turn, affect what techniques we will use to address everyday challenges and even affects what we call a "best practice". Hence the way we should look at Kubernetes is more like a fundamental paradigm that has implications in multiple dimensions, rather than an API to interact with.

The Kubernetes effect

The container and orchestrator features provide a new set of abstractions and primitives. To get the best value out of these new primitives and balance their forces, we need a new set of design principles to guide us. Subsequently, the more we use these new primitives, the more we will end up solving repeating problems and reinventing the wheel.

This is where the patterns come into play. Design patterns provide recipes on how to structure the new primitives to solve repeating problems more quickly. While principles are more abstract, more fundamental, and change less often, the patterns may be affected by a change in the primitive behavior. A new feature in the platform may make the pattern an anti-pattern or less relevant. Then, there are also practices and techniques we use daily. The techniques range from small technical tricks use to perform a task more efficiently to more extensive practices. We change the techniques and practices as soon as we find a slightly better way of doing things easier and faster. This is how we and the platform evolve, where we find the benefits and value of using this new platform and satisfy our needs. In

Figure 2: The effect of Kubernetes on the software development lifecycle.

a sense, the Kubernetes effect is self-enforcing and multifaceted.

Let's look at examples from each of the categories in Figure 2.

Distributed abstractions and primitives

In order to explain what I mean by new abstractions and primitives, I will compare them to the well-known object-oriented programming (OOP) and Java specifically. In the OOP universe, we have concepts such as class, object, package, inheritance, encapsulation, polymorphism, etc. The Java runtime provides certain features and guarantees for how it manages the lifecycle of our objects and the application as a whole. The Java language and the JVM runtime provide local, in-process building blocks for creating applications.

Kubernetes adds a completely new dimension to this well-known mindset by offering a new set of distributed primitives and runtime for creating distributed systems that spread across multiple processes and nodes. With Kubernetes at hand, I don't rely only on the Java primitives to implement the entire application behavior. I still need to use the object-oriented building blocks to create the components of the distributed application, but I can also use Kubernetes primitives for some of the application behavior. A few examples of distributed abstractions and primitives from Kubernetes are:

- Pod — the deployment unit for a related collection of containers.
- Service — service discovery and load-balancing primitive.
- Job — an atomic unit of work scheduled asynchronously.
- CronJob — an atomic unit of work scheduled at a specific time in the future or periodically.
- ConfigMap — a mechanism for distributing configuration data across service instances.
- Secret — a mechanism for management of sensitive configuration data.
- Deployment — a declarative application release mechanism.
- Namespace — a control unit for isolating resource pools.

For example, I can rely on Kubernetes health checks (e.g., readiness and liveness probes) for some of my application reliability. I can use Kubernetes Services for service discovery rather than doing client-side service discovery from within the application. I can use Kubernetes Jobs to schedule an asynchronous atomic unit of work. I can use ConfigMap for configuration management and Kubernetes CronJob for periodic task scheduling, rather than using the Java-based Quartz library or an implementation of the ExecutorService interface.

The in-process primitives and the distributed primitives have commonalities, but they are not directly comparable or interchangeable. They operate at different abstraction levels and have different preconditions and guarantees. Some primitives are supposed to be used together. For example, we still have to use classes to create objects and put them into container images. But some other primitives such as

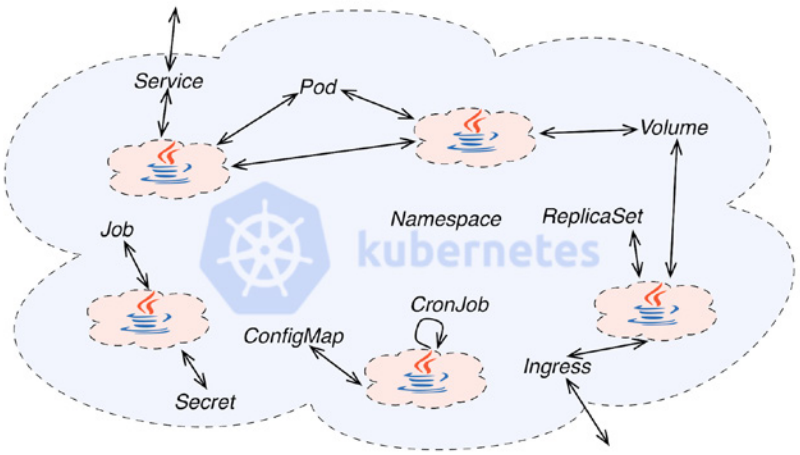


Figure 3: Local and distributed primitives as a part of a distributed system.

Concern	Java & JVM	Kubernetes
Behaviour encapsulation	Class	Container Image
Behaviour instance	Object	Container
Unit of reuse	.jar	Container Image
Deployment unit	.jar/.war/.ear	Pod
Buildtime/Runtime isolation	Module, Package, Class	Container Image, Namespace
Initialization preconditions	Constructor	Init-container
Post initialization	init-method	PostStart
Pre destroy	destroy-method	PreStop
Cleanup procedure	finalize(), ShutdownHook	Defer-container*
Asynchronous & Parallel execution	ThreadPoolExecutor, ForkJoinPool	Job
Periodic task	Timer, ScheduledExecutorService	CronJob
Background task	Daemon Thread	DaemonSet
Configuration management	System.getenv(), Properties	ConfigMap, Secret

Figure 4: Local and distributed primitives categorized.

CronJob in Kubernetes can completely replace the ExecutorService behavior in Java. Here are few concepts that share commonalities between the JVM and Kubernetes, but don't take it too far.

I've blogged about distributed abstractions and primitives in the past. The point here is that a developer can use a richer set of local and global primitives to design and implement a distributed solution. With time, these new primitives give birth to new ways of solving problems, and some of these repetitive solutions become patterns. This is what we will explore further down.

Container design principles

Design principles are fundamental rules and abstract guidelines for writing quality software. The principles do not specify concrete rules, but they represent a language and common wisdom that many developers understand and refer to regularly. Similar to the SOLID principles that Robert C. Martin introduced, which represent guidelines for writing better object-oriented software, there are also design principles for creating better containerized applications. The SOLID principles use object-oriented primitives and concepts such as classes, interfaces, and inheritance for reasoning about object-oriented designs. In a similar way, the principles for creating containerized applications listed below use the container image as the basic primitive and the container orchestration platform as the target container runtime environment. Principles for container-based application design are, for building:

- Single concern — every container should address a single concern and do it well.
- Self-containment — a container should rely only on the presence of the Linux kernel and have any additional libraries added to it at the time the container is built.
- Image immutability — containerized applications are meant to be immutable, and once built are not expected to change between different environments.

And for runtime:

- High observability — every container must implement all necessary APIs to help the platform observe and manage the application in the best way possible.

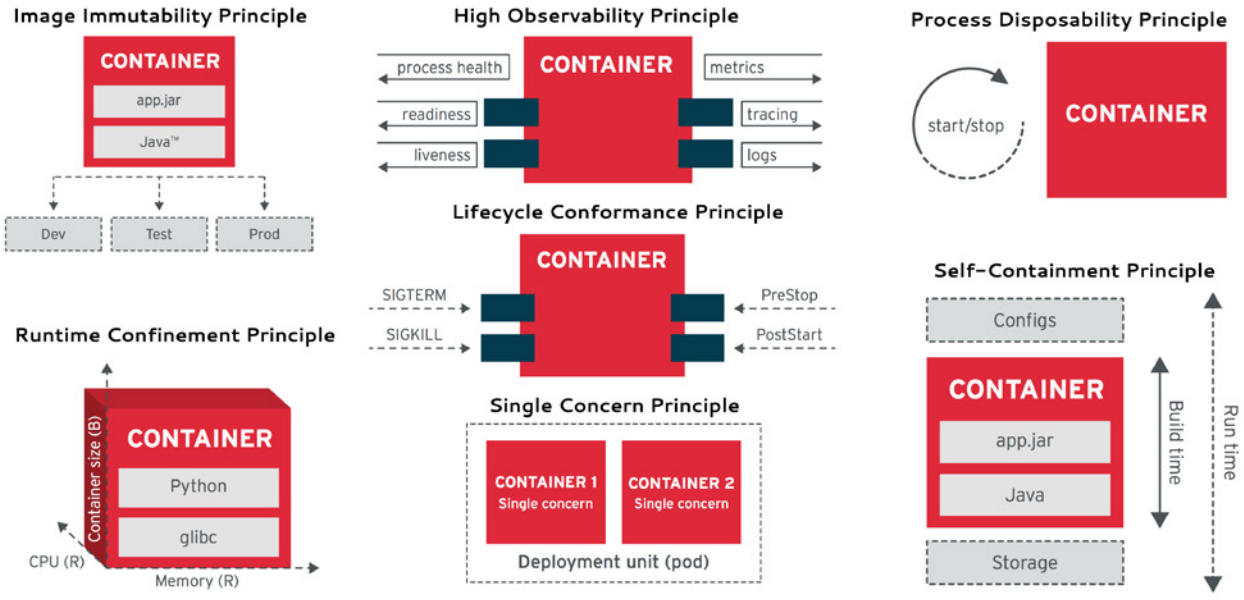


Figure 5: Container design principles.

- Lifecycle conformance — a container should have a way to read the events coming from the platform and conform by reacting to those events.
- Process disposability — containerized applications need to be as ephemeral as possible and ready to be replaced by another container instance at any point in time.
- Runtime confinement — every container should declare its resource requirements and it is also important that the application stay confined to the indicated resource requirements.

Following these principles, we are more likely to create containerized applications that are better suited for cloud-native platforms such as Kubernetes.

These principles are well documented and freely available for download as a white paper from Red Hat.

Container design patterns

New primitives need new principles that explain the forces between the primitives. The more we use the primitives, the more we end up solving repeating problems, which leads us to recognize repeating solutions called patterns. Container design patterns are focused on structuring the containers and other distributed primitives in the best possible way for solving the challenges at hand. A short list of container-related design patterns is:

- Sidecar pattern — a sidecar container extends and enhances the functionality of a pre-existing container without changing it.
- Ambassador pattern — this pattern hides complexity and provides a unified view of the world to your container.
- Adapter pattern — an adapter is kind of reverse ambassador and provides a unified interface to a pod from the outside world.
- Initializer pattern — init containers allow separation of initialization-related tasks from the main application logic.
- Work-queue pattern — a generic work-queue pattern based on containers allows taking arbitrary processing code packaged as a container and arbitrary data to build a complete work-queue system.
- Custom-controller pattern — a controller watches for changes to objects and acts on those changes to drive the cluster to a desired state. This reconciliation pattern can be used to implement custom logic and extend the functionality of the platform.
- Self-awareness pattern — describes occasions where an application needs to introspect and get metadata about itself and the environment where it is running.

Brendan Burns and David Oppenheimer laid out the foundational work in this area in their "Design patterns for container-based distributed systems" paper. Since then, Burns has written a book that also covers design patterns and related topics on distributed systems. Roland Huß and I are writing a book titled Kubernetes Patterns that will cover these design patterns and use cases for container-based applications. Figure 6 shows a few of these patterns.

Primitives need principles and makeup patterns. Let's look at some of the best practices and overall benefits of using Kubernetes.

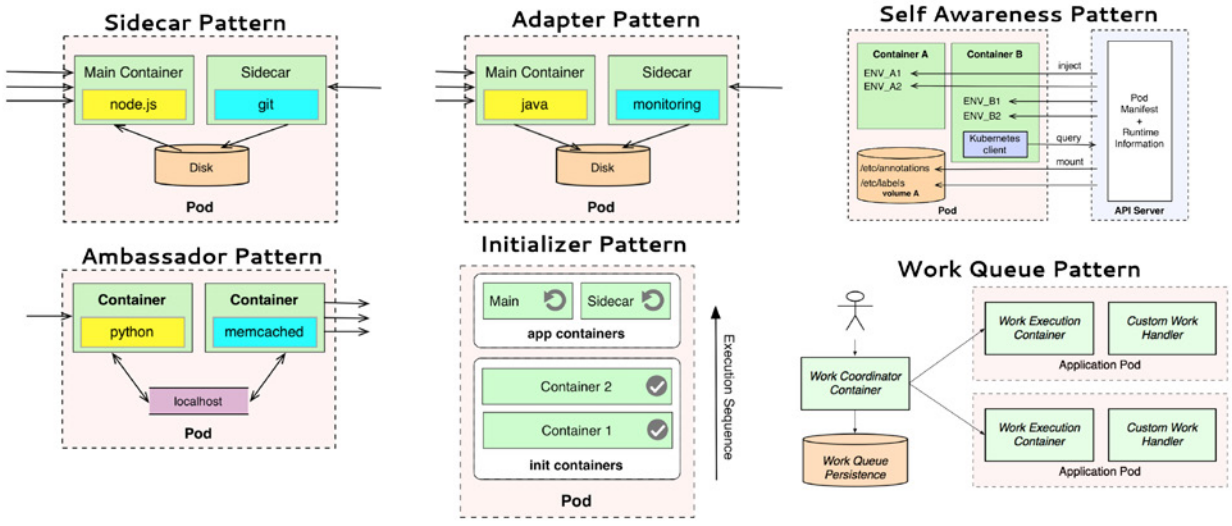


Figure 6: Container design patterns.

Practices and techniques

In addition to the principles and patterns, creating good containerized applications requires familiarity with other container-related best practices and techniques. Principles and patterns are abstract, fundamental ideas that change less often. Best practices and the related techniques are more concrete and may change more frequently. Here are some of the common container-related best practices:

- Aim for small images — this reduces container size, build time, and networking time when copying container images.
- Support arbitrary user IDs — avoid using the `sudo` command or requiring a specific `userid` to run your container.
- Mark important ports — specifying ports using the `EXPOSE` command makes it easier for both humans and software to use your image.
- Use volumes for persistent data — the data that needs to be preserved after a container is destroyed must be written to a volume.
- Set image metadata — image metadata in the form of tags, labels, and annotations makes your container images more discoverable.
- Synchronize host and image — some containerized applications require the container to be synchronized with the host on certain attributes such as time and machine ID.
- Log to STDOUT and STDERR — logging to these system streams rather than to a file will ensure container logs are picked up and properly aggregated.

Here are few links to resources with container-related best practices:

- Best practices for writing Dockerfiles
- OpenShift Guidelines
- Kubernetes Production Patterns
- Kubernetes By Example

Kubernetes benefits

Kubernetes dictates the design, development, and day-two operations of distributed systems in a fundamental way. The learning curve is not short — crossing the Kubernetes chasm takes time and patience. That's why I'd like finish with a list of benefits that Kubernetes brings to developers. Hopefully, this will help prove the worth of jumping into Kubernetes and using it to steer your IT strategy:

- Self-service environments — enables teams and team members to instantly carve isolated environments from the cluster for CI/CD and experimentation purposes.
- Dynamically placed applications — allows applications to be placed on the cluster in a predictable manner based on application demands, available resources, and guiding policies.
- Declarative service deployments — encapsulates the upgrade and rollback process of a group of containers and makes executing it a repeatable and automatable activity.
    - infinite loops — CPU shares and quotas;
    - memory leaks — OOM yourself;
    - disk hogs — quotas;
    - fork bombs — process limits;
    - circuit breaker, timeout, retry as sidecar;
    - failover and service discovery as sidecar;
    - process bulkheading with containers;
    - hardware bulkheading through the scheduler; and
    - autoscaling and self-healing.
- Service discovery, load balancing, and circuit breaker — the platform allows services to discover and consume other services without in-application agents. Further, the usage of sidecar containers and tools such as Istio framework allow us to move the networking responsibilities completely outside of the application to the platform level.
- Declarative application topology — using Kubernetes API objects allow us to describe how our services should be deployed, their dependency on other services, and resource prerequisites. And having all this information in an executable format allows us to test the deployment aspects of the application in the early stage of development and treat it as programmable application infrastructure.

There are many more reasons why the IT community is getting so excited about Kubernetes. The ones I listed above are the ones that I, as somebody with a developer background, find really useful.

Hopefully, I've managed to describe how Kubernetes is affecting the daily life of developers. If you want to read more about Kubernetes from developer's point

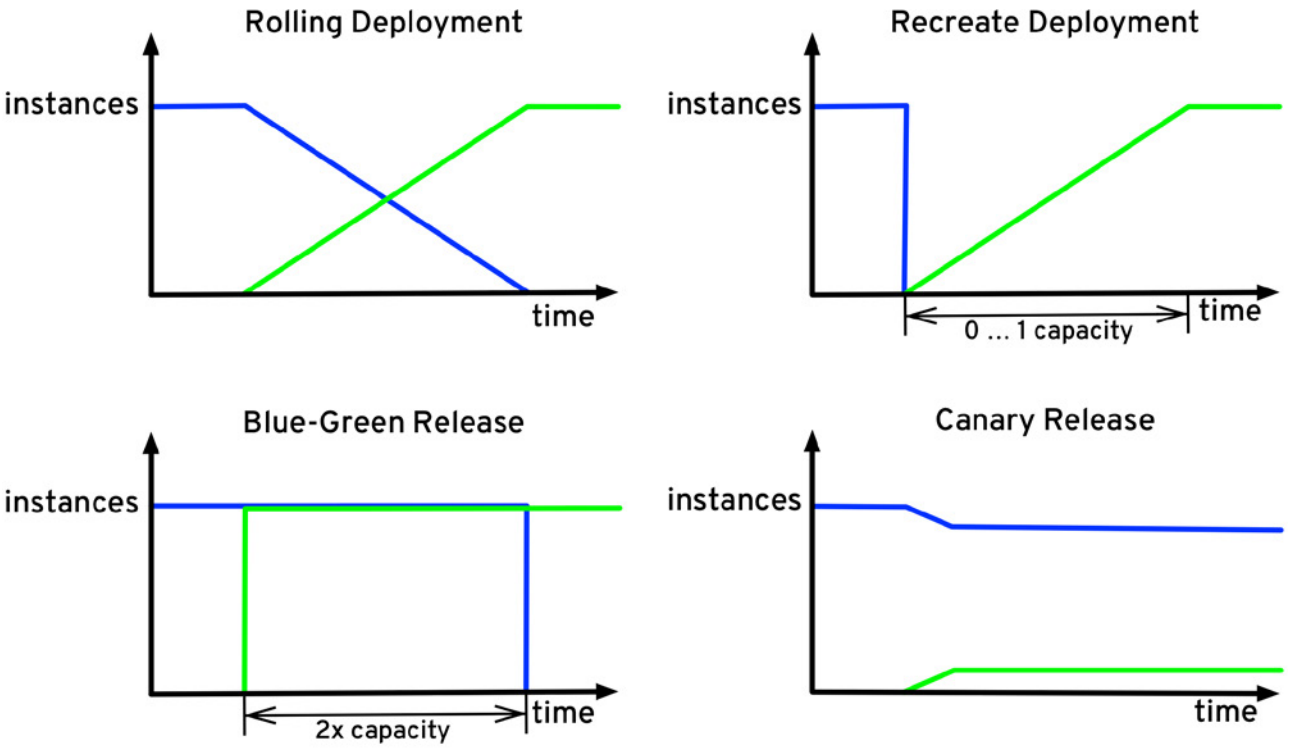


Figure 7: Examples of deployment and release strategies with Kubernetes.

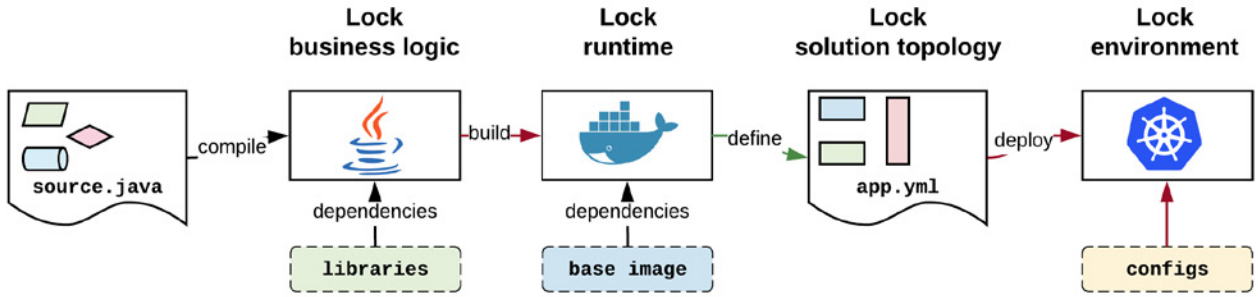


Figure 8: Declarative application topology using Kubernetes resource descriptor files.

of view, check out my book, follow me on Twitter at @bibryam, or take a look at these

resources that focus on Kubernetes from a developer point of view:

- Design patterns for container-based distributed systems (white paper),
- Principles of container-based application design (white paper),
- Designing Distributed Systems (ebook),
- Kubernetes Patterns (ebook), and
- Kubernetes in Action (ebook).

Read online on InfoQ

KEY TAKEAWAYS

Most developers who build applications on top of Kubernetes still mainly rely on stateless protocols and design. The problem is that focusing exclusively on a stateless design ignores the hardest part in distributed systems: managing state — your data.

The challenge is not designing and implementing the services themselves, but managing the space in between the services: data consistency guarantees, reliable communication, data replication and failover, component failure detection and recovery, sharding, routing, consensus algorithms, and so on.

Akka has been growing rapidly in the last two years. Today, it has around 5 million downloads a month, compared with 500,000 downloads a month two years ago.

STATEFUL-SERVICE DESIGN CONSIDERATIONS FOR THE KUBERNETES STACK

by **Charles Humble**

At 2018's QCon in New York, Jonas Bonér delivered one of the most popular talks of the conference with his focus on "Designing Events-First Microservices". We asked Bonér to explain how "bringing bad habits from monolithic design" is a road to nowhere for service design, and where he sees his Akka framework fitting in the cloud-native stack.

THE PANELIST

Jonas Bonér
is the founder and CTO of Lightbend, inventor of the Akka project, initiator and co-author of the Reactive Manifesto, and a Java Champion.

InfoQ: At QCon you said, "When you start with a microservices journey you should take care not to end up with microliths because you may bring bad habits from monolithic design to microservices, which creates a strong coupling between services." Can you explain?

Jonas Bonér: In the cloud-native world of application development, I'm still seeing a strong reliance on stateless, and often synchronous, protocols.

Most of the developers building applications on top of Kubernetes are still mainly relying on stateless protocols and design. They embrace containers and too often hold on to old architecture, design, habits, patterns, practices, and tools — made for a world of monolithic single-node systems running on top of the almighty RDBMS.

The problem is that focusing exclusively on a stateless design ignores the hardest part in distributed systems: managing state — your data.

It might sound like a good idea to ignore the hardest part and push its responsibility out of the application layer — and sometimes it is. However, as applications today are becoming increasingly data-centric and data-driven, taking ownership of your data by having an efficient, performant, and reliable way of managing, processing, transforming, and enriching data close to the application itself, is more important than ever.

Many applications can't afford the roundtrip to the database for each data access or storage and need to continuously process data in close to real-time, mining knowledge from never-ending streams of data. This data also often needs to be processed in a distributed way — for scalability, low-latency, and throughput — before it is ready to be stored.

InfoQ: Stateful services have long been cited as the toughest obstacle for mainstream container adoption. Tell us a little bit more about why this is such a tricky area.

Bonér: The strategy of treating containers as logically identical units that can be replaced, spun up, and moved around without much thought works really well for stateless services but is the opposite of how you want to manage distributed stateful services and databases. First, stateful instances are not trivially replaceable since each one has its own state that needs to be taken into account. Second, deployment of stateful replicas often requires coordination among replicas — things like bootstrap dependency order, version upgrades, schema changes, and more. Third, replication takes time, and the machines from which the replication is done will be under a heavier load than usual, so if you spin up a new replica under load, you may actually bring down the entire database or service.

One way around this problem — which has its own problems — is to delegate the state management to a cloud service or database outside of your Kubernetes cluster. That said, if we want to manage all of our infrastructure in a uniform fashion using Kubernetes then what do we do?

At this time, the Kubernetes answer to the problem of running stateful services that are not cloud-native is the concept of a StatefulSet, which ensures that each pod is given a stable identity and dedicated disk that is maintained across restarts (even after it's been rescheduled to another physical machine). As a result, it is now possible, albeit still quite challenging, to deploy distributed databases, streaming data pipelines, and other stateful services on Kubernetes.

What is needed is a new generation of tools that allow developers to build truly cloud-native stateful services that only have the infrastructure requirements of what Kubernetes gives to stateless services. This is not an argument against the use of low-level infrastructure tools like Kubernetes and Istio — they clearly bring a ton of value — but a call for closer collaboration between the infrastructure and application layers in maintaining holistic correctness and safety guarantees.

InfoQ: Where does Akka fit into this set of stateful service requirements for the Kubernetes stack?

Bonér: Really, the hard part is not designing and implementing the services themselves, but in managing the space in between

the services. Here is where all the hard things enter the picture: data consistency guarantees, reliable communication, data replication and failover, component failure detection and recovery, sharding, routing, consensus algorithms, and much more. Stitching all that together, and maintaining it over time, yourself is very, very hard.

End-to-end correctness, consistency, and safety mean different things for different services. It's completely dependent on the use case, and can't be outsourced completely to the infrastructure. What we need is a programming model for the cloud, paired with a runtime that can do the heavy lifting that allows us to focus on building business value instead of messing around with the intricacies of network programming and failure modes — I believe that Akka paired with Kubernetes can be that solution.

Akka is an open-source project created in 2009, designed to be a fabric and programming model for distributed systems, for the cloud. Akka is cloud-native in the truest sense; it was built to run natively in the cloud before the term "cloud-native" was even coined.

Akka is based on the actor model and built on the principles outlined in the Reactive Manifesto, which defines a "reactive system" as a set of architectural design principles that are geared toward meeting the demands that systems face, today and tomorrow.

In Akka, the unit of work and state is called an actor and can be seen as a stateful, fault-tolerant, isolated, and autonomous component or entity. These actors/entities are extremely lightweight in terms of resources — you can easily run millions of them concurrently on a single machine — and communicate using asynchronous messaging. They have built-in mechanisms for automatic self-healing and are distributable and location transparent by default. This means that they can be scaled, replicated, and moved around in the cluster on demand — reacting to how the application is being used — in a way that is transparent to the user of the actor/entity.

InfoQ: So where is the separation of concerns between Kubernetes and Akka when they are used together?

Bonér: One way to look at it is that Kubernetes is great in managing and orchestrating "boxes" of software (the containers) but managing these boxes only gets you halfway there. Equally important is what you put inside the boxes, and this is what Akka can help with.

Kubernetes and Akka compose very well, each being responsible for a different layer and function in the application stack. Akka is the programming model in which to write the application and its supporting runtime — it helps to manage business logic, data consistency and integrity, operational semantics, distributed and local workflow and communication, integration with other systems, etc. Kubernetes is the tool that operations can use to manage large numbers of container instances in a uniform fashion — it helps manage container lifecycle, versioning and grouping of containers, routing communication between containers, security, authentication, and authorization between containers, etc.

In essence, Kubernetes's job is to give your application enough compute resources, get external traffic to a node in your application, and manage things like access control while Akka's job is deciding how to distribute that work across all the computing resources that has been given to it.

InfoQ: Akka embraces a "let it crash" approach — i.e., losing one actor shouldn't matter because another will pick up the work. Can you explain how this works in a container environment? Does an admin need to intervene?

Bonér: Traditional thread-based programming models only give a single thread of control, so if this thread crashes with an exception, you are in trouble. This means that you need to make all error handling explicit within this single thread. Exceptions do not propagate between threads, or across the network, so there is no way of even finding out that something has failed. But losing the thread or, in the worst case, the whole container is very expensive. To make things worse, the use of synchronous protocols can cause these failures to cascade across the whole application. We can do better than this.

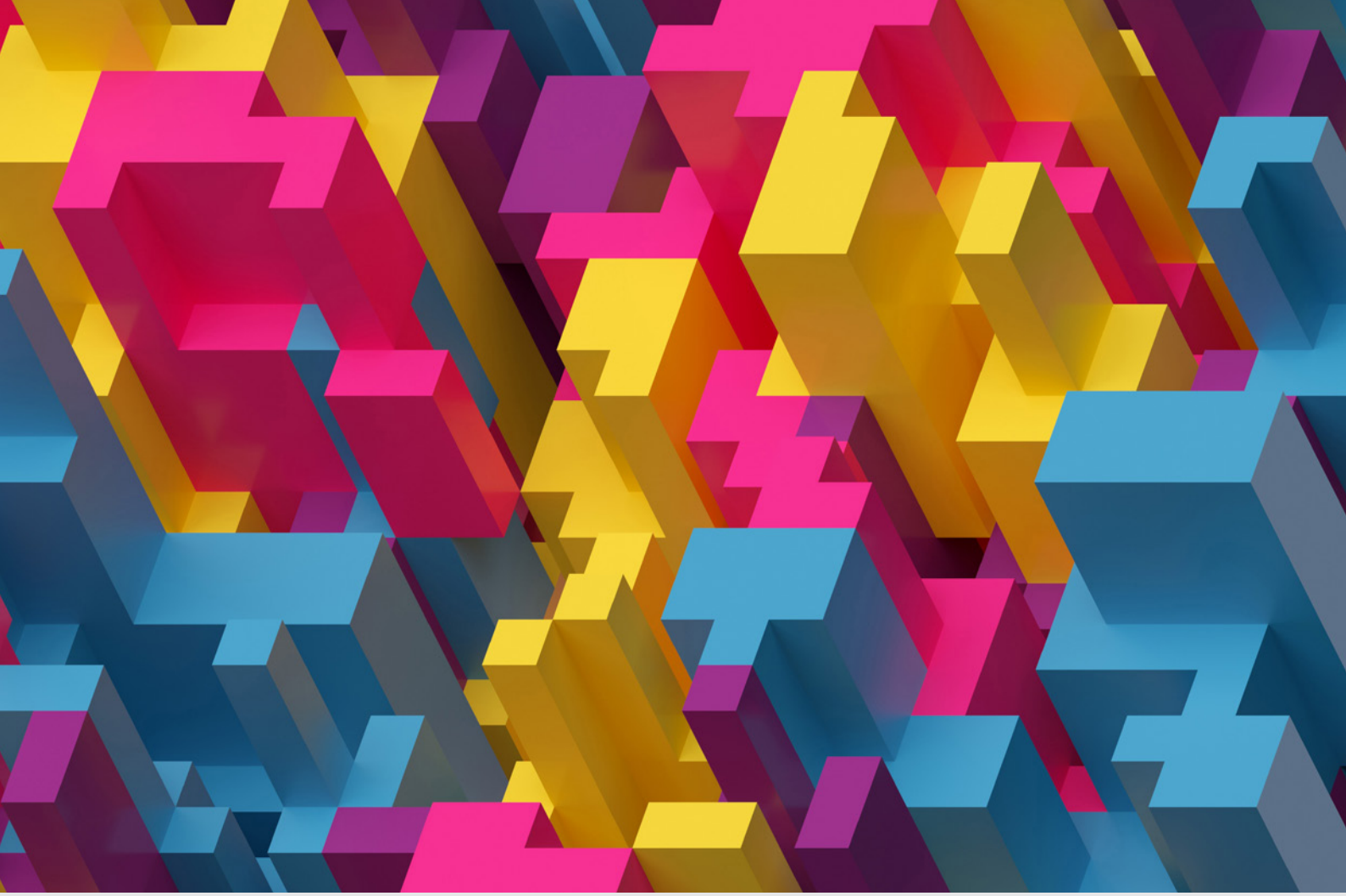
In Akka, you design your application in so-called "supervisor hierarchies" where the actors are watching out for each other's health and manage each other's failures. If an actor fails, its error is isolated and contained, reified as a message that is sent asynchronously — across the network if needed — to its supervising actor, which can handle the failure in a safe, healthy context and restart the failed actor automatically according to declaratively defined rules. This naturally yields a non-defensive way of programming and a fast fail (and recover) approach that is also called "let it crash".

This might sound like it is overlapping with the roles of Kubernetes, and it's true that both Kubernetes and Akka help to manage resilience and scalability, but they do so at distinct granularity levels in the application stack. You can also look at the two technologies in terms of fine-grained versus coarse-grained resilience and scalability.

Kubernetes allows for coarse-grained container-level management of resilience and scalability, where the container is replicated, restarted, or scaled out/in as a whole. Akka allows for fine-grained entity-level management of resilience and scalability — working closely with the application — where each service in itself is a cluster of entity replicas that are replicated, restarted, and scaled up and down as needed, automatically managed by the Akka runtime, without the operator or Kubernetes having to intervene.

InfoQ: Could you give us an idea of Akka adoption? How many monthly downloads do you currently see?

Bonér: Akka has been growing rapidly in the last two years. Today, it has around 5 million downloads a month, compared with 500,000 downloads a month two years ago. If you're interested in some of the earliest history and milestones around the project, this infographic has some cool data points.



KEY TAKEAWAYS

Container runtime choices have grown to include other options beyond the popular Docker engine.The Open Container Initiative (OCI) has successfully standardized the concepts of a container and container image in order to guarantee interoperability between runtimes.

Kubernetes has added a Container Runtime Interface (CRI) to allow container runtimes to be pluggable underneath the Kubernetes orchestration layer.

Innovation is allowing containers to use lightweight virtualization and other unique isolation techniques for increased security requirements.

WHO'S RUNNING MY KUBERNETES POD? THE PAST, PRESENT, AND FUTURE OF CONTAINER RUNTIMES

by **Phil Estes**

In the Linux operating-system world, container technology has existed for quite some time, reaching back over a decade to the initial ideas around separate namespaces for file systems and processes. At some point in the more recent past, LXC was born and became the common way for Linux users to access this powerful isolation technology hidden within the Linux kernel.

Even with LXC masking some of the complexity of assembling the various technology underpinnings of what we now commonly call a "container", containers still seemed like a bit of wizardry and, other than niche uses for those versed in this art of containers, were not broadly used.

Docker changed all this in 2014 with developer-friendly packaging of the same Linux kernel technology that powered LXC — in fact, early versions of Docker used LXC behind the scenes — and containers truly came to the masses as developers were drawn to the simplicity and re-use of Docker's container images and simple runtime commands.

Of course, Docker wasn't alone in wanting to realize market share for containers given that the hype cycle showed no signs of slowing down after the initial explosion of interest in 2014. Over the past few years, various alternatives for container runtimes appeared, such as CoreOS (rkt), lightweight virtualization married to containers from Intel Clear Containers, and hyper.sh, and high-performance computing for scientific research from Singularity and shifter.

As the market continued to mature, attempts to standardize the initial ideas that Docker promoted via the Open Container Initiative (OCI) took hold. Today, many container runtimes are either OCI compliant or on the path to OCI compliance, providing a level playing field for vendors to differentiate by their features or specifically focused capabilities.

Kubernetes Popularity

The next step in the evolution of containers was to bring distributed computing, à la microservices, and containers together in this new world of rapid development and deployment iteration — DevOps, we might say — that quickly arose alongside Docker's growing popularity.

While Apache Mesos and other distributed software-orchestration platforms existed prior to its ascendancy, Kubernetes had a similar meteoric rise, from a small open-source project at Google to the flagship project of the Cloud Native Computing Foundation (CNCF). Even Docker's competing orchestration platform (Swarm) built into Docker itself with Docker's style of simplicity and a focus on secure-by-default cluster configuration didn't seem to be enough to stem the growing tide of interest in Kubernetes.

It was unclear to many interested parties outside the bubble of cloud-native communities whether it was Kubernetes versus Docker or Kubernetes and Docker. Since Kubernetes was simply the orchestration platform, it required a container runtime to do the work of managing the actual running containers it orchestrated. From day one, Kubernetes had been using the Docker engine, and even with the tensions of Swarm as an orchestration competitor, Kubernetes still kept Docker the default runtime required by an operational Kubernetes cluster.

With a short list of container runtimes available beyond Docker, it seemed clear that interfacing a container runtime to Kubernetes would require a specially written interface, or shim, for each runtime. Due to the lack of a clear interface to implement for container runtimes, adding more runtime options to Kubernetes was getting messy.

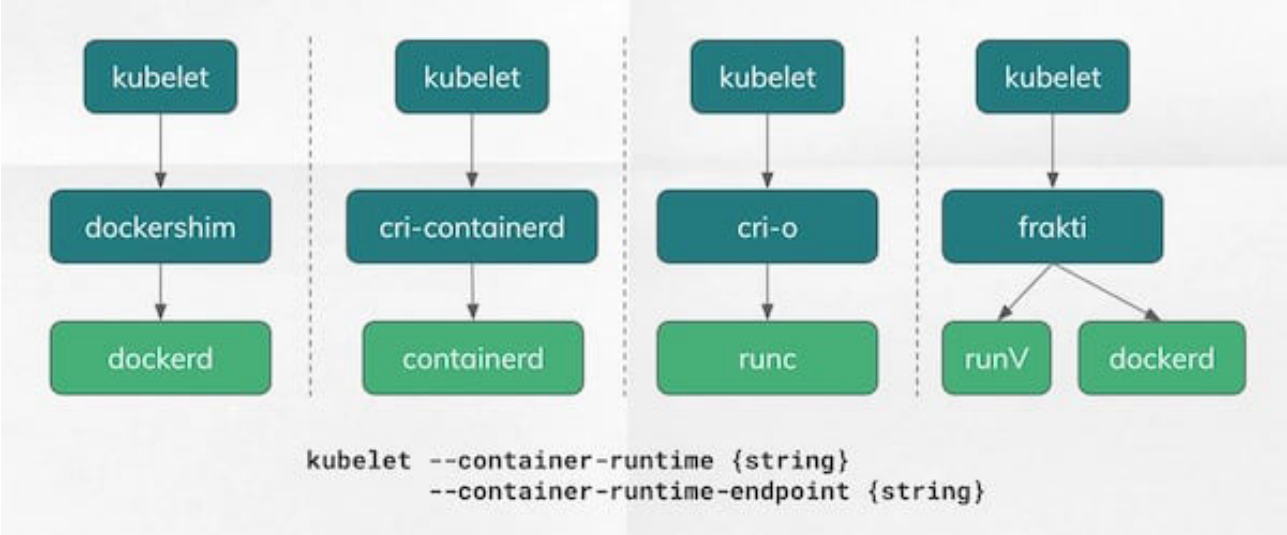
The Container Runtime Interface (CRI)

To solve the growing challenge of incorporating runtime choice into Kubernetes, the community defined an interface — specific functions that a container runtime would have to implement on behalf of Kubernetes — called the Container Runtime Interface (CRI). This both corrected the problem of having a sprawling list of places within the Kubernetes codebase in which container runtimes changes would have to be applied and clarified to potential runtimes exactly what functions they would have to support to be a CRI runtime.

As you might expect, the CRI's expectations for a runtime are fairly straightforward. The runtime must be able to start/stop pods, handle all container operations within pods (start, stop, pause, kill, delete), as well as handle the image management with a container registry. Utility and helper functions around gathering logs, metrics collection, and so on also exist.

If a new feature enters Kubernetes and that feature has a dependency on the container runtime layer, then changes are made to the versioned CRI API, and new versions of runtimes that support the new feature (user namespaces, for one recent example) will have to be released to handle the new functional dependency from Kubernetes.

As of 2018, several options exist for container runtimes underneath Kubernetes. As the image below depicts, Docker is still a viable choice for Kubernetes, with its shim now implementing the CRI API. In fact, in most cases today, Docker is still the default runtime for Kubernetes installations.



One of the interesting outcomes of the tensions between Docker's orchestration strategy with Swarm and the Kubernetes community was that a joint project was formed, taking the core of Docker's runtime and breaking it out as a new jointly developed open-source project named containerd. Containerd was then also contributed to the CNCF, the same foundation that manages and owns the Kubernetes project.

With containerd as a core, unopinionated pure runtime underneath both Docker and Kubernetes via the CRI, it has gained popularity as a potential replacement for Docker in many Kubernetes installations. Today, both IBM Cloud and Google Cloud offer containerd-based clusters in a beta/early-access mode. Microsoft Azure has also committed to moving to containerd in the future, and Amazon is still reviewing options for runtime choice underneath their ECS and EKS container offerings, continuing to use Docker for the time being.

Red Hat also entered the container runtime space by creating a pure implementation of the CRI, called CRI-O, around the OCI reference implementation, runc. Docker and containerd are also based around the runc implementation, but CRI-O states that it is "just enough" of a runtime for Kubernetes and nothing more, adding just the functions necessary above the base runc binary to implement the Kubernetes CRI.

The lightweight virtualization projects Intel Clear Containers and hyper.sh merged underneath an OpenStack Foundation project, Kata Containers, and are also providing their style of virtualized container for extra isolation via the frakti CRI implementation. Both CRI-O and containerd are also working with Kata Containers so that their OCI-compliant runtime can be a pluggable runtime option within those CRI implementations as well.

Predicting the Future

While it is rarely wise to claim to know the future, we can at least observe some emerging trends as the container ecosystem moves from mass levels of excitement and hype into a more mature phase of existence.

There were early concerns that disputes across the container ecosystem would create a fractured environment in which various parties would end up with different and incompatible ideas of what a container is. Thanks to the work of the OCI and commitment by key vendors and participants, we see healthy investment across the industry in software offerings prioritizing OCI compliance.

In newer spaces where standard use of Docker had gained less traction due to unique constraints — e.g., in high-performance computing — even those non-Docker-based attempts at a viable container runtime are now taking notice of the OCI. Discussions are underway and there is hope that OCI can be viable for the science and research communities' specific needs as well.

Adding in the standardization on pluggable containers runtimes in Kubernetes via the CRI, we can envision a world where developers and operators can pick the tools and software stacks that work for them and still expect and experience interoperability across the container ecosystem.

The following example demonstrates this:

1. A developer on a MacBook uses Docker for Mac to develop her application, and uses

the built-in Kubernetes support (using Docker as the CRI runtime) to try out deploying her new app within Kubernetes pods.

2. The application goes through CI/CD on a vendor product that uses runc and some vendor-written code to package an OCI image and push it to the enterprise's container registry for testing.

3. An on-premises Kubernetes cluster using containerd as the CRI runtime runs a suite of tests against the application.

4. This particular enterprise has chosen to use Kata Containers for certain workloads in production, and when the application is deployed, it runs in pods with containerd, configured to use Kata Containers as the base runtime instead of runc.

This scenario works flawlessly because of compliance to the OCI specification for runtimes and images, and because the CRI allows for this flexibility on runtime choice.

This flexibility and choice in the container ecosystem is a great advantage and is truly important for the maturity of the industry that has risen only since 2014. The future is bright for continued innovation and flexibility for those using and creating container-based platforms as we enter 2019 and beyond!

Additional information can be found from my recent QCon NY talk. "CRI Runtimes Deep Dive: Who's Running My Kubernetes Pod!?"

Read online on InfoQ

KEY TAKEAWAYS

Kubernetes offers many tools to help make applications scalable and fault tolerant.

Setting resource requests for pods is important.

Use affinities to spread your apps across nodes and availability zones.

Add pod disruption budgets to allow cluster administrators to maintain clusters without breaking your apps.

Autoscaling of pods in Kubernetes means apps should always be available as demand increases.

SIX TIPS FOR RUNNING SCALABLE WORKLOADS ON KUBERNETES

by **Joel Speed**

As an infrastructure engineer at Pusher, I work with Kubernetes on a daily basis. And while I've not been actively developing applications to run on Kubernetes, I have been configuring deployments for many applications. I've learned what Kubernetes expects for workloads and how to make them as tolerant as possible.

This article is all about ensuring Kubernetes knows what is happening with your deployment: where best to schedule it, knowing when it is ready to serve requests, and ensuring work is spread across as many nodes as possible. I will also discuss pod disruption budgets and horizontal pod autoscalers, which I've found get overlooked all too often.

I work on the Infrastructure team at Pusher. I like to describe what my team does as "providing a PaaS for the SaaS". My team of three primarily works on building a platform using Kubernetes for our developers to build, test, and host new products.

In a recent project, I cut the costs of our EC2 instances by moving our Kubernetes workers onto spot instances. Spot instances are the same as other AWS EC2 instances, but you pay a lower price for them with the proviso that you could get a two-minute warning that you are losing the instance.

One of the requirements for the project was to ensure that our Kubernetes clusters were tolerant to losing nodes at such short notice. This also then extended to the monitoring, alerting, and cluster components (kube-dns, metrics) managed by my team. The result of the project was a cluster (with cluster services) that tolerated node losses, but unfortunately this behavior didn't extend to our product workloads.

Shortly after moving our clusters onto spot instances, an engineer came to me to try to minimize any potential downtime for his application:

"Is there any way to stop my pod scheduling on the spot instances?"

To me, this is the wrong approach. Instead of trying to avoid node failures, the engineer should be taking advantage of Kubernetes and the tools it offers to help ensure applications are scalable and fault tolerant.

Kubernetes has many features to help application developers ensure that their applications are deployed in a highly available, scalable, and fault tolerant way. When deploying a horizontally scalable application to Kubernetes, you'll want to ensure you have configured the following:

- resource requests and limits,
- node and pod affinities and anti-affinities,
- health checks,
- deployment strategies,
- pod disruption budgets, and
- horizontal pod autoscalers.

The following sections will describe the part that each of these concepts plays in making Kubernetes workloads both scalable and fault tolerant.

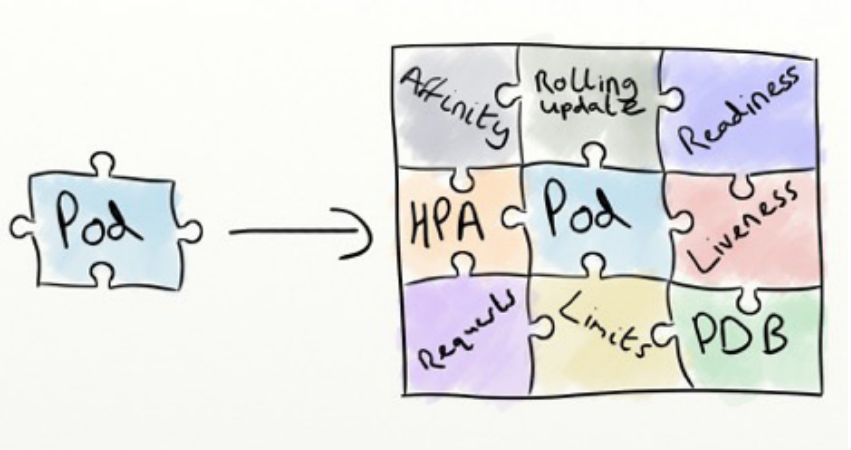
Resource requests and limits

Resource requests and limits tell Kubernetes how much CPU and memory you expect your application will use. Setting requests and limits on a deployment is one of the most important things you can do. Without requests, the Kubernetes scheduler cannot ensure that workloads are spread across your nodes evenly and this may result in an unbalanced cluster with some nodes overcommitted and some nodes underutilized.

Having appropriate requests and limits will allow autoscalers to estimate capacity and ensure that the cluster expands (and contracts) as the demand changes.

```
spec:
  containers:
   - name: example
     resources:
       requests:
         cpu: 100m
         memory: 64Mi
         limits:
         cpu: 200m
         memory: 128Mi
```

Requests and limits are set on a per container basis within your pod, but the scheduler will take into consideration the requests of all the containers. For example, a pod with three containers, each requesting 0.1 CPUs and 64 MB of memory (as in the spec example above) will produce a total request of 0.3 CPUs and 192 MB of memory. If you are running pods with multiple containers, be wary of the total requests for the pod: the higher the total, the more restricted scheduling (finding a node with available resources to match the pod's total request) becomes.



```
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - nginx-ingress-controller
        topologyKey: failure-domain.beta.kubernetes.io/zone
```

Code 1

In Kubernetes, CPUs are measured as decimals of CPU cores. If a pod requests 0.3 CPUs, it will be limited to using up to 30% of one core of the processor. Memory is measured in megabytes as you might expect. Requests should represent a reasonable guess (a lot of this is guesswork, sorry) at what you expect your container might use during normal operations.

Load testing can be a good way to get initial values for your requests, for example, when deploying Nginx as an ingress controller in front of a service. Imagine you are expecting 30,000 requests per second under normal load and you start with three replicas of Nginx. Load-testing a single container for 10,000 requests per second and recording its resource usage might give you a reasonable starting point for your requests per pod.

Limits, on the other hand, are hard limits. If a container reaches its CPU limit, it will be throttled; if it reaches its memory limit, it will be killed.

You should, therefore, set limits higher than the requests and such that they are only reached in exceptional circumstances. For instance, you might want to kill something deliberately if there was a memory leak to stop it crashing your entire cluster.

You must be careful here, though. In the case that a single node in the cluster starts running out of memory or CPU, a pod with containers over their requests might be killed/limited before they reach their limits. The first pod to be killed in a "lack of memory" scenario will be the one whose containers exceed their requests quota by the biggest margin.

With appropriate requests on all pods within a Kubernetes cluster, the scheduler can almost always guarantee that the workloads will have the resources they need for their normal operation.

With appropriate limits set on all containers within a Kubernetes cluster, the system can ensure that no single pod starts hogging all the resources, nor can it affect the running of other workloads. Perhaps even more importantly, no single pod will be able to bring down an entire node, since memory consumption is limited.

Node and pod affinities and anti-affinities

Affinities and anti-affinities are another kind of hint that the scheduler can use when trying to find the best node to assign your pod to. They allow you to spread or confine your workload based on a set of rules or conditions.

At present, there are two kinds of affinity/anti-affinity: required and preferred. Required affinities will stop a pod from scheduling if the affinity rules cannot be met by any node. With preferred affinities, on the other hand, a pod will still be scheduled even if no node was found that matches the affinity rules.

The combination of these two types of affinity allows you to place your pods according to your specific requirements, for example: "

If possible, run my pod in an availability zone without other pods labelled `app=nginx-ingress-controller`"

or "

Only run my pods on nodes with a GPU."

Below is an example of an anti-affinity. This pod anti-affinity ensures that pods with the label `app=nginx-ingress-controller` are scheduled in different availability zones. In a scenario where you have nodes in three different zones in a cluster and you want to run four of these pods, this rule would stop the fourth pod from scheduling. (Code 1)

If we changed the `requiredDuringSchedulingIgnoredDuringExecution` line to `preferredDuringSchedulingIgnoredDuringExecution`, that would tell the scheduler to spread the pods across the availability zones until that's no longer possible. With the prefer rule, the scheduler will balance a fourth, fifth, or more pods across the availability zones as well.

The `topologyKey` field in the affinity specification is based on the labels on your nodes. You can use the labels to ensure that pods are only scheduled on nodes with certain storage types or nodes that have GPUs. You can also prefer that they are scheduled on one type of node over another or even a node with a particular hostname.

For more details of how the scheduler manages distributing pods across nodes, check out this post from my colleague Alexandru Topliceanu.

Health checks

Health checks come in two flavors with Kubernetes: readiness and liveness probes. A readiness probe tells Kubernetes that the pod is ready to start receiving requests (usually HTTP). A liveness probe on a pod tells Kubernetes that the pod is still running as expected.

HTTP readiness/liveness probes are very similar to traditional load-balancer health checks. Their configuration normally specifies a path and port, but can also define timeouts, success/failure thresholds, and initial delays. The probe passes for any response with a status code between 200 and 399.

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 10254
    scheme: HTTP
  initialDelaySeconds: 10
  timeoutSeconds: 5
livenessProbe:
  httpGet:
    path: /healthz
    port: 10254
    scheme: HTTP
  initialDelaySeconds: 10
  timeoutSeconds: 5
```

Kubernetes uses liveness probes to determine whether the container is healthy. Should a liveness probe start failing while the pod is running, Kubernetes will restart the pod in accordance with its restart policy. Every pod should have a liveness probe if possible so that Kubernetes can tell whether the application is working as expected or not.

Readiness probes are for containers that expect to be serving requests; these will typically have a service receiving requests in front of them. A liveness probe and a readiness probe may well be the same thing in certain cases. However, in the case where your container may start up and have to process some data or do some calculation before serving requests, the readiness probe tells Kubernetes when the container is ready to be registered with the service and receive requests from the outside world.

While both of these probes are often HTTP callbacks, Kubernetes also supports TCP and Exec callbacks. TCP probes check that a socket is open within the container and Exec probes execute a command within the container, expecting a 0 exit code:

```
livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
```

These checks, when set up correctly, help ensure that requests to a service always hit a container that is in a state capable of processing that request. The probes are also used in other core Kubernetes functions when autoscaling and performing rolling updates. In the remainder of this article, when I refer to a pod being ready, that means its readiness probe is passing.

Deployment strategies

Deployment strategies determine how Kubernetes replaces running pods when you want to update their configuration (changing the image tag, for instance). There are currently two kinds of strategies: recreate and rolling update.

The "recreate" strategy kills all pods managed by the deployment before bringing up a new one. This might sound dangerous — and it is, in most scenarios. The intended use of this strategy is for preventing two different versions running in parallel. Realistically, this means it is used for databases or applications where the running replica must shut down before the new instance is launched.

The "rolling update" strategy, on the other hand, runs the old and the new configurations side by side. It has two configuration options: `maxUnavailable` and `maxSurge`. They define how many pods Kubernetes can remove and how many extra pods Kubernetes can add as it starts a rolling update.

They can both be set as an absolute number or a percentage. The default for both values is 25%. When `maxUnavailable` is set to a percentage and a rolling update is in progress, Kubernetes will calculate (rounding down) how many replicas it can terminate to allow new replicas to come up. When `maxSurge` is set to a percentage, Kubernetes will calculate (rounding up) how many extra replicas it can add during the update process.

For example, in a deployment with six replicas: when there's an update to the image tag and the rolling update strategy has the default configuration, Kubernetes will terminate one instance

(6 instances * 0.25 = 1.5, rounded down to 1) and then introduce a new replicaset with three new instances (6 instances * 0.25 = 1.5, rounded up to 2 , plus 1 instance to compensate for the one terminated = 3) to give a total of eight running replicas. Once the new pods become ready, it will terminate two more instances from the old replicaset to bring the deployment back to the desired replica count, before repeating this process again and again until deployment is finished.

If a deployment fails (that means a liveness or readiness probe failed on the new replicaset) during this method, the rolling update will stop and your running workload would remain with the old replicaset, albeit potentially slightly smaller if the rolling update had already removed some old instances.

You can configure the `maxSurge` and `maxUnavailable` based on your needs; you can set either of them to zero if you so desire (although they can't both be zero simultaneously), which would result in you never running more than your desired replica count or never running fewer than your desired replica count respectively.

Pod disruption budgets

Disruptions in Kubernetes clusters are almost inevitable. The days of pet VMs with two-year uptimes are behind us. Today's VMs tend to be more like cattle, disappearing at a moment's notice.

There is an almost endless list of reasons why a node in Kubernetes might disappear. Here are just a few we have seen:

- A spot instance on AWS gets taken out of service.
- Your infrastructure team wants to apply a new config and starts replacing nodes within a cluster.
- An autoscaler discovers your node is underutilized and removes it.

The job of the pod disruption budget is to ensure that you always have a minimum number of ready pods for your deployment. A pod disruption budget allows you to specify either the `minAvailable` or `maxUnavailable` number of replicas within a deployment. These values can be a percentage of the desired replica count or an absolute number.

By configuring a pod disruption budget for your deployment, Kubernetes can start rejecting voluntary disruptions, which are those caused by the Eviction API (for example, a cluster administrator draining nodes in the cluster).

With a pod disruption budget of `maxUnavailable` equal to 1, the first of these drain attempts will evict the first of your pods. While this pod is being rescheduled and waiting to become ready (i.e., is unavailable), all requests to evict other pods in your deployment will fail. Applications that use the Eviction API, such as kubectl drain, are expected to retry eviction attempts until they are successful or the application times out, so both your administrator and your developers can achieve their goals without affecting each other.

While the pod disruption budget cannot protect you from involuntary disruptions, it can ensure that you don't reduce capacity further during these events, halting any attempt to drain a node or move pods until new replicas are scheduled and ready.

Horizontal pod autoscalers

One of the best things about the Kubernetes control plane is the ability to scale your applications based on their resource utilization. As your pods become busier, it can automatically bring up new replicas to share the load.

The controller periodically looks at metrics provided by metrics-server (or heapster for Kubernetes versions 1.8 and earlier) and, based on the load of the pods, scales up or scales down the deployment as necessary.

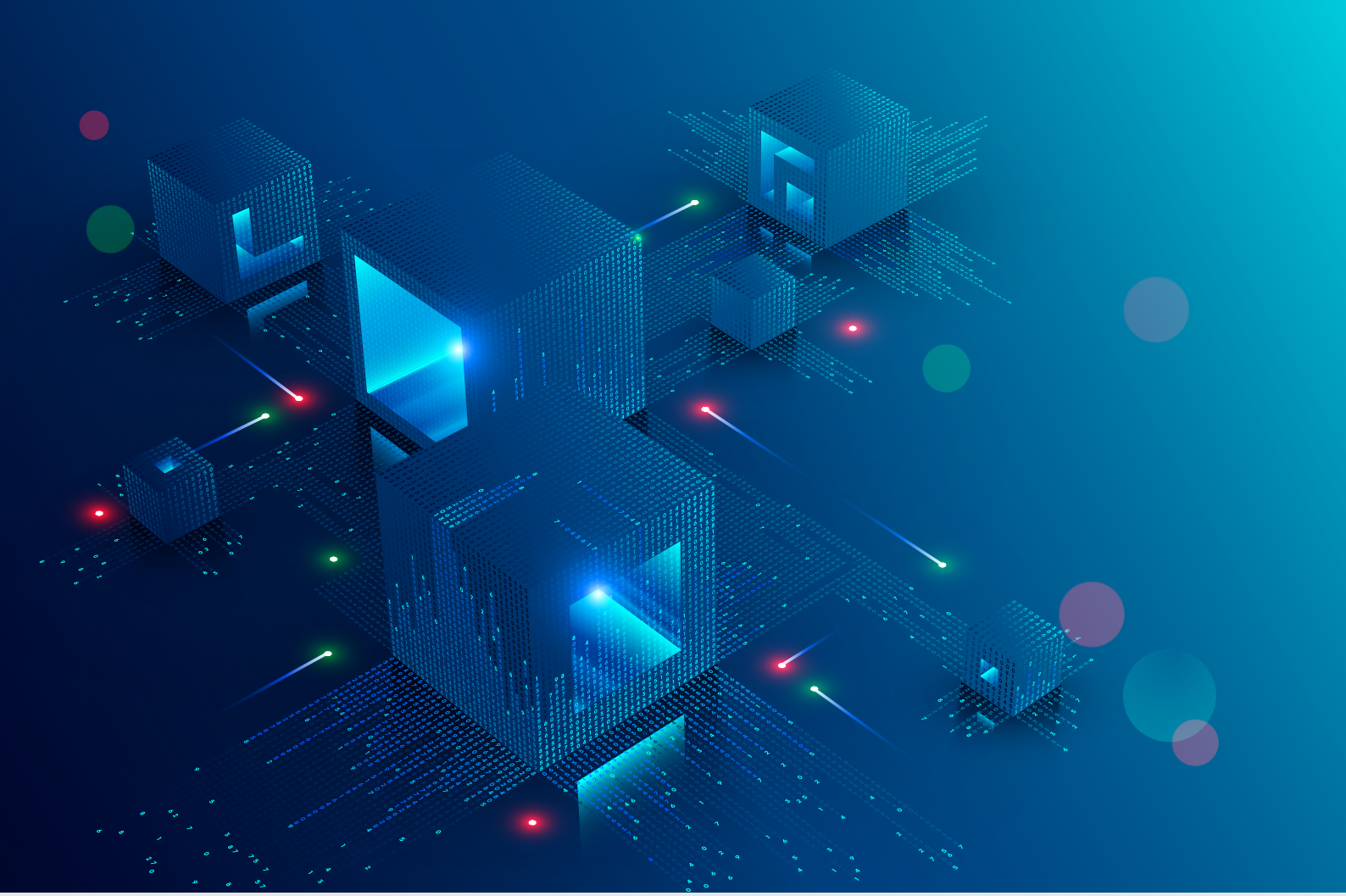
When configuring a horizontal pod autoscaler, you can scale based on CPU and memory usage. Custom metrics servers could extend the metrics you have available and may even let you scale based on, for example, requests per second to a service.

Once you have chosen which metric to scale on (default or custom), you can define your `targetAverageUtilization` for resources or `targetValue` for custom metrics. This is your desired state.

The resource values are based on the requests set for the deployment. If you set the CPU `targetAverageUtilization` to 70%, the autoscaler will try to keep the average CPU utilization across the pods at 70% of their requested CPU value. Additionally, you must set the range of replicas you would like the deployment to have: a minimum and a maximum number of replicas that you think your deployment will need.

Conclusion

By applying the configuration options discussed above, you can take advantage of everything Kubernetes has to offer for making your applications as redundant and available as possible. While not everything will apply to every application, I do strongly urge you to start setting appropriate requests and limits, as well as health checks where you can.



Read online on InfoQ

KEY TAKEAWAYS

The microservice architecture is still the most popular architectural style for distributed systems. But Kubernetes and the cloud-native movement have redefined certain aspects of application design and development at scale. Observability of services is not enough on a cloud-native platform. A more fundamental prerequisite is to make microservices automatable by implementing health checks, reacting to signals, declaring resource consumption, etc.

In the post-Kubernetes era, using libraries to implement operational networking concerns (such as Hystrix circuit-breaking) has been completely overtaken by service mesh technology.

MICROSERVICES IN A POST-KUBERNETES ERA

by **Bilgin Ibryam**

The microservices hype started with a bunch of extreme ideas about the organizational structure, team size, size of the service, rewriting and throwing services rather than fixing, avoiding unit tests, etc. In my experience, most of these ideas were proven wrong, impractical, or at least not generally applicable. Most of the surviving principles and practices are so generic and loosely defined that they may stand true for many years to come without meaning much in practice.

Adopted a couple of years before Kubernetes was born, microservices is still the most popular architectural style for distributed systems. But Kubernetes and the cloud-native movement has redefined certain aspects of application design and development at scale. I question some of the original microservices ideas and hold that they are not standing as strongly in the post-Kubernetes era as they were before.

Not only observable but automatable

Observability has been a fundamental principle of microservices from the very beginning. While it remains true for distributed systems in general, a lot of it (such as process health checks, CPU, and memory consumption) today comes out of the box at the platform level (on Kubernetes specifically). The minimal requirement is for an application to log into the console in JSON format. From there, the platform can track resource consumption, trace requests, and gather all kinds of metrics without much service-level development effort.

On cloud-native platforms, observability is not enough. A more fundamental prerequisite is to make microservices automatable, by implementing health checks, reacting to signals, declaring resource consumption, etc. It is possible to put almost any application in a container and run it. But to create a containerized application that a cloud-native platform can effectively automate and orchestrate requires following certain rules. Following these principles and patterns will ensure that the resulting containers behave like a good cloud-native citizen in most container orchestration engines, allowing them to be automatically scheduled, scaled, and monitored.

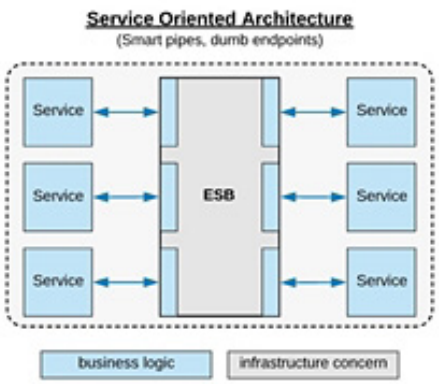
Rather than observing what is happening in a service, we want the platform to detect anomalies and reconcile them as declared. Whether that occurs by stopping traffic to a service instance, restarting, scaling, moving a service to a healthy host, retrying a failing request, or something else doesn't matter. If the service is automatable, all corrective actions occur automatically, and we only have to describe the desired state rather than observing and reacting. A service should be observable, but also rectifiable by the platform without human intervention.

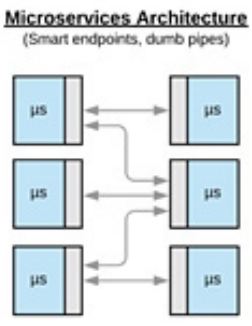
Smart platform and services with the right responsibilities

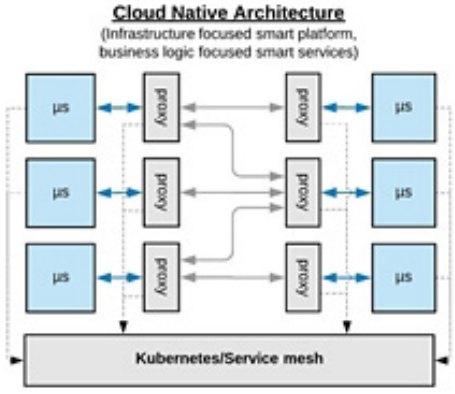
While transitioning from the SOA to the microservices world, the notion of smart endpoints and dumb pipes in service interactions was another fundamental shift. In the microservices world, the services would not rely on the presence of a centralized, smart routing layer, but would instead rely on the smart endpoints that possess some platform-level features. That was achieved by embedding some of the capabilities of the traditional enterprise service bus (ESB) in every microservice and transitioning to lightweight protocols that don't have business-logic elements.

While this is still a popular way to implement service interaction over a unreliable networking layer (with libraries such as Hystrix), it has been completely overtaken by service-mesh technology in the Kubernetes era. Interestingly, the service mesh is even smarter than the traditional ESBs. The mesh can do dynamic routing, service discovery, load balancing based on latency, response type, metrics and distributed tracing, retries, timeouts… you name it.

Rather than use an ESB's one centralized routing layer, each microservice in a service mesh typically has its own router: a sidecar container that performs the proxying logic with an additional central management layer. More importantly, the pipes (the platform and the service mesh) hold no business logic; they are purely focused on infrastructure concerns, leaving the service to focus on







the business logic. As shown on the diagram below, this evolved out of the lessons learned from ESBs and microservices to fit the dynamic and non-reliable nature of cloud environments.

SOA versus MSA versus CNA

Looking at the other aspects of the services, we notice that cloud-native affects not only endpoints and service interactions. The Kubernetes platform (with all the additional technologies) also takes care of resource management, scheduling, deployment, configuration management, scaling, service interaction, etc. Rather than again calling it a smart proxy and dumb endpoints, I think it is better described as a smart platform and smart services with the right responsibilities. It is not only about the endpoints; it is instead a complete platform, automating all the infrastructure aspects of services that primarily focus on business functionality.

Design for recovery, not failure

Microservices running on cloud-native environments, where the infrastructure and networking are inherently unreliable, have to be designed for failure. There is no question about it. But more and more failures are detected and handled by the platform, and there is less provision left for catching failures from within a microservice. Instead, think about designing your service for recovery by implementing idempotency from multiple dimensions.

The container technology, the container orchestrators, and the service mesh can detect and recover from many failures: infinite loops, memory leaks, etc. With the transition to the serverless model, where a service lives only for the few milliseconds required to handle a single request, concerns around garbage collection, thread pools, and resource leakage are less and less relevant, as well.

With all this and more handled by the platform, think about your service as a hermetic black box that will be started and stopped many times — so make the service idempotent for restarts. Your service will be scaled up and down multiple times, so make it safe for scaling by making it stateless. Assume many incoming requests will eventually time out and make the endpoints idempotent. Assume many outgoing requests will temporarily fail and the platform will retry them for you, so make sure you consume idempotent services.

In order to be suitable for automation in cloud-native environments a service must be:

- idempotent for restarts (a service can be killed and started multiple times),idempotent for scaling up/down (a service can be autoscaled to multiple instances),an idempotent service producer (other services may retry calls), andan idempotent service consumer (the service or the mesh can retry outgoing calls).

If your service always behaves the same way when the above actions are performed once or many times, then the platform will be able recover your services from failures without human intervention.

Lastly, keep in mind that all the recoveries the platform provides are only local optimizations. As nicely put by Christian Posta, application safety and correctness in a distributed system is still the responsibility of the application. An overall business process-wide mindset (which may span multiple services) is necessary for designing a holistically stable system.

Hybrid development responsibilities

More and more of the microservices principles are implemented and provided as capabilities by Kubernetes and its complementary projects. As a consequence, a developer has to be fluent in a programming language to implement the business functionality and equally fluent in cloud native technologies to address the non-functional infrastructure-level requirements to fully implement a capability.

The line between business requirement and infrastructure (operational or cross-functional requirements or system-quality attributes) is always blurry and it is not possible to take one aspect and expect someone else to do the other. For example, if you implement the retry logic in the service-mesh layer, you have to make the consumed service idempotent at the business-logic or database layer within the service. If you use a timeout at service-mesh level, you have to synchronize the service consumer timeouts within the service. If you have to implement a recurring execution of a service, you have to configure a Kubernetes job to do the temporal execution.

Going forward, some service capabilities will be implemented within the services as business logic and others will be provided as platform capabilities. While using the right tool for the right task is a good separation of responsibilities, the proliferation of technologies hugely increases the overall complexity. Imple-

menting even a simple service in terms of business logic requires a good understanding of distributed-technology stacks, as the responsibilities are spread at every layer.

It is proven that Kubernetes can scale up to thousands of nodes, tens of thousands of pods, and millions of transactions per second. But can it scale down? The threshold of your application size, complexity, or criticality that justifies the introduction of the cloud-native complexity is not clear to me yet.

Conclusion

It is interesting to see how the microservices movement pushed the adoption of container technologies such as Docker and Kubernetes. While the microservices practices initially drove these technologies forward, now it is Kubernetes that defines microservices architecture principles and practices.

As a recent example, we are not far from accepting the function model as a valid microservices primitive, rather than considering it as an anti-pattern for nano services. We are insufficiently questioning the cloud-native technologies for their practicality and applicability for small and medium-sized cases, but instead are jumping in somewhat carelessly with excitement.

Kubernetes took advantage of many of the lessons we learned from ESBs and microservices, and as such, it is the ultimate distributed-system platform. It is the technology defining the architectural styles rather than the other way around. Whether that is good or bad, only time will show.

Read online on InfoQ

KEY TAKEAWAYS

Kubernetes is experiencing phenomenal growth since it solves specific pain points with respect to application portability and deployment.

Kubernetes is already eliminating vendor lock-in and enabling cloud portability with the choice of offerings on the different clouds.

Although Kubernetes is already established in multiple clouds, "multi-cloud" means more than that.

The Kubernetes community is coming together to address the challenges related to multi-cloud.

VIRTUAL PANEL: KUBERNETES AND THE CHALLENGES OF MULTI-CLOUD

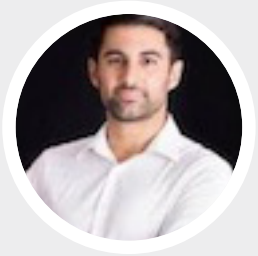
by **Rags Srinivas**

At the recently concluded sold-out Kubecon+CloudNativeCon 2018 conference in Seattle, the opening keynote and many technical sessions discussed the multiple Kubernetes services offered by the major cloud providers.

THE PANELISTS

Lew Tucker

is the former VP/CTO at Cisco Systems and served on the board of directors of the Cloud Native Computing, OpenStack, and Cloud Foundry Foundations. He has more than 30 years of experience in the high-tech industry, ranging from distributed systems and artificial intelligence to software development and parallel system architecture. Prior to Cisco, Tucker was VP/CTO Cloud Computing at Sun Microsystems where he led the development of the Sun Cloud platform. He was also a member of the JavaSoft executive team, launched java.sun.com, and helped to bring Java into the developer ecosystem. Tucker moved into technology following a career in neurobiology at Cornell University Medical school and has a Ph.D. in computer science.



Marco Palladino

is an inventor, software developer, and Internet entrepreneur based in San Francisco. CTO and co-founder of Kong, he is Kong's co-author, responsible for the design and delivery of the company's products, while also providing technical thought leadership around APIs and microservices within both Kong and the external software community. Prior to Kong, Marco co-founded Mashape in 2010, which became the largest API marketplace and was acquired by RapidAPI in 2017.

Janet Kuo

is a software engineer for Google Cloud. She has been a Kubernetes project maintainer since 2015. She currently serves as co-chair of KubeCon + CloudNativeCon.

Sheng Liang

is co-founder and CEO of Rancher Labs. Rancher develops a container-management platform that helps organizations adopt Kubernetes. Previously, he was CTO of Cloud Platform at Citrix and CEO and founder of Cloud.com (acquired by Citrix).

Although cloud providers have their respective Kubernetes offerings, with each trying to distinguish itself with complementary services, the Kubernetes community's goal is application portability across the spectrum of these Kubernetes offerings. InfoQ caught up with experts in the field at Kubecon 2018 in Seattle: Lew Tucker, former cloud CTO at Cisco; Sheng Liang, CEO of Rancher Labs; Marco Palladino, CTO of Kong; and Janet Kuo, Kubecon+CloudNativeCon 2018 co-chair and software engineer at Google to ask them whether a single solution can really span multiple cloud products or do organizations have to face dealing with multiple clouds. We discussed more multi-cloud aspects of Kubernetes and the challenges that remain.

InfoQ: Let's talk about the Kubernetes community in general, and the recently concluded Kubecon 2018 in particular. As folks being involved with the community and the conference right from its inception, what reasons do you attribute to the growth and how does it affect developers and architects in particular going forward?

Lew Tucker: It's true — Kubernetes is seeing amazing growth and expansion of the developer and user communities. As an open-source orchestration system for containers, it's clear that Kubernetes makes it easier for developers to build and deploy applications with resiliency and scalability, while providing portability across multiple cloud platforms. As the first project to graduate from the Cloud Native Computing Foundation, it is quickly becoming the de facto platform for cloud native apps.

Sheng Liang: Kubernetes became popular because it solved an important problem really well: how to run applications reliably. It is the best container orchestrator, cluster manager, and scheduler out there. The best technology combined with a very-well-run open-source community made Kubernetes unstoppable.

Marco Palladino: The growth of Kubernetes cannot be explained without taking into consideration very disruptive industry trends that transformed the way we build and scale software: microservices and the rise of containers. As businesses kept innovating to find new ways of scaling their applications and their teams, they discovered that decoupling and distributing both the software and the organization would provide a better framework to ultimately enable their business to also grow exponentially over time. Large teams and large monolithic applications were, therefore, decoupled — and distributed — into smaller components.

While a few companies originally led this transformation a long time ago (Amazon, Netflix, and so on) and built their own tooling to enable their success, every other large enterprise organization lacked both the will, the know-how and the R&D capacity to also approach a similar transition. With the mainstream adoption of containers (Docker in 2013) and the emergence of platforms like Kubernetes (in 2014) shortly after, every enterprise in the world could now approach the microservices transition by leveraging a new, easy-to-use self-service ecosystem. Kubernetes, therefore, became the enabler not just to a specific way of running and deploying software, but also to an architectural modernization that organizations were previously cautious to adopt because of lack of tooling, which was now accessible to them.

It cannot be ignored that Docker and Kubernetes, and most of the ecosystem surrounding these platforms, are open source — to understand the Kubernetes adoption, we must also understand the decisional shift of enterprise software adoption from being top-down (like SOA, driven by vendors) to being bottom-up (driven by developers). As a result of these industry trends that in turn further fed into Kubernetes adoption, developers and architects now can leverage a large ecosystem of self-service open-source technologies that's unprecedented compared to even half a decade ago.

Janet Kuo: One of the key reasons is that Kubernetes has a very strong community is it is made up of a diverse set of end users, contributors, and service providers. The end users don't choose Kubernetes because they love container technology or Kubernetes, they choose Kubernetes because it solves their problems and allows them to move faster. Kubernetes is also one of the largest and one of the most active open-source projects, with contributors from around the globe. This is because there's no concentration of power in Kubernetes, and that encourages collaboration and innovation regardless of whether or not someone works on Kubernetes as part of their job or as a hobby. Lastly, most of the world's major cloud providers and IT service providers have adopted Kubernetes as their default solution for container orchestration. This network effect make Kubernetes grow exponentially.

InfoQ: Some of us can recount the Java days that eliminated vendor lock-in. Kubernetes is similar in that vendors seem to cooperate on standards and compete on implementations. Eliminating vendor lock-in might be good overall but what does cloud portability or multi-cloud mean, and does it even matter to the customer?

Tucker: Layers of abstraction are typically made to hide the complexity of underlying layers, and as they become platforms, they also provide a degree of portability between systems.

In the early Java days, we talked about the promise of "write once, and run anywhere" reducing or eliminating the traditional operating system lock-in associated with either Unix or Windows. The degree to which this was realized often required careful coding, but the value was clear. Now with competing public cloud vendors,

we have a similar situation. The underlying cloud platforms are different, but Docker containers and Kubernetes provide a layer of abstraction and high degree of portability across clouds. In a way, this forces cloud providers to compete on the services offered. Users then get to decide how much lock-in they can live with in order to take advantage of the vendor-specific services. As we move more and more towards service-based architectures, it's expected that this kind of natural vendor lock-in will become the norm.

Liang: While eliminating cloud lock-in might be important for some customers, I do not believe a product aimed solely at cloud portability can be successful. Many other factors, including agility, reliability, scalability, and security are often more important. These are precisely some of the capabilities delivered by Kubernetes. I believe Kubernetes will end up being an effective cloud-portability layer, and it achieves cloud portability almost as a side-effect, sort of like how the browser achieved device portability.

Palladino: Multi-cloud is often approached from the point of view of a conscious top-down decision made by the organization as part of a long-term strategy and roadmap. However, the reality is much more pragmatic. Large enterprise organizations really are the aggregation of a large variety of teams, agendas, strategies and products that happen to be part of the same complex "multicellular" organism.

Multi-cloud within an organization is bound to happen simply because each different team/product inevitably makes different decisions on how to build their software, especially in an era of software development where developers are leading, bottom-up, all the major technological decisions and experimentations. Those teams that are very close to the end-user and to the business are going to adopt whatever technology (and cloud) better allows them to achieve their goals and ultimately scale the business. The traditional central IT — far from the end users and from the business but closer to the teams — will then need to adapt to a new hybrid reality that's being developed beneath them as we speak. Corporate acquisitions over time of products and teams that are already using different clouds will also lead to an even more distributed and decoupled multi-cloud organization.

Multi-cloud is happening not necessarily because the organization wants to, but because it has to. Containers and Kubernetes, by being extremely portable, are therefore a good technological answer to these pragmatic requirements. They offer a way to run software among different cloud vendors (and bare metal) with semi-standardized packaging and deployment flows, thus reducing operational fragmentation.

Kuo: Cloud portability and multi-cloud gives users the freedom to choose best-fit solutions for different applications for different situations based on business needs. Multi-cloud also enables more levels of redundancy. By adding redundancy, users achieve more flexibility to build with the best technology, and also helps them optimize operations and stay competitive.

InfoQ: Kubernetes products on different clouds try to differentiate their respective offerings, which seems the natural course of "co-opetition". What pitfalls should the Kubernetes community watch out for?

Tucker: It's natural to expect that different vendors will want to differentiate their respective offerings in order to compete in the market. The most important thing for the Kubernetes community is to remain true to its open-source principles and put vendor-based or infrastructure-based differences behind standard interfaces to keep the platform from fragmenting into proprietary variants. Public interfaces, such the device plugin framework (for things such as GPUs) and CNI (Container Networking Interface) isolate infrastructure-specific differences behind a common API, allowing vendors to compete on implementation while offering a common layer. Vendors today also differentiate in how they provide managed Kubernetes. This sits outside the platform, leaving the Kubernetes API intact, which still leaves it up the user whether or not they wish to adopt a vendor's management framework in their deployment model.

Liang: The user community should avoid using features that make their application only work with a specific Kubernetes distro. It is easy to stand up Kubernetes clusters from a variety of providers now. After creating the YAML files or Helm charts to deploy your application on your distro of choice, you should also try the same YAML files or Helm charts on GKE or EKS clusters.

Palladino: The community should be wary of any cloud vendor hinting at an "embrace, extend, extinguish" strategy that will, in the long term, fragment

Kubernetes and the community and pave the road for a new platform "to rule them all".

Kuo: Fragmentation is what the Kubernetes community should work together to avoid. Otherwise, end users cannot get consistent behavior in different platforms and lose the portability and the freedom to choose — which brought them to Kubernetes in the first place. To address this need, first identified by Google, the Kubernetes community has invested heavily in conformance, which ensures that every service provider's version of Kubernetes supports the required APIs and give end users consistent behavior.

InfoQ: Can you mention some no-brainer distributed system or application patterns that make it a shoe-in for multi-cloud Kubernetes? Is it microservices?

Tucker: Microservice-based architectures are a natural fit with Kubernetes. But when breaking apart monolithic apps into a set of individual services, we've now brought in the complexity of a distributed system that relies on communication between its parts. This is not something that every application developer is prepared to take on. Service meshes, such as Istio, seem like a natural complement for Kubernetes. They offload many networking and traffic-management functions, freeing the app developer from having to worry about service authentication, encryption, key exchange, traffic management, and others while providing uniform monitoring and visibility.

Liang: While microservices obviously fit multi-cloud Kubernetes, legacy deployment architecture fits as well. For example, some of our customers deploy multi-cloud Kubernetes for the purpose of disaster recovery. Failure of the application in one cloud does not impact the functioning of the same app in another cloud. Another use case is geographic replication. Some of our customers deploy the same application across many different regions in multiple clouds for the purpose of geographic proximity.

Palladino: Microservices is a pattern that adds a significant premium on our architecture requirements because it leads to more moving parts and more networking operations across the board — managing a monolith is a O(1) problem, while managing microservices is a O(n) problem. Kubernetes has been a very successful platform for managing microservices, since it provides useful primitives that can be leveraged to automate a large variety of operations that in turn remove some, if not most, of that "microservices premium" from the equation. The emergence of API platforms tightly integrated with those Kubernetes primitives — like sidecar and ingress proxies — are also making it easier to build network-intensive, decoupled, and distributed architectures.

With that said, any application can benefit from running on top of Kubernetes, including monoliths. By having Kubernetes running as the underlying abstraction layer on top of multiple cloud vendors, teams can now consolidate operational concerns — like distributing and deploying their applications — across the board without having to worry about the specifics of each cloud provider.

Kuo: Microservices is one of the best-known patterns. Sidecar pattern is also critical for modern applications to integrate functionalities like logging, monitoring, and networking into the applications. Proxy pattern is also useful for simplifying application developers' lives so that it's much easier to write and test their applications, without needing to handle network communication between microservices, and this is commonly used by service mesh solutions, like Istio.

InfoQ: What does the emerging service-mesh pattern do for multi-cloud platform implementation?

Tucker: I believe Kubernetes-based apps based on a microservices and service-mesh architecture will likely become more prevalent as the technology matures. The natural next step is for a service mesh to connect multiple Kubernetes clusters running on different clouds. A multi-cloud service mesh would make it much easier for developers to move towards stitching together the very best components and services from different providers into a single application or service.

Liang: The emerging service-mesh pattern adds a lot of value for multi-cloud Kubernetes deployment. When we deploy multiple Kubernetes clusters in multiple clouds, the same Istio service mesh can span these clusters, providing unified visibility and control for the application.

Palladino: Transitioning to microservices translates to a heavier use of networking across the services that we are trying to connect. As we all know, the network is implicitly unreliable and cannot be trusted, even within the private organization's network. Service mesh is a pattern that attempts to make our inherently un-

reliable network reliable again by providing functionalities (usually deployed in a sidecar proxy running alongside our services) that enable reliable service-to-service communication (like routing, circuit breakers, health checks, and so on). In my experience working with large enterprise organizations implementing service mesh across their products, the pattern can help with multi-cloud implementations by helping routing workloads across different regions and data centers, and enforcing secure communication between the services across different clouds and regions.

Kuo: Container orchestration is not enough for running distributed applications. Users need tools to manage those microservices and their policies, and they want those policies to be decoupled from services so that policies can be updated independent of the services. This is where service-mesh technology comes into play. The service-mesh pattern is platform independent, so a service mesh can be built between clouds and across hybrid infrastructures. There are already several open-source service-mesh solutions available today. One of the most popular open-source service-mesh solutions is Istio. Istio offers visibility and security for distributed services, and ensures a decoupling between development and operations. As with Kubernetes, users can run Istio anywhere they see fit.

InfoQ: Vendors and customers usually follow the money trail. Can you talk specifically about customer success stories or case studies where Kubernetes and/or multi-cloud matters?

Liang: Rancher 2.0 has received tremendous market success precisely because it is capable of managing multiple Kubernetes clusters across multiple clouds. A multinational media company uses Rancher 2.0 to stand up and manage Kubernetes clusters in AWS, Azure, and their in-house vSphere clusters. Their IT department is able to control which application is deployed on which cloud depending on its compliance needs. In another case, Goldwind, the third-largest wind-turbine manufacturer in the world, uses Rancher 2.0 to manage multiple Kubernetes clusters in the central data center and in hundreds of edge locations where wind turbines are installed.

Palladino: I have had the pleasure of working very closely with large enterprise organizations and seeing the pragmatic challenges that these organizations are trying to overcome. In particular, one large enterprise customer of Kong decided to move to multi-cloud on top of Kubernetes due to the large number of acquisitions executed over the past few years by the organization. Each acquisition would bring new teams, products, and architectures under the management of the parent organization, and you can imagine how hard it became to grow existing teams within the organization with so much fragmentation. Therefore, the organization decided to standardize how applications are being packaged (with Docker) and executed (with Kubernetes) in an effort to simplify ops across all the teams.

Although sometimes very similar, different cloud vendors actually offer a different set of services with different quality and support, and it turns out that some clouds are better than others when it comes to certain use cases. As a result, many applications running within the organization also ran on different clouds depending on the services they implemented, and the company was already a multi-cloud reality by the time they decided to adopt Kubernetes. In order to keep the latency low between the applications and the specific services that those products implemented from each cloud vendor, they also decided to start a multi-cloud Kubernetes cluster. It wasn't by any means a simple task, but the cost of keeping things fragmented was higher than the cost of modernizing their architecture to better scale it in the long-term. In a large enterprise, it's all about scalability — not just technical but also organizational and operational.

Kuo: There are a wide variety of customer success stories covered in Kubernetes case studies. My favorite is actually one of the oldest: the one about how Pokemon Go (a mobile game that went viral right after its release) runs on top of Kubernetes, which allows game developers to deploy live changes while serving millions of players around the world. People were surprised and excited to see a real use case of large-scale, production Kubernetes clusters. Today, we've learned about so many more Kubernetes customer success stories — such as the ones we just heard from Uber and Airbnb on the KubeCon keynote stage. Diverse and exciting use cases are now the norm within the community.

InfoQ: From a single project or solution viewpoint, is it even feasible for the different components or services to reside in multiple clouds? What are the major technical challenges that need to be solved before Kubernetes is truly multi-cloud?

Tucker: Yes. Work on a dedicated federation API apart from the Kubernetes API is already in progress (see on Kubernetes and on GitHub). This approach is not limited to clusters residing at the same cloud provider. But it's still very early days and many of the pragmatic issues beyond the obvious ones such as increased latency and different cloud-service APIs are still under discussion.

Liang: Yes, it is. Many Rancher customers implement this deployment model. We are developing a number of new features in Rancher to improve multi-cloud experience: 1) a mechanism to orchestrate applications deployed in multiple clusters that reside in multiple clouds; 2) integration with global load balancers and DNS servers to redirect traffic; 3) a mechanism to tunnel network traffic between pods in different clusters; and 4) a mechanism to replicate storage across multiple clusters.

Palladino: Federation across multiple Kubernetes clusters, a control-plane API that can help run multiple clusters and security (users and policies). Synchronizing resources across multiple clusters is also a challenge, as is observability across the board. Some of these problems can be fixed by adopting third-party integrations and solutions, but it would be nice if Kubernetes provided more out-of-the box support in this direction. There is already an alpha version of an experimental federation API for Kubernetes, but it's unfortunately not mature enough to be used in production (with known issues).

Thinking of a multi-cloud Kubernetes is akin to thinking of managing separate Kubernetes clusters at the same time. The whole release lifecycle (packaging, distributing, testing, and so on) needs to be applied and synced across all the clusters (or a few of them based on configurable conditions), while having at the same time a centralized plane to monitor the status of operations across the entire system. Both application and user security also become more challenging across multiple clusters. At runtime, we may want to enforce multi-region routing (for failover) between one cluster and another, which also means introducing more technology (and data plane overhead) in order to manage those use cases. All of the functionality that we normally use in a single cluster — like CI/CD, security, auditing, application monitoring, and alerting — has to be rethought in a multi-cluster, multi-cloud environment, not just operationally but also organizationally.

Kuo: Kubernetes already does a good job of abstracting the underlying infrastructure layer so that cloud-provider services, such as networking and storage, are simply resources in Kubernetes. However, users still face a lot of friction when running Kubernetes in multi-cloud. Technical challenges that need to be solved include connectivity between different regions and clouds, disaster discovery, logging, and monitoring. We need to provide better tooling to make the user experience seamless and the community is dedicated to providing those resources.

InfoQ: Can you briefly talk about products or technologies that may not yet be mainstream but which might help obviate some of the issues that we've been talking about so far?

Liang: Our flagship product, Rancher, is designed specifically to manage multiple Kubernetes clusters across multiple clouds

Palladino: Open-source products like Kong can help in consolidating security, observability, and traffic control for our services across multi-cloud deployments by providing a hybrid control plane that can integrate with different platforms and architectures and by enabling large legacy applications to be decoupled and distributed in the first place.

Open-source ecosystems, like CNCF, also provide a wide variety of tooling that helps developers and architects navigate the new challenges of multi-cloud and hybrid architectures that the enterprise will inevitably have to deal with (monoliths, SOA, microservices, and serverless). It's very important that as the scale of our systems increase, we avoid fragmentation of critical functions (like security) by leveraging existing technologies instead of reinventing the wheel. And open-source, once again, is leading this trend by increasing developer productivity, while at the same time creating business value for the entire organization.

Kuo: One of the common patterns I see is to manage everything using Kubernetes API. To do this, you most likely need to customize Kubernetes APIs, such as using Kubernetes's CustomResourceDefinition. A number of technologies built around Kubernetes, such as Istio, rely heavily on this feature. Kubernetes devel-

opers are improving the custom Kubernetes APIs feature, to make it easier to build new tools for development and deployment on multi-cloud.

InfoQ: Where do you see the Kubernetes community headed and how important is multi-cloud in defining the roadmap? Have any other thoughts about Kubernetes and KubeCon to share with developers and architects?

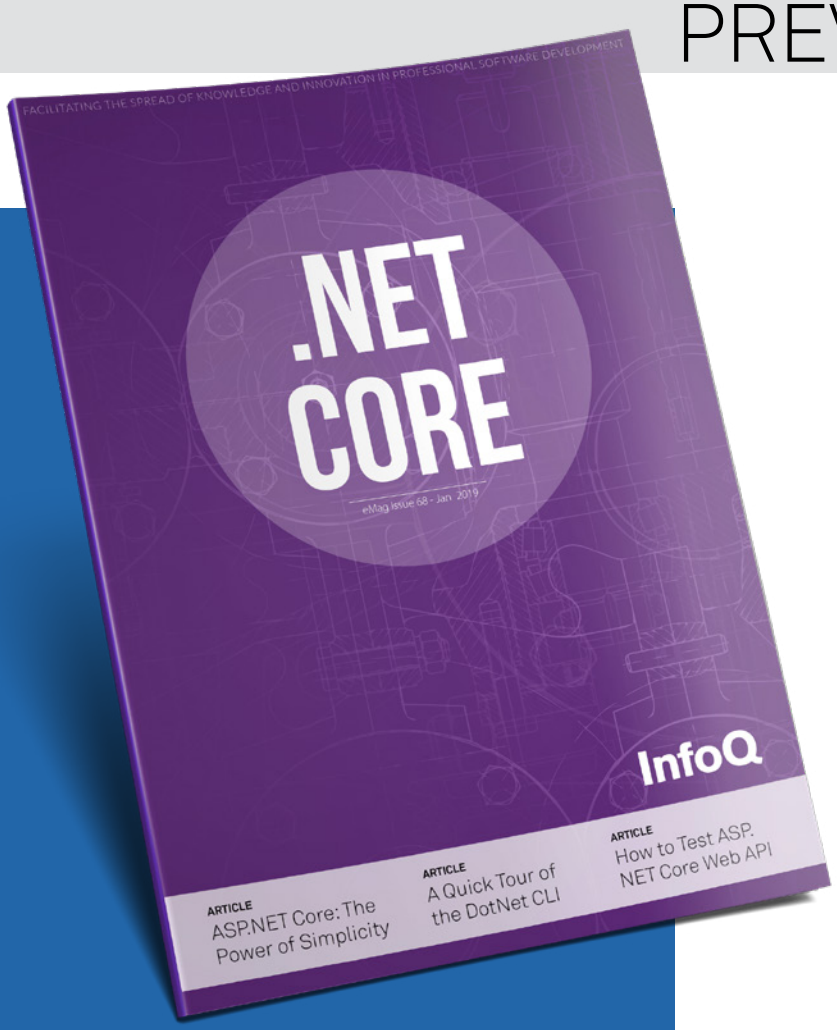
Tucker: Most user surveys show that companies use more than a single cloud vendor and often include both public and private clouds. So multi-cloud is simply a fact. Kubernetes therefore provides an important common platform for application portability across clouds. The history of computing, however, shows that we continually build up layers of abstraction. When we therefore look at multi-cloud, it may be that other "platforms" such as serverless or a pure services-based architecture built on top of Kubernetes might be where we are really headed.

Liang: Multi-cloud started as a wonderful side benefit of Kubernetes. I believe multi-cloud is now a core requirement when people plan for Kubernetes deployment. The community is working in a number of areas to improve multi-cloud support: Kubernetes conformance, SIG multi-cluster, and federation. I'm extremely excited about where all these efforts are headed. I encourage all of you to take a look at these projects if you are interested in multi-cloud support for Kubernetes.

Palladino: Kubernetes and containers enabled entire organizations to modernize their architectures and scale their businesses. As such, Kubernetes is well positioned to be the future of infrastructure for any modern workload. As more and more developers and organizations deploy Kubernetes in production, the more mature the platform will become to address a larger set of workloads running in different configurations, including multi-cloud. Multi-cloud is a real and pragmatic topic that every organization should plan for in order to continue being successful as the number of products and teams — with unique requirements and environments — keeps growing over time. Like a multicellular organism, the modern enterprise will have to adapt to a multi-cloud world in order to keep building scalable and efficient applications that ultimately deliver business value to their end users.

Kuo: The Kubernetes community has a special interest group for cloud providers to ensure that the Kubernetes ecosystem is evolving in a way that's neutral to all cloud providers. I've already seen many Kubernetes users choose Kubernetes for the benefit of multi-cloud. I envision that more enterprise users will join the Kubernetes community for that very same reason.

PREVIOUS ISSUES



68

.NET Core

In this eMag covering .NET Core, we will explore the benefits of .NET Core and how it can benefit not only traditional .NET developers but all technologists that need to bring robust, performant and economical solutions to market.

Chaos Engineering

67

This eMag will inspire you to dig deeper into your systems, question your mental models, and use chaos engineering to build confidence in your system's behaviors under turbulent conditions.

Tech Ethics

66

In an ideal world, devs would like to be ethical in their work but they ultimately don't consider it to be part of their responsibilities. This eMag sets out to understand why they might feel that way and whose job it is to take reasonable steps to ensure that tech products don't harm users or anyone else.

Domain-Driven Design in Practice

65

This eMag highlights some of the experience of real-world DDD practitioners, including the challenges they have faced, missteps they've made, lessons learned, and some success stories.