WHITE PAPER

Fast, Lean, and Unbreakable: Building Cloud Apps with Akka and Kubernetes

by Jonas Bonér | CTO and Co-Founder of Lightbend, Inc.

akka by Lightbend

Everything You Need to Know in 150 Words (TL;DR)

If you're short on time, and would like to digest the core points of this white paper quickly, then this page is for you.

- The use of containers and container orchestration—in particular, Docker and Kubernetes—have become ubiquitous and indispensable tools for deploying and managing distributed systems at scale.
- The need to manage ever-increasing volumes of data, reliably, in close to real-time, calls for Reactive data-driven services and streaming pipelines—forcing us to move:
  - from traditional stateless behavior-driven services and high-latency batch-oriented data processing,
  - to stateful data-driven services and low-latency stream-oriented data processing.
- Kubernetes is great for managing stateless services, but developing and operating stateful data-driven services on Kubernetes is challenging.
- End-to-end correctness requires that the infrastructure and application work together in maintaining data consistency and correctness guarantees.
- Akka—an open source programming model and fabric for the cloud—can do the heavy lifting of managing distributed state and communication, maintaining application consistency and delivery guarantees.
- Akka lets you run *stateful services* on Kubernetes *as if they are stateless*.

If you have time to read more, then our next section starts with examining the rise of containers and the prominence of Docker, especially. If you are familiar with containers and orchestration tools, however, then you can skip the upcoming two sections—covering the rise of containers and how to manage them at scale—and go directly to "Towards A Holistic View On System Design".

Table of Contents

Executive Summary

Why Stateful Services for Cloud Native Applications?

"The next frontier of competitive advantage is the ***speed*** *with which you can extract value from data."*
—Mike Gualtieri, VP, Principal Analyst Forrester Research, Inc.

Tools like Docker and Kubernetes have emerged as front-runners in the container and container orchestration space, used by many enterprises as the best way to simplify running distributed systems at scale. These enterprises are now focusing on the new gold—data—to differentiate their customer experience.

Here, speed counts: managing a firehose of never-ending data streams requires Reactive, data-driven services that can operate in near-real-time. This forces us to shift from a traditional *stateless* approach to data—where a database somewhere is constantly being hit with requests each time information is needed—to a *stateful* data-driven approach, where data is owned and accessed locally, with low latency and high-availability.

Kubernetes is a strong choice for managing distributed, and in particular stateless, services. But it has—given its low-level and foundational role in the infrastructure stack—little influence on the underlying business logic and operational semantics of your application. The next generation of cloud-native, stateful, data-centric services requires a complementary application architecture designed to serve high scalability, cloud-native environments.

In this white paper, we will show how Akka—an open source programming model and fabric for the cloud—can complement Kubernetes and do the heavy lifting of managing distributed state and communication, maintaining application consistency and delivery guarantees, while Kubernetes manages the underlying infrastructure. Akka lets you efficiently *run stateful services* on Kubernetes *as if they are stateless*, with minimal infrastructure requirements.

The Rise of Containers

If you are familiar with containers and orchestration tools then you can skip the upcoming two sections which talks about the rise of containers and how to manage them at scale.

The concept of computer virtualization is quite old and has been used in different forms for decades. The previous generation tools were based on VM hypervisors—like Xen, KVM, and Hyper-V—and were emulating the hardware, running a full OS in each instance, which made them quite heavyweight in terms of resources. Containers address this by instead of virtualizing hardware they run on a shared Linux instance, which makes them a lot more lightweight and allows you to run many more of them on the same underlying hardware while still providing the same isolation and control.

The recent trends of DevOps, **continuous integration and delivery** of software (CI/CD), and the move to Cloud Computing—where you pay for the resources you use rather than making capital expenditures on hardware that isn't 100% utilized at all times—have driven the need for more lightweight and easily managed containers and have helped popularize containers in the industry at large.

Containers can be dated back to the year 2000 and FreeBSD Jails. Since then we have seen many different projects—such as **LXC** (Linux Containers) and **OpenVZ**—pushing the technology forward.

Docker is an open source container platform built on LXC, first released in 2013, that has grown to become immensely popular the last 4-5 years and is now considered the de facto standard in containers. It has a whole suite of developer and management tools around it that helps packaging services or applications in a straightforward and standardized way.

Some of the things that Docker can help you with include:

- Uniformity of infrastructure and application instance management.
- Allowing for polyglot programming, building services in different languages and platforms.
- Adding and removing resources as needed, on demand.
- Rapid deployments and continuous integration through the management of deployment pipelines.

Some of the things that Docker cannot help you with:

- Managing container life-cycle in a uniform fashion.
- Automating application work scheduling, scalability, resilience, and self-healing.
- Consolidated management of configuration, load-balancing, security and authentication/authorization.

Managing Containers at Scale

While running and managing a handful of containers using CLI tools and shell scripts can work reasonably well, as soon as you need to scale the system of containers it quickly becomes overwhelming.

Now you need a better way to manage all these containers—starting, stopping, upgrading, and replicating them, manage security and authentication/authorization, in a holistic fashion. This is the job for the container orchestrator.

There are several out there on the market with the most popular ones being **Mesos**, **AWS ECS**, **Docker Swarm**, **Nomad**, and **Kubernetes**. The last couple of years Kubernetes has risen in popularity and use to become the de facto standard in container orchestration and is what we will focus on in this article.

Kubernetes was created by Google, influenced by its internal system **Borg**, and was released to the public as an open source project in mid-2014. Since then the list of contributing companies has grown extensively which has helped to turn it into a vibrant ecosystem.

Kubernetes and its ecosystem of tools make it easier to manage large numbers of Docker containers in a standardized and uniformed way and can help you with:

- Managing container life-cycle—through the grouping of containers, so-called pods, including starting, stopping, upgrading, and replicating pods.
- Automation of scheduling, scaling, and recovery of container pods.
- Automation of deployment pipelines for CI/CD.
- Consolidated management of configuration management, routing, load-balancing, security, authentication/authorization, and more.

While we believe that Kubernetes is the best way to manage and orchestrate containers in the cloud, it’s not a cure-all for programming challenges at the application level, such as:

- The underlying business logic and operational semantics of the application.
- Data consistency and integrity.
- Distributed and local workflow and communication.
- Integration with other systems.

Kubernetes has been central in the recent cloud-native application development trend and is one of the projects in the **CNCF** (Cloud Native Computing Foundation), alongside gRPC, Prometheus, Envoy, and many others.

But what does “cloud-native” mean? CNCF **defines it** as:

> *“Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.*
>
> *These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.”*

As we will see, there is more to cloud-native—the art of building scalable, resilient, and observable distributed systems—than orchestration of containers.

Towards a Holistic View on System Design

One question that we frequently hear is: "Now that my application is containerized, do I still need to worry about all that hard distributed systems stuff? Will Kubernetes solve all my problems around cloud resilience, scalability, stability, and safety?" Unfortunately, no—it is not that simple.

In the now classic paper "**End-To-End Arguments In System Design**" from 1984, Saltzer, Reed, and Clark discusses the problem that many functions in the underlying infrastructure (the paper talks about communication systems) can only be completely and correctly implemented with the help of the application at the endpoints.

Here is an excerpt:

> *"This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument,* ***suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level.*** *[...]*
>
> ***Therefore, providing that questioned function as a feature of the communication system itself is not possible*** *(sometimes an incomplete version may be useful as a performance enhancement). [...]*
>
> *We call this line of reasoning against low-level function implementation the end-to-end argument."*

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.

End-to-end correctness, consistency, and safety mean different things for different services. It's totally dependent on the use-case, and can't be outsourced completely to the infrastructure. To quote[1] Pat Helland: "The management of uncertainty must be implemented in the business logic."

In other words, a holistically stable system is still the responsibility of the application, not the infrastructure tooling used. Some examples of this include:

- What constitutes a successful message delivery? When the message makes it across the network and arrives in the receive buffer on the destination host's NIC? When it is relayed through the network stack up to the application's networking library? When it is application's processing callback is invoked? When the callback has completed successfully? When the ACK has been sent and gone through the reversed path as the message and consumed by the sender? The list goes on, and as you can see, it all depends on the expected semantics and requirements defined by the use-case: application-level semantics.
- Flow control—where fast producers can't overload slow consumers—is all about maintaining end-to-end guarantees, and requires that all components in the workflow participate to ensure a steady and stable flow of data across the system. This means that business logic, application-level libraries, and infrastructure all collaborate in maintaining end-to-end guarantees.

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—we believe that Akka paired with Kubernetes can be that solution.

We will discuss this in more depth soon but let's first look at a—somehow unfortunate—trend in the cloud-native world of application development: the strong reliance on stateless protocols.

1 This quote is from Pat Helland's highly influential paper "**Life Beyond Distributed Transactions**", which is essential reading.

The Unfortunate Trend of Stateless Protocols in Cloud-Native Apps

Most of the developers building applications on top of Kubernetes are still mainly relying on **stateless protocols** and design. They embrace containers but 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: literally, managing the *state* of 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. But 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 becoming more important than ever.

Many applications can't afford the round-trip to the database for each data access or storage but 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 and throughput—before it is ready to be stored by the persistence layer.

This requires a different way of looking at the problem, a different design, one that puts data at the center, not on the sidelines.

Finding Success With Data-Driven Stateful Services

Data is the new gold, becoming more and more valuable every year. Companies are trying to capture "everything"—aggregating and storing away ever-increasing volumes of data in huge data lakes with the hope of gaining unique insights that will lead to a competitive advantage in the race to attract more customers. But running nightly jobs processing all this data through traditional batch processing (using Hadoop) often turns out to be too slow.

Nowadays we need to be able to extract knowledge from the data as it flies by, in close to real time, processing never-ending streams of data, from thousands of users, at faster and faster speeds—using low-latency data streaming tools like **Akka Streams**, **Apache Spark**, and **Apache Flink**.

In order to do this efficiently, we need to take back the ownership of our data, model our systems with data at the center[2]. We need to move from high-latency, Big Data, and traditional stateless behavior-driven services, to low-latency, **Fast Data**, and stateful data-driven services.

Some of the benefits we will get are:

- *Data locality*—having the data close to the context where it's being used leads to lower latency, less bandwidth consumption, and higher throughput.
- More models for *highly available data consistency*—stateful sticky sessions open up for new consistency models previously only available to centralized systems.[3]

2 By for example leveraging design techniques like **Event Storming** and Events-first **Domain Driven Design**.

3 These models include 'causal', 'read-your-writes', and 'pipelines random access memory'. See Caitie McCaffrey great talk Building Scalable Stateful Services more details, and other benefits of being stateful.

A core infrastructure piece in this architecture is the *event log*, functioning as:

- The data distribution, coordination, and communication backbone—helping the application to maintain order, data consistency, reliable communication, and data replication.
- The *service of record* and *source of truth* for the events in the application—often using a pattern called ***event sourcing***, which means that every state change is materialized as an event/fact and stored in the event log, on disk, in order. Here, the state is persisted in-memory (so-called ***memory image***) and rehydrated from disk on failure recovery, replication, and migration.

The Challenges of Orchestrating Stateful Services

Container orchestration is built around the idea that containers are cheap and disposable. It allows you to deploy tens or hundreds of identical replicas on-demand for elasticity, and if one of them fails it simply redirects traffic to another replica, resuming business with very little downtime.

This pattern 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 a unique 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 which the replication is done from 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. But if we want to manage all of your infrastructure in a uniform fashion using Kubernetes then what do we do?

At this time, 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[4] that is maintained across restarts (even after it's been rescheduled to another physical machine). As a result, it is now possible—but still quite challenging[5]—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.

Let's take a look at one of these tools: Akka.

4 A limitation has been that disks need to be remote and therefore managed by external services. The good news for latency sensitive services is that support for **local disks** is coming soon (currently in beta)—but keep in mind that this model can reduce availability.
5 StatefulSets don't solve all of the challenges of distributed stateful applications. Stateful use-cases can be quite different, which makes it hard to solve in a generic fashion, and is why a custom controller—so-called **Operator**—is usually necessary to actually manage the application, as **described in this article**.

In Need of a Programming Model for the Cloud

A single service is not that useful—services come in systems, and are only useful when they can collaborate, as systems. As soon as they start collaborating, we need ways to coordinate across a distributed system, across a network that is inherently unreliable.

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 yourself is very very hard.

Akka—an open source project created in 2009—was built to address these challenges from the start, 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 coined.

Akka is based on the **Actor Model** and built on the principles outlined in the **Reactive Manifesto**, which defines Reactive Systems[6] as a set of architectural design principles that are geared toward meeting the demands that systems face—today and tomorrow.

Leverage Reactive Design Principles

These principles are most definitely not new; they can be traced back to the '70s and '80s and the seminal work by Jim Gray and Pat Helland on the Tandem System, as well as Joe Armstrong and Robert Virding on Erlang. However, these pioneers were ahead of their time, and it was not until the past 5-10 years that the technology industry was forced to rethink current best practices for enterprise system development and learned to apply the hard-won knowledge of the Reactive principles to today's world of multicore architectures, Cloud Computing, and the Internet of Things.

We believe that these principles for building distributed applications are not only highly complementary with the cloud-native philosophy and the Kubernetes ecosystem of tools, but essential for making the most out of it. Akka—being a programming model used to build the application in, as well as the runtime to maintain its guarantees and semantics—complements Kubernetes very well in the pursuit of end-to-end correctness and safety in cloud-native applications.

Let Akka Do the Heavy Lifting

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.

6 For a good primer on Reactive Systems vs Reactive Programming see **this article**.

What Akka provides:

- **Guaranteed message delivery**—through durable journaling, at-least-once message delivery, and deduplication.
- **Persistent state**—state persisted in memory using **event sourcing**, which is rehydrated from disk when the entity is created, recovered from the failure, or migrated (even **replicated across data-centers**, if needed), avoiding the object-relational impedance mismatch.
- **Highly fault-tolerant stateful service management**—using decentralized peer-to-peer masterless **clustering** (through gossip and consensus protocols) and failure detectors, allowing for self-healing services with no single point of failure.
- **The Actor Model**—which gives you a unified model for high-performance **distributed and local communication**, that is location transparent (maintaining consistent semantics for both local and distributed), supporting uni- and bi-directional messaging, publish-subscribe, and streaming semantics.
- **Cluster sharding of entities**—automatic state replication, dynamic rebalancing, re-partitioning of entities, and location transparent routing (using consistent hashing and other algorithms).
- **Strong eventual consistency**—highly scalable and resilient management of global cluster state through automatic replication (and merging) of **CRDTs**.
- **Ultra low-latency**—including ingestion, transformation, enrichment, merging, and splitting of data streams using **stream processing**.
- **Back-pressure and flow control**—through **Reactive Streams**-compliant libraries.
- **A wide range of supported protocols for client communication**—such as **HTTP** and **gRPC**—and system integration—through the **Alpakka project**.

Examples of where Akka Cluster can be very helpful are when your application is complicated enough that simple load balancer rules can't distribute the work across computing resources in an efficient and sensible manner, or there is more coordination required than going to a central data store per request.

Let's now take a look at how Akka and Kubernetes can complement each other, compose, and operate in concert, at different levels in the software stack.

Getting the Best of Both Worlds: Kubernetes With Akka

As discussed above, Kubernetes alone is not capable in maintaining holistic end-to-end *application* correctness and safety, but need a way to work efficiently in concert with the application—especially when it comes to stateful applications and services. Akka—with its distributed programming model and runtime for cluster and state management—has proven to be an ideal bridge, mediator, and maintainer of these guarantees and semantics. But how should one think about these two models for distributed systems and how can they be composed?

Composing Kubernetes With Akka

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

Looking at it this way, Kubernetes and Akka compose very well, each being responsible for a different layer and function in the application stack:

- Akka is the *programming model* to write the application in, and its supporting *runtime*—helps to manage business logic; data consistency and integrity; operational semantics; distributed and local workflow and communication; integration with other systems, etc.
- Kubernetes the *tool for operations* to manage large numbers of container instances in a uniform fashion—helps managing container life-cycle; versioning and grouping of containers; routing communication between containers; managing security, authentication, and authorization between containers, etc.

Here is an illustration of the overall architecture of a composed Akka and Kubernetes stack, each tool working its distinct layer, maintaining either application or infrastructure-level concerns, but also in concert, maintaining the application's end-to-end guarantees:

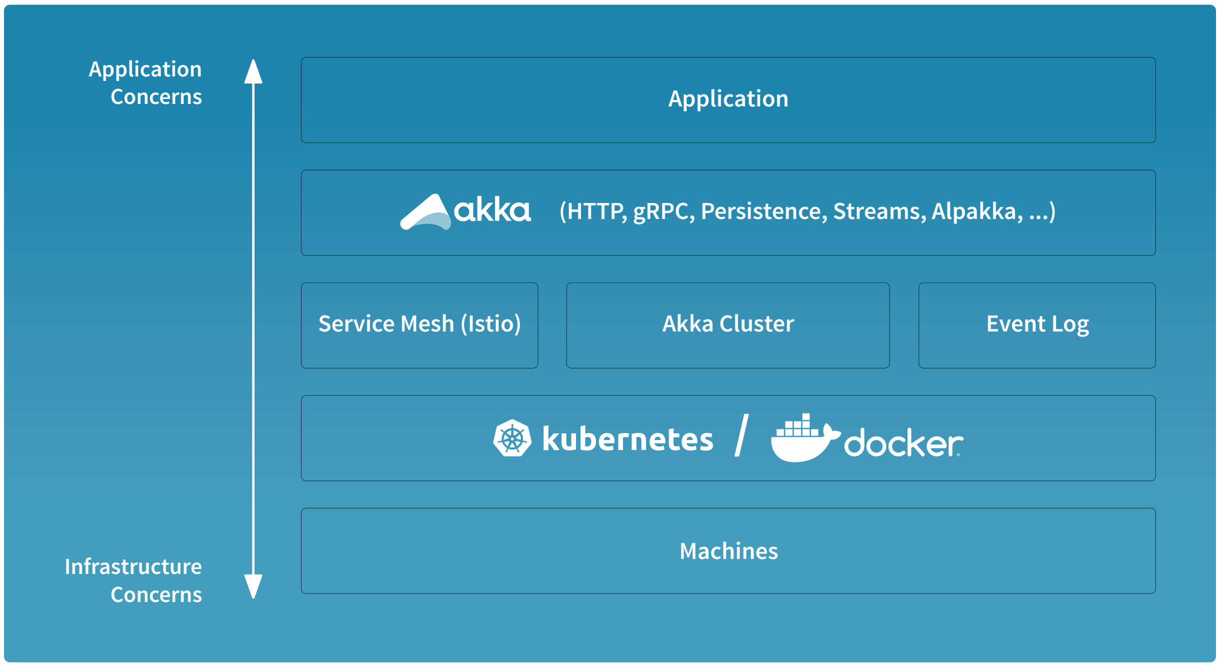

Figure 1

Here, the **service mesh** composes nicely with Akka, managing all RPC communication such as Akka gRPC or Akka HTTP endpoints, while Akka is used for other communication patterns, like pub-sub event-based messaging and point-to-point messaging.

Fine-Grained vs. Coarse-Grained Scalability and Resilience

What about scalability and resilience, why would I need two ways of managing that? It's true that both Kubernetes and Akka helps to manage resilience and scalability, but at distinct granularity levels in the application stack. The end-to-end argument applies again and we have a lot to gain by composing the two models:

- 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.
- 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.

In essence: Kubernetes' job is to give your application enough compute resources, getting 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 resource that has been given to it.

Here—as seen in the illustration below—the logical concept of a "service" is made up of a set of Akka entities (actors), managed in a decentralized peer-to-peer fashion, as one single Akka cluster. This cluster of entities spans multiple Kubernetes pods[7] and is usually a mix of replicas of different entity types—most often with persisted state—all working together, growing and shrinking on demand, and covering up for each other's failures. The Akka cluster is deployed to Kubernetes as a **Headless Service** and is utilizing **Akka Cluster Bootstrap**[8] to coordinate cluster formation.

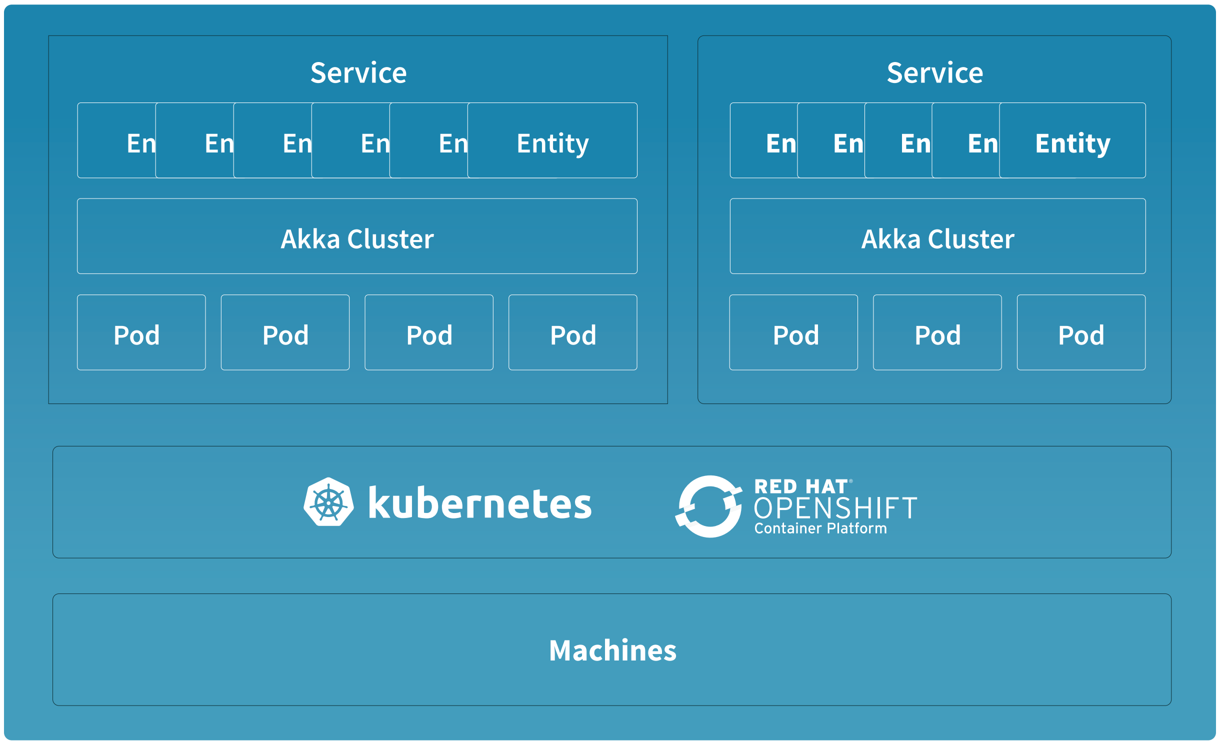

Figure 2

Maintaining these guarantees in a software layer above the containers themselves can be very efficient in terms of minimizing resources and allows for the service to self-heal without external intervention by the underlying infrastructure.

Also, if you try to let the infrastructure alone provide your guarantees, then your infrastructure is a part of your application—leading to a lack of separation and in essence infrastructure lock-in.

Looking at a Concrete Example

To make things a bit more concrete, let's take a look at an example: migration and recovery of a persisted stateful entity on container failure.

7 They can, if needed, even span multiple data centers while retaining the strong consistency of the entity and its underlying durable event log.
8 Akka Cluster Bootstrap implementations include DNS, Kubernetes, Marathon, AWS EC2 or ECS, and HashiCorp Consul.

If we implement the stateful entity[9] using Akka then we make the entity a **persistent actor**. Being persistent means that it is backed by a durable replicated *event log*[10]—using the *event sourcing pattern* (as discussed above). Once this is implemented, the entity will run **sharded** on a highly-available decentralized Akka cluster of nodes.

This allows Akka to ensure that in the event of node failure:

1. The failure detector will trigger and the cluster decides the best course of action.
2. The entity will be restarted on another—healthy—node.
3. The event log will be replayed—from the latest snapshot and forward—bringing the entity fully up to speed, to the state it was in just before the failure.
4. The clustered entities will be rebalanced on the remaining nodes in the cluster.
5. Client communication is seamlessly and automatically resumed.

Kubernetes' job is to eventually reboot the failed node or bring up a new one in its place—and when that is done, Akka cluster will automatically make use of the new node and rebalance the entities as needed.

In Conclusion

To sum things up: *Akka lets you run stateful services on Kubernetes as if they are stateless*. No fuzz, no StatefulSets, no Operators, no messing around with remotely mounted disks, just using the infrastructure—and guarantees—that Kubernetes gives stateless services.

This allows us to leverage Kubernetes for what it is great at while relying on Akka for the management of state and application-level guarantees.

9 In Akka you would normally have one microservice consist of 1-N number of persistent entities, all backed by a single Akka cluster.
10 The storage medium for the event log in **Akka Persistence** is pluggable, with a long **list of storage backends** ranging from SQL to distributed NOSQL databases—the default being Cassandra.

akka
by Lightbend

Akka by Lightbend (**@Lightbend**) is used by many of the world's largest brands as the foundation for their multi-cloud, mission-critical applications. Through Akka, the industry's most powerful distributed application platform, Lightbend provides scalable, high-performance microservices frameworks and streaming engines for building data-centric systems optimized to run on cloud native infrastructure.

For more information, visit **www.lightbend.com**