



Lightbend

@deanwampler



The SMACK Stack: Emerging Fast Data and Microservice Architectures

©Dean Wampler 2014-2016, All Rights Reserved

Photos, Copyright (c) Dean Wampler, 2014-2016, All Rights Reserved, unless otherwise noted. From the Ansel Adams Wilderness and Yosemite National Park, both in the Sierra Nevada Range, California, USA.

Other content Copyright (c) 2015-2016, Dean Wampler, but is free to use with attribution requested.

<http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>

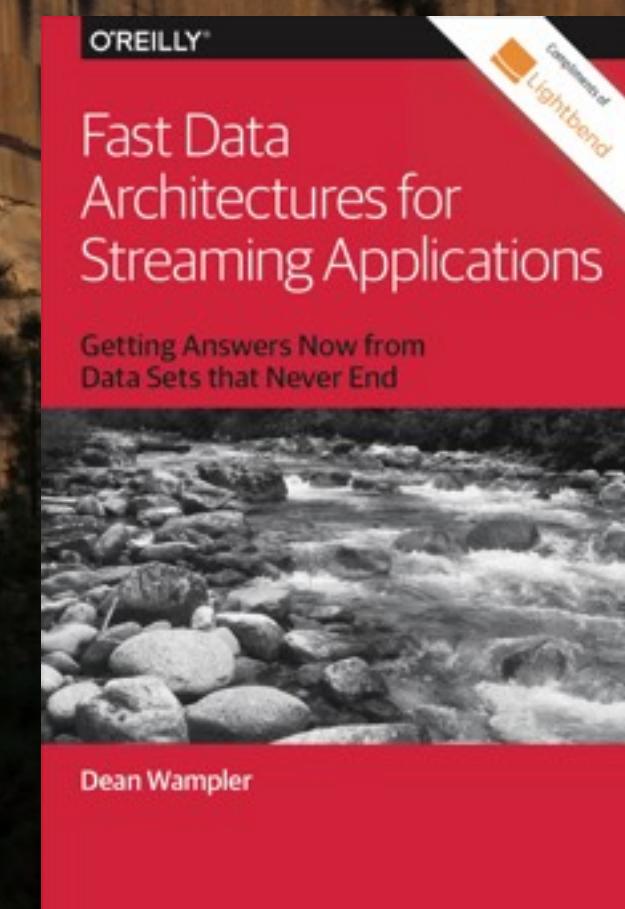
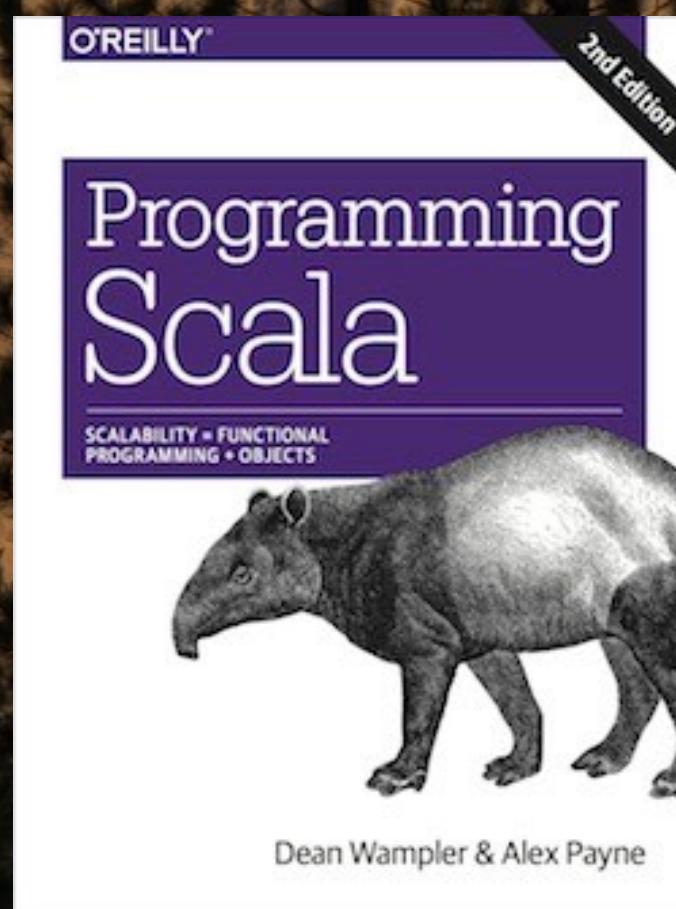
photo: Half Dome in morning light, from Little Yosemite Valley campground. You're looking at several thousand vertical feet of rock!

Dean Wampler

dean@lightbend.com

polyglotprogramming.com/talks

@deanwampler





About the acronym...

SMACK



Jamie Allen
@jamie_allen



Following

The SMACK stack for fast data: [@ApacheSpark](#),
[@ApacheMesos](#), [#Akka](#), [#Cassandra](#) & [#Kafka](#).
Name coined by [@TheOTown](#)

RETWEETS

37

LIKES

29



7:19 AM - 25 Jun 2015



37



29

...

Apparently coined by Oliver White, in the Lightbend Marketing team. Jamie's tweet is the first mention of it.



Skills Matter

@skillsmatter



Following

Final Call: Join us on 24th Oct for
@deanwampler's talk on why #SMACK is a good
stack for #FastData buff.ly/2dDX08p #BigData
#spark



Yesterday, I retweeted this reminder of today's talk...



Jarrod Brockman

@DgtlNmd



Following

@skillsmatter @deanwampler #SMACK is a very unfortunate hashtag.



To which Jarrod Brockman replied this gif. Click this link to see it.
<https://twitter.com/DgtlNmd/status/790113697545027584>



Jarrod Brockman

@DgtlNmd



Following

@skillsmatter @deanwampler #SMACK is a very unfortunate hashtag.



Here's a screen capture after one smack...



Dean Wampler

@deanwampler

@DgtlNmd @skillsmatter Not to mention the association with Heroin, but I didn't mention it...

3:55 AM - 23 Oct 2016

📍 City of London, London



...



Jarrod Brockman
@DgtlNmd



Following

@deanwampler @skillsmatter it came to mind
but I did not mention it either. 😊

4:09 AM - 23 Oct 2016



...

Where are we going?



10

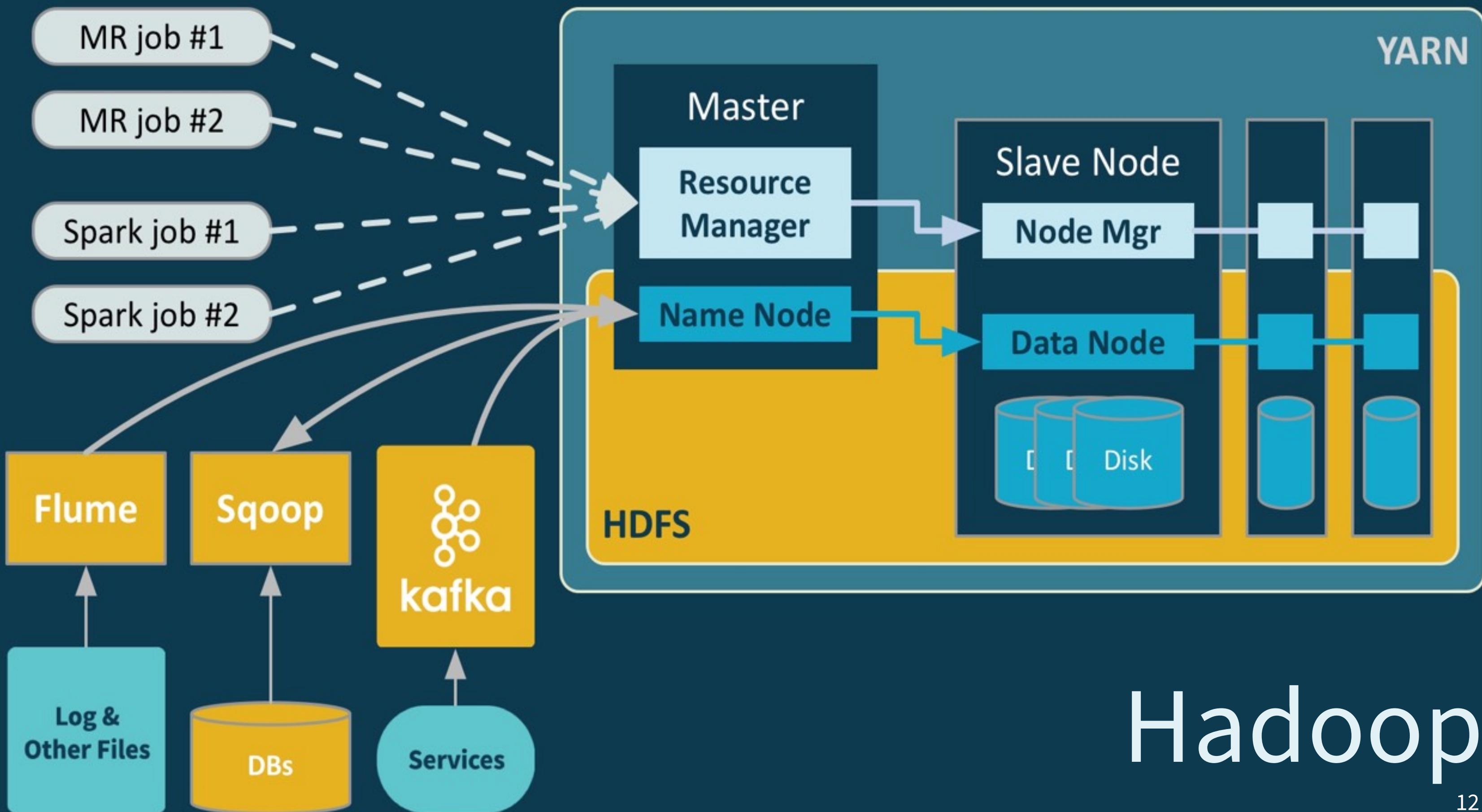
Let's start with two areas of change right now for architectures: data-centric systems and general-purpose systems.
photo: Pack train below Donahue Pass, Ansel Adams Wilderness

A wide-angle photograph of a rocky mountain landscape. In the foreground, a hiker wearing a large backpack and trekking poles walks away from the camera along a rocky path. The terrain is covered in large, light-colored granite boulders and patches of green grass and small yellow flowers. In the background, more mountain ridges are visible under a blue sky with scattered white clouds.

Big Data

11

Starting with data-centric systems, the Big Data world is now fairly mature.
photo: Climbing to Donahue Pass, Ansel Adams Wilderness

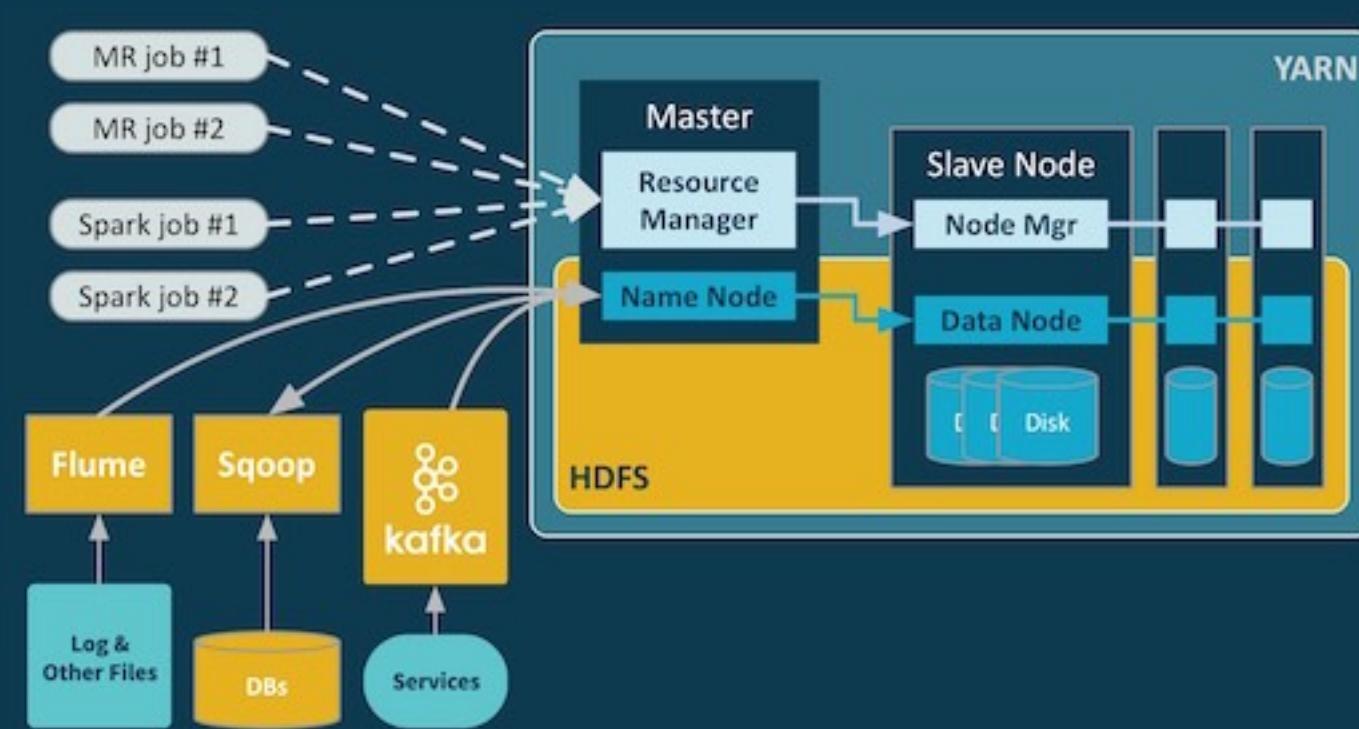


Hadoop

12

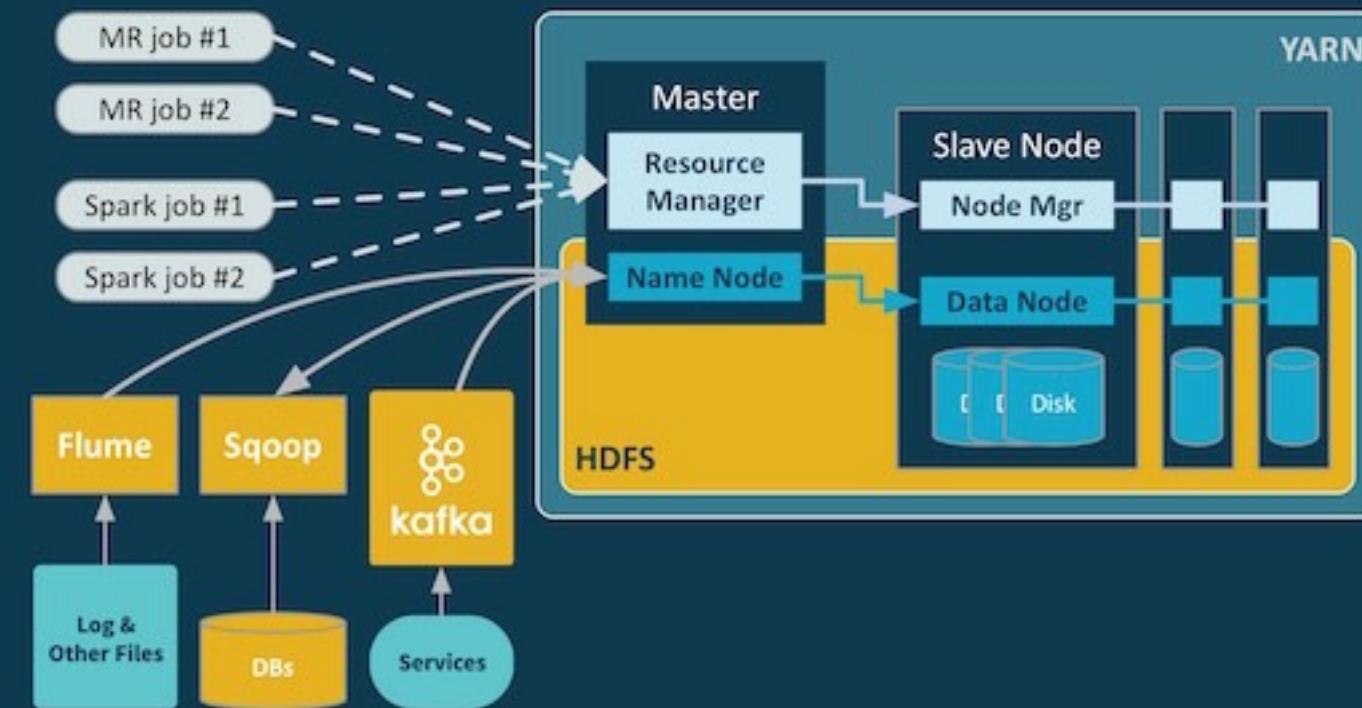
Hadoop is the dominant, general purpose architecture for big data systems. NoSQL Databases are more specialized big data systems, which we won't consider further. Three major components: 1) a distributed filesystem (storage), 2) a processing engine (MapReduce and Spark, with jobs and constituent tasks run by services in light blue, 3) and YARN, the service that manages resources and schedules jobs.

Hadoop



- Very large data sets (HDFS)
- Batch jobs: “Table scans”
- Job durations: minutes to hours
- Latencies: minutes to hours

YARN



- Resources are *dynamic*
 - CPU cores & memory
 - Global, top-down scheduler
 - Best for “compute” jobs

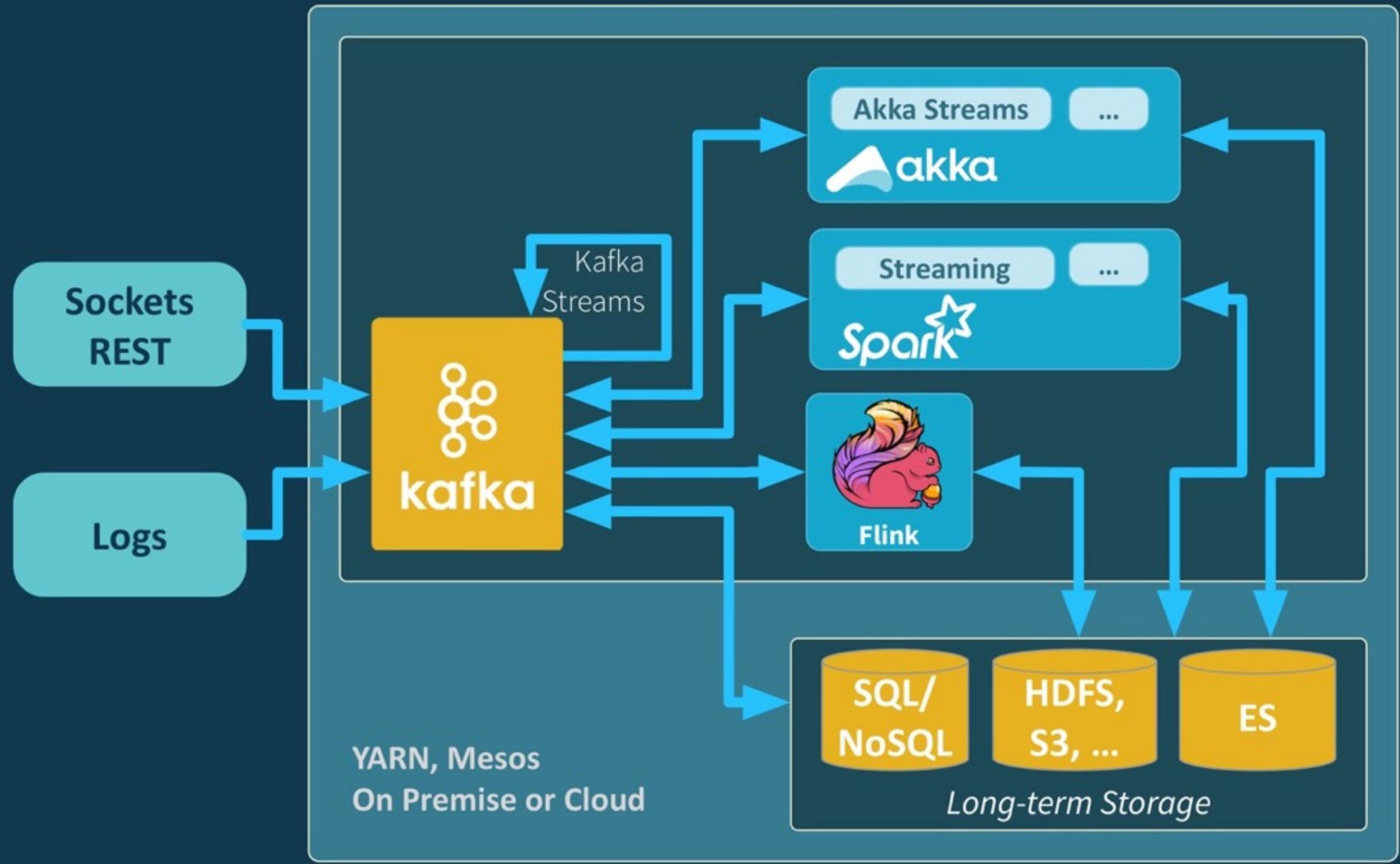
We'll compare with Mesos. YARN doesn't yet manage disk space and network ports, but they are being considered. Scheduling is primarily a global concern and uses the Fair Scheduler, Capacity Scheduler, etc. It's ill-suited to manage things that aren't like MapReduce or Spark. It can't even manage HDFS resources, although attempts are being made to address this limitation: <http://hortonworks.com/blog/evolving-apache-hadoop-yarn-provide-resource-workload-management-services/>

A wide-angle photograph of a mountainous landscape. In the foreground, a river flows from the bottom right towards the center-left, its surface reflecting the surrounding environment. The banks of the river are rocky and covered with small, green shrubs and a few small pine trees. Behind the river, the terrain rises into large, rugged mountains. These mountains are composed of light-colored, possibly granite, rock and are sparsely vegetated with patches of green grass and small pine trees. Some snow is visible on the higher peaks and in the upper reaches of the mountains. The sky above is a clear blue with a few wispy white clouds.

Streaming

15

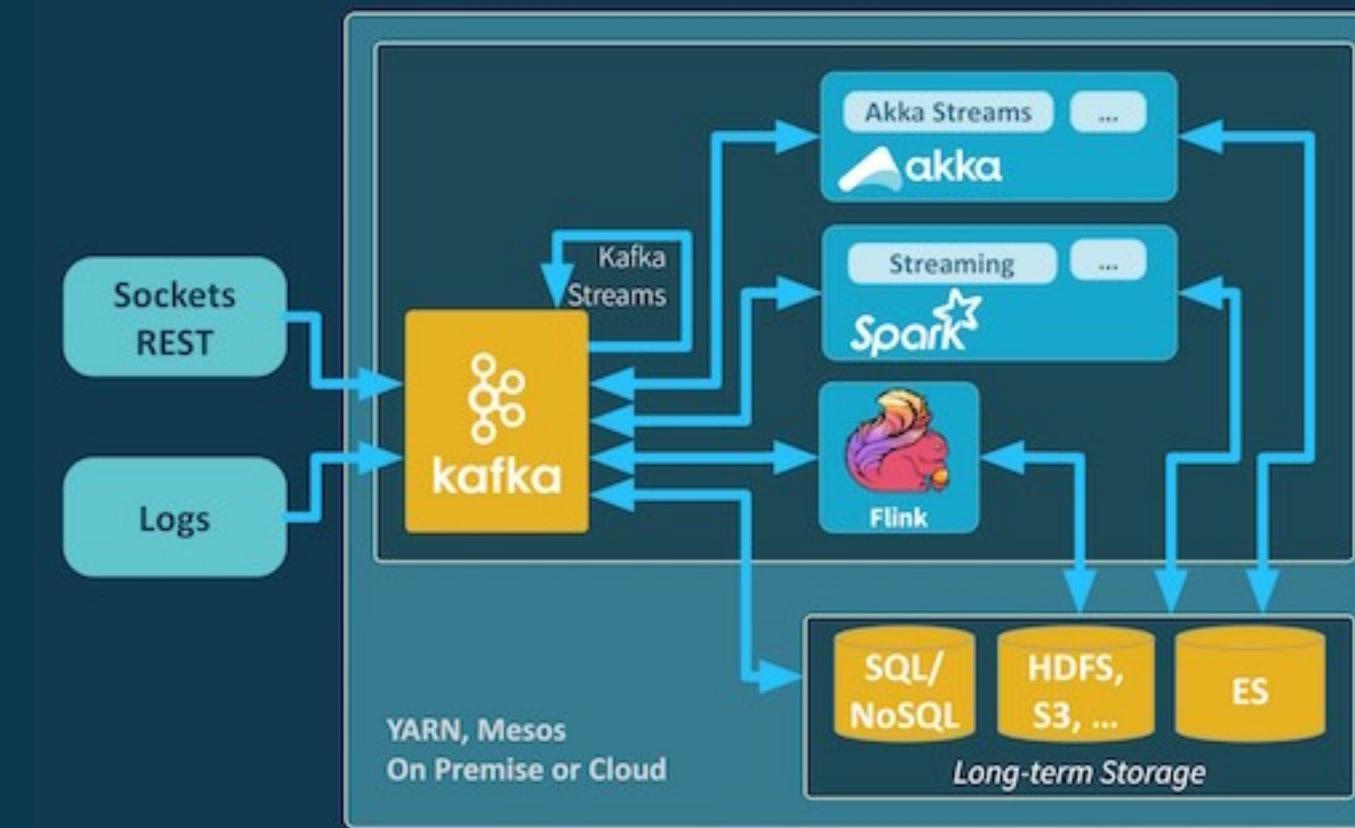
This is the new directory for data-centric systems.
photo: Lyell Creek below Donahue Pass, Yosemite National Park



16

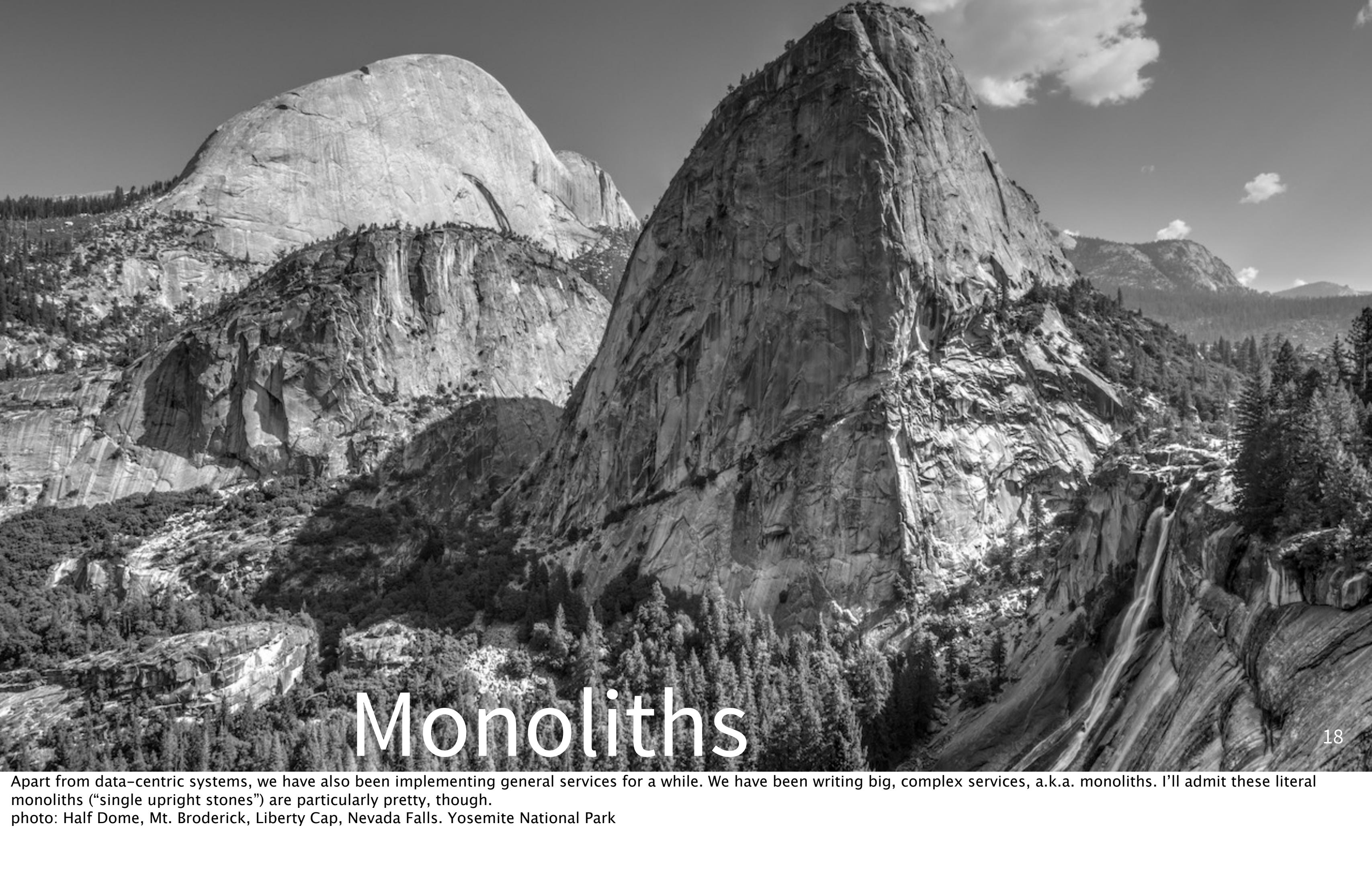
A partial set of possible, sometimes competing streaming engines. The SMACK stack “mentions” Spark, Mesos, Akka, Cassandra, and Kafka. We’ll talk about the elements here in more detail later, but notice a few are the same from the Hadoop diagram, including HDFS, Spark, and YARN.
 The major components are 1) storage (Kafka, for durability and temp. storage of inflight data, HDFS, databases, and Elastic Search), 2) streaming compute engines (Akka, Spark, and Flink, plus many more not shown), and 3) a resource manager & scheduling system (YARN, as before, but also Mesos and cloud services).

Streaming



- Never ending sequences (Kafka)
- Incremental processing
- Job durations: forever!
- Latencies: μ secs - seconds

Data “sets” are sequences that could go forever. Instead of being rooted in file system storage (although that’s still present...), a message queue/data bus, especially Kafka is the core. In streaming, they are processed within seconds, in “real-time” event systems possibly down to microseconds. These jobs could run forever, although in practice they are often replaced with updated jobs (or they crash and have to be restarted...)

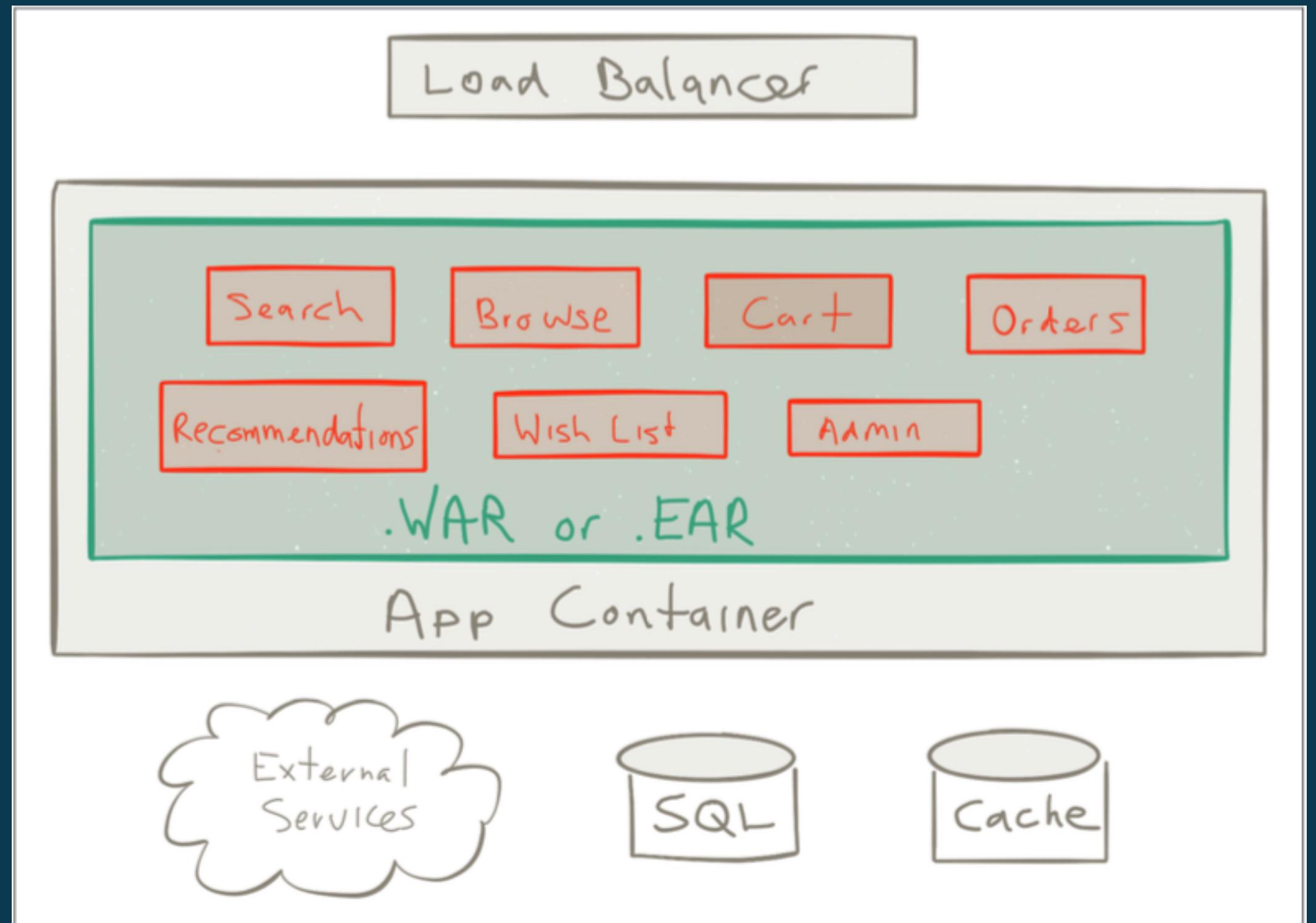


Monoliths

18

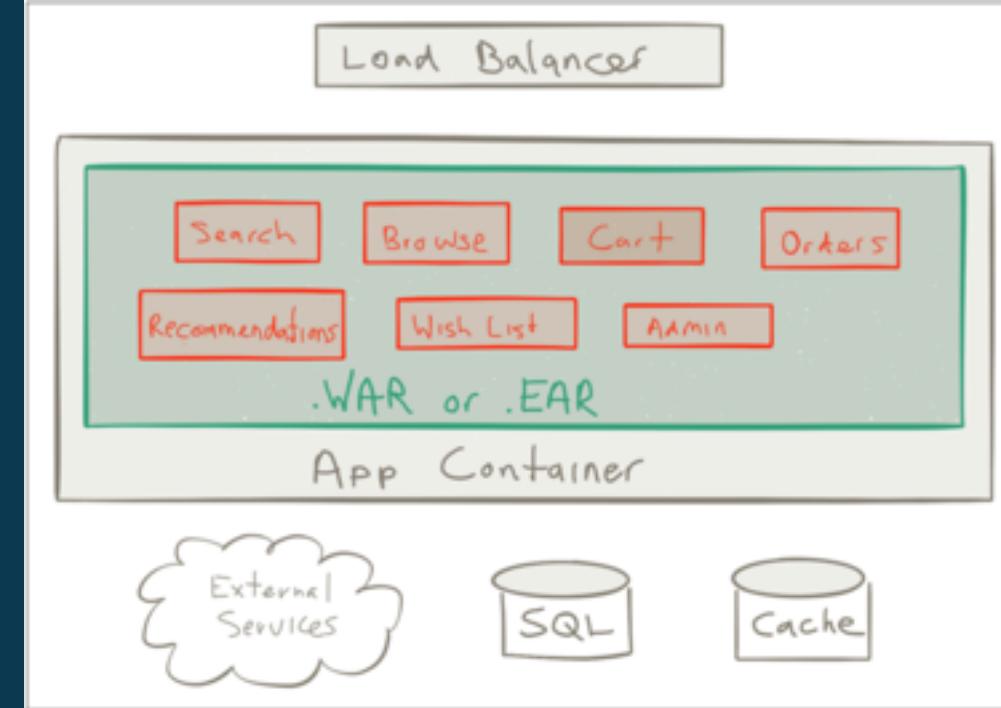
Apart from data-centric systems, we have also been implementing general services for a while. We have been writing big, complex services, a.k.a. monoliths. I'll admit these literal monoliths ("single upright stones") are particularly pretty, though.

photo: Half Dome, Mt. Broderick, Liberty Cap, Nevada Falls. Yosemite National Park



A classic, JEE approach for “macroscopic” services (monoliths). App Containers are too heavyweight to run one per microservice. So, you have a lot of concerns and dependencies in one place. Also, it’s a synchronous model, so throughput and other benefits of asynchrony are not natural outcomes.
Drawing by Kevin Webber in Reactive Microservices Architecture, Jonas Bonér, O'Reilly Media, 2016

Monoliths



- Tangled responsibilities
- Difficult, infrequent deployments
- Durations: months to years
- Latencies: μ secs to seconds

Monoliths mean fewer things to manage and intraprocess function calls are faster than interprocess communications (the picoseconds), but they tend to become bloated with tangled dependencies, making them fragile and difficult to engineer. Hence, deployments are often “big bang” and too painful to do frequently.

A wide-angle photograph of a mountainous landscape. In the foreground, there is a field of large, light-colored granite boulders of various sizes, some partially covered by low-lying green shrubs and small patches of grass. In the middle ground, the terrain continues with more boulders and shrubs. The background features majestic, rugged mountains with sharp peaks. One prominent peak in the center-left has patches of white snow clinging to its upper slopes. The sky above is a clear, vibrant blue with a few wispy white clouds.

Microservices

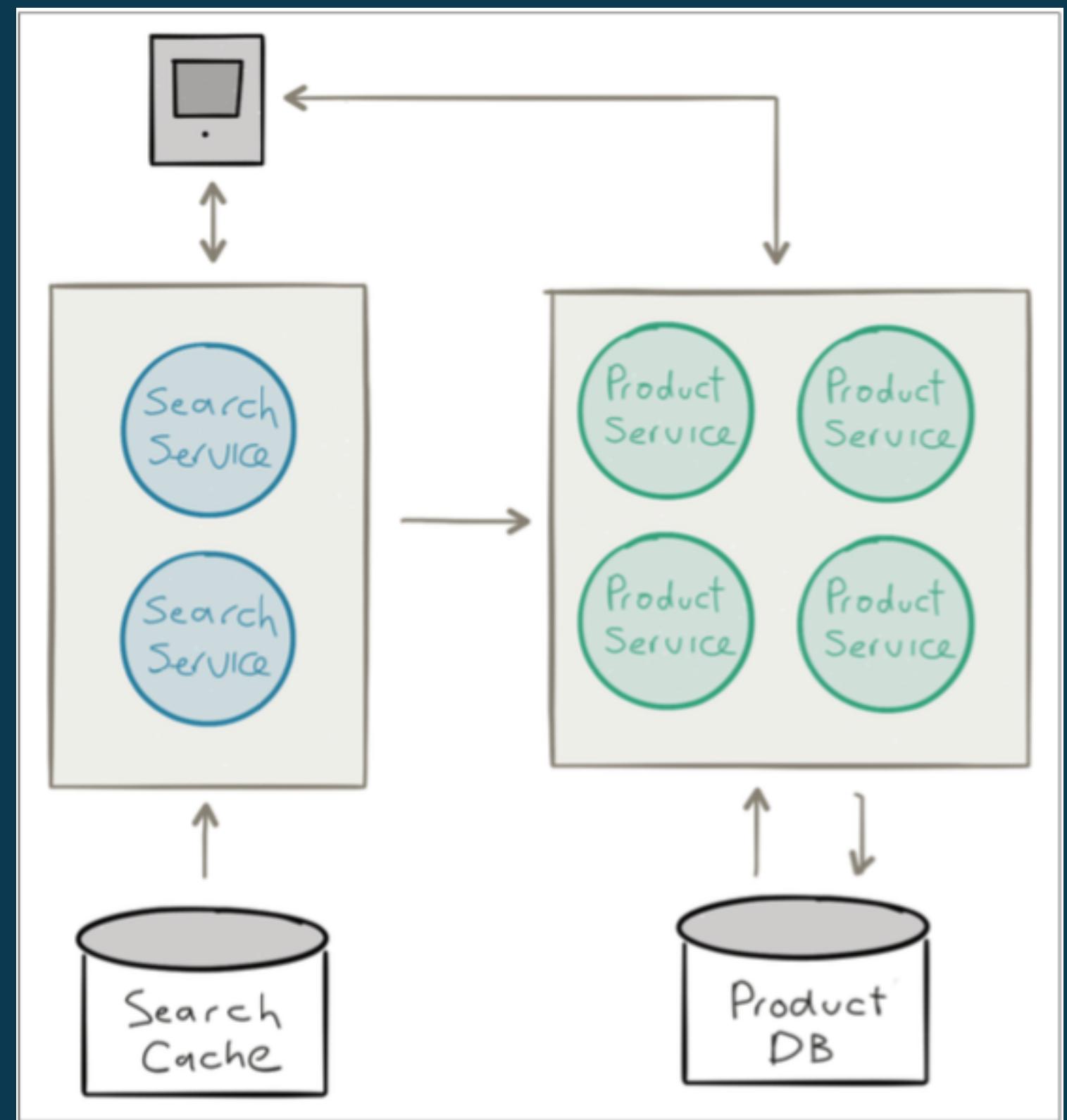
21

Smaller rocks...
photo: Mt. Lyell and boulders just North of Donahue Pass.



22

<https://twitter.com/alexcrdean/status/790494111396691968>



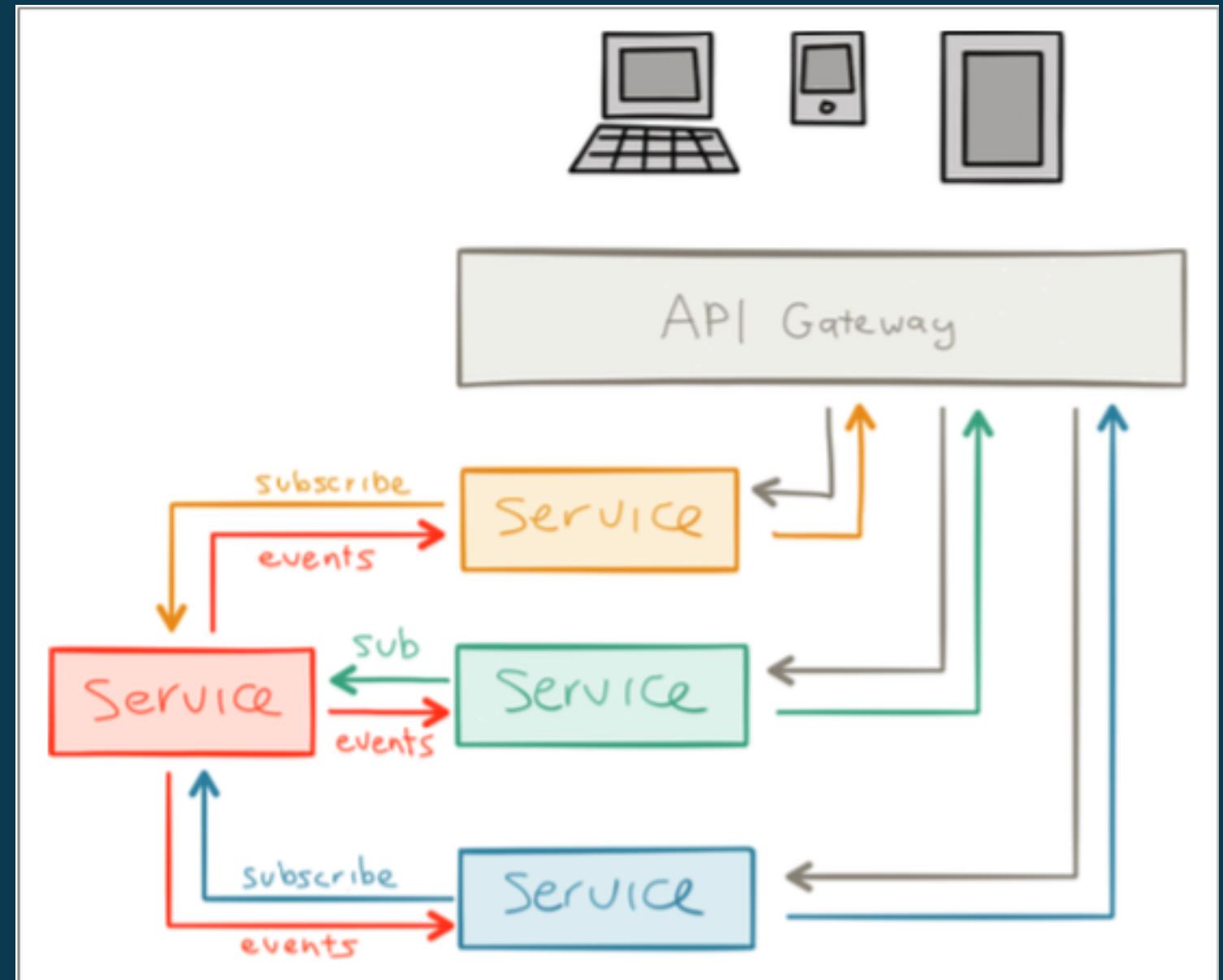
Microservices try to do one thing and do it well. They must manage their own data, because a shared data store is a monolith in disguise.
Drawing by Kevin Webber in Reactive Microservices Architecture, Jonas Bonér, O'Reilly Media, 2016

Reactive Microservices Architecture

Design Principles for Distributed Systems



Jonas Bonér



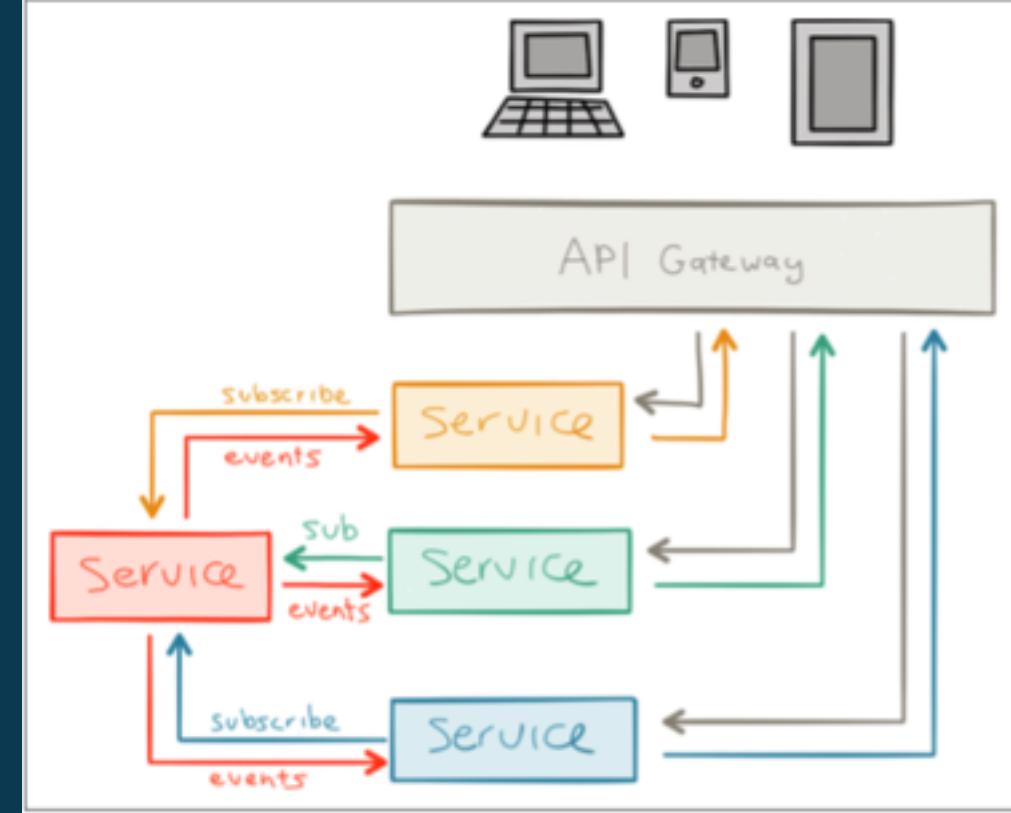
It's also common to provide a uniform API abstraction to clients which hides the independent APIs of the underlying sources and also provides a level of indirection, so it's easier to swap out instances of these services.

Drawing by Kevin Webber in *Reactive Microservices Architecture*, Jonas Bonér, O'Reilly Media, 2016



Microservices

- Each does one thing
- Embrace Conway's Law
- Message driven & asynchronous
- Durations: minutes! to forever
- Latencies: higher than func. calls



They might be smaller in memory and CPU footprint, but only as a side effect of their focus on doing one thing. They communicate with each other through messages and should be asynchronous to maximize throughput, although this isn't always best. Because they do one thing and have a very clear boundary and interfaces to other services, it's easy to organize their development into teams, essentially a Reverse Conway's Law. This makes it easier to evolve and deploy them independently of other microservices, too. Very small "dockerized" services might last just a few minutes, but could run a very long time. A big drawback of microservices is the longer communication latency of calls between them compared to function calls in the same process.



Spark Mesos Akka

Cassandra

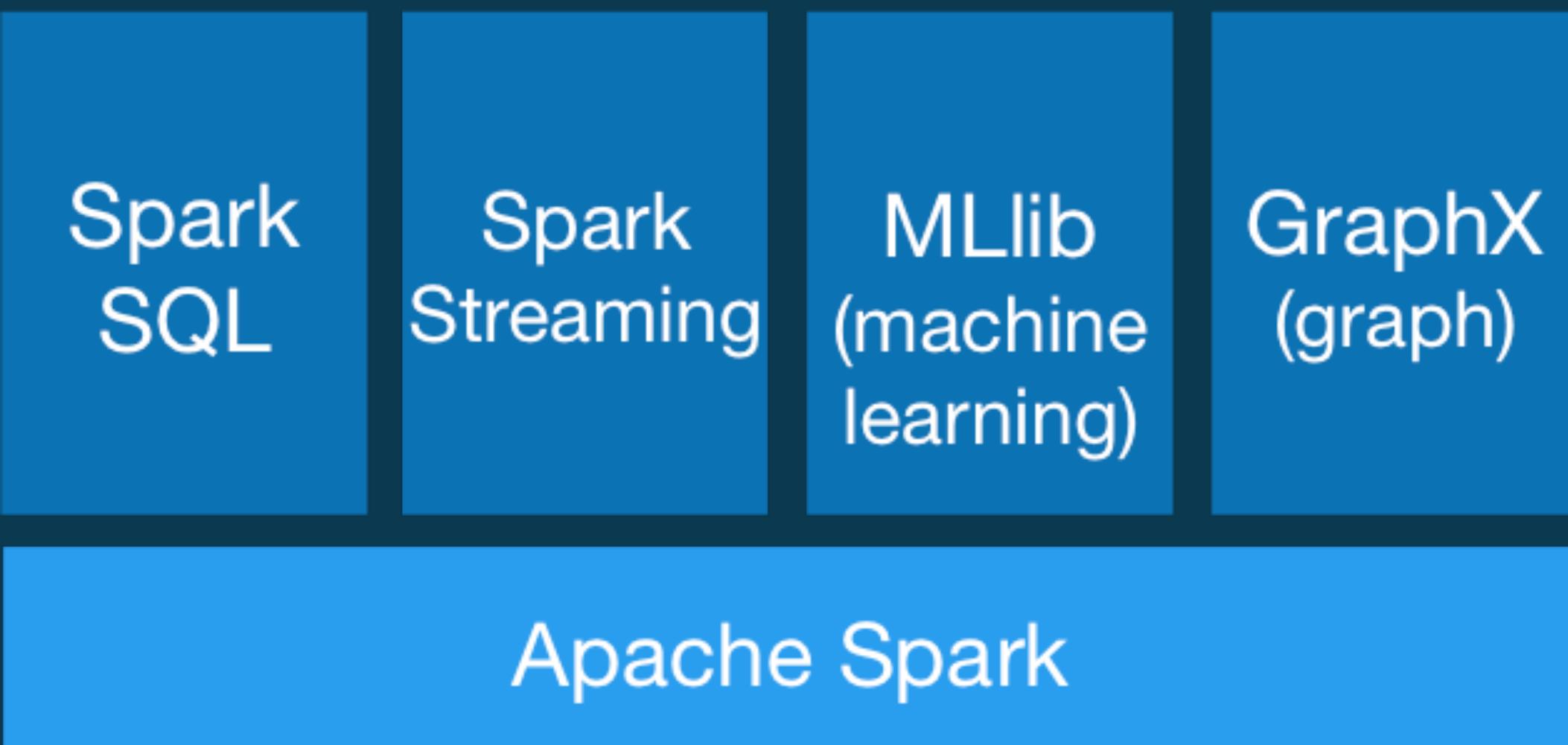
Kafka

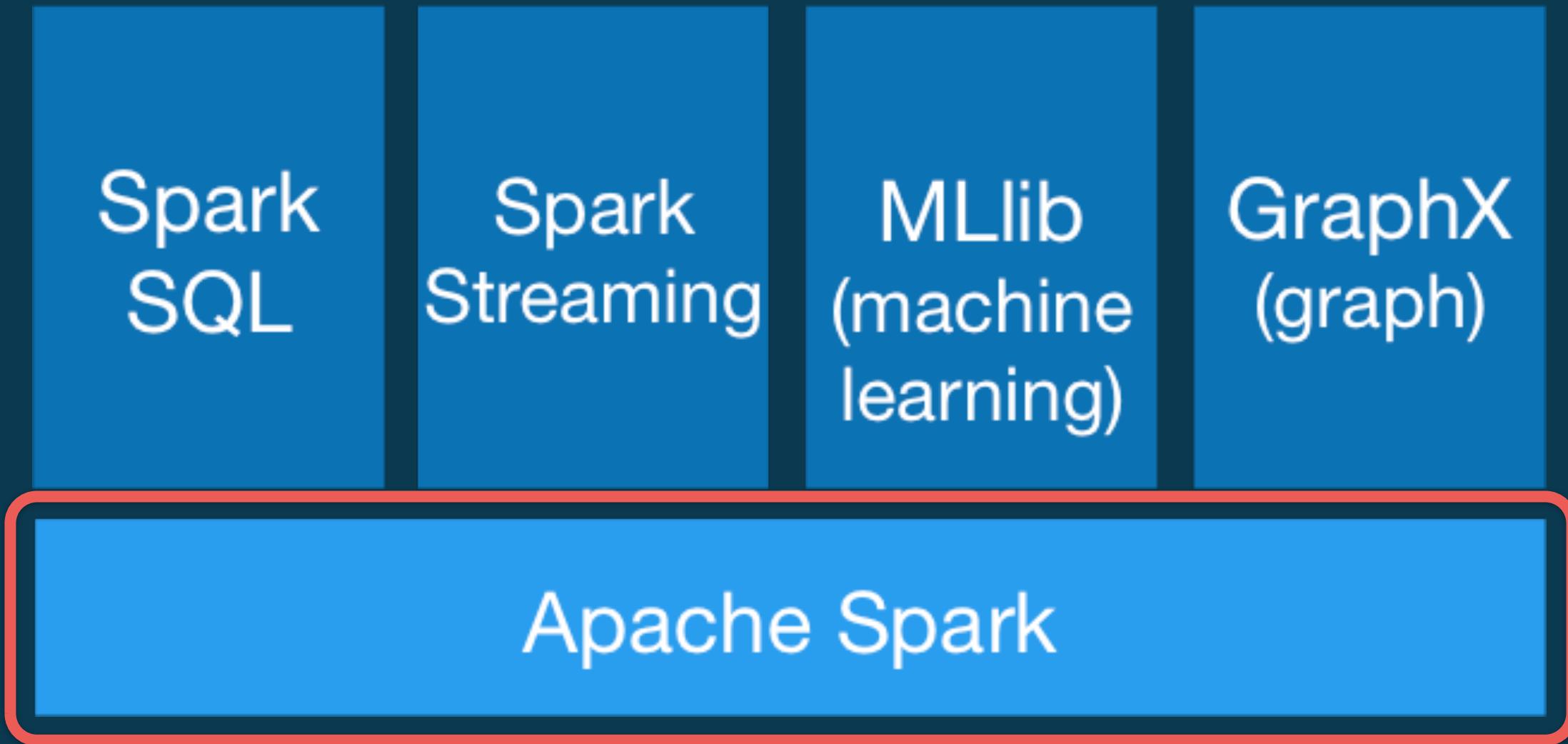
SMACK

A wide-angle photograph of a mountain lake. The water is very still, creating a perfect mirror that reflects the towering granite peaks and the dense forest of tall evergreen trees lining the shore. The sky is a clear, pale blue with a few wispy clouds.

Spark

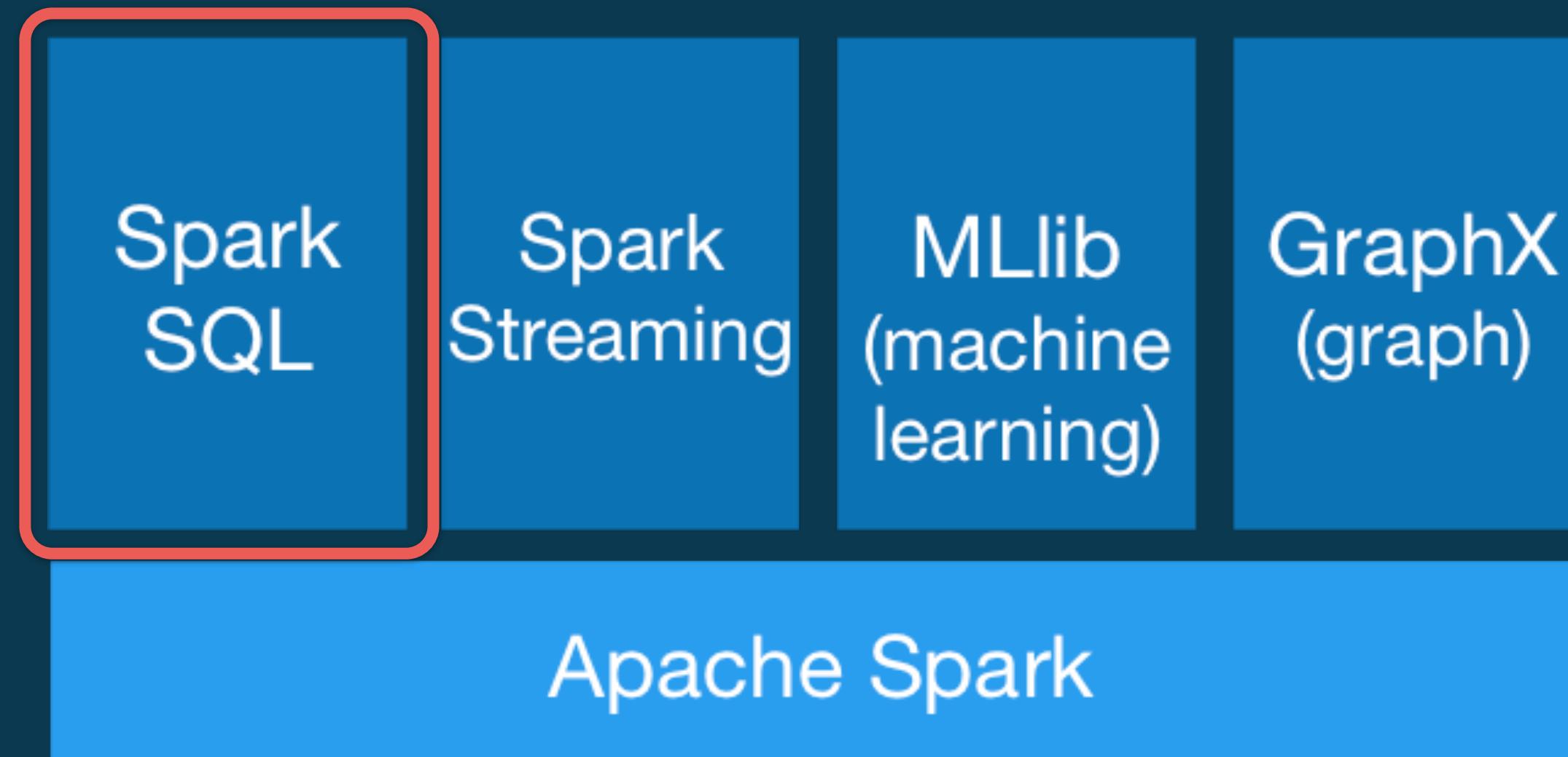
To replace Hadoop, we need the same kinds of components that it provides. Spark, the “S” provides the compute component for batch and streaming.
Photo: Upper Cathedral Lake with Tressider Peak on the left, Yosemite National Park (Spark)



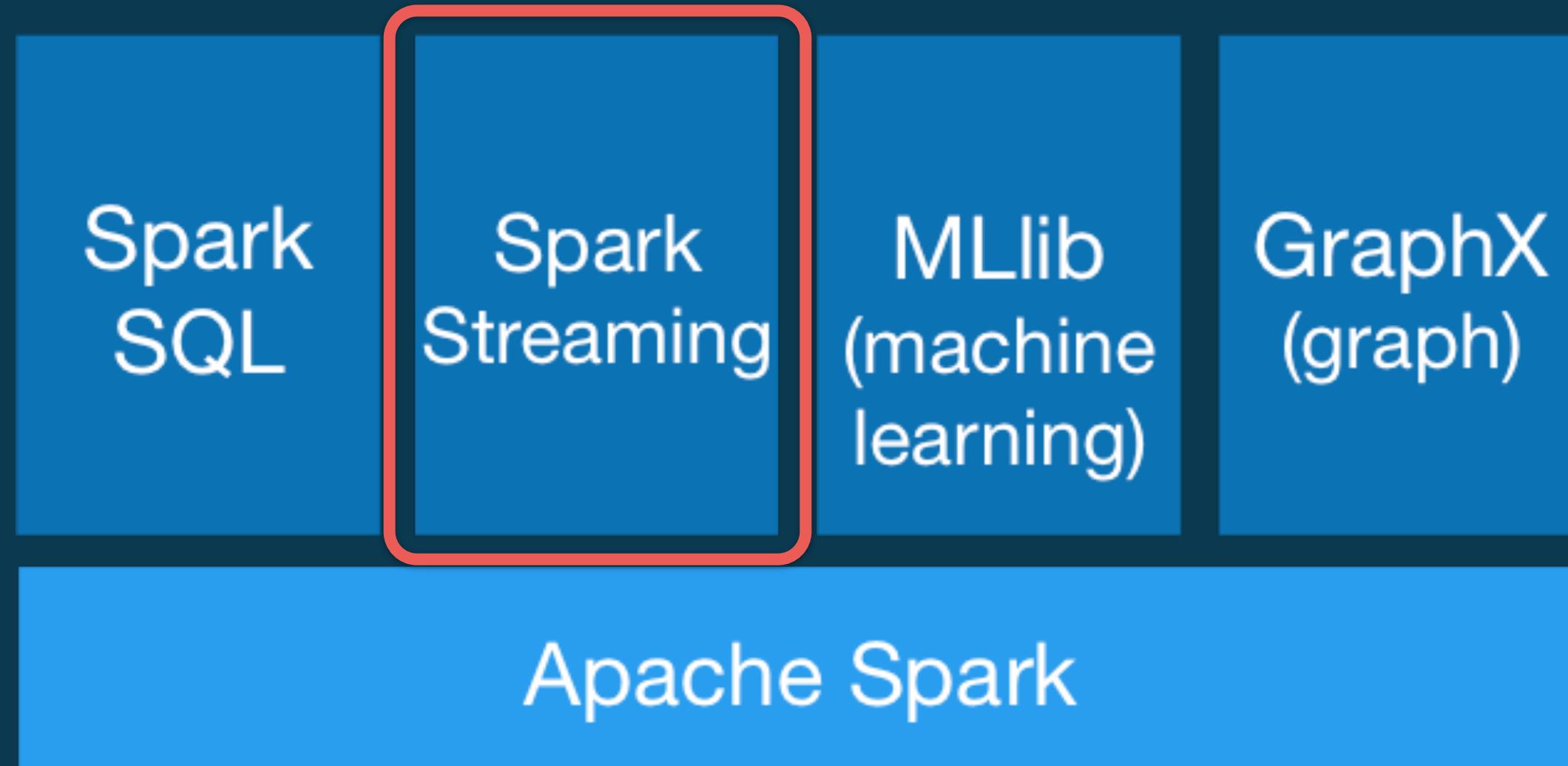


Core “RDD” API

Highly-optimized SQL

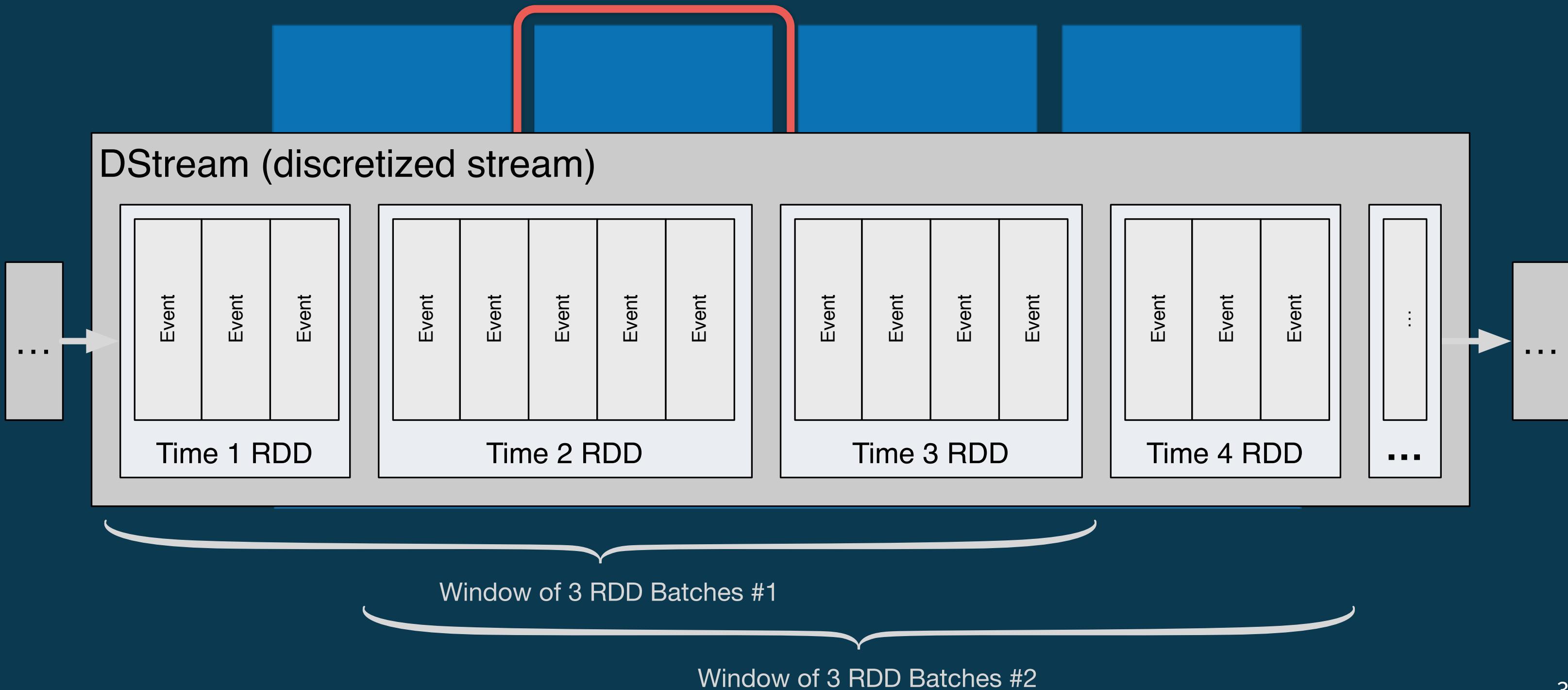


Stream processing



When streaming started getting popular, the Spark community realized that batch-mode Spark is efficient enough that it could be adapted to streaming using a “mini-batch model, with latencies (batch interval durations) down to ~0.5 seconds or so.

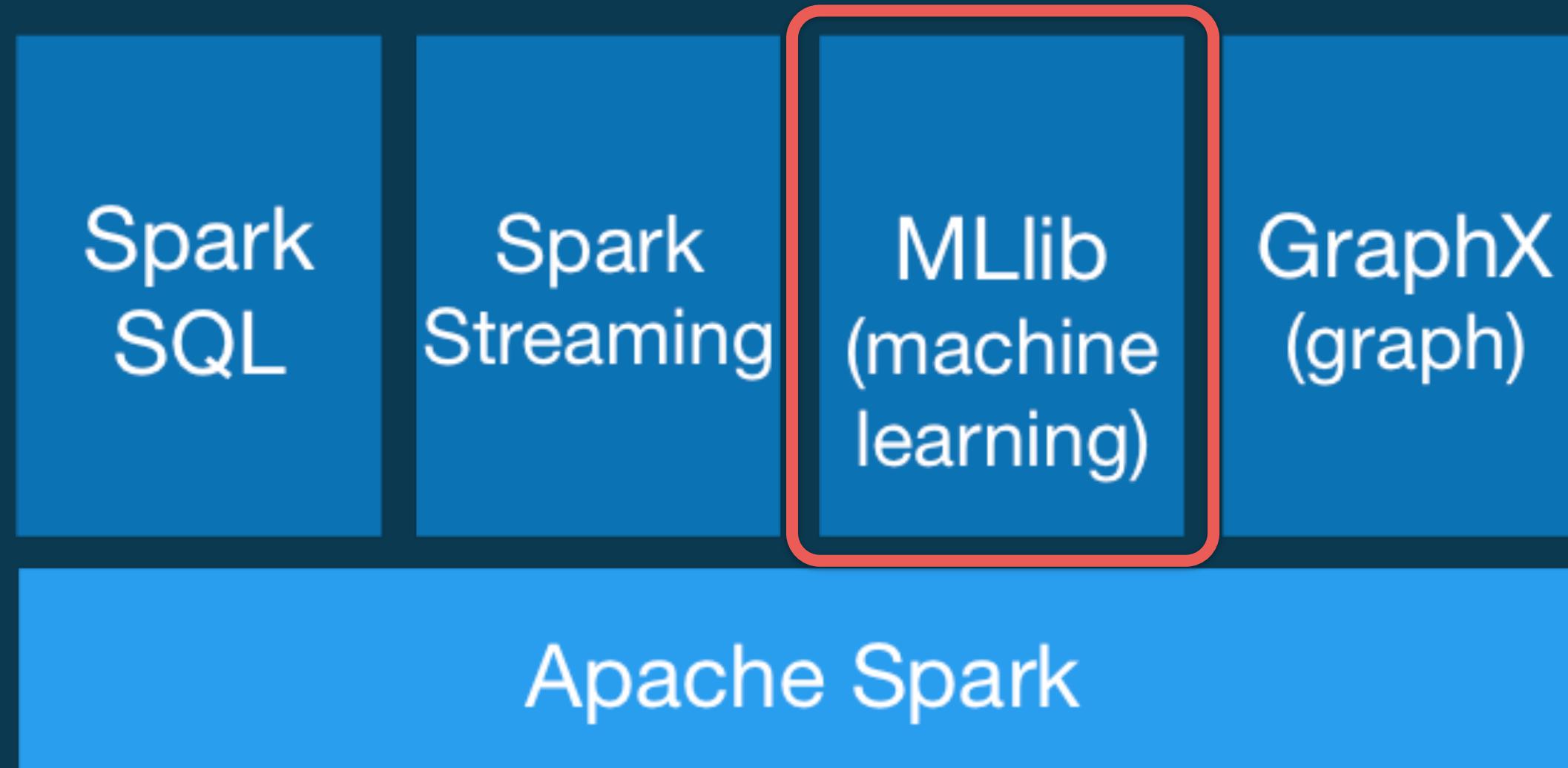
Stream processing



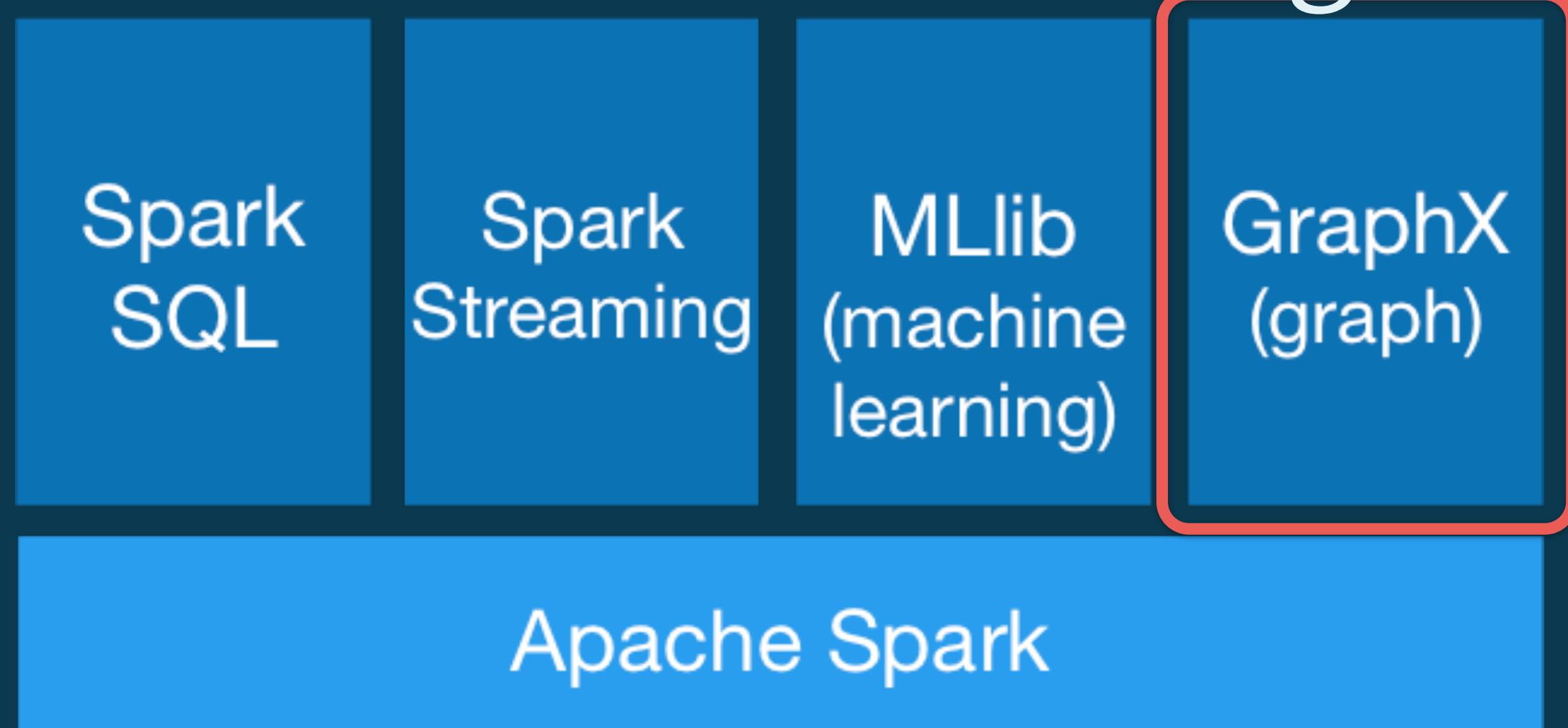
32

A fixed time interval is used and the events captured in each interval are put into an RDD and then processed using the RDD API and extensions, such as window functions as indicated. Spark Streaming is now evolving towards a more pure streaming model with lower latency.

Machine Learning



Graph data structures and algorithms





Other Streaming Engines?

SMACK is just an reference architecture; what about other streaming engines?
photo: Fairview Dome(?), West of Tuolumne Meadows, Yosemite National Park.

Streaming Tradeoffs

- Low latency? How low?
- High volume? How high?

Some tasks require a few microseconds or less, while others can tolerate more latency, especially if it allows the job to do more sophisticated or expensive things, like train machine learning models iteratively, write to databases, etc.

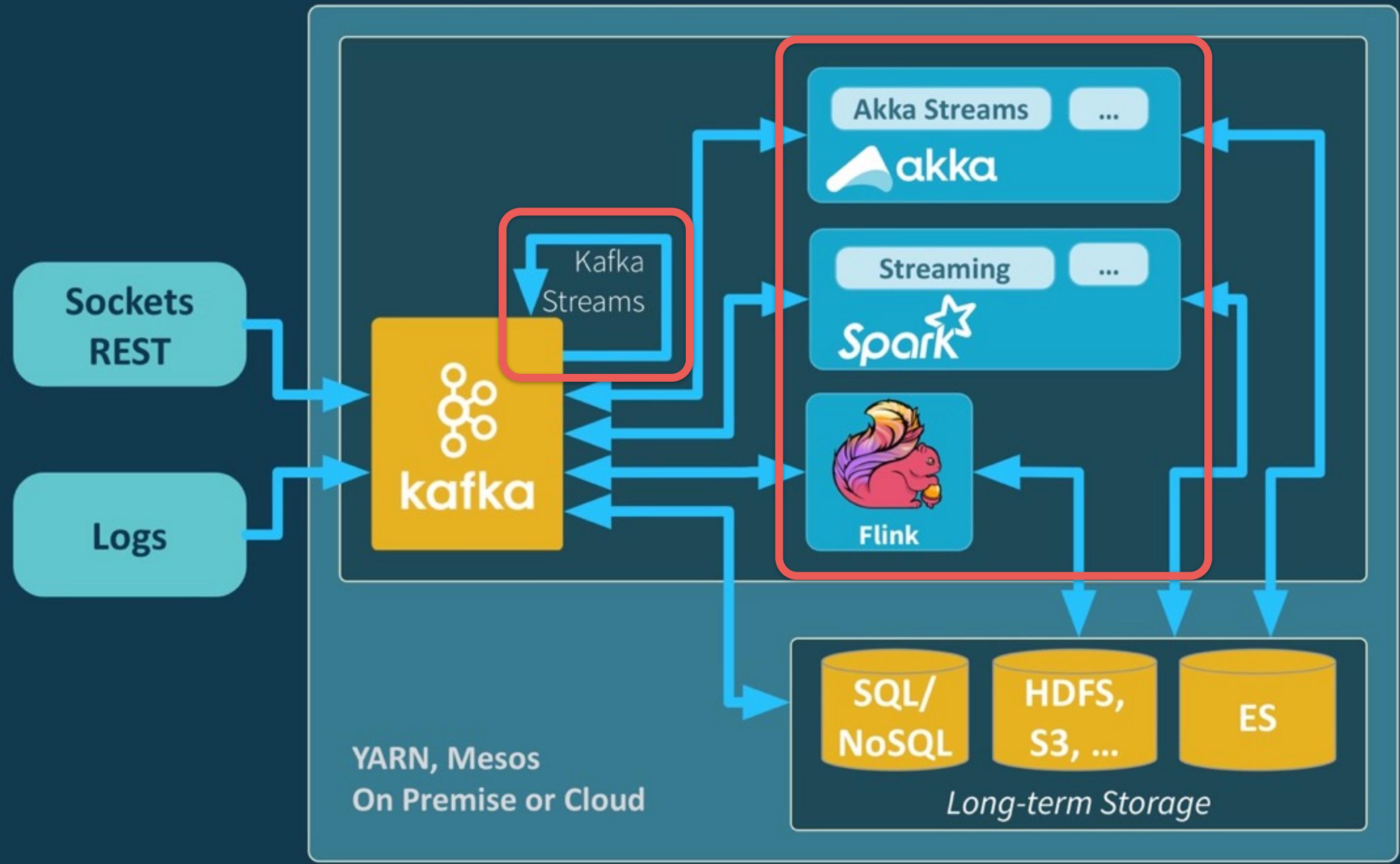
At high volumes, you might have to pick a very scalable tool with amortized excellent performance per event, but not when processing low volumes (e.g., due to the infrastructure it uses to support high volumes). Alternatively, a tool with excellent per-event performance might not scale well.

Streaming Tradeoffs

- Which kinds of data processing, analytics are required?
- How?
 - Bulk processing of records?
 - Individual processing of events?

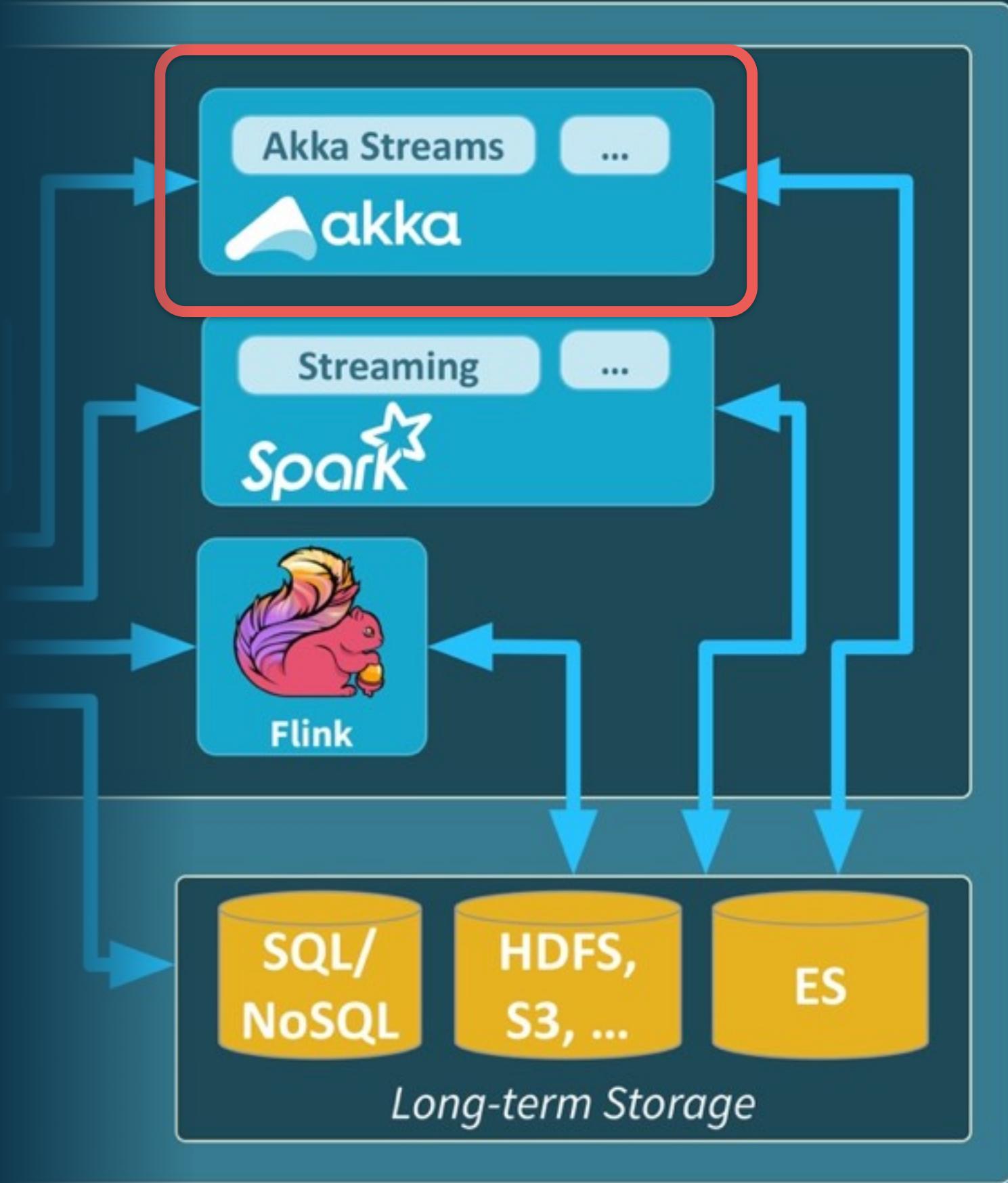
Are you doing complex event processing (CEP)? Aggregations? ETL? Others?

CEP is (usually) best done with a tool that processes each event individually, whereas other kinds of data can be processed “en masse” and it’s more efficient to do so, (like joins and group-bys).



I showed four of dozens(?) of possibilities, Akka, Spark, and Flink. I picked these three because they offer interesting choices in these tradeoffs...

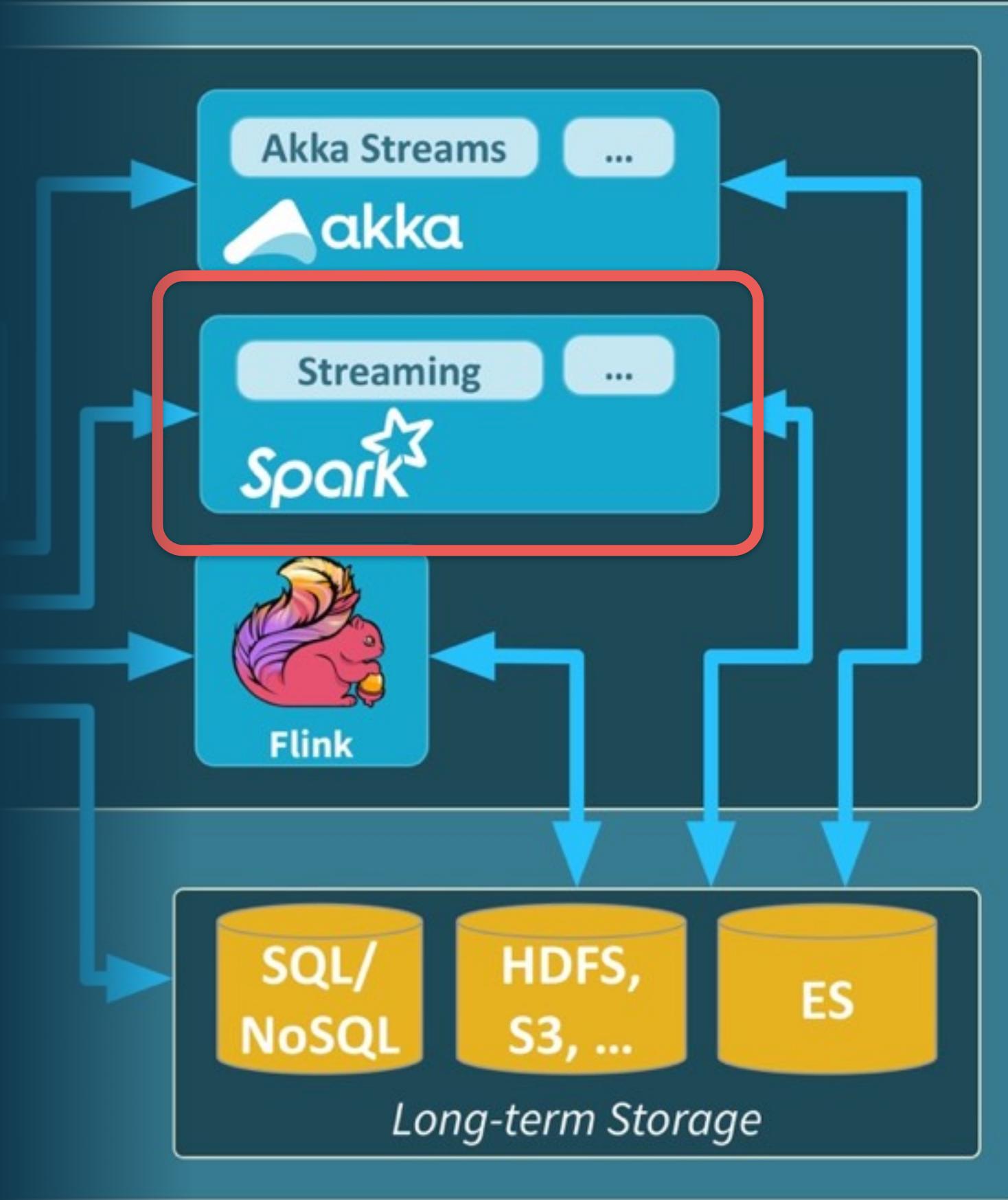
- Low latency
- Low volume
- Complex flows
- Per Event



39

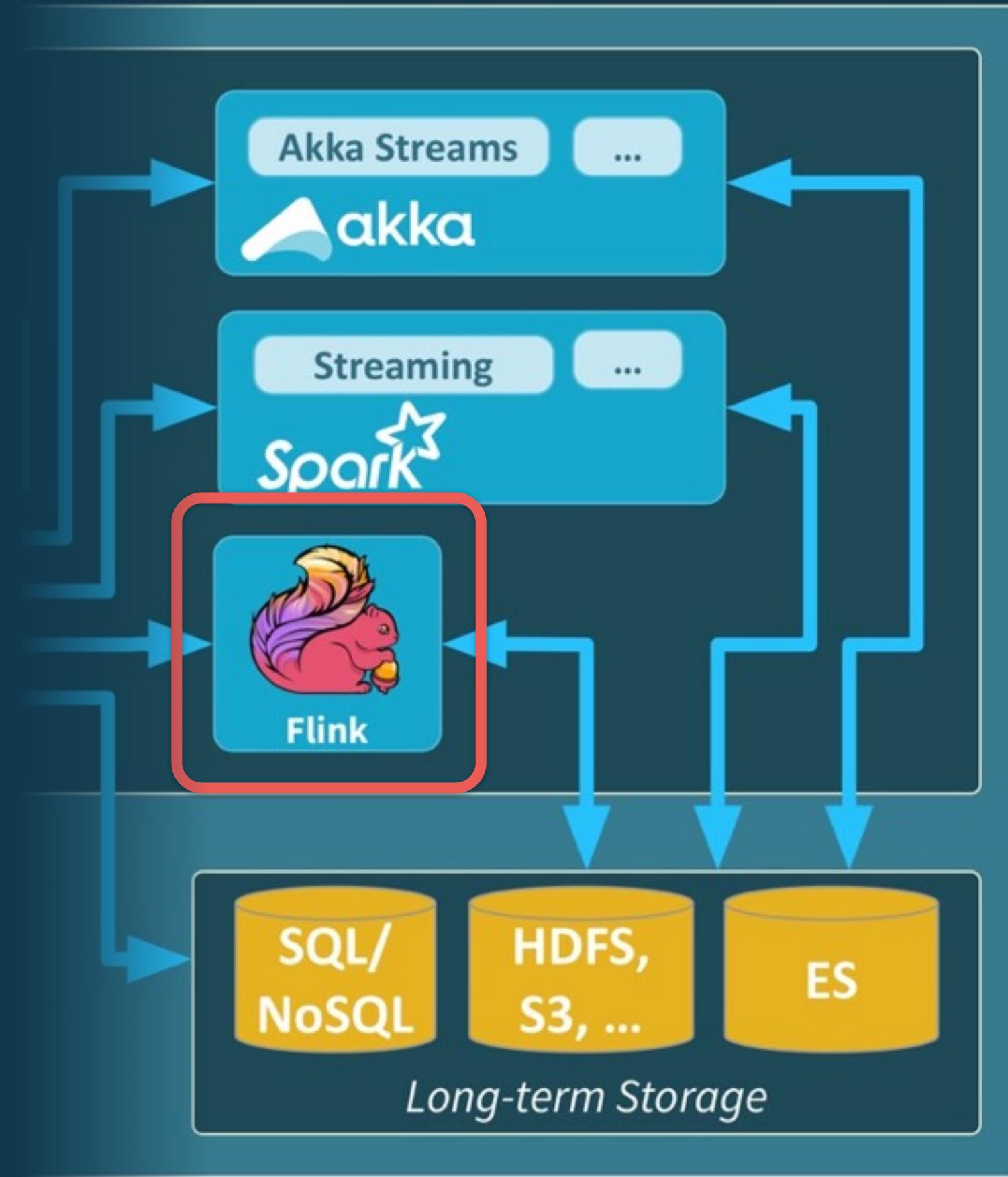
Akka is very low latency, optimized for excellent performance per event instead of high volume processing. You can do arbitrarily complex processing, including a sophisticated “flow graph” model. It is ideal for per-event processing.

- Med. latency
- High volume
- Data flows, SQL
- En masse

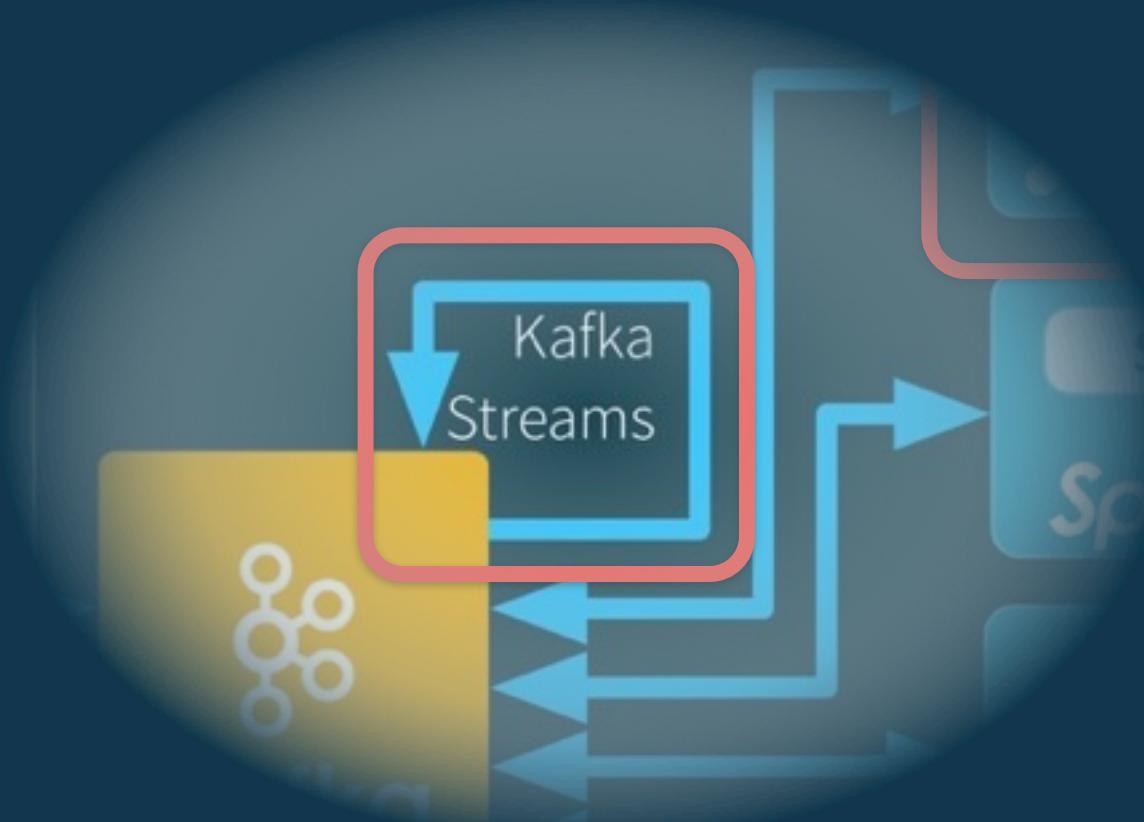


Spark has medium latency (~0.5 seconds and up), optimized for excellent, scalable performance at high volumes. The model is either data flows (think sequential processing nodes) or SQL queries. It is not designed for per-event processing, but “en masse” processing of records.

- Low latency
- High volume
- Data flows, correctness
- En masse



Flink contrasts mostly with Spark. It has low instead of medium latency, both with excellent, scalable performance at high volumes. The model is also primarily data flow oriented (SQL is coming), but it also supports very sophisticated correctness semantics, such as handling windows of events, processing by event time (not system arrival time), handling late-arriving data, etc.

- 
- Low latency
 - Med. volume
 - ETL, “tables”
 - Data flow / Per Event

42

Kafka Streaming is focused on reading data in Kafka topics, processing it, and writing the results to new topics. It's ideal for many common scenarios, such as ETL, but also supports running aggregations including the last seen values for keys (like DB tables work). Using the API, you write data flow code, but the implementation is more like a per-event processor.



43

R to L: Tressider Peak, Columbia Finger, Cathedral Peak, Echo Peaks, and Matthes Crest, Yosemite National Park

MESOS



photo: Columbia Finger, Yosemite National Park (Mesos)

Mesos

- Treats your cluster like a large set of resources

Mesos: Analogous to YARN

- Resources are *dynamic*
 - CPU cores & memory
 - but also network, disk, ...

Mesos: Analogous to YARN

- Each application *framework* provides its own scheduler
- Resources are offered
 - They can be refused

YARN has to hard-code knowledge about how any application will need and use resources. This centralization makes it impossible to plug in arbitrary, new apps with very different needs. Mesos delegates this app-specific knowledge to the app. Instead, it naively offers available resources to each running app (a framework in Mesos terms). The app's scheduler decides whether or not to accept any or all of the offered resources. If it doesn't, then Mesos will offer them to another framework. If it does, then the framework tells Mesos how to start the process that will use the resources. This makes Mesos far more flexible than YARN; it can not only run HDFS and databases, which YARN can't, it can even run YARN itself (See <https://myriad.apache.org/>)
There are other advantages, see my Spark on Mesos talk: <https://deanwampler.github.io/polyglotprogramming/papers/SparkOnMesos.pdf>

[http://mesos.berkeley.edu/
mesos_tech_report.pdf](http://mesos.berkeley.edu/mesos_tech_report.pdf)

Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center

Benjamin Hindman, Andy Konwinski, Matei Zaharia,
Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion Stoica

University of California, Berkeley

Thursday 30th September, 2010, 12:57

Abstract

We present Mesos, a platform for sharing commodity clusters between multiple diverse cluster computing frameworks, such as Hadoop and MPI. Sharing improves cluster utilization and avoids per-framework data replication. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. To

The solutions of choice to share a cluster today are either to statically partition the cluster and run one framework per partition, or allocate a set of VMs to each framework. Unfortunately, these solutions achieve neither high utilization nor efficient data sharing. The main problem is the mismatch between the allocation granularities of these solutions and of existing frameworks. Many frameworks, such as Hadoop and Dryad, employ a fine-

The Mesos research paper. Ben lead the development as a Berkeley grad student. Matei was a fellow grad student who created Spark...

[http://mesos.berkeley.edu/
mesos_tech_report.pdf](http://mesos.berkeley.edu/mesos_tech_report.pdf)

Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center

“Our results show that Mesos can achieve near-optimal data locality when sharing the cluster among diverse frameworks, can scale to 50,000 (emulated) nodes, and is resilient to failures.”

Abstract

We present Mesos, a platform for sharing commodity clusters between multiple distributed computation frameworks, such as Hadoop and MPI. Sharing improves cluster utilization and avoids per-framework data replication. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. To

The solutions of choice to share a cluster today are either to statically partition the cluster and run one framework per partition, or allocate a set of VMs to each framework. Unfortunately, these solutions achieve neither high utilization nor efficient data sharing. The main problem is the mismatch between the allocation granularities of these solutions and of existing frameworks. Many frameworks, such as Hadoop and Dryad, employ a fine-grained allocation granularity, while the VM-based solu-

<http://mesos.berkeley.edu/>
mesos_tech_report.pdf

Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center

“To validate our hypothesis ... ,

Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion Stoica

University of California, Berkeley

Thursday 30th September, 2010, 12:57

we have also built a new framework

on top of Mesos called Spark... ”

Abstract

We present Mesos, a platform for sharing commodity clusters between multiple diverse cluster computing frameworks, such as Hadoop and MPI. Sharing improves cluster utilization and avoids per-framework data replication. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. To

The solutions of choice to share a cluster today are either to statically partition the cluster and run one framework per partition, or allocate a set of VMs to each framework. Unfortunately, these solutions achieve neither high utilization nor efficient data sharing. The main problem is the mismatch between the allocation granularities of these solutions and of existing frameworks. Many frameworks, such as Hadoop and Dryad, employ a fine-grained allocation granularity, while the underlying hardware and the operating system provide a coarse-grained allocation granularity.

Adoption

- Twitter
- Apple's Siri
- Airbnb
- Verizon
- CERN, ...





Other
Clustering
Platforms?

Clouds?

- Compelling, but it's also common to use Mesos on top of clouds:
 - Virtual cluster of resources
 - Uniform deployment, management on-premise & cloud

53

Will Clouds make Mesos unnecessary? Maybe, but there is cost advantages to spin up long-running virtual instances, then manage them as a cluster of resources using Mesos. Also, for hybrid on-premise and cloud environments, uniformity is useful for everything above the server level.



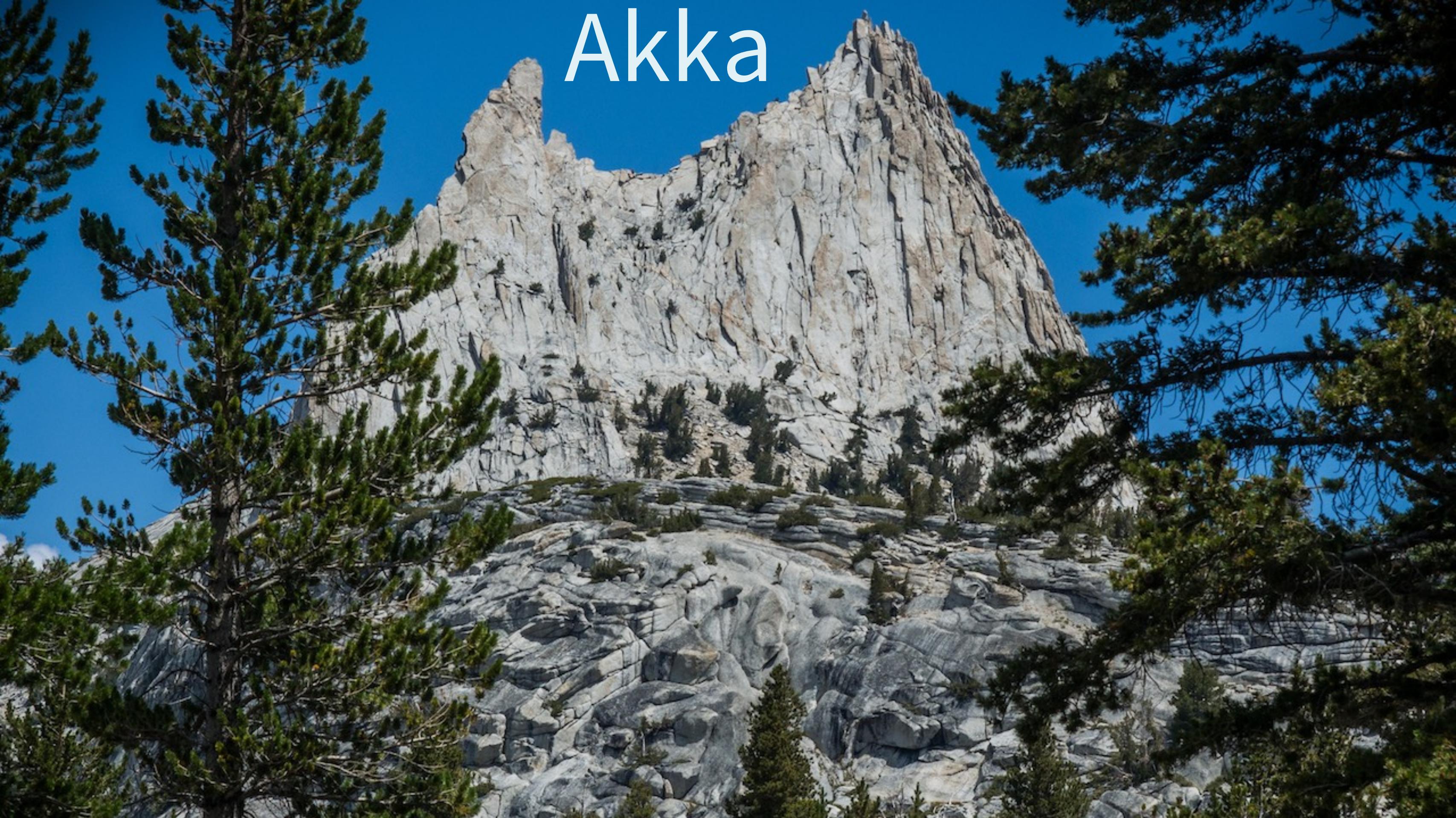
Spark Mesos Akka

Cassandra

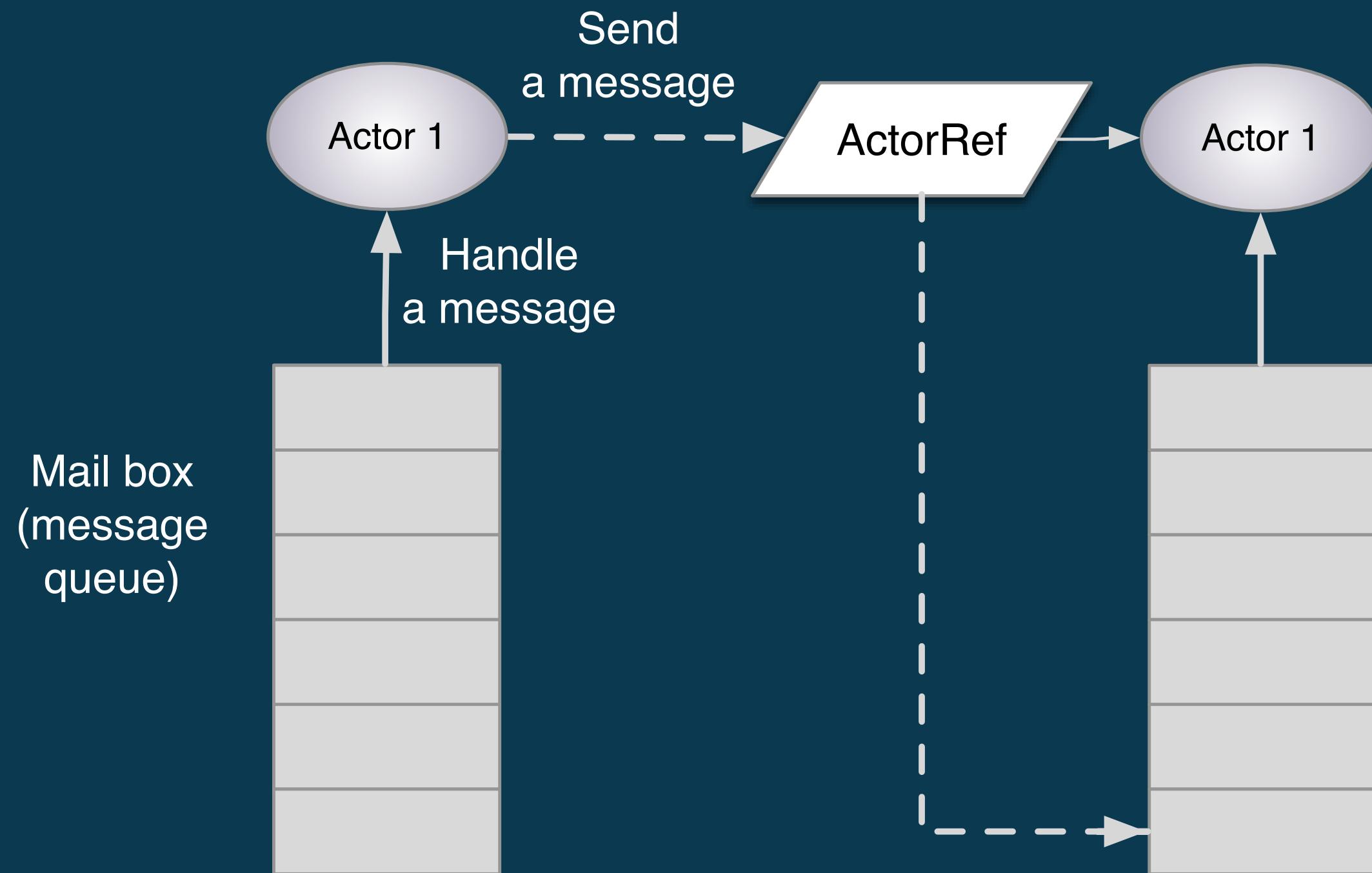
Kafka

SMACK

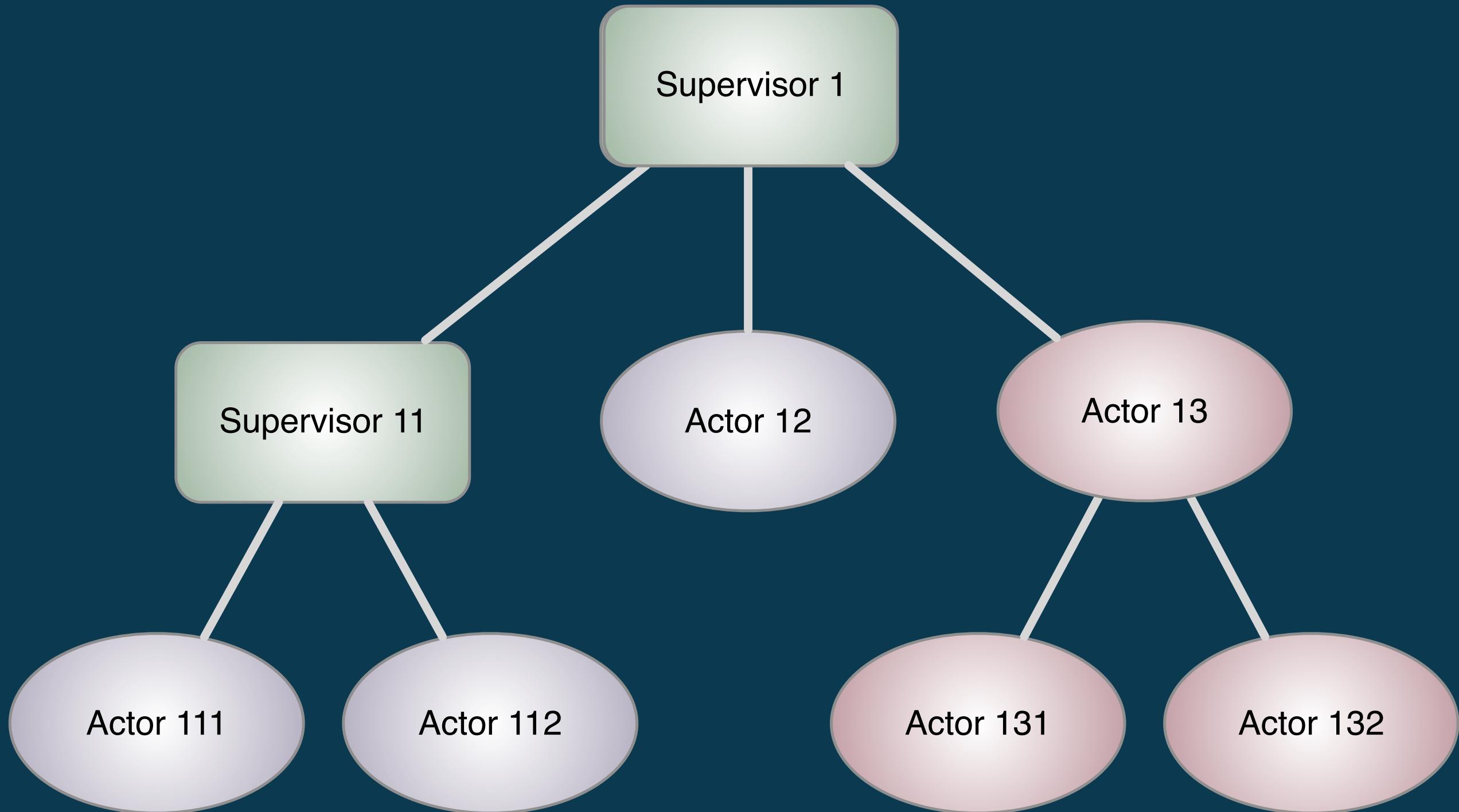
Akka

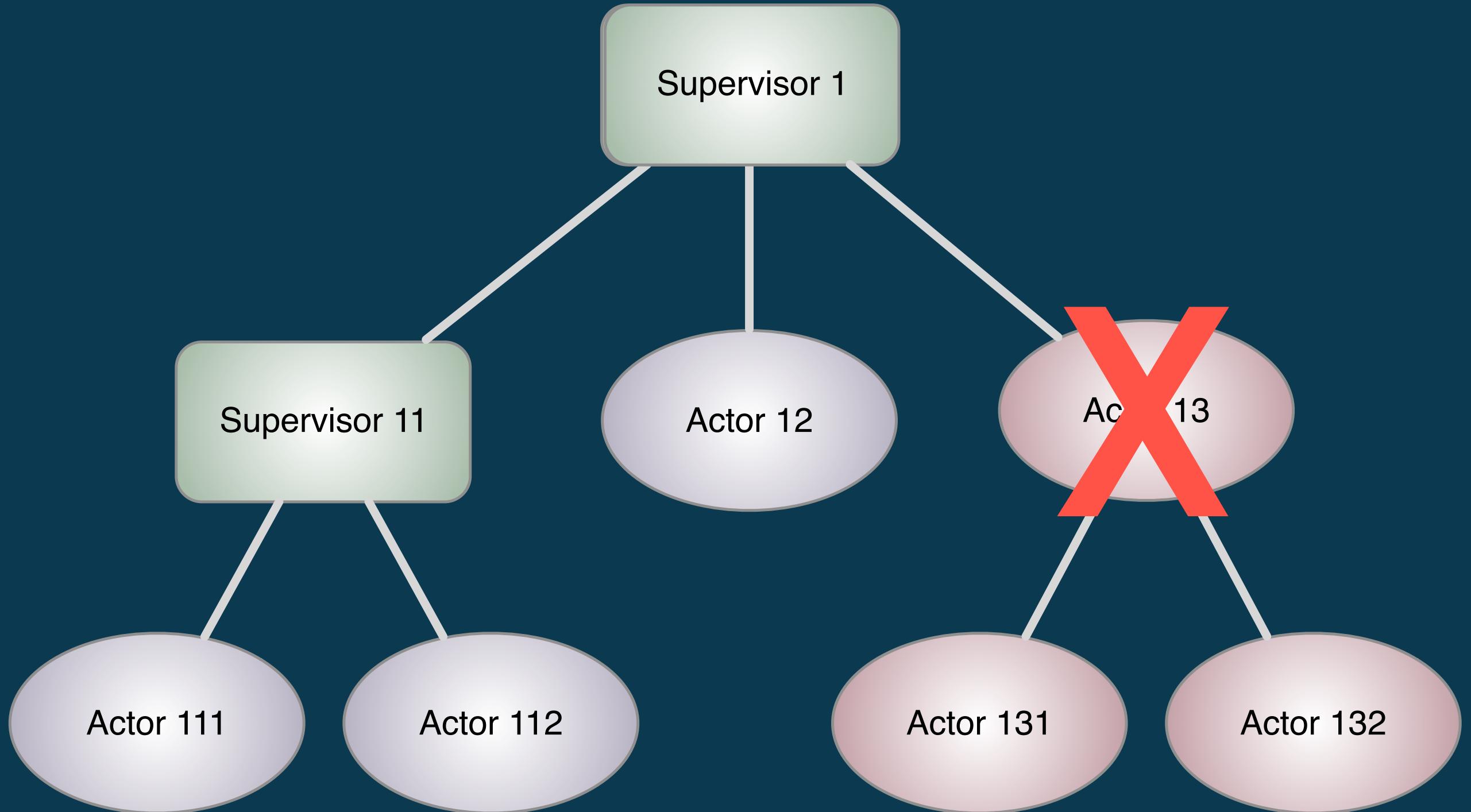


Akka is a set of tools for building resilient, distributed, concurrent apps on the JVM.
for photo: Cathedral Peak, Yosemite National Park (Akka)

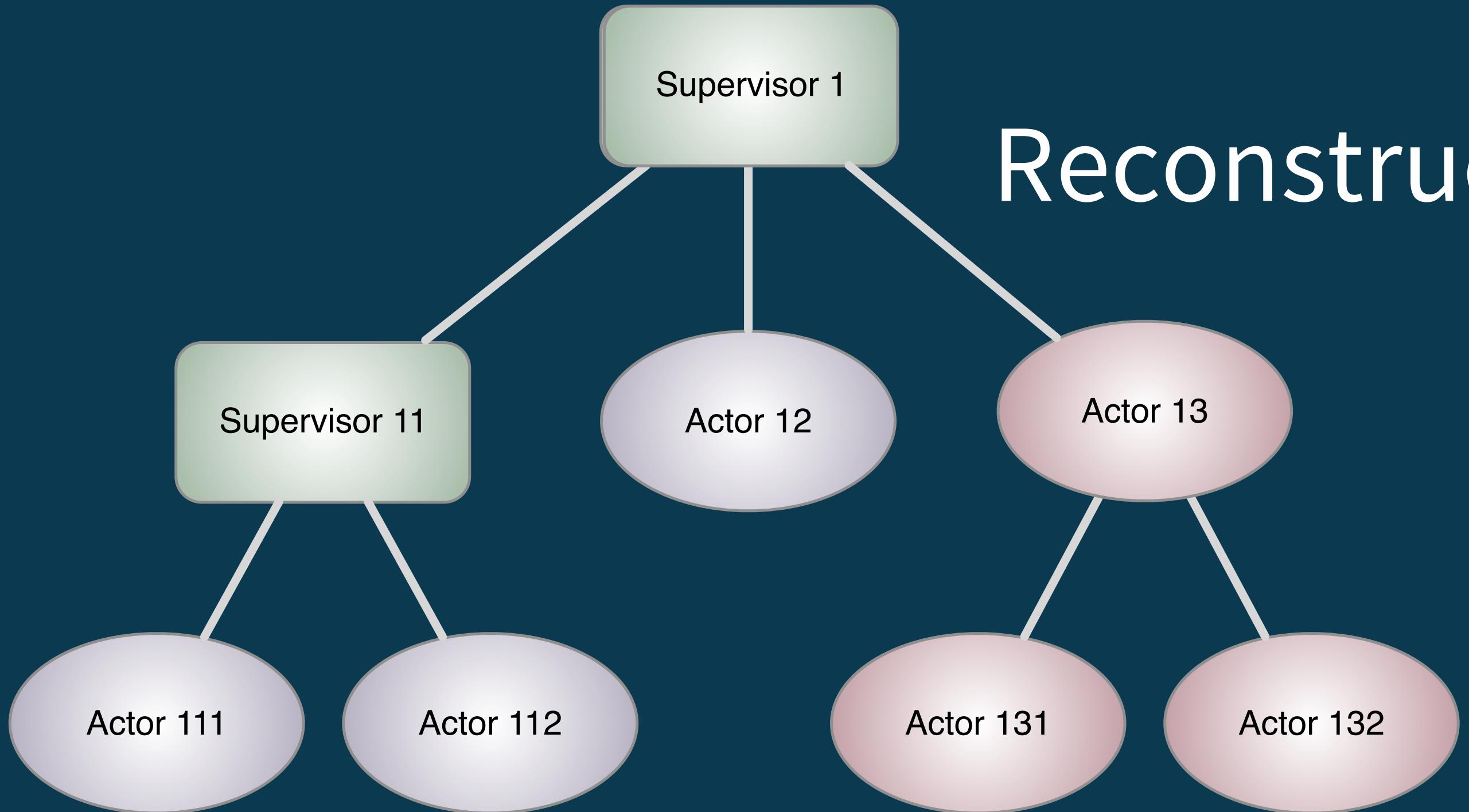


At the lowest level, you program to an Actor model, autonomous agents where your code inside them is guaranteed thread safe. Actors send messages to each other to send information and invoke actions asynchronously.
 There is also the higher level Akka Streams API we discussed previously.





Reconstructed



One of the most powerful features for resiliency is that supervision of your domain actors is a separate concern (special-purpose actors), which manage the lifecycles of domain actors, triggering recovery when failure happens.

Distributed Apps



- High performance
 - ~50M msgs/sec on a laptop
- Elastic and decentralized
- Modules for clustering, CQRS, HTTP, ...

Lightbend Reactive Platform

- Akka is one piece of SMACK. If you add the rest of RP, you get:

SMRCK

61

If you throw in the rest of our stuff, you get the SMRCK (“smirk”) stack...

I'll be here all week folks. You've been a great audience! Don't forget to tip your waitress!



Other App Middleware?

photo: Cathedral Peak... or is it the Eye of Sauron??!!

Yes, but it must be *Reactive*

- There are alternatives, but:
 - Is it message driven?
 - Is it scalable up and down?
 - Is it resilient against failure?
 - Is it always responsive?



Spark Mesos Akka

Cassandra

Kafka

SMACK

Cassandra



photo: The really beautiful Echo Peaks, Yosemite National Park (Cassandra)



Distributed Databases

- Cassandra is in SMACK (SMRCK?) because it's so widely used.
- Spark + Cassandra + Kafka is *very* common in streaming systems.

The second bullet is based on surveys that Lightbend and other organizations have done.

I won't discuss the advantages of Cassandra further, as this is already a long talk and you may already be familiar with it or other NoSQL databases.

Other Databases/file systems?

- Scalable? distribution with partitioning
- Resilient? distribution, replication, availability (CAP) better

67

Lots of other databases could be used instead of (or along side of Cassandra). Even a distributed file system might be all your need. What about traditional relational (SQL) databases?? NoSQL databases that embrace availability (CAP theorem) will remain available a higher % of time in the face of failure, but note that many exports are concerned that eventual consistency is very hard to do correctly, so some organizations are actually choosing consistency instead! TCO is generally lower for NoSQL, too.

Other Databases/file systems?

- Cost? NoSQL *has been* cheaper

Lots of other databases could be used instead of (or along side of Cassandra). Even a distributed file system might be all your need. What about traditional relational (SQL) databases?? NoSQL databases that embrace availability (CAP theorem) will remain available a higher % of time in the face of failure, but note that many exports are concerned that eventual consistency is very hard to do correctly, so some organizations are actually choosing consistency instead! TCO is generally lower for NoSQL, too.



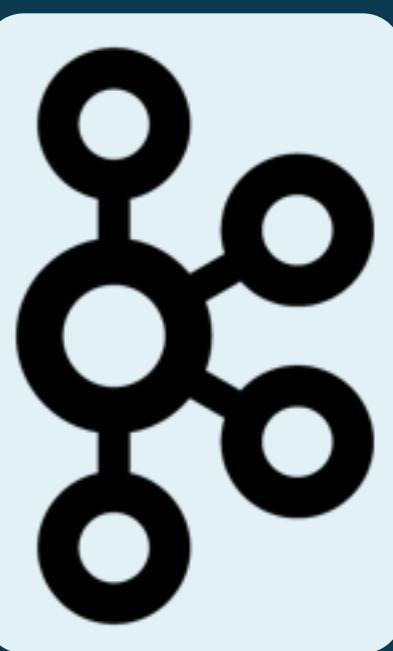
69

R to L: Tressider Peak, Columbia Finger, Cathedral Peak, Echo Peaks, and Matthes Crest, Yosemite National Park

Kafka



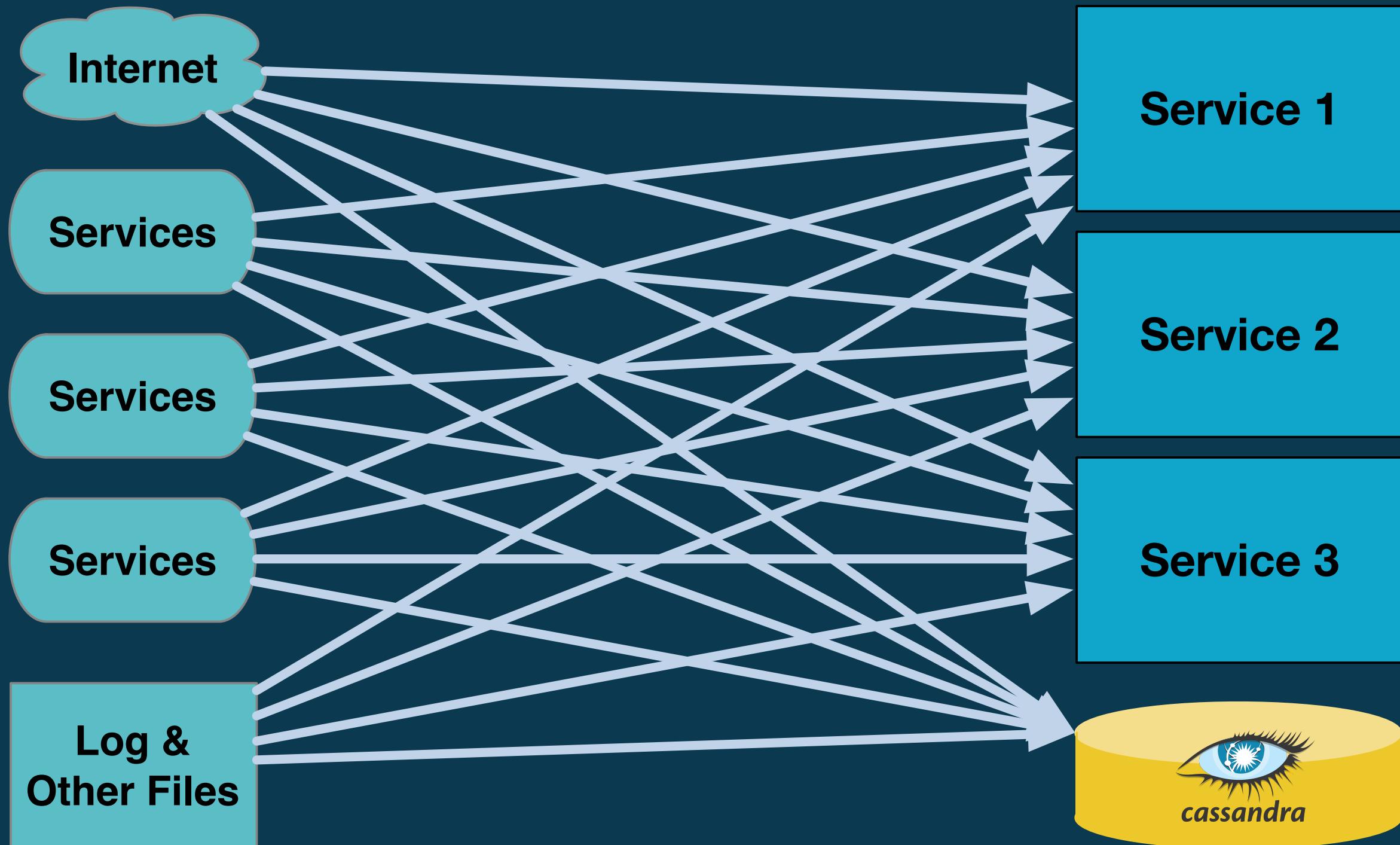
photo: The dramatic Matthes Crest, Yosemite National Park (Kafka)



“Message Bus”

- Pub/Sub model
 - Organized by topic
 - Short term, resilient storage
 - Massive scalability
 - 1.2+T messages/day at LinkedIn

From This...



Producers

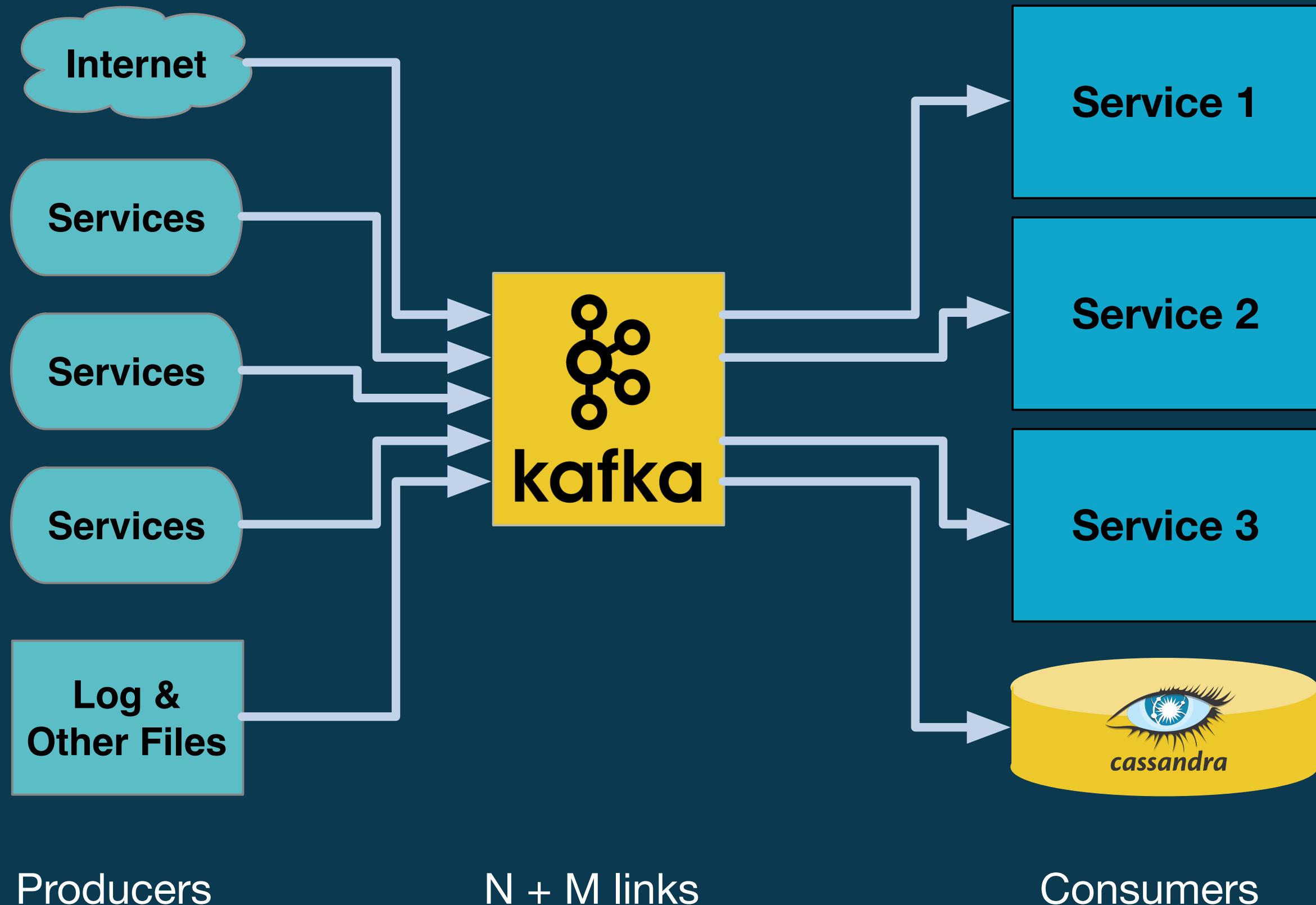
$N * M$ links

Consumers

72

As an architect tool, Kafka fixes the flaw of having direct point-to-point connections (up to $N*M$). This is both messy to manage and fragile. If Service1 drops, for instance, all the connections to it are broken, affecting the producers and losing service.

To This.



By routing messages through Kafka, you reduce the connections to $N+M$, allow 1+ producers and 1+ consumers per topic, and provide much greater resiliency. Should Service 1 go down (or need to be upgraded), the messages in the topic will remain there until a replacement comes up.



Other “Messageware”?

photo: The dramatic Matthes Crest, Yosemite National Park

Kinesis, Message Queues, ...

- AWS Kinesis
- Traditional message queues:
ZeroMQ, RabbitMQ, ...

Kinesis, Message Queues, ...

- Evaluate the differences:
 - Scalability & resiliency
 - Pub/Sub semantics different
 - E.g., Kafka readers don't consume the message!

76

Be sure that the alternative scale as much as you need them to scale.

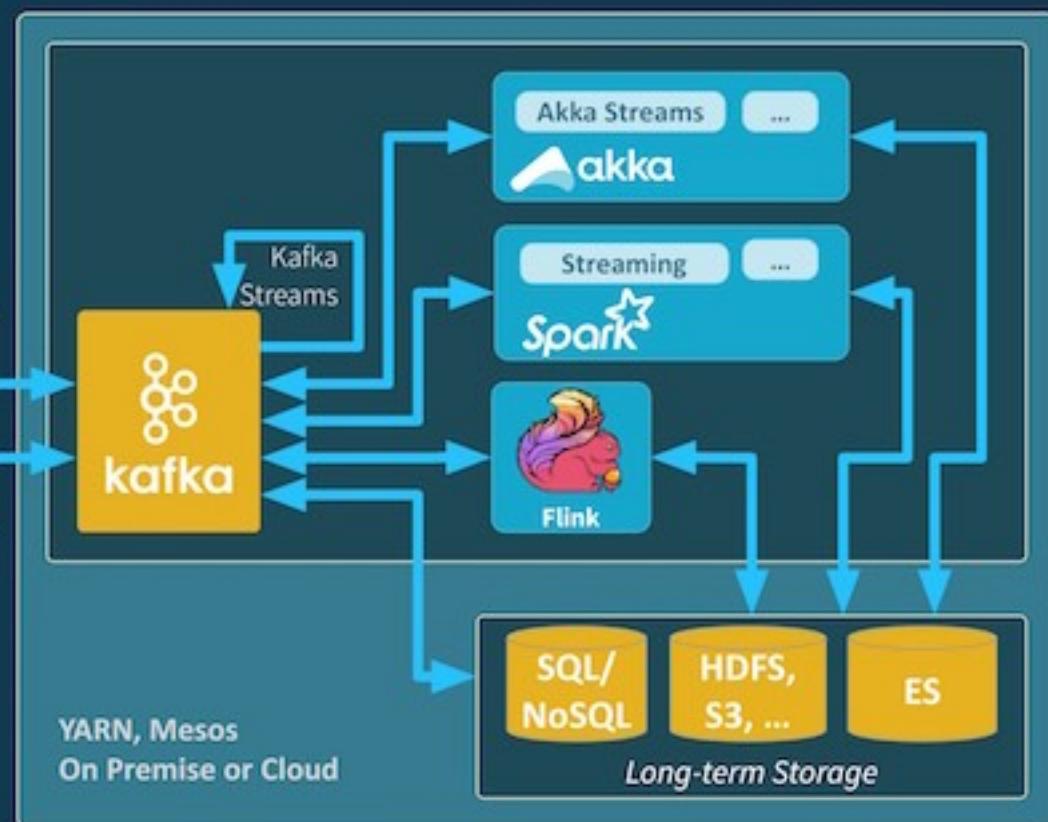
Study the pub/sub semantics carefully. For most message queues, reading a message consumes it. That simplifies the implementation, but it complicates writing many applications that need to see the whole stream. Every reader of a Kafka topic will see all the messages. Kafka manages the offsets into the topics for each reader.



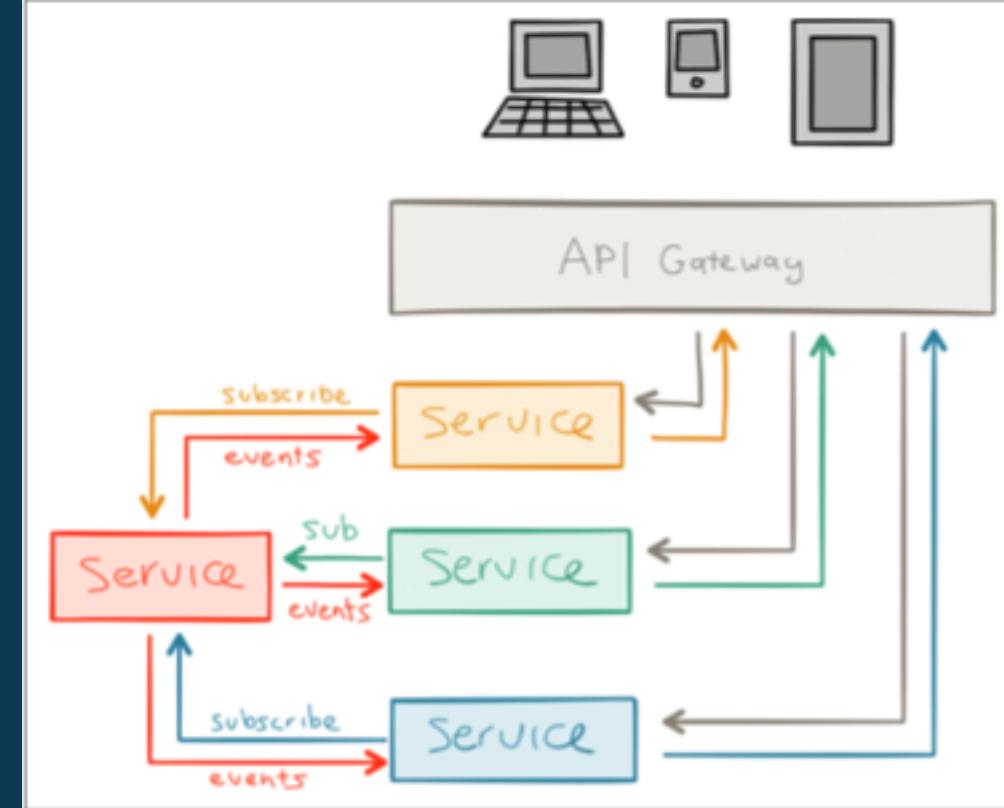
Fast Data & Microservices: Will They Converge?

photo: First light on Banner Peak from 1000 Island Lake.

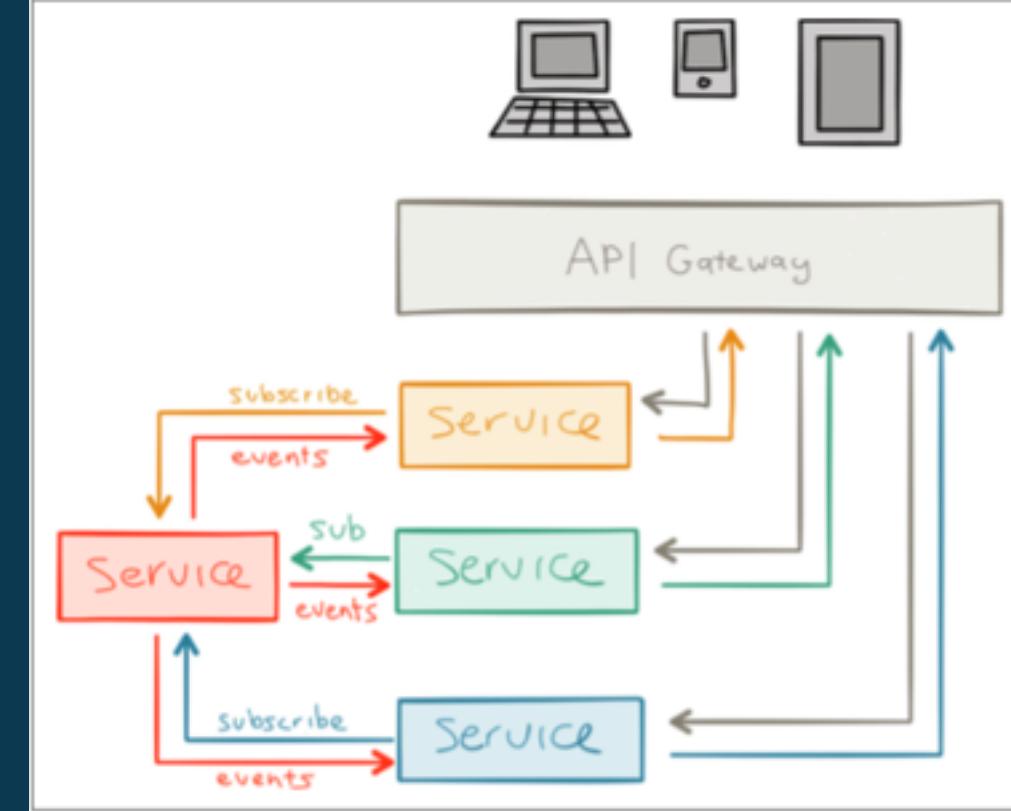
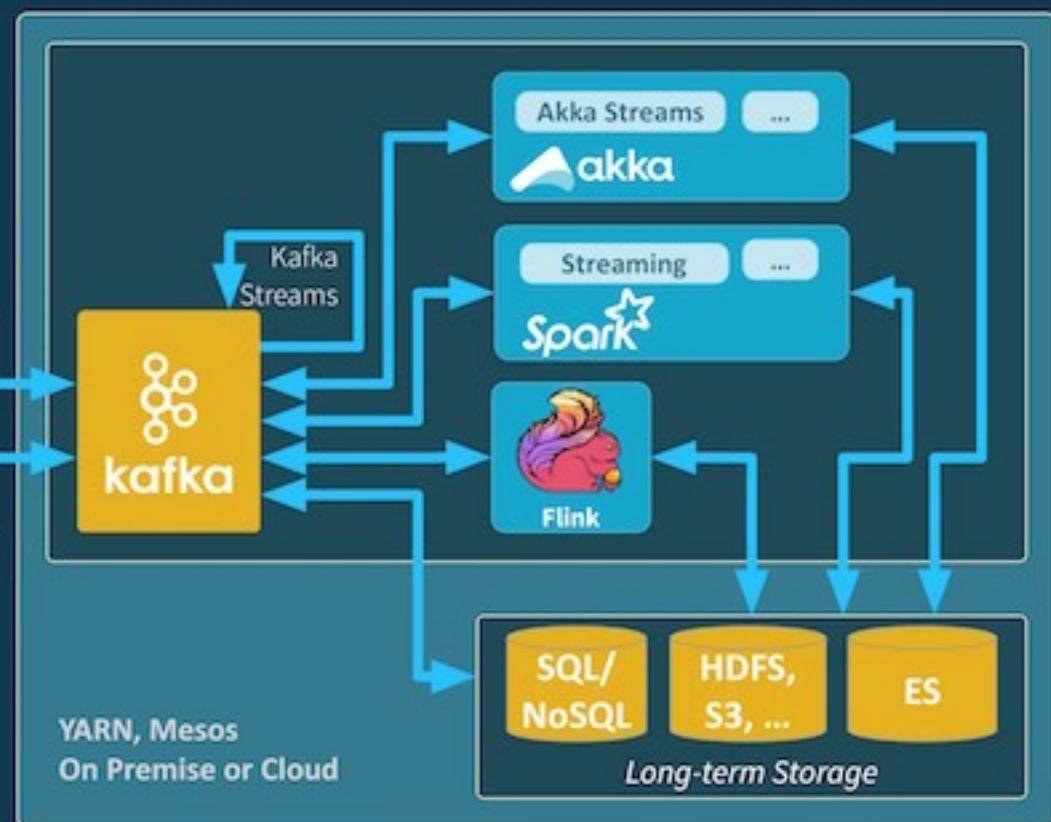
Synergies



- Each (stream|μservice):
 - does one thing
 - has unending (data|messages)



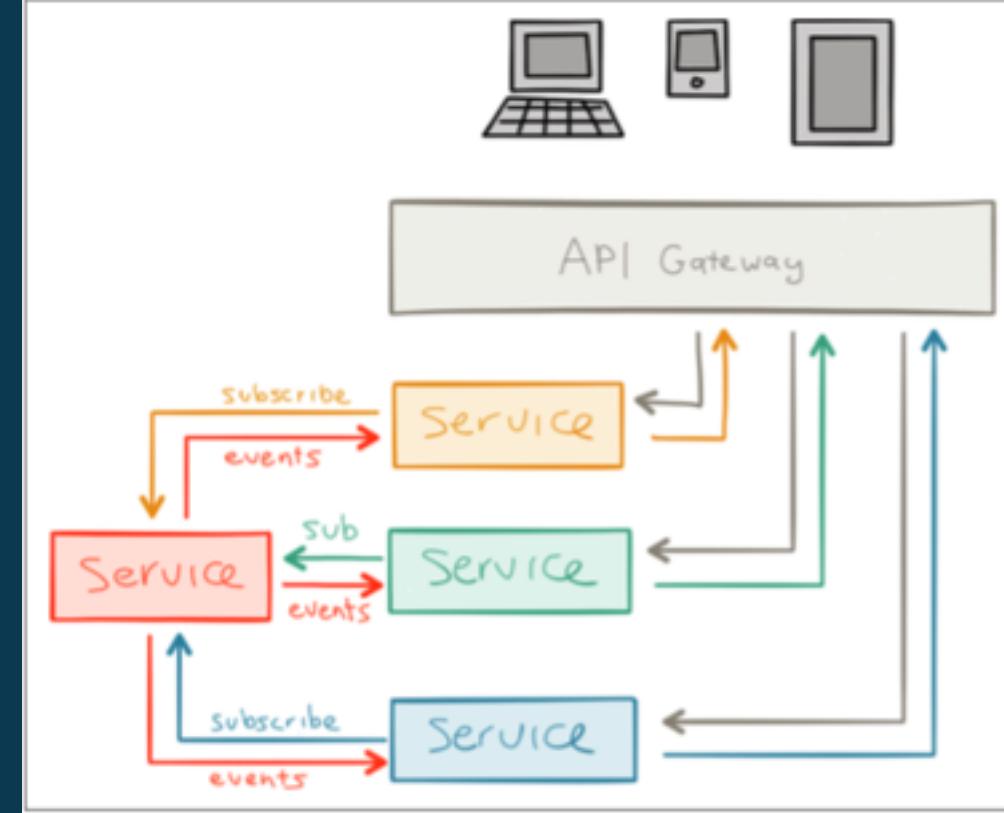
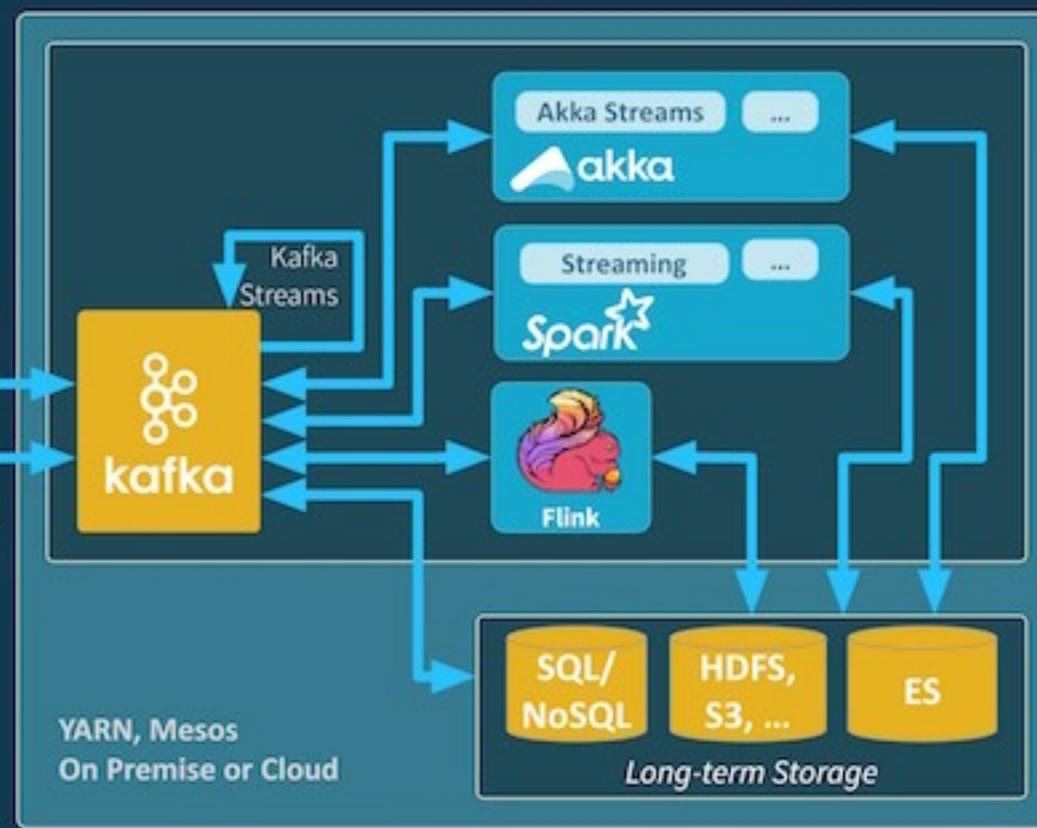
Synergies



- Each (stream|μservice):
 - encourage asynchrony
 - offers unbounded service

Thesis

- These architectures will converge:
 - Similar design problems
 - Data will dominate microservices



For More...



photo: Banner Peak reflected in 1000 Island Lake, Ansel Adams Wilderness

For More...

Reactive Microservices Architecture

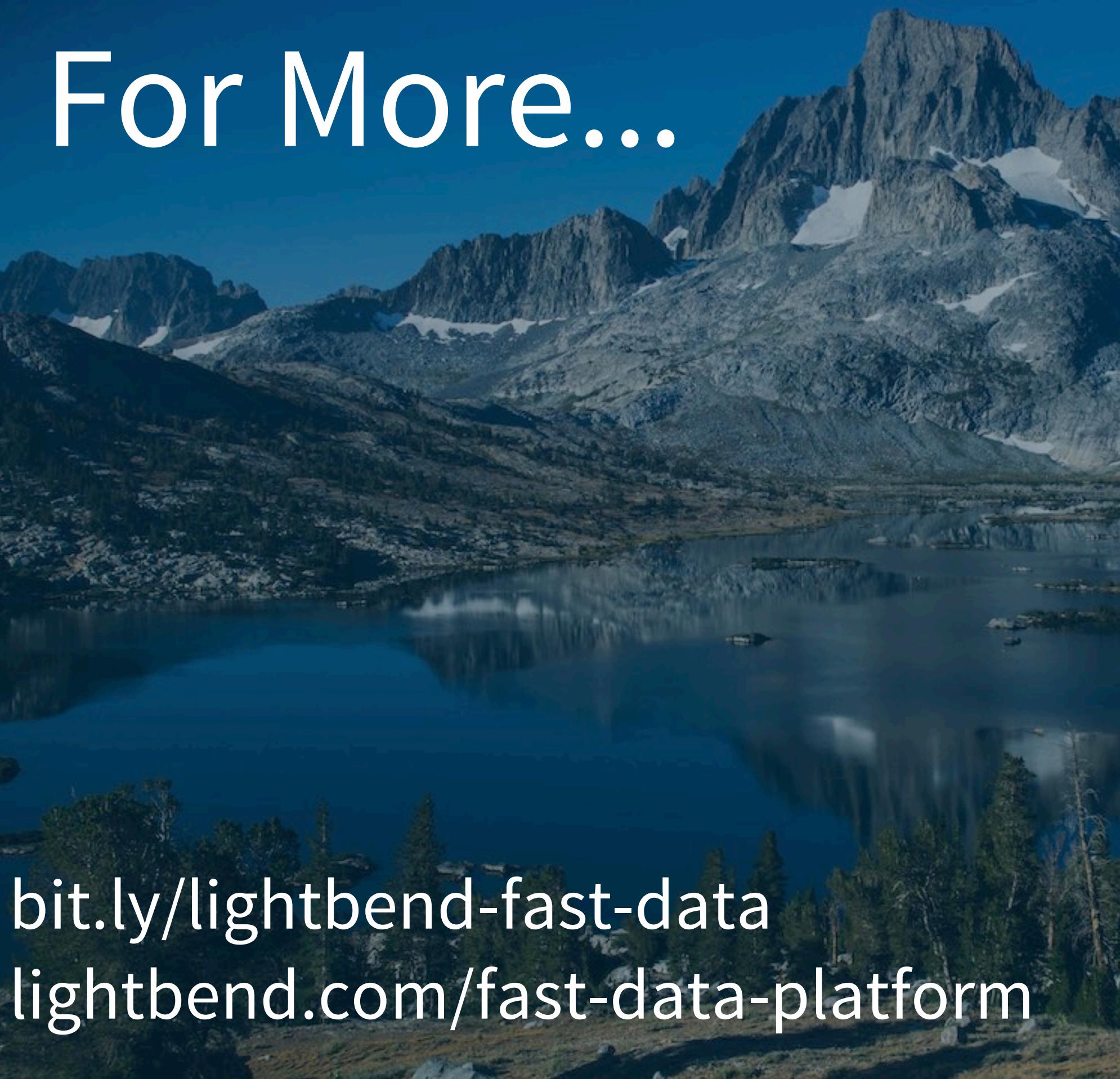
Design Principles for Distributed Systems

Jonas Bonér

lightbend.com/reactive-microservices-architecture

Jonas Bonér's book on architectures for Reactive Microservices

For More...



bit.ly/lightbend-fast-data
lightbend.com/fast-data-platform

O'REILLY®

Fast Data Architectures for Streaming Applications

Getting Answers Now from
Data Sets that Never End



Dean Wampler, PhD

First link is to this report I wrote, published by O'Reilly.
Second link is to learn more about what Lightbend is doing to help teams build fast-data, streaming architectures.

Compliments of
Lightbend

Thank You



dean@lightbend.com
@deanwampler

Photos, Copyright (c) Dean Wampler, 2014-2016, All Rights Reserved, unless otherwise noted. From the Ansel Adams Wilderness and Yosemite National Park, both in the Sierra Nevada Range, California, USA.

Other content Copyright (c) 2015-2016, Dean Wampler, but is free to use with attribution requested.

<http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>

photo: Morning Light and reflections in 1000 Island Lake of Banner and Davis Peaks, Ansel Adams Wilderness