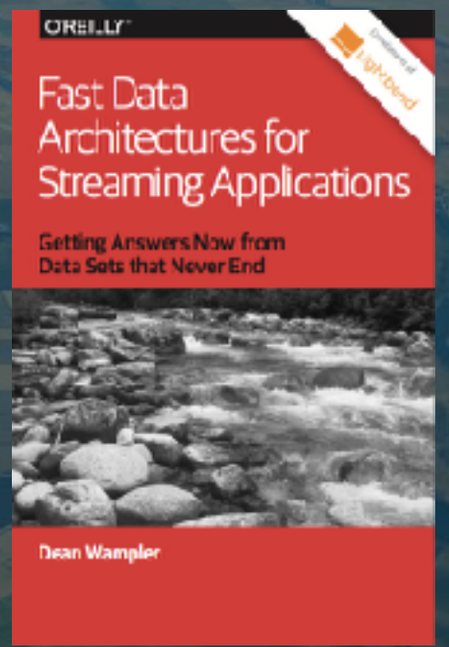
You can download this and my other talks from the polyglotprogramming.com/talks link.

Photograph: Tower Bridge, time-lapse photo at night.

Free as in 🍺

bit.ly/fastdata-ORbook

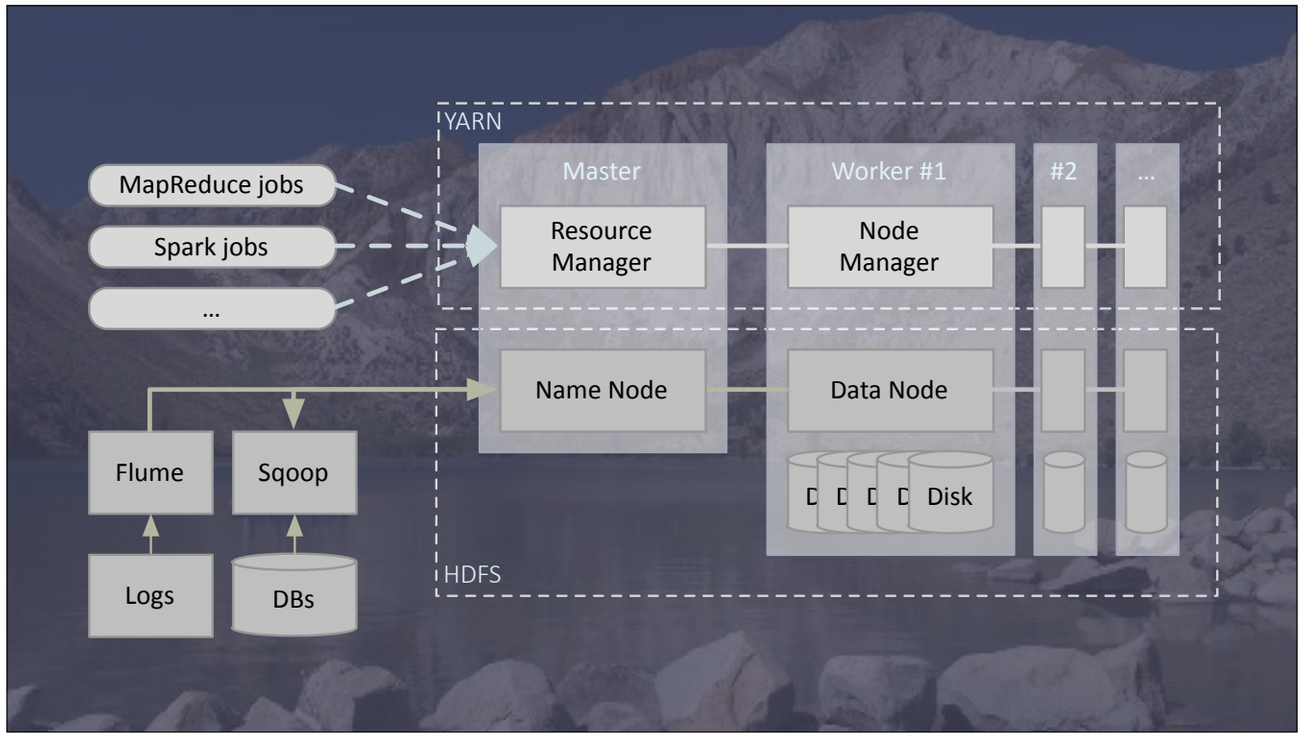


My book, published last fall that describes the points in this talk in greater depth. I've refined the talk a bit since this was published.

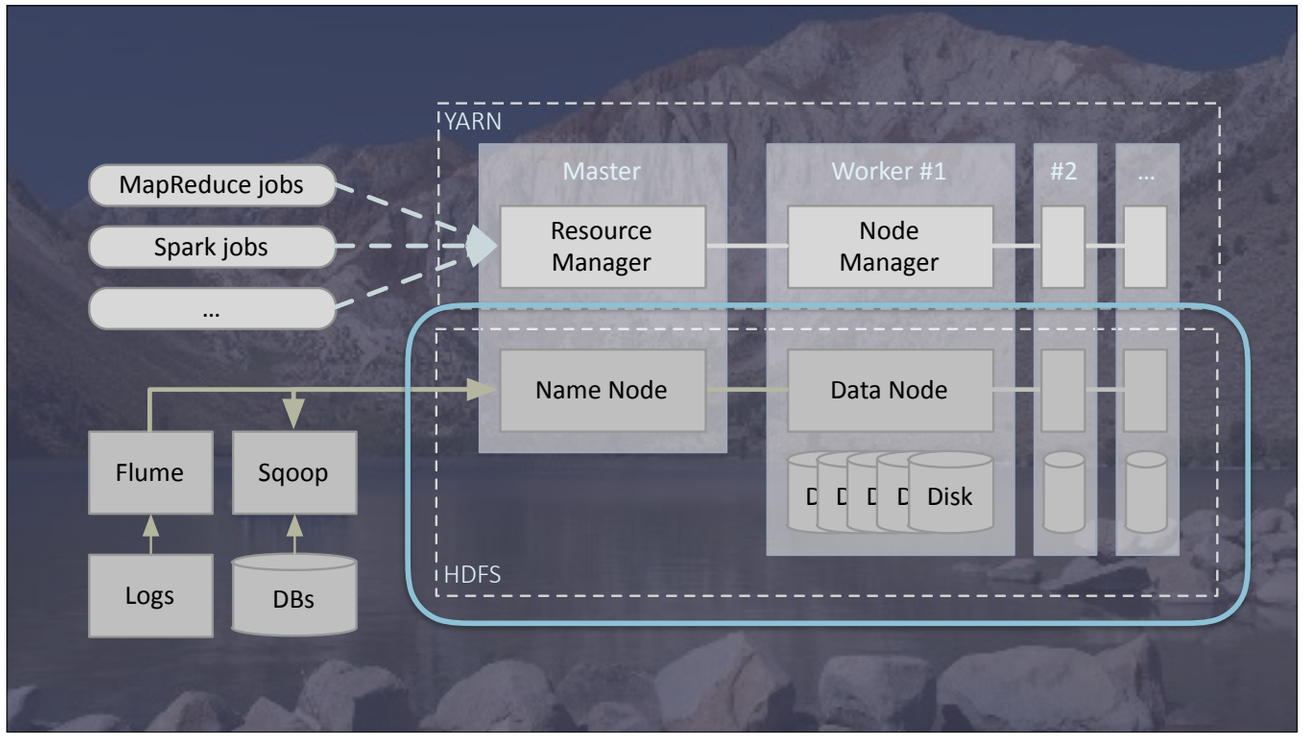
You can download this and my other talks from the polyglotprogramming.com/talks link.

Photograph: Farms in the Sacramento River delta, ENE of San Francisco .

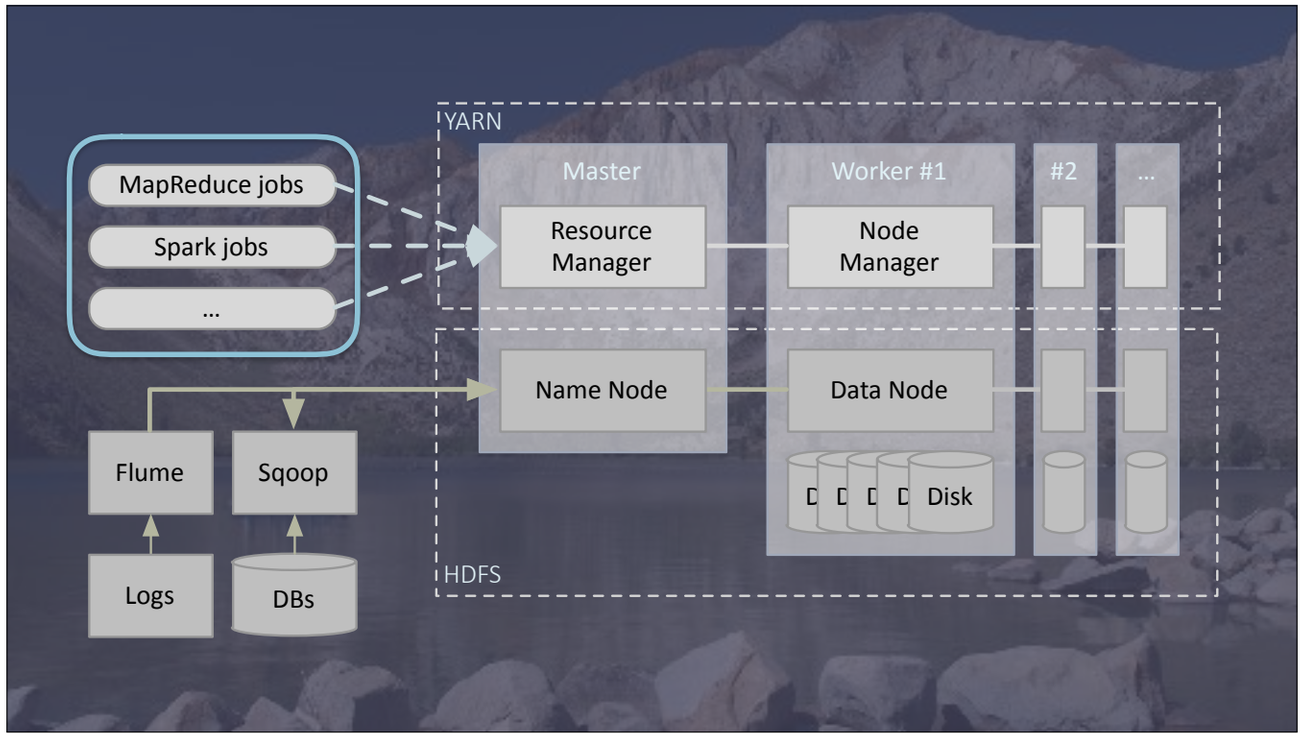
Photo: Convict Lake in the Sierras, California



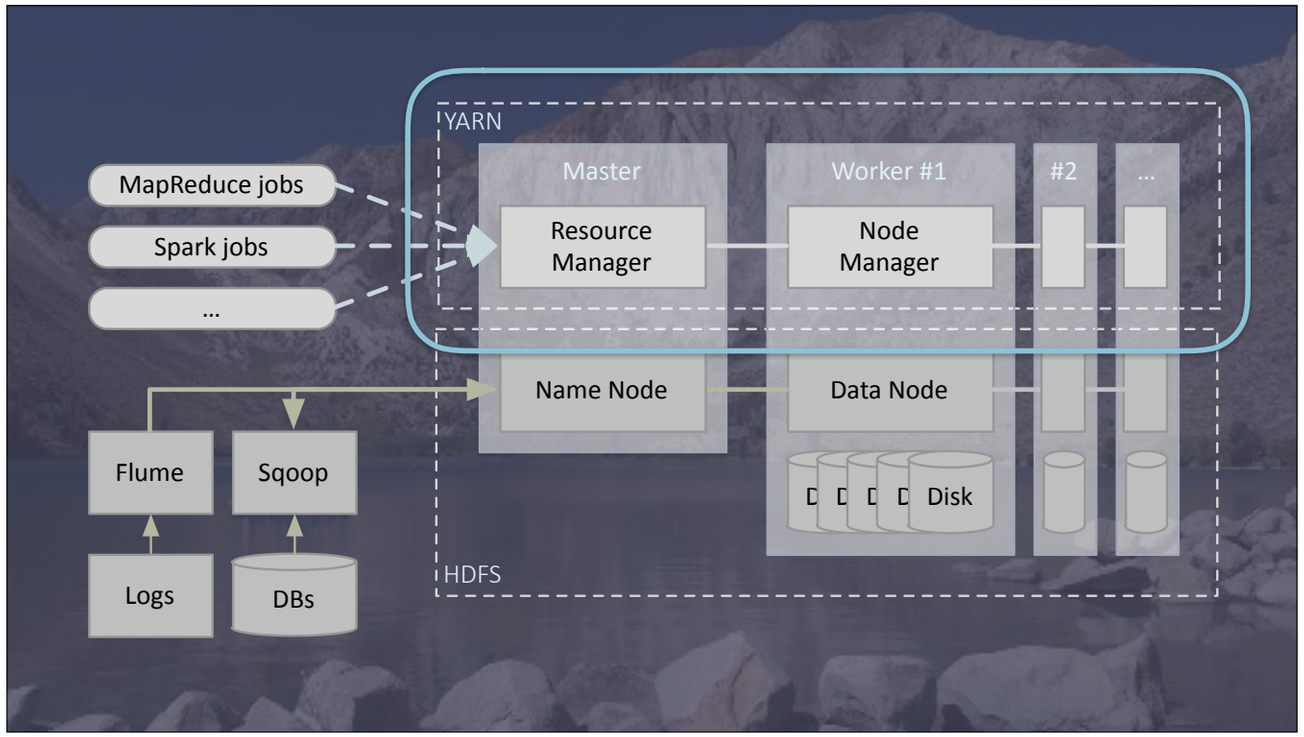
Schematic view of Hadoop. You need 3 core features...



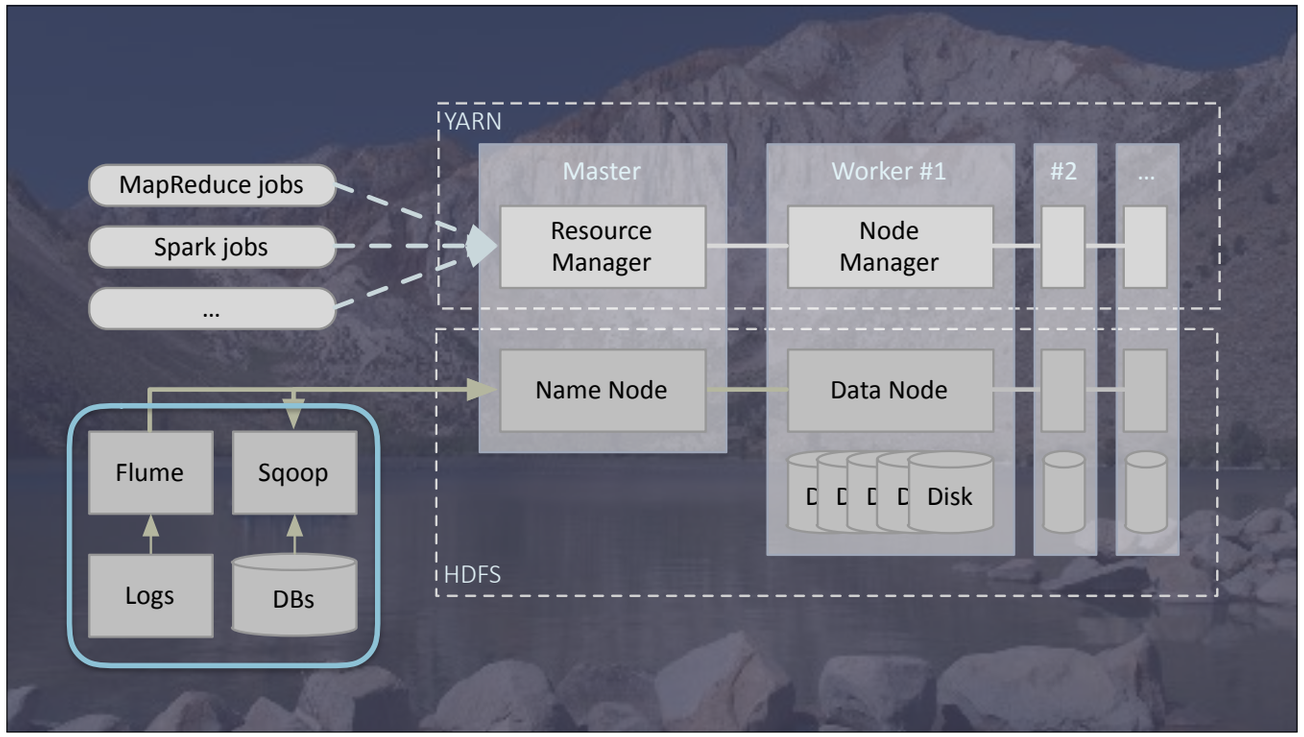
1. Storage tier, either HDFS (Hadoop Distributed File System) or alternatives like S3 or databases.



2. You need a compute engine for processing data. First we had MapReduce, then a few other tools to replace it, finally Spark has emerged as the most viable successor.

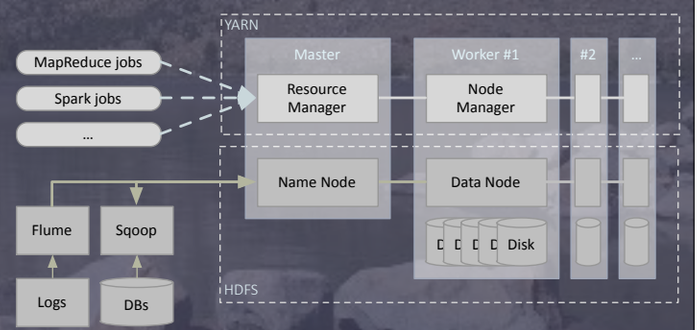


3. Finally, you need a manager of resources, scheduler of jobs and tasks, etc.



Other tools are built on this foundation or supplement it, like tools for ingesting data, such as Sqoop and Flume.

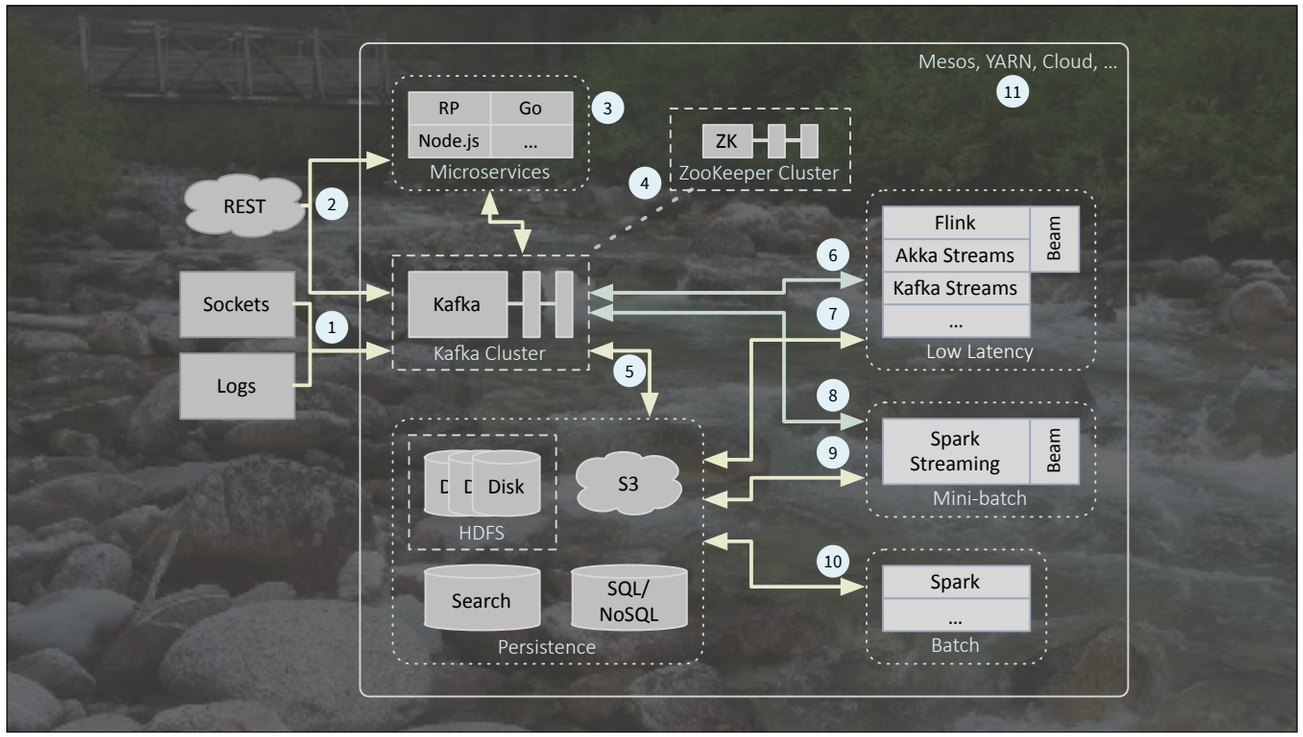
- Characteristics
  - Batch oriented
  - Massive storage
  - Multiuser jobs
- Data warehouse replacement



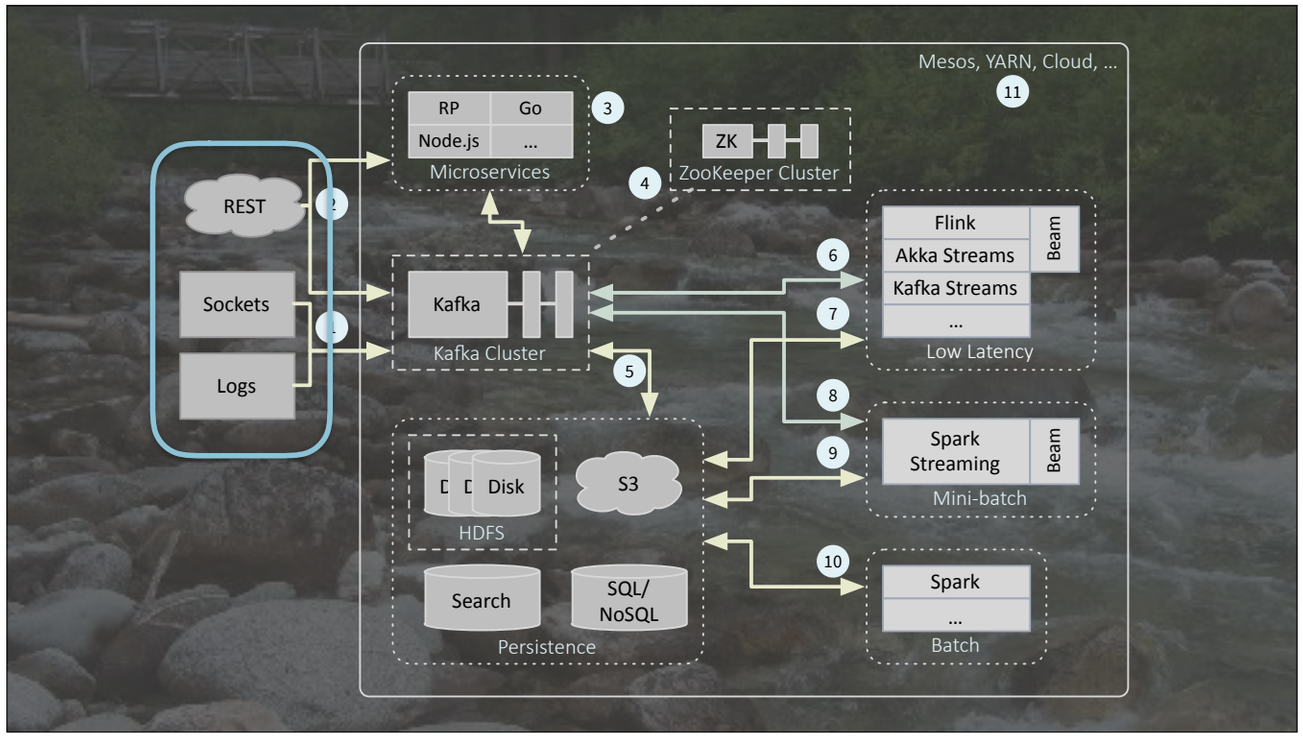
DW replacement is a common use of Hadoop, because it nicely supports SQL analysis over large data sets, using Hive, Impala, Kudu, etc.

But it will also support batch!!

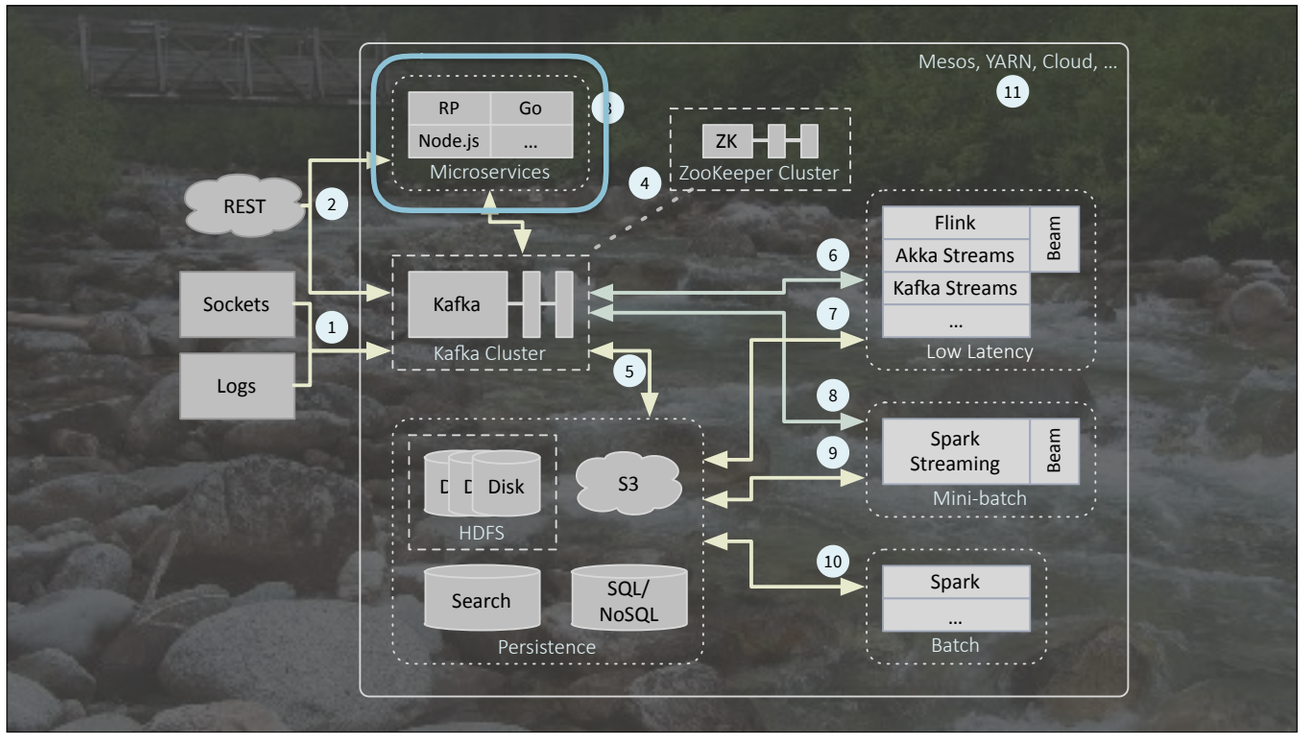
Photo: Stream in North Cascades National Park, Washington State.



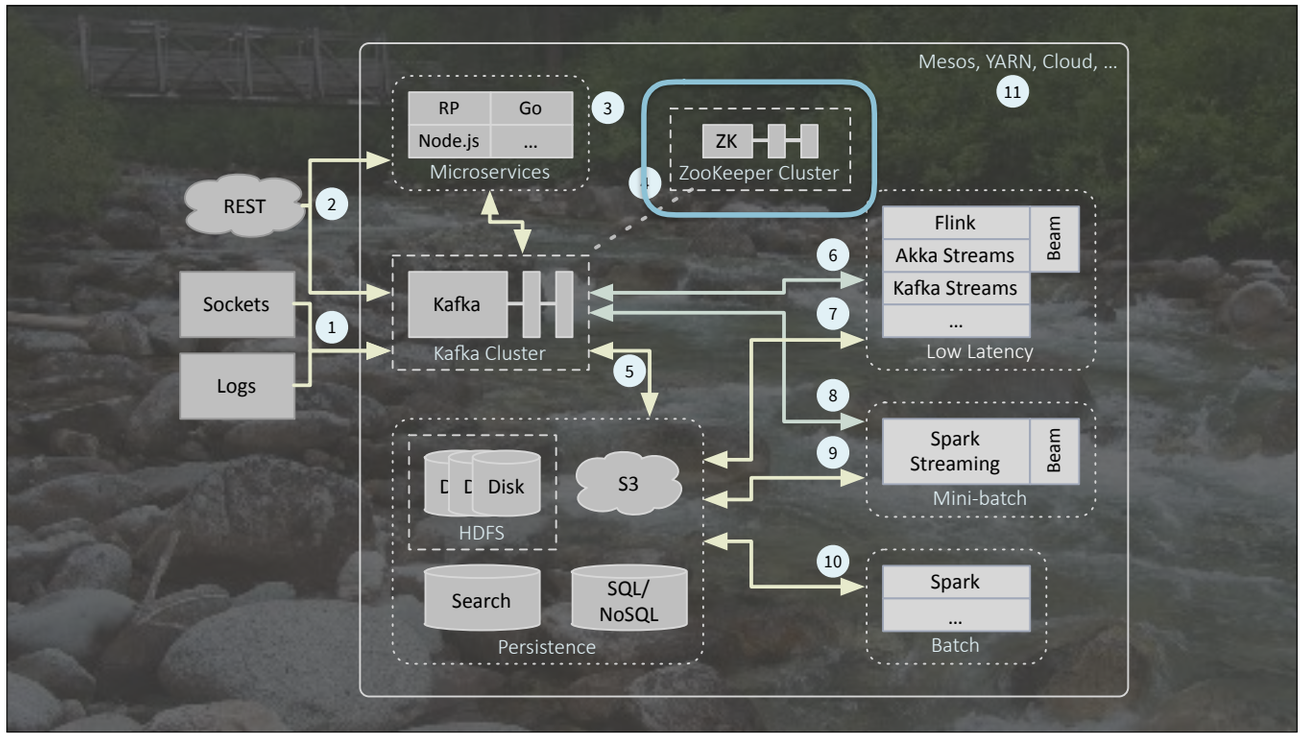
Numbering is in the report, so it’s easier for the text to refer back to the figure.



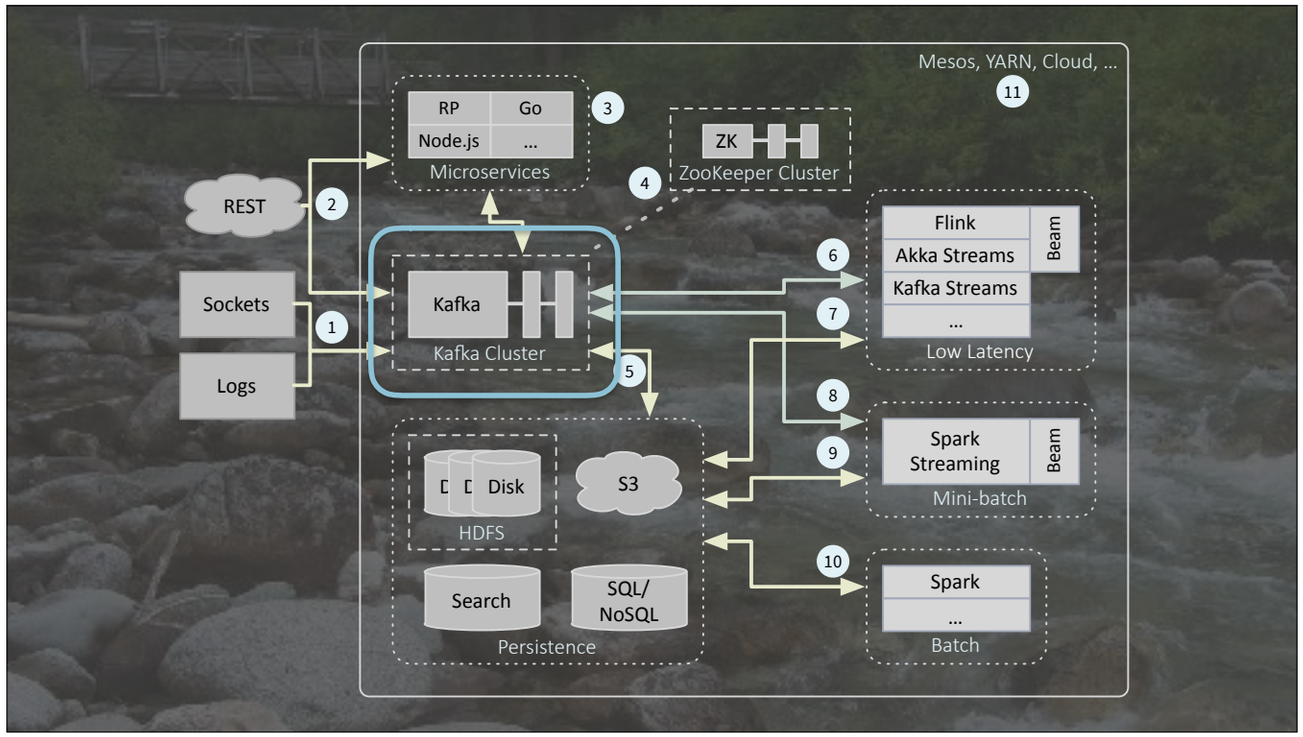
1 & 2. Data sources include streams of data over sockets and from logs. You might also ingest through REST channels, but unless they are async, the overhead will be too high, so REST might be used only for communicating with the microservices (3) you write to complete your environment.



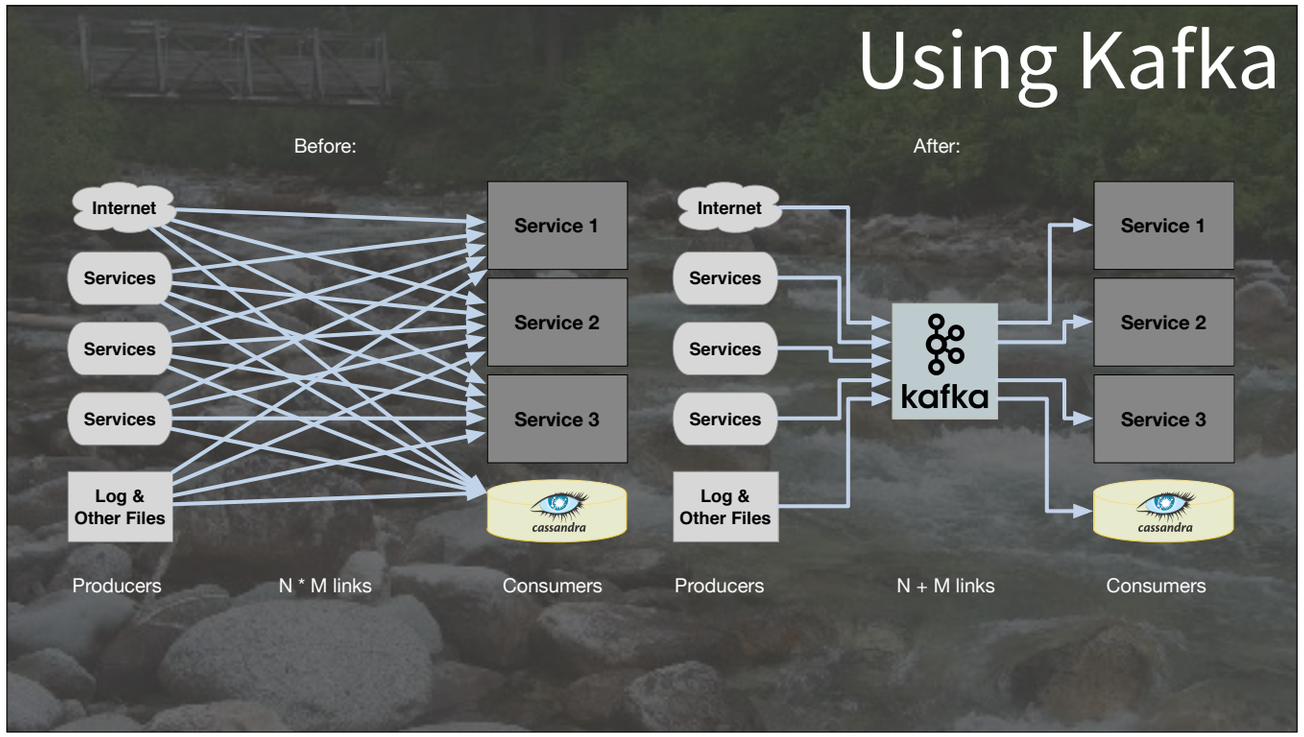
3. A real fast data environment is similar to more general services, you'll have the "heavy hitters" like Spark, Kafka, etc., but you'll also need to write support microservices to complete the environment.



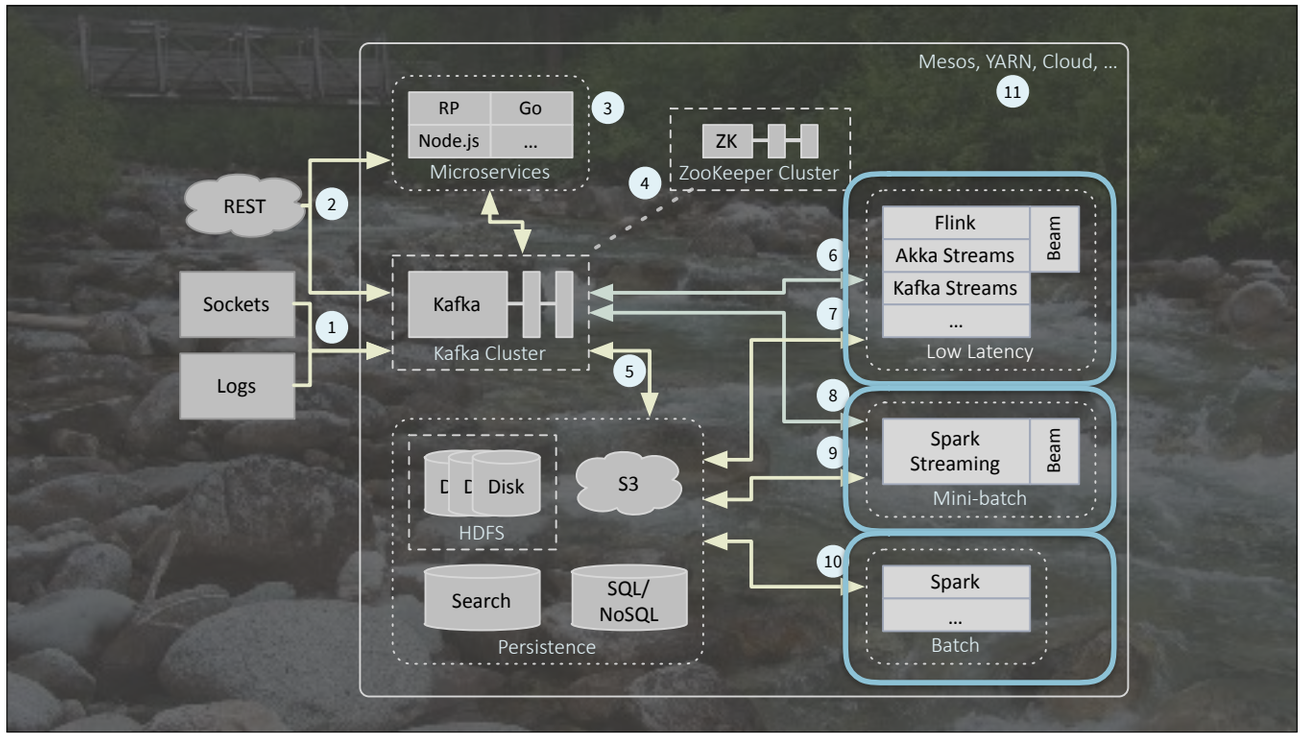
4. Some services, like Kafka, require ZooKeeper for managing shares state, such as leaders.



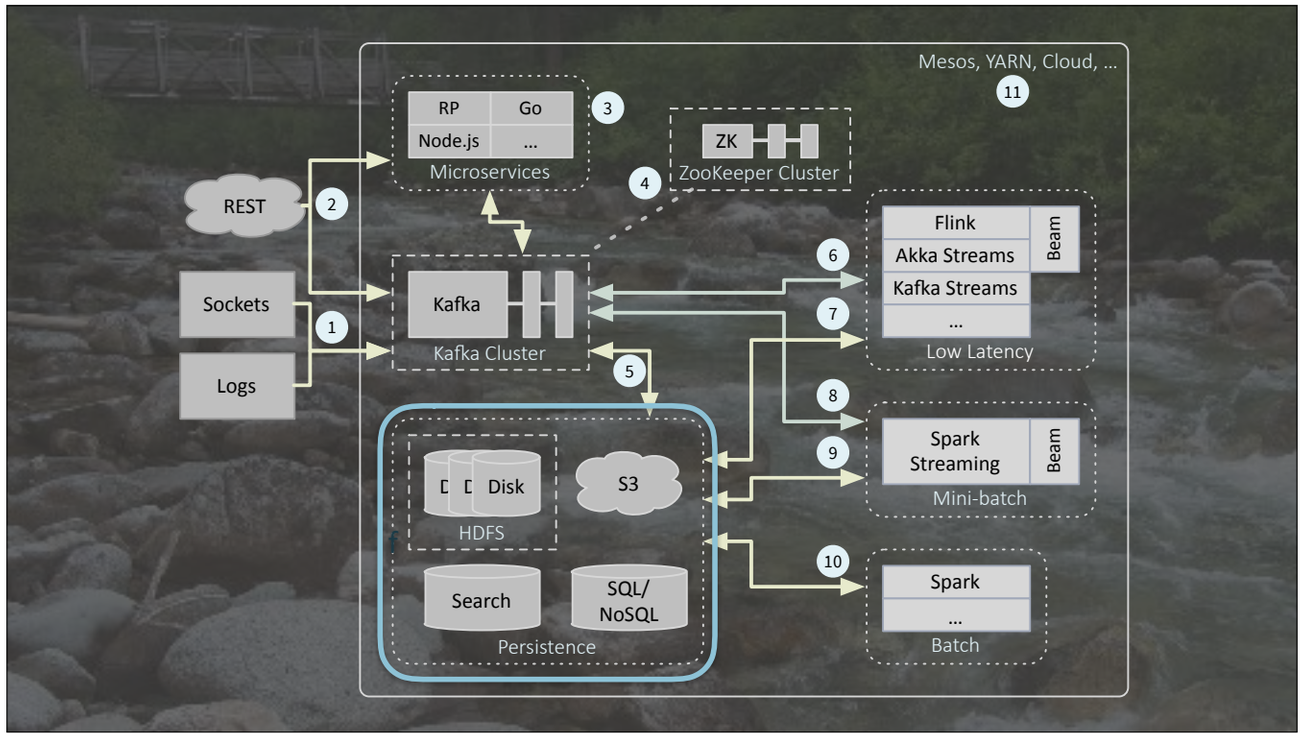
5. Kafka is the core of this architecture, the place where data is ingested in queues, organized into topics. Publishers are consumers are decoupled and N-to-M. Kafka has massive scalability and excellent resiliency and data durability. All services can communicate through each other using Kafka, too, rather than having to manage arbitrary point-to-point connections.



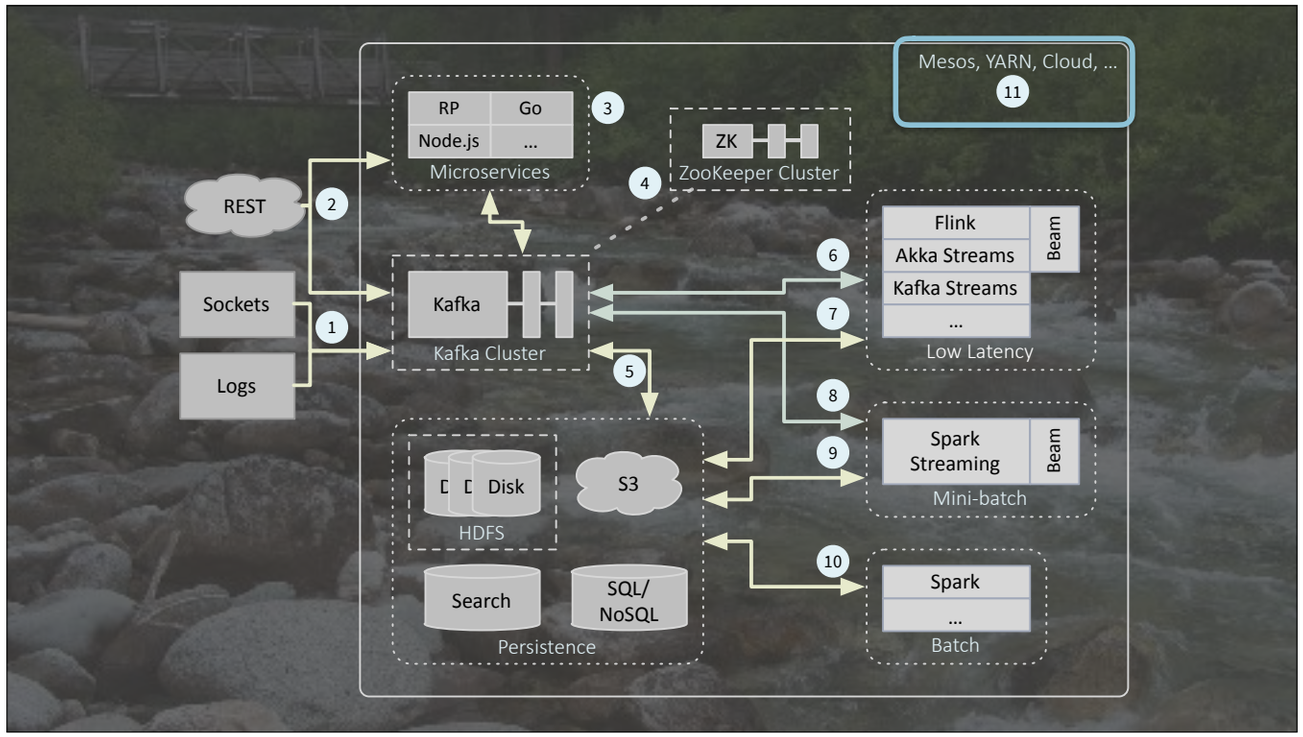
Kakfa can simplify your architecture, too. Not only is this cleaner, it’s more robust, as the point-to-point connections are more fragile, more likely to result in data and service loss if one point goes down, where Kafka keeps the other point “healthy” while the failed point is restored. Kafka also lets you easily support multiple consumers or producers per topic (the way data is organized, as in classic message queues). Finally, the uniformity of always (or most of the time) communicating through Kafka simplifies the challenge of connecting services together with different APIs.



6, 8, and 10. Many pure streaming, mini-batch streaming, and batch engines are vying for your attention. We'll focus into this area after this overview.



5, 7, 9, and 10. Data can also be read and written between these compute engines and storage, not just Kafka. For 5, Kafka Connect is used.



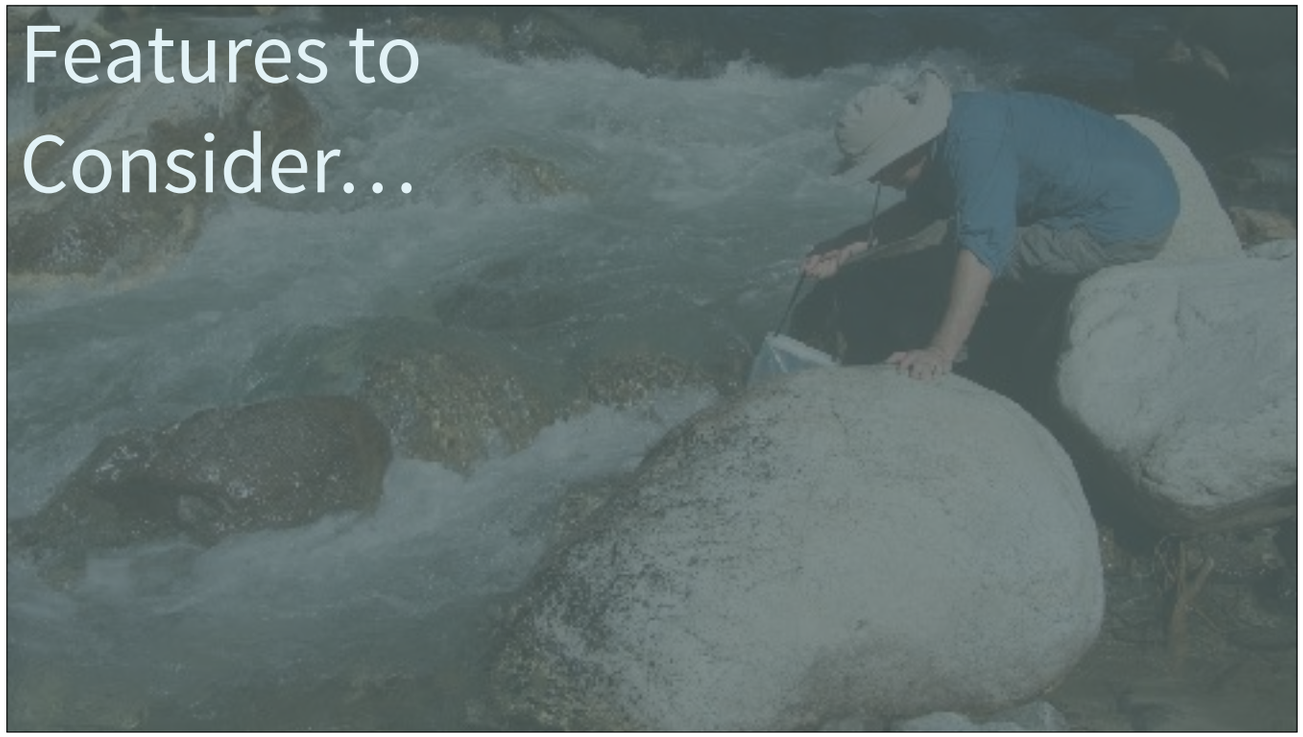
11. You can run this architecture on Mesos, YARN (Hadoop), on premise or in the cloud. (At Lightbend, we think Mesos is the best choice.)

Streaming Engines

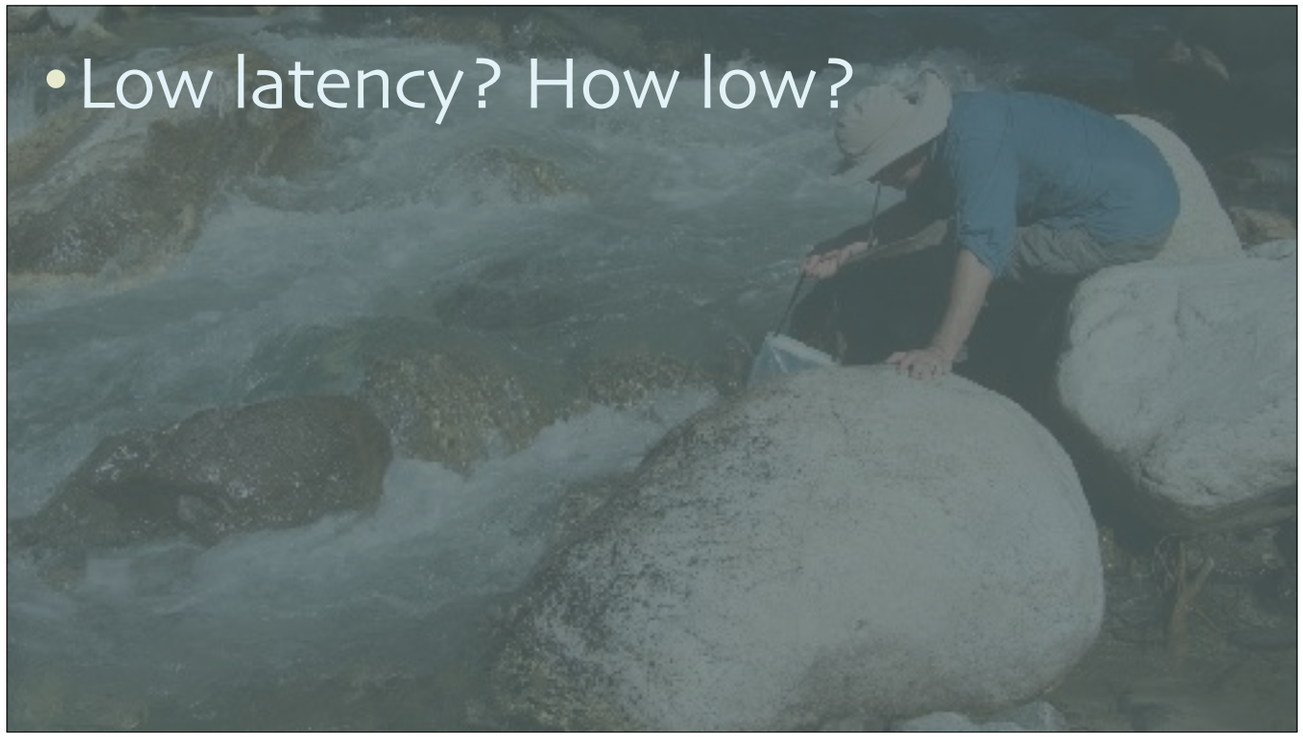
Let’s dive into a representative set of available streaming engines.

Photo: Stream in North Cascades National Park, Washington State.

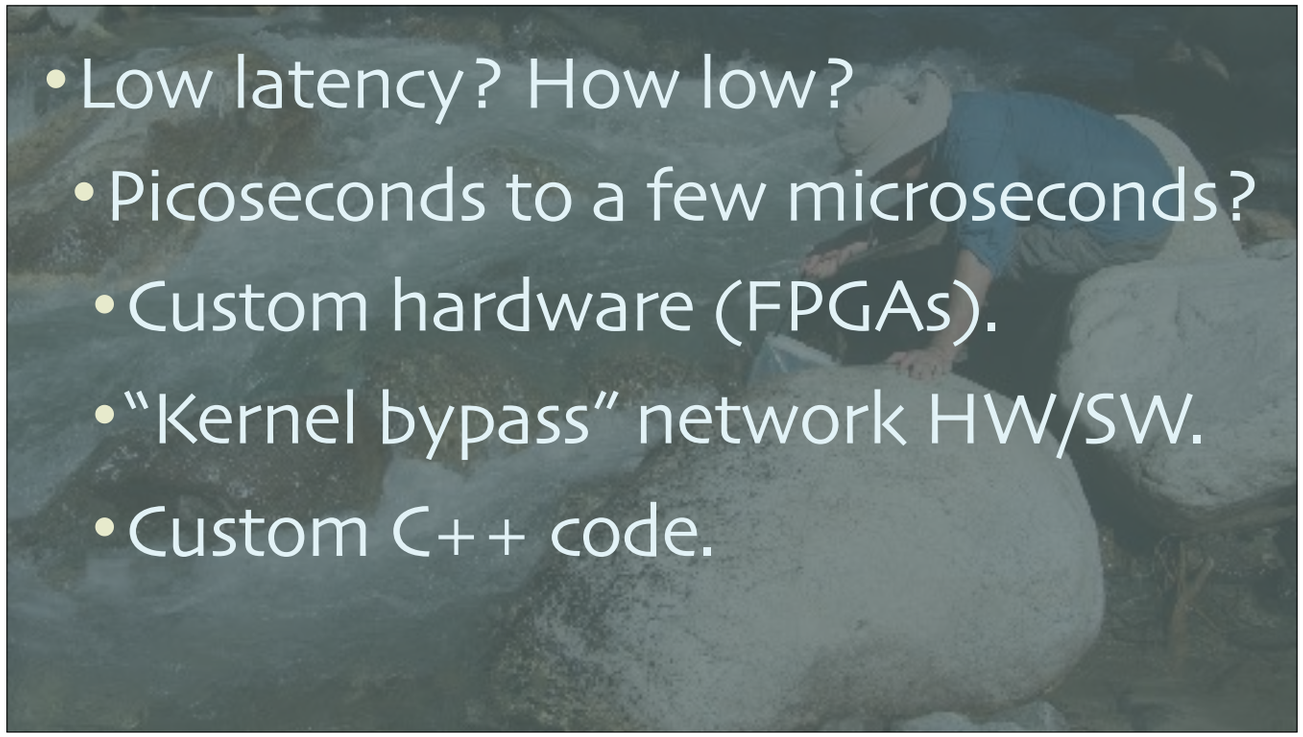
Features to Consider…



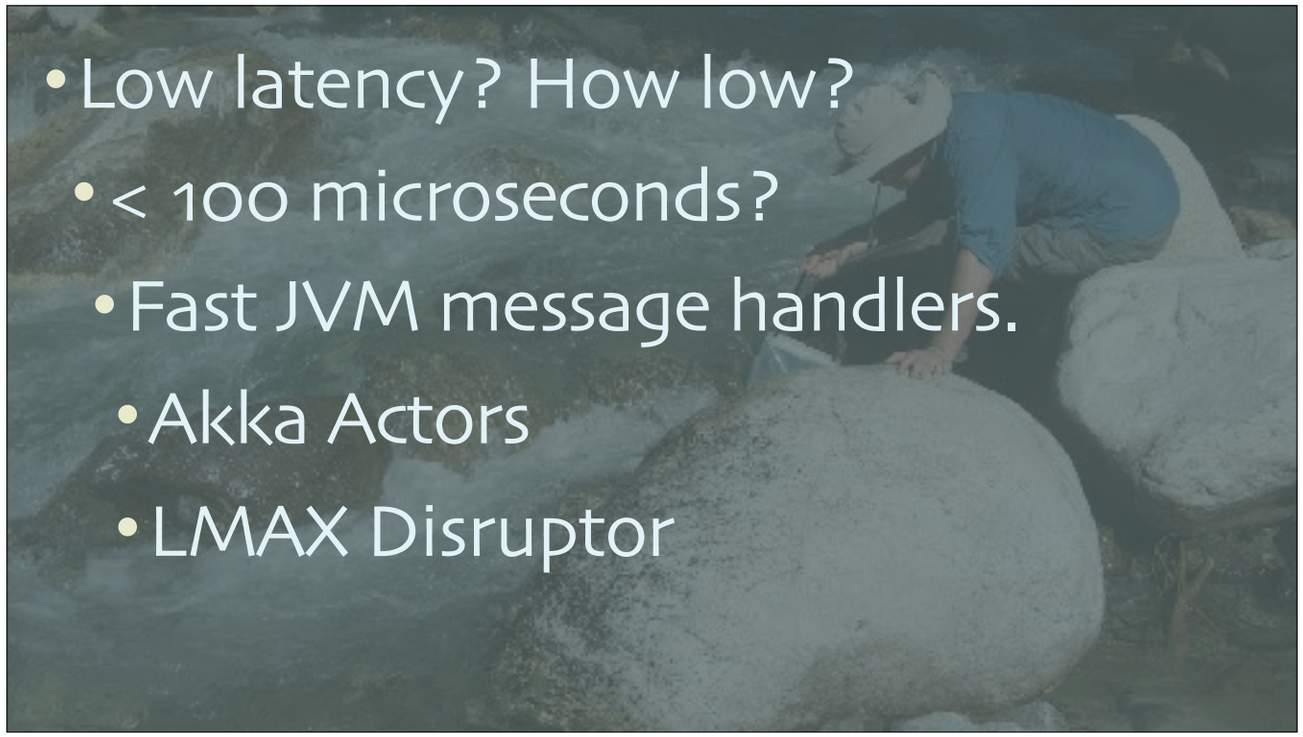
When deciding what to use, there are several dimensions or features to consider.



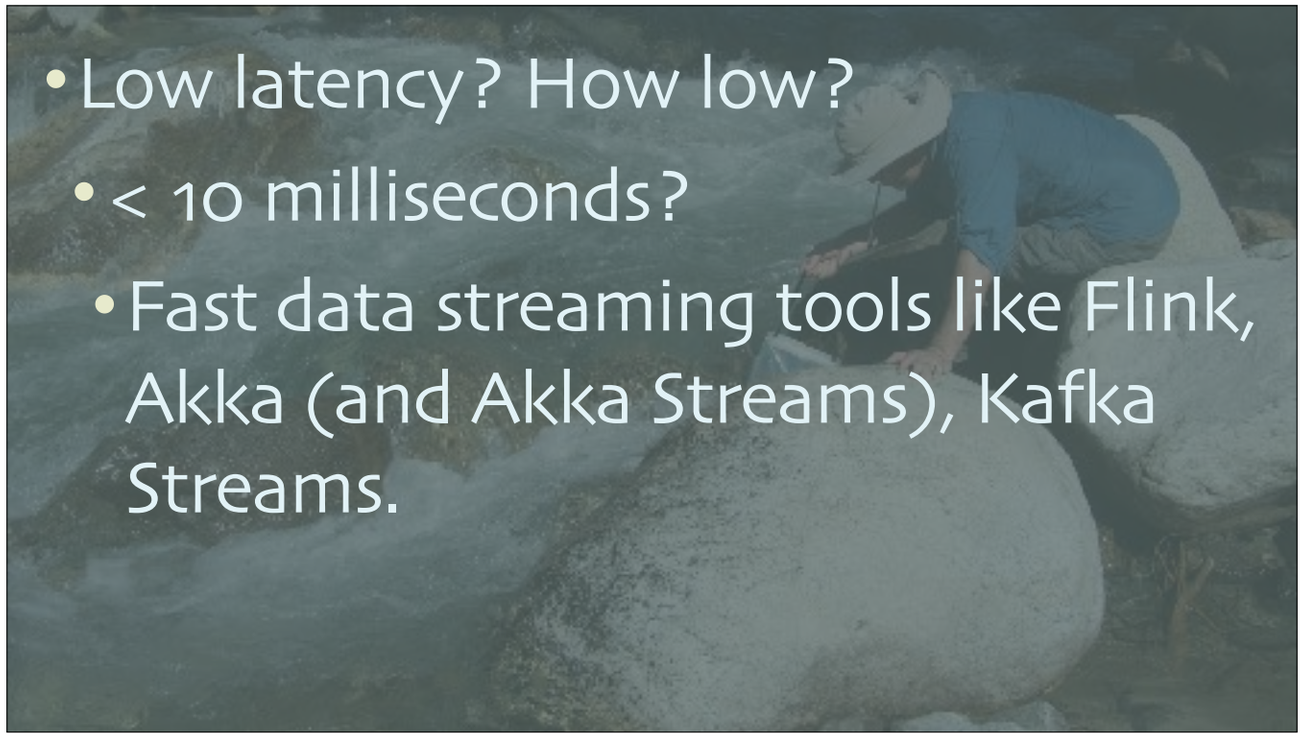
If you have tight latency requirements, you can’t use a mini-batch or batch engine. If you’re constraints are more flexible, you can do more sophisticated things, like training ML models, writing to databases, etc.



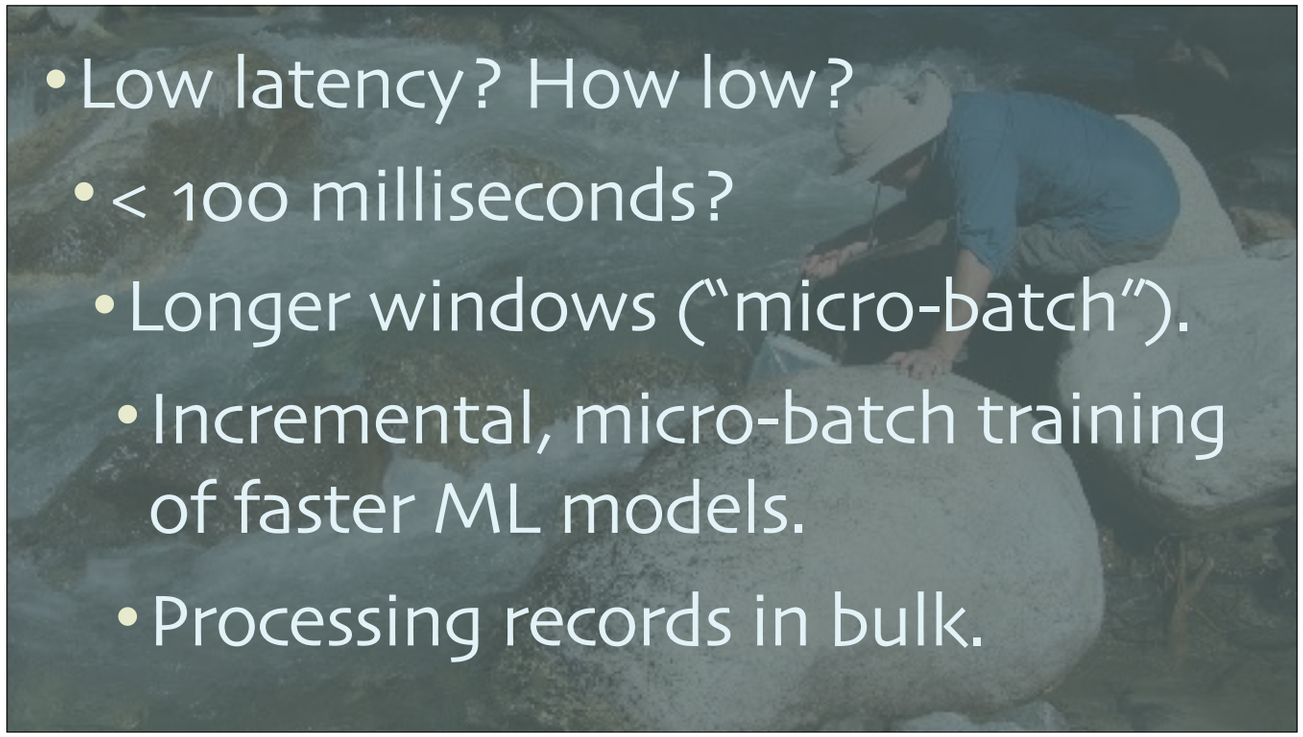
Extremely low latency, high-frequency trading territory, requires custom everything.



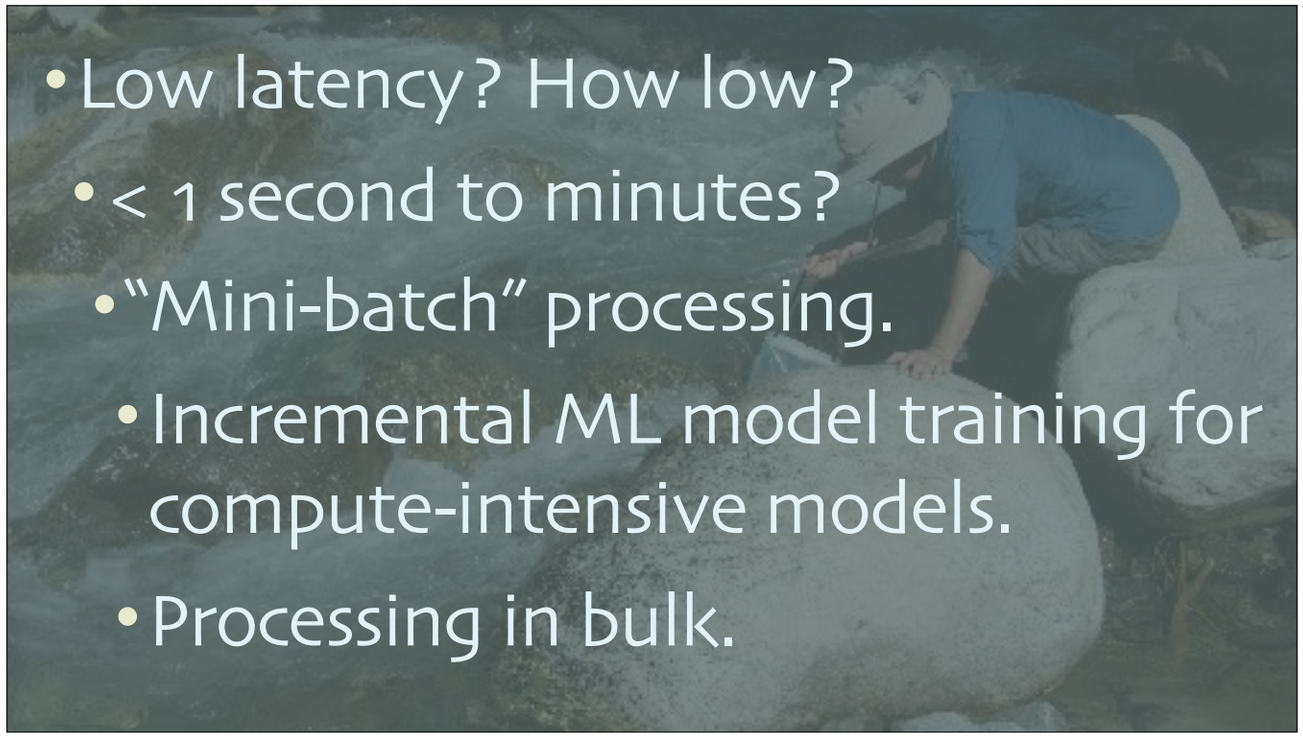
With more latency tolerance, efficient JVM tools can be used, like Akka Actors and Disruptor (https://lmax-exchange.github.io/disruptor/).



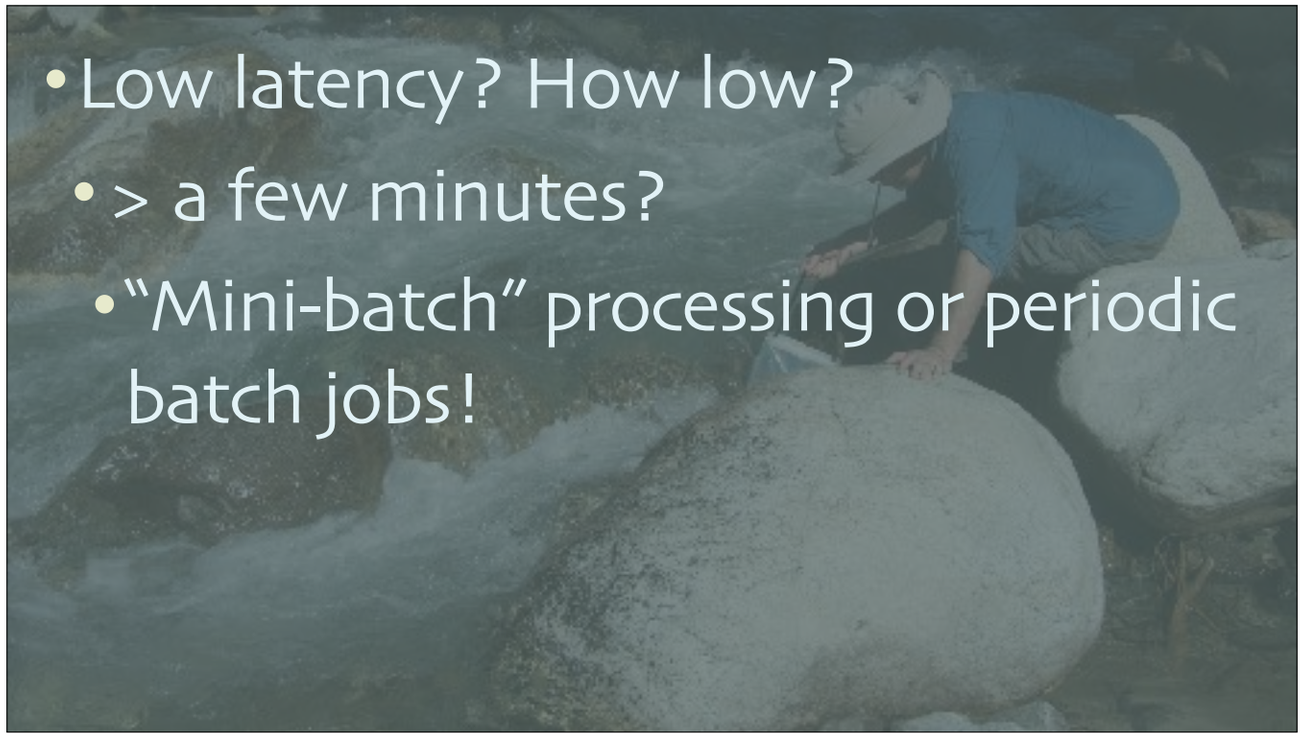
Now we’re in the low end of what high volume, fast data architectures are designed for. Note that part of the latency here is the end-to-end system latency for passing through Kafka (with short topic queues!!), etc.



The mainstream of fast data processing, where you want quick results, but you can tolerate some time for incremental training of ML models and/or you prefer to process records in bulk, rather than individually. At the upper end of usability for responding to human requests (using the usual 200 msec round-trip response time upper limit), but don't forget about the rest of the latency in the roundtrip request-response pipeline!

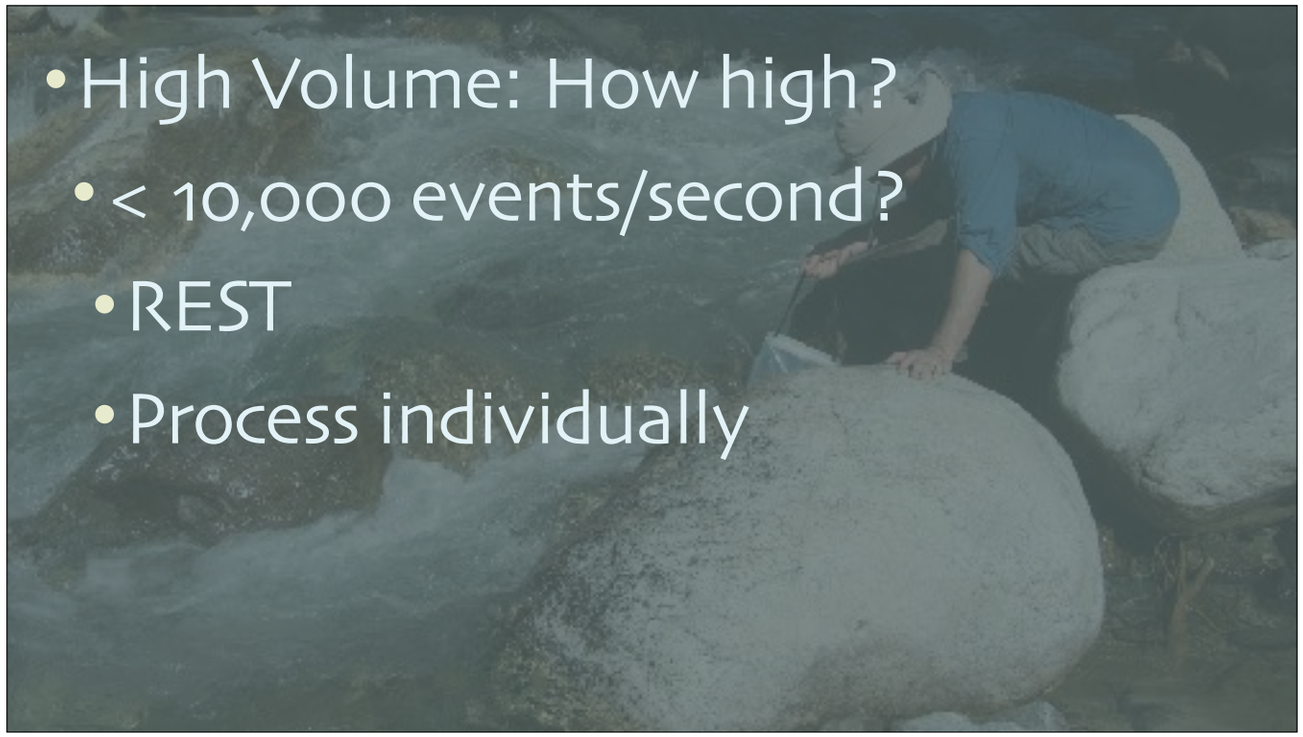


The realm of tools like Spark Streaming, where larger "mini-batches" (vs. micro-batches) are processed. Used for more expensive, incremental model training and larger bulk processing.

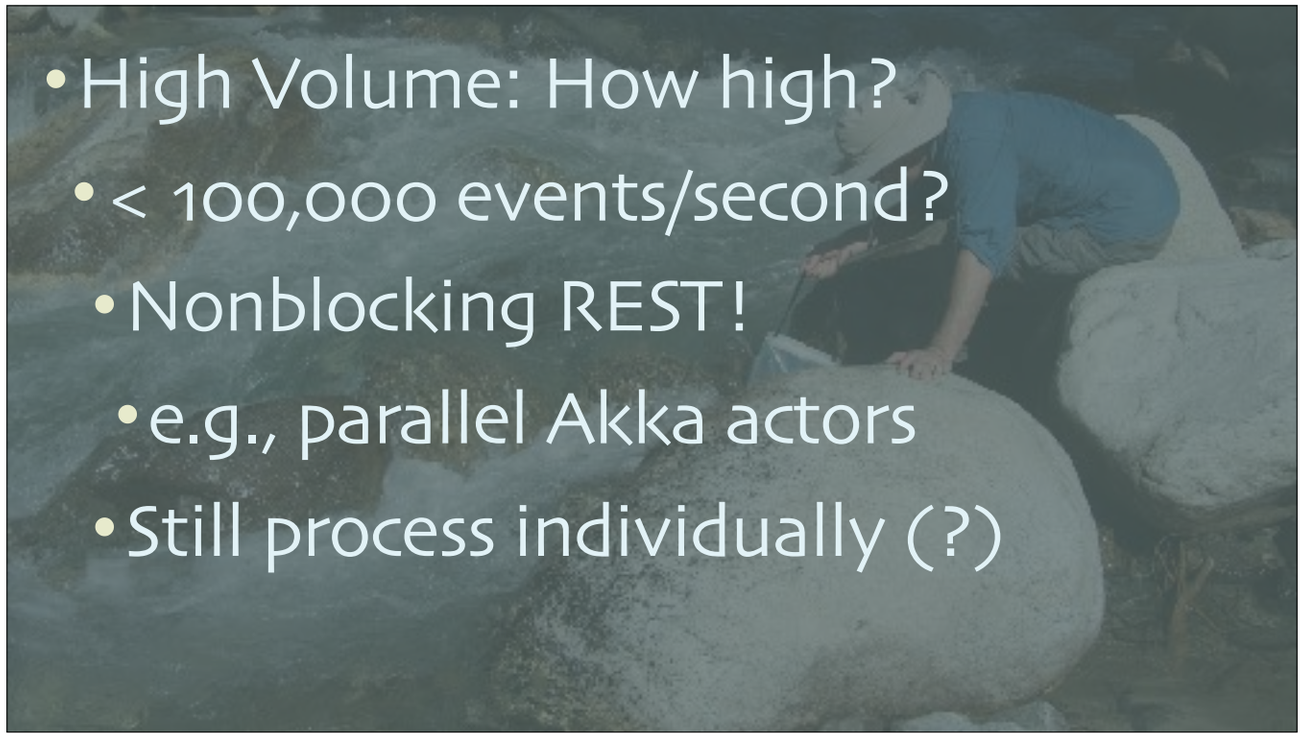


Once the latency is this high, it might actually make more sense to run frequent batch jobs, rather than keep a long-running, mini-batch system healthy.

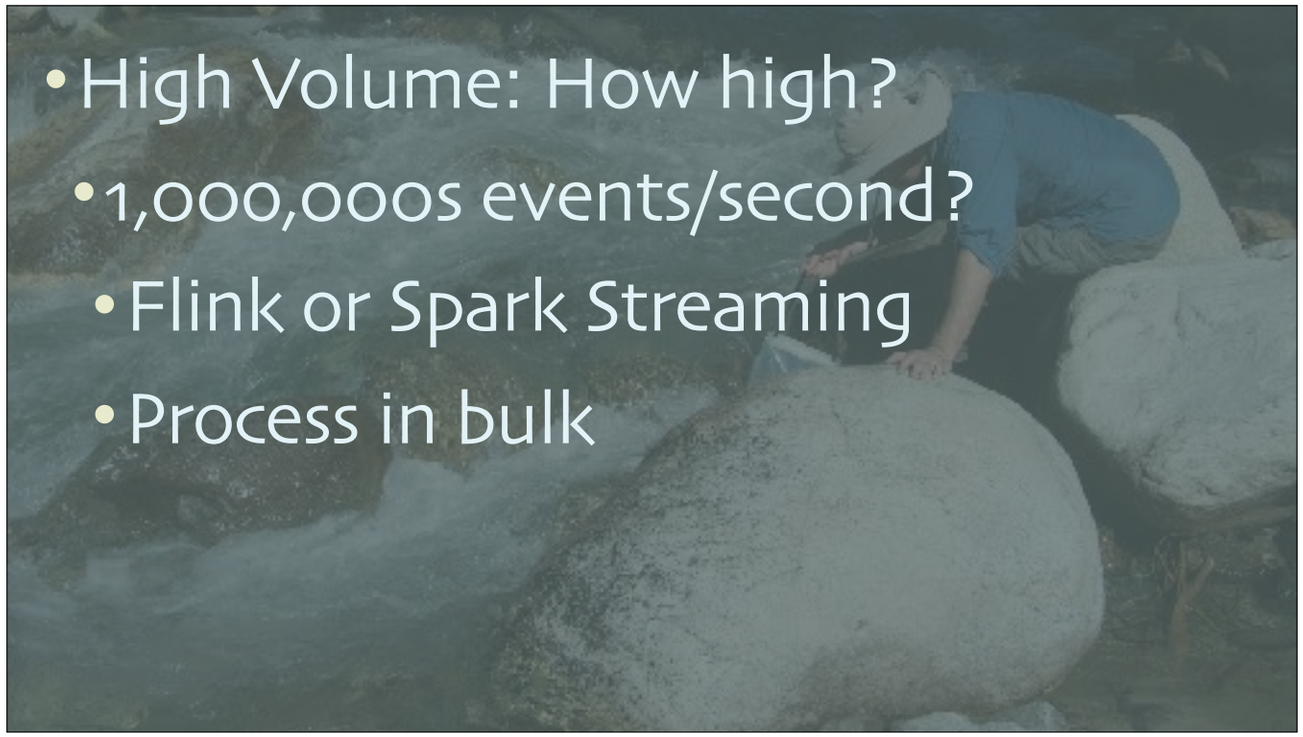
Some tools are better at large volumes than others. If you have low volumes, you'll want tools that are efficient at low volumes, whereas some of the high-volume tools are efficient per record when amortized over the stream.



These are relatively small volumes by modern standards and can be serviced individually by standard REST approaches.



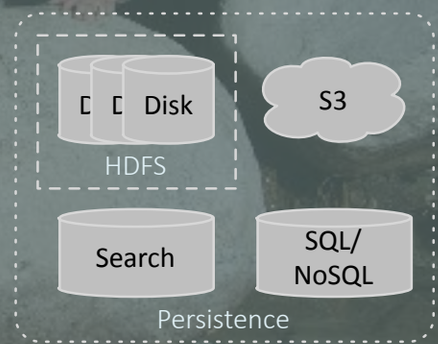
Some tools are better at large volumes than others. If you have low volumes, you'll want tools that are efficient at low volumes, whereas some of the high-volume tools are efficient per record when amortized over the stream.



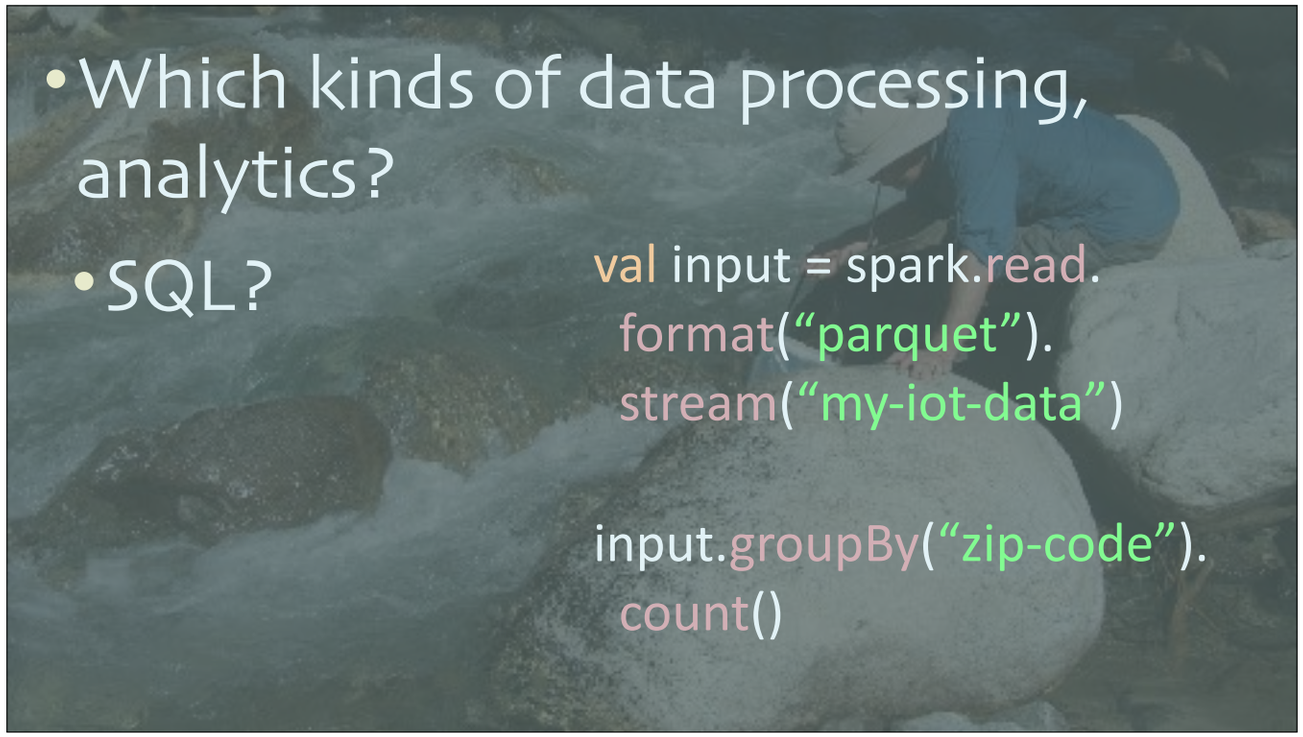
Some tools are better at large volumes than others. If you have low volumes, you'll want tools that are efficient at low volumes, whereas some of the high-volume tools are efficient per record when amortized over the stream.

Connecting everything through Kafka can eliminate this requirement, but often that’s not ideal and you may need some tools to have direct connections to other tools.

- Integration with other tools.
  - Akka, Flink, & Spark integrate with Databases, Kafka, file systems, REST, …
  - Kafka Streams only read & write Kafka topics.

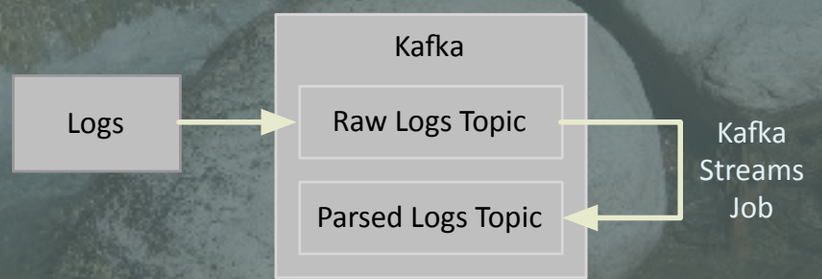


Connecting everything through Kafka can eliminate this requirement, but often that's not ideal and you may need some tools to have direct connections to other tools. In most cases, there are plenty of connection options, but Kafka Streams deliberately only reads and writes Kafka topics. (However, Kafka Connect can be used to move data between Kafka and other stores.)

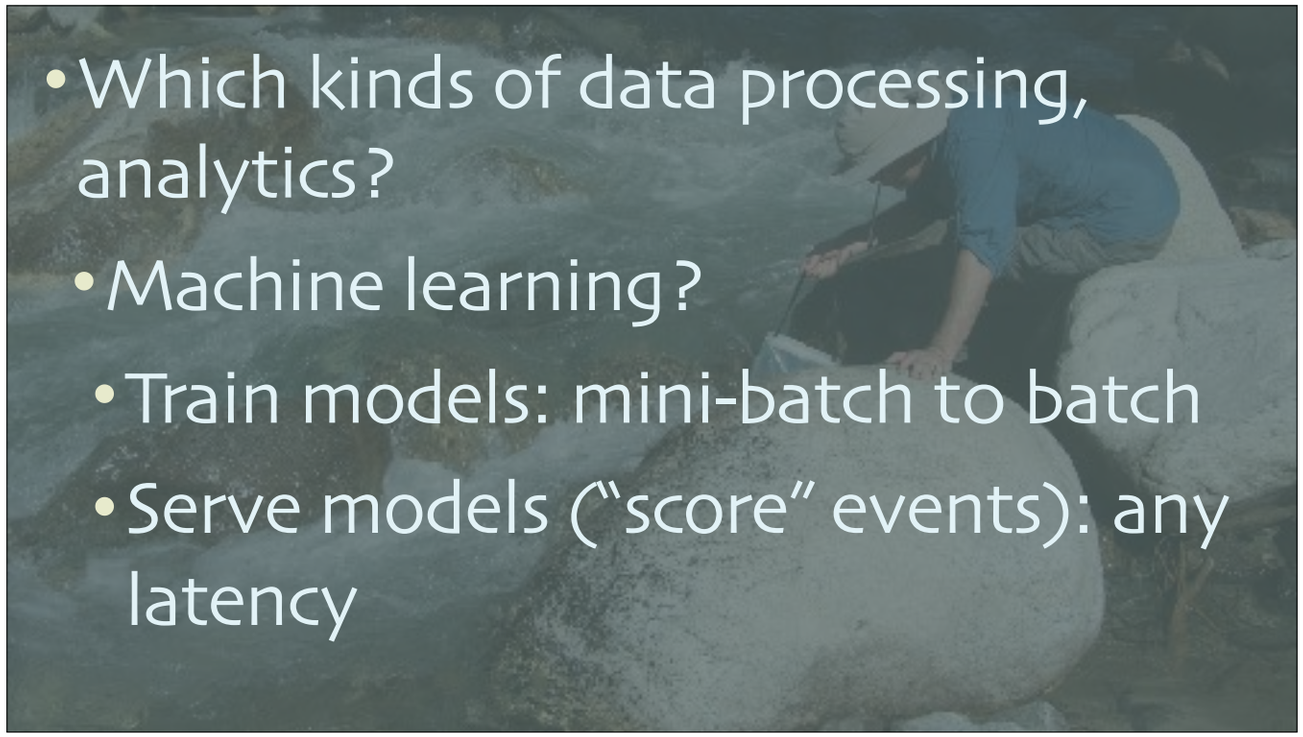


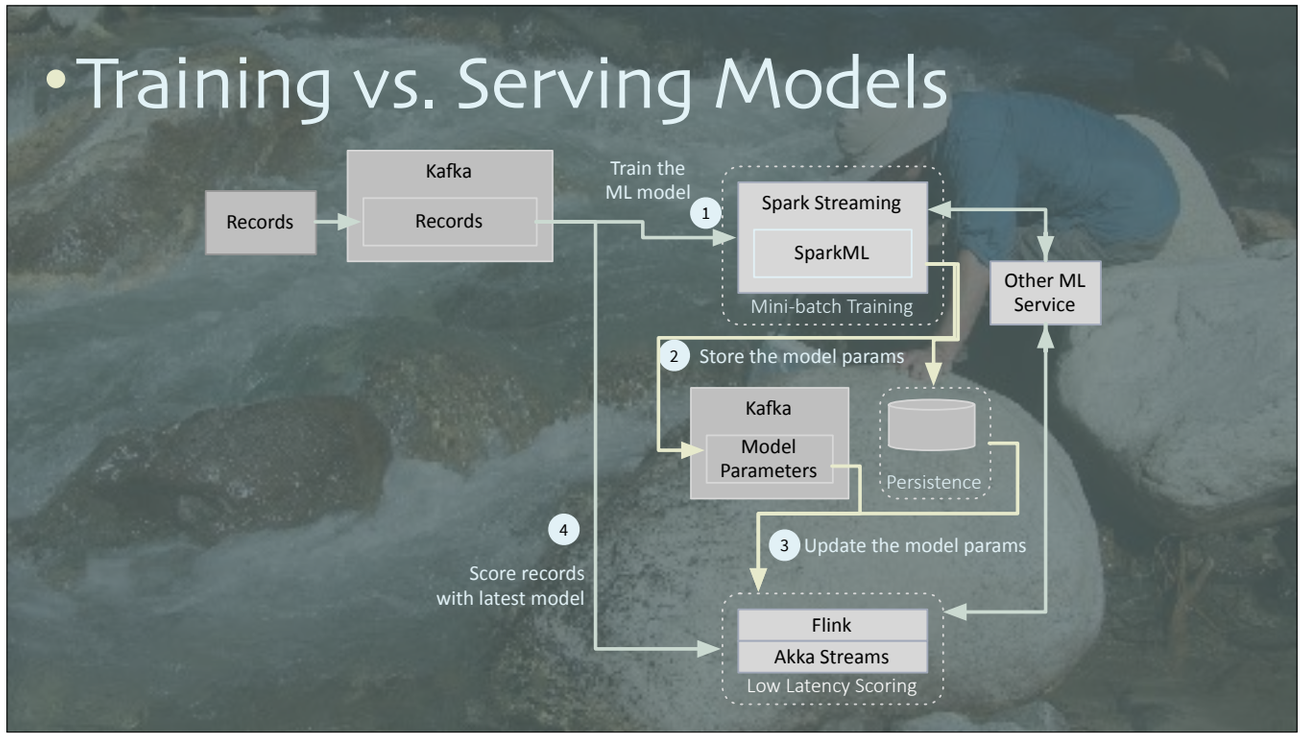
Spark Structured Streaming example using the SQL DSL in Scala, but I could have used actual SQL.

- Which kinds of data processing, analytics?
  - Extract, transform, and load (ETL)?

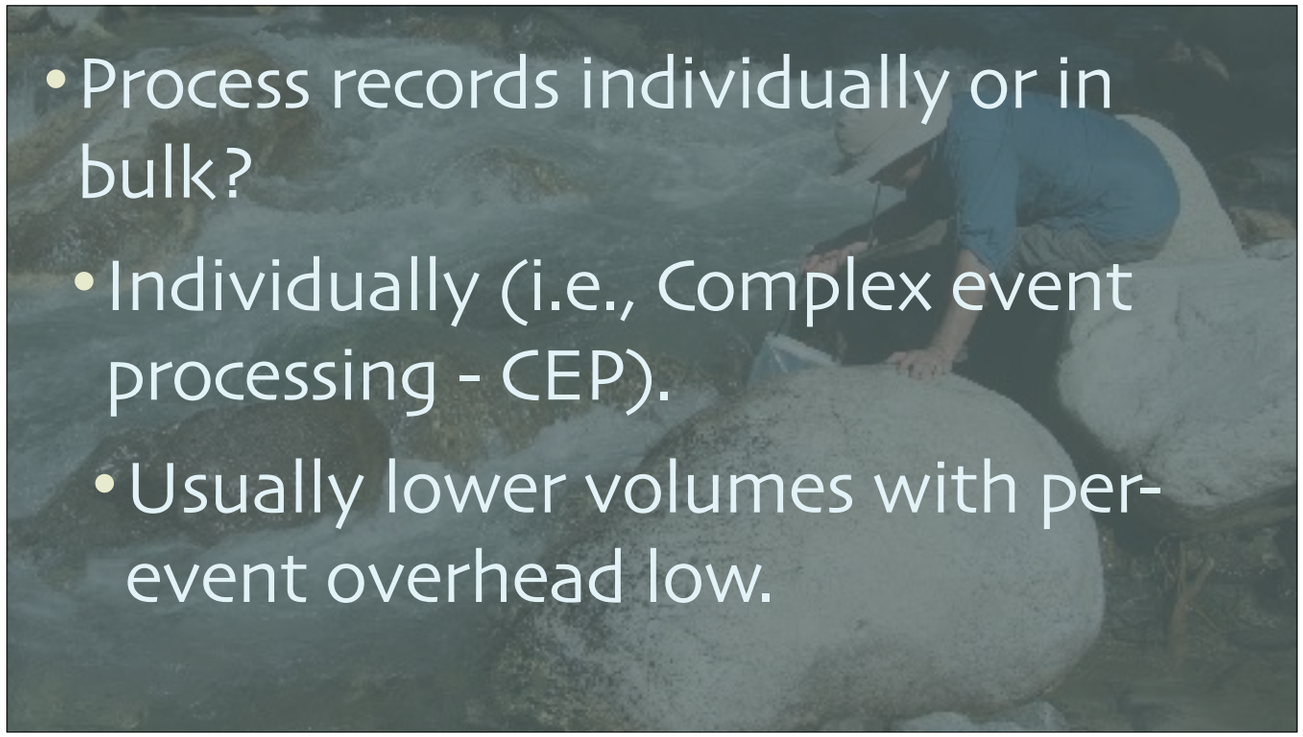


Simple example where raw (text?) logs are ingested into one Kafka topic, then processed with a Kafka Streams job to clean them and transform them into a more suitable format for downstream consumption.

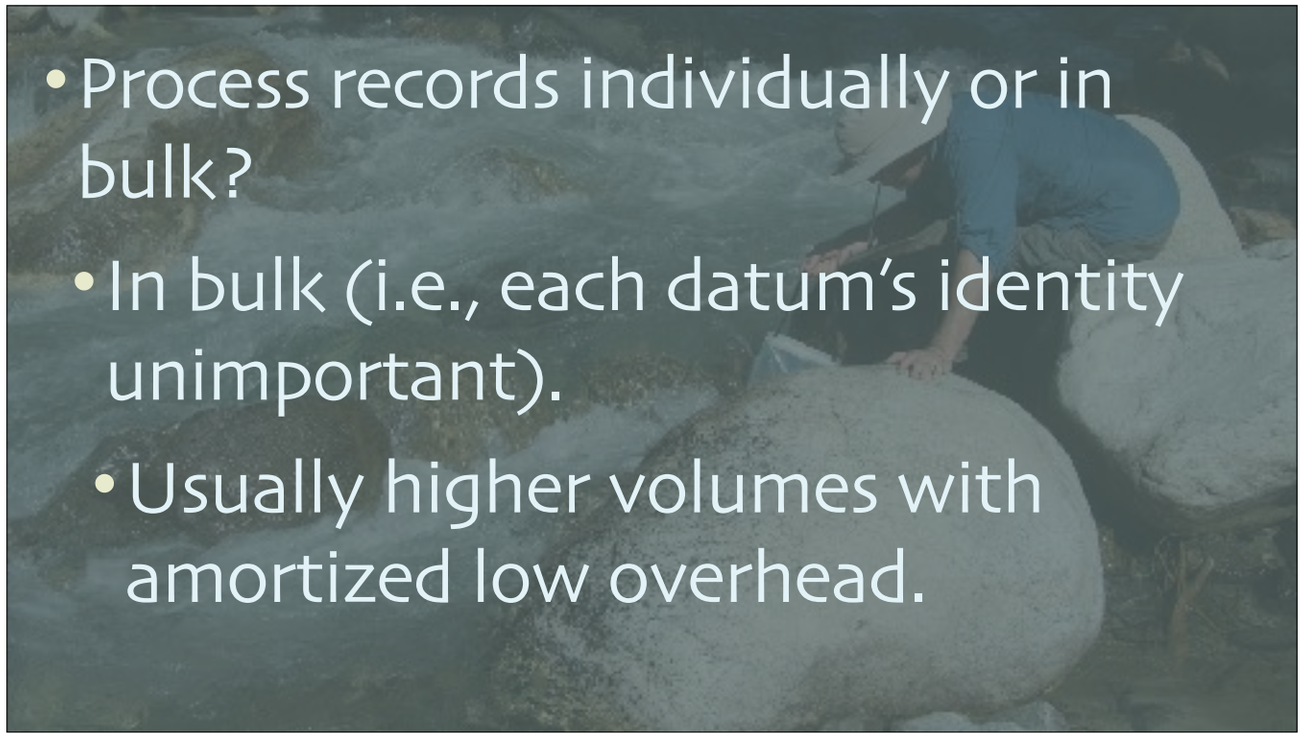

• Which kinds of data processing, analytics?
• Machine learning?
• Train models: mini-batch to batch
• Serve models ("score" events): any latency



Records/events are ingested into SparkML (or another ML service) for incremental model training using “mini-batch” training. The Spark Streaming job could also call a 3rd-party ML service for training. Updates to model parameters are written to one of several places. Shown here is writing to a special Kafka topic or another persistence store. One of several processes supported by Flink or Akka Streams is used to periodically ingest model parameter updates. OR, Flink and Akka will call to the external ML service to do scoring. Either way, while the data is used to train the “next” model, the current model is used to score the records with low latency. Hence, the models are always slightly “obsolete”, but this is rarely an issue.

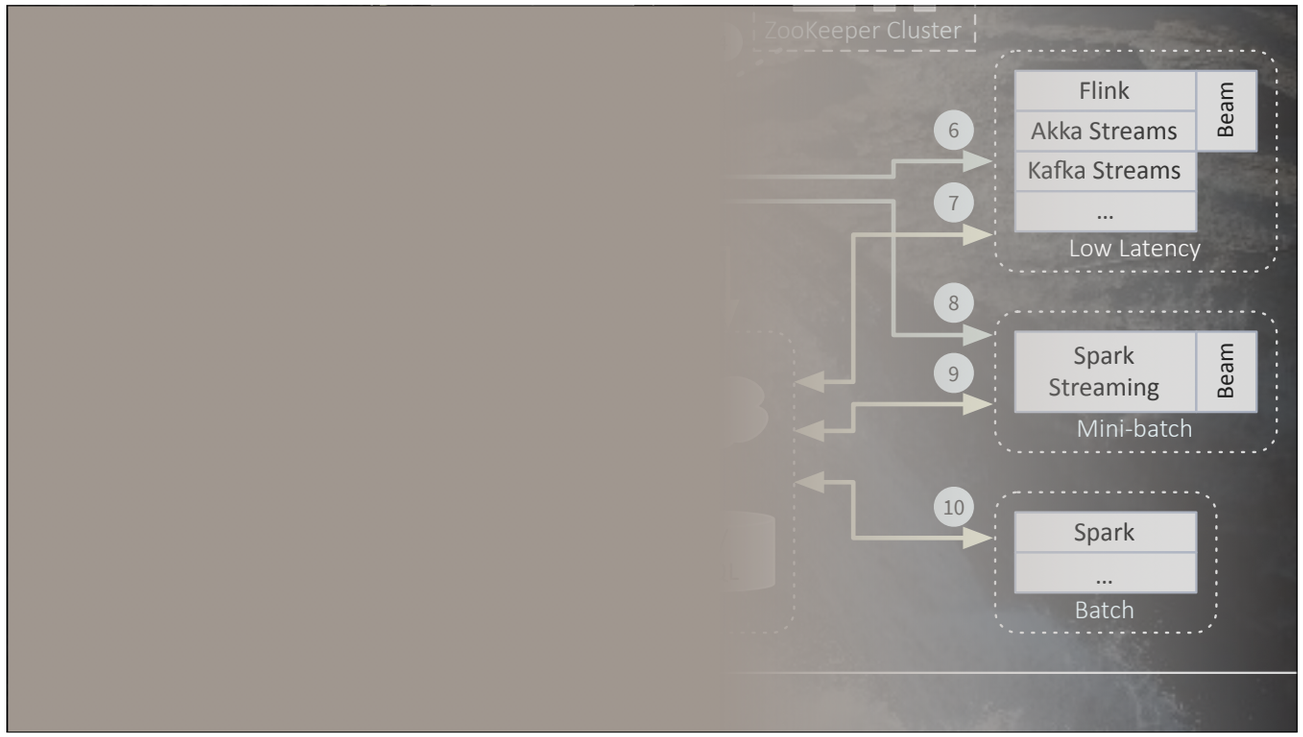


Per event is best for complex event processing, where each event needs to be analyzed, routed, etc. separately. Best when volumes are relatively low and the overhead per event is also low.

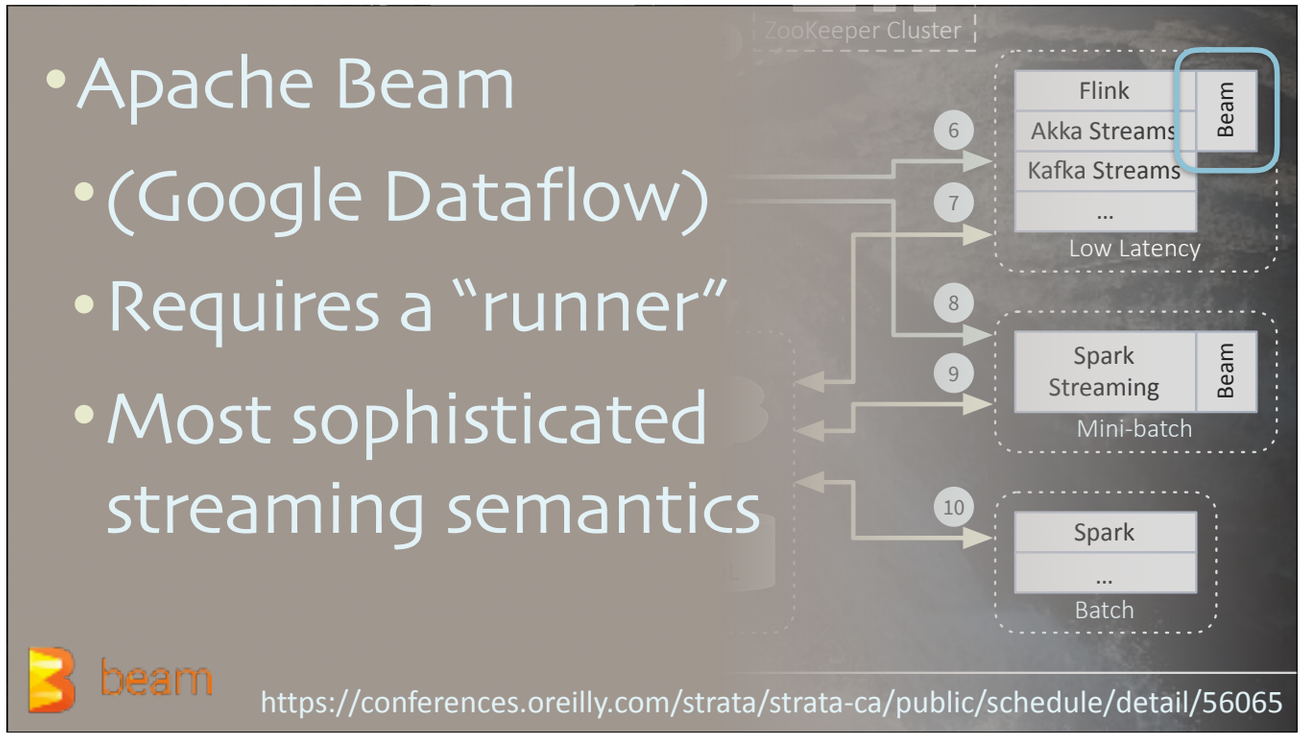


Processing in bulk, like SQL queries, training ML models, etc. best for large volumes where the system is efficient at scale (i.e., the overhead amortized over the bulk events is good), but low overhead per event at low volumes isn't important.

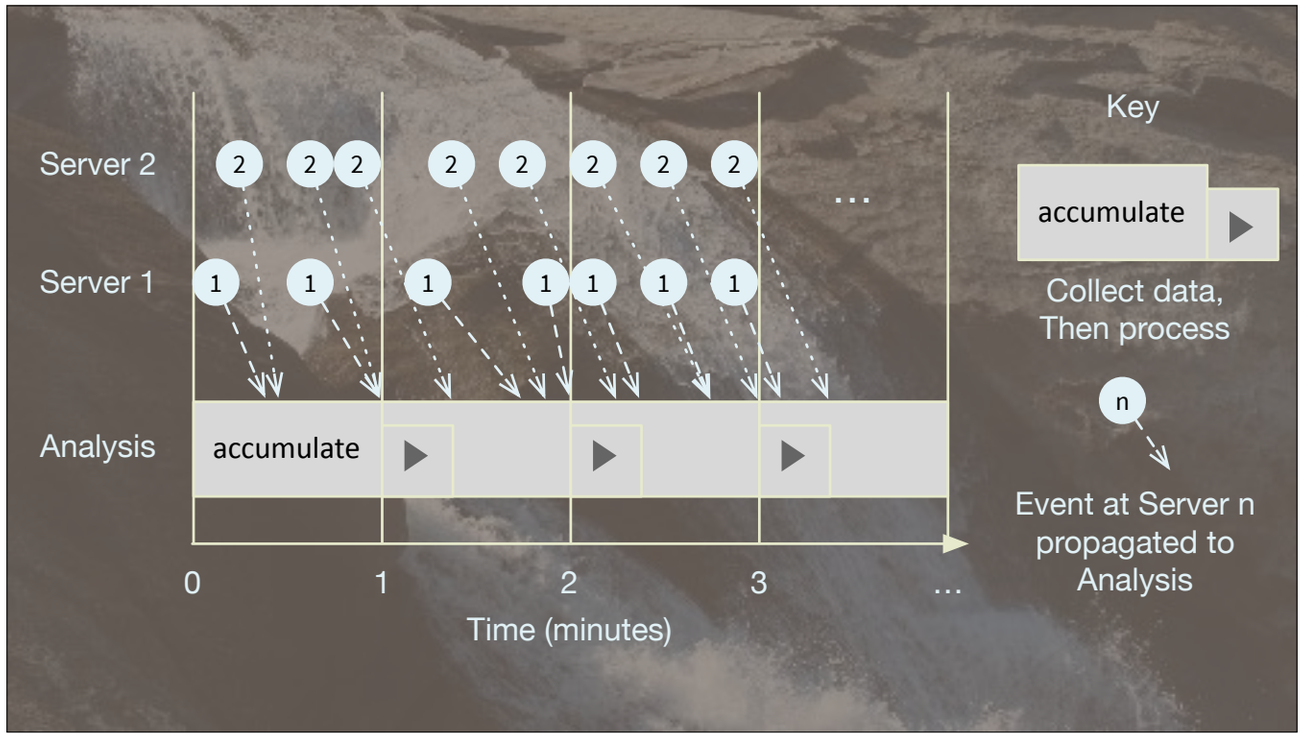
Photo: Top of Nevada Falls, Yosemite National Park, California.



Focusing in on the pure (low-latency) streaming and mini batch engines on the right…



Google has spent years thinking about all the things a streaming engine has to handle if you want to do accurate calculations on streams, accounting for all the contingencies that can happen. The latest incarnation is Google Dataflow, available as a service in Google Cloud Platform. Google recently open-sourced the part of Dataflow for defining "data flows" with these sophisticated semantics, called Apache Beam. Beam doesn't provide a runner, so third-party tools provide this capability.
I recommend the talk by Tyler Akidau from Google at the recent Strata conference, where he explains the history leading up to Beam.

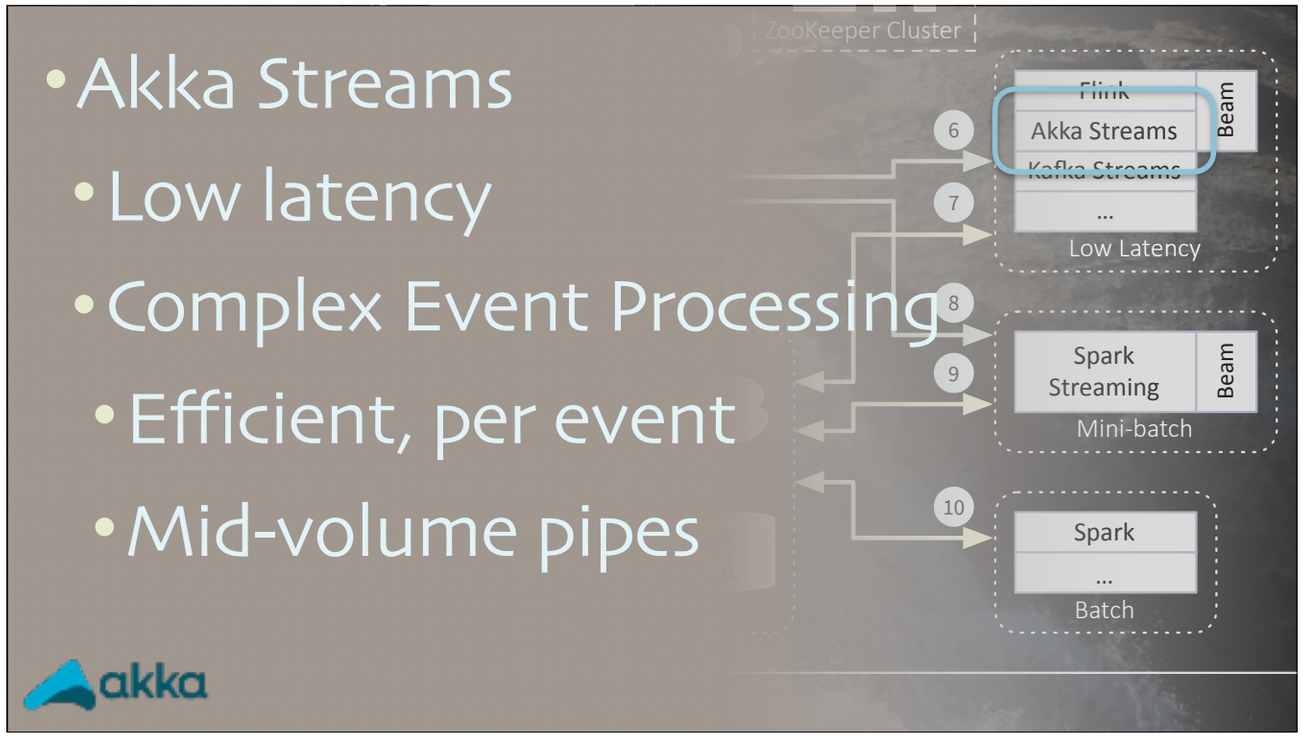


Here is an example of the scenarios you have to handle. Suppose you are computing per-minute aggregations (like sales per minute). The analysis machine has one clock that is not necessarily in sync with the other servers processing sales. Worse, there is an unavoidable time delay for events to arrive to the analysis server. Hence you need to process event time and you need to account for late arriving data, not only small delays where some events/minute will arrive within the following minute, but even large delays due to network partitions, servers busy, etc., etc. This is one example of the challenges posed by streaming when you want accurate vs. approximate results.

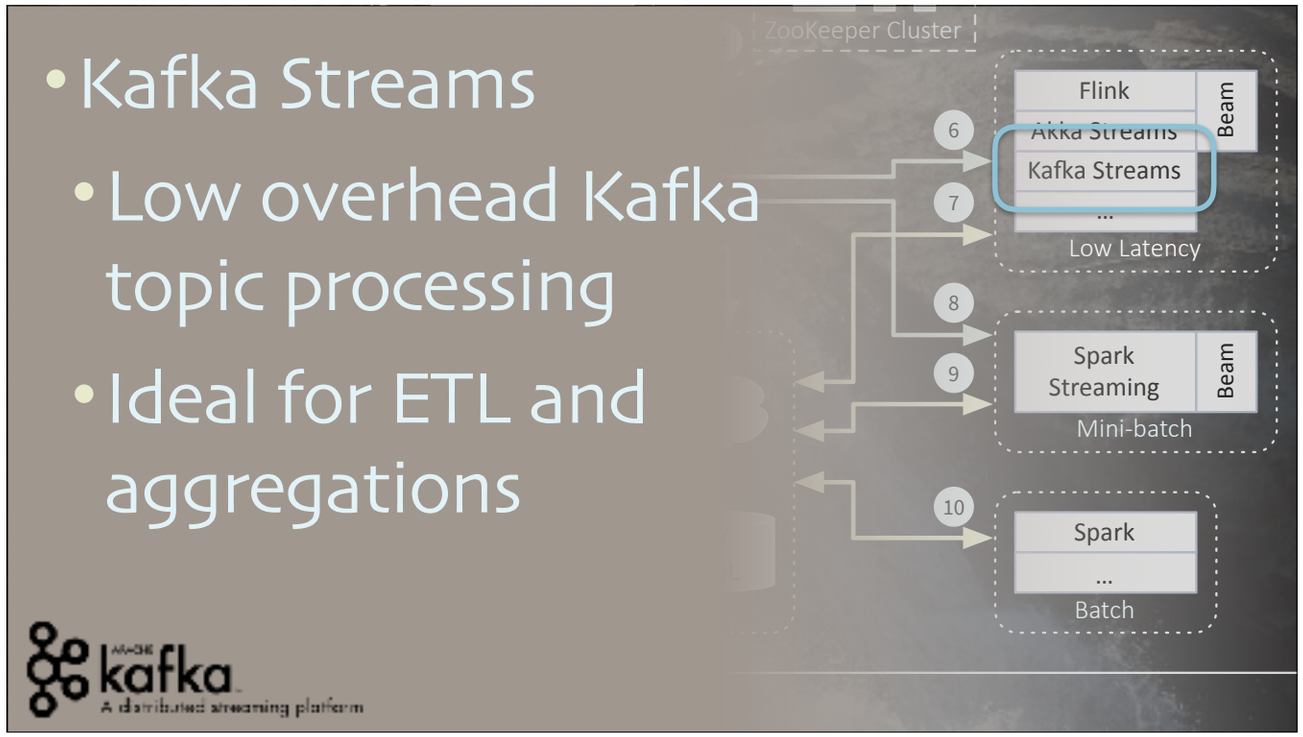
- Apache Flink
  - High volume
  - Low latency
  - Beam Runner
  - Evolving SQL, ML

Flink

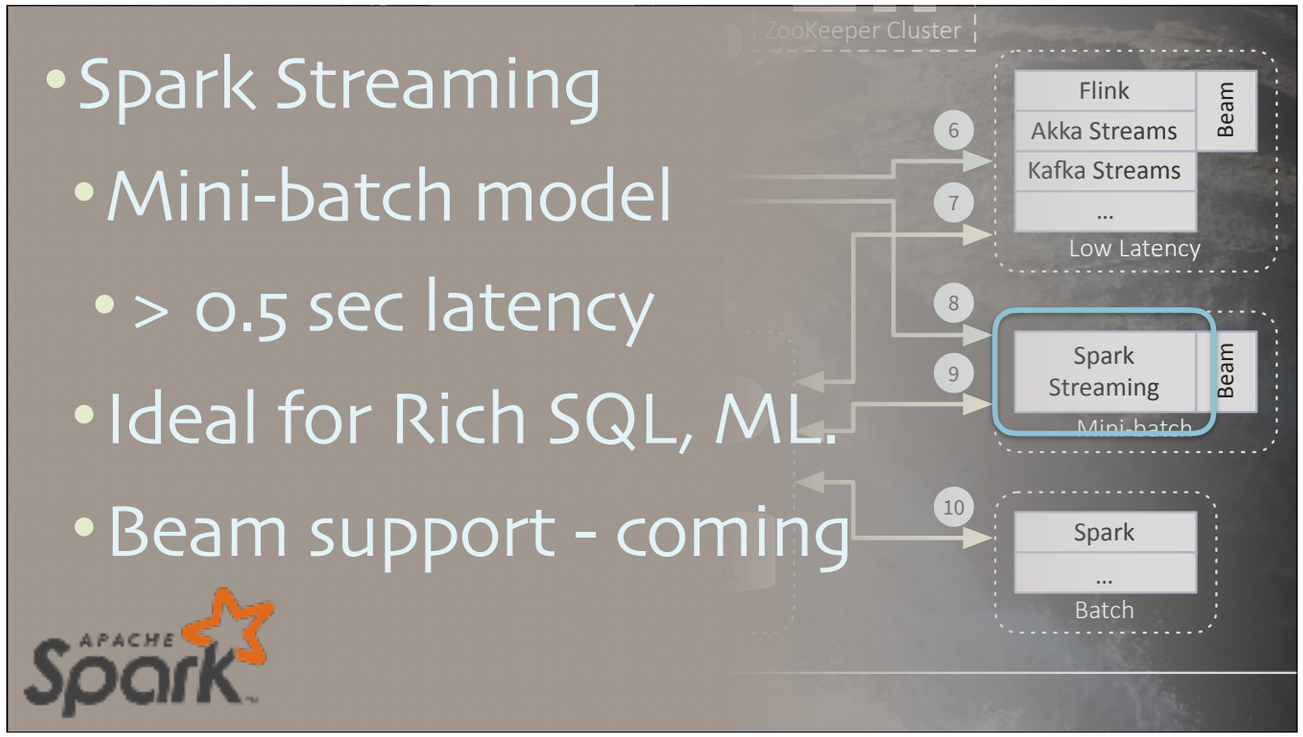
Flink provides two unique capabilities among these choices, 1) low-latency processing at scale and 2) it is the most mature runner for Apache Beam data flows (at least that I know of) outside Google's own Dataflow engine.



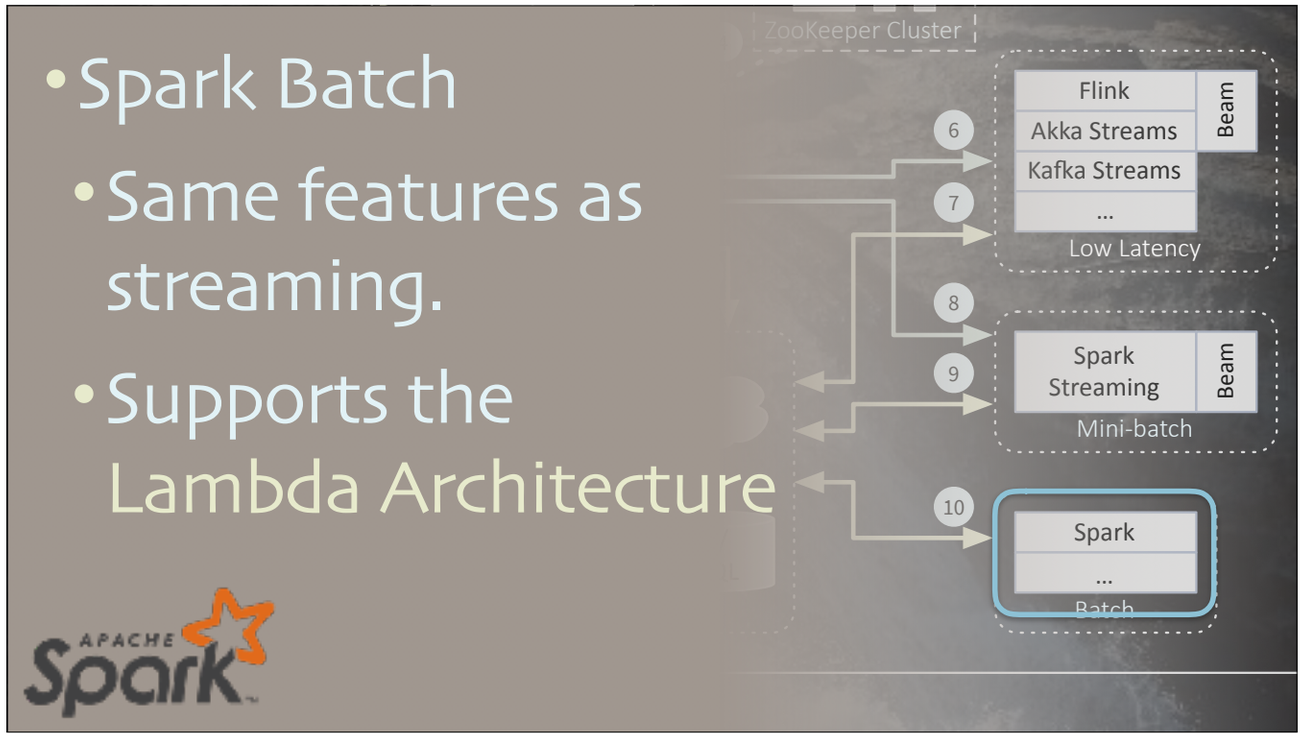
Akka Streams adds a streaming abstraction on top of Akka actors. It's ideal when complex event processing (CEP) is the preferred model, as opposed to in bulk processing of data. Akka's powerful Actor model abstracts over the details of thread programming for highly concurrent apps, with libraries for clustering, persisting state, and data interchange with many sources and sinks (the "Alpakka" project).



Kafka Streams is a great 80% solution that works close with Kafka (in terms of production deployment and overhead). It is designed to read Kafka topics, perform processing like transformations, filtering, aggregations, “last-seen” value for keys, etc. then write results back to Kafka. It nicely addresses many common design challenges, but isn’t designed to be a complete solution for stream processing, e.g., running SQL queries and training ML models.

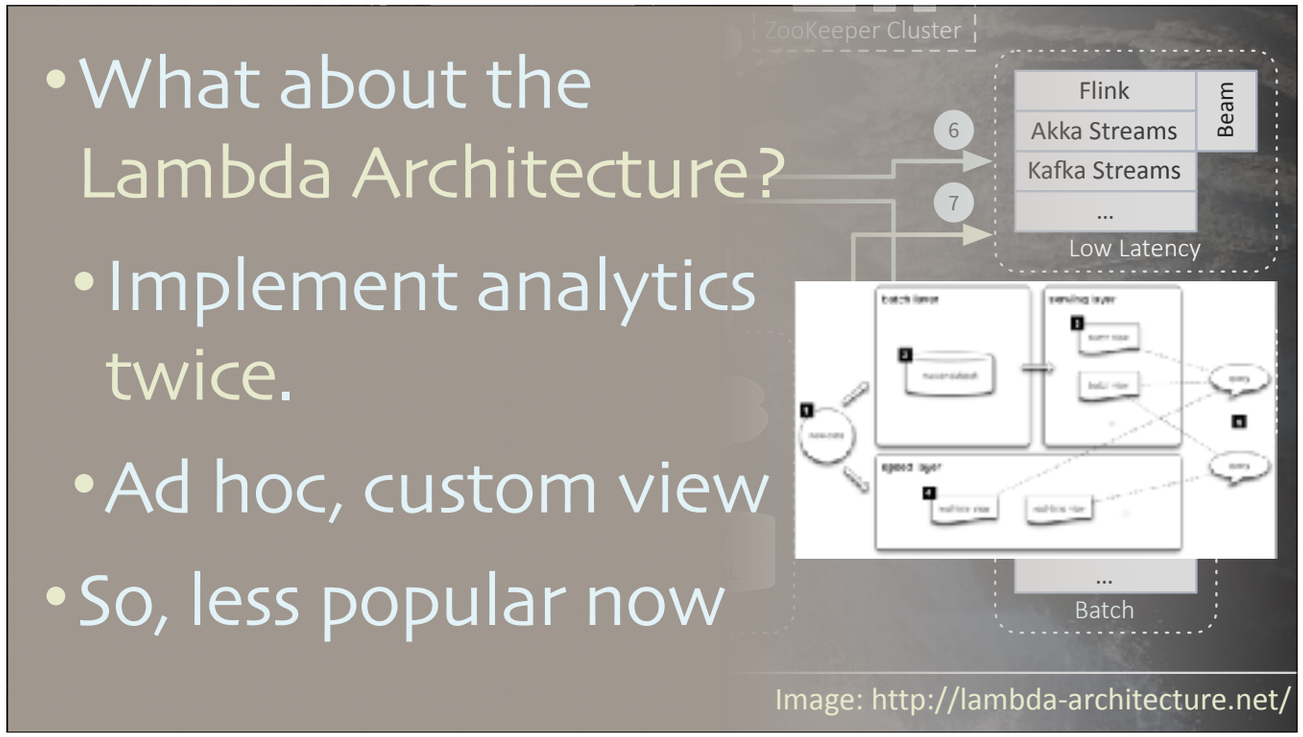


Spark Streaming is a mini-batch model, although this is slowly being replaced with a true, low-latency streaming model. So, today, use it for more expensive calculations, like training ML models, but don't use it when the lowest-latency processing is required. Lightbend's Fast Data Platform is working on tools to make it easy to train ML models in Spark and serve them with the other tools. Another advantage of Spark is the rich ecosystem of tools for a wide variety of batch and streaming scenarios. You can maybe push the latency down to 200 or even 100 msecs, but that's difficult.



If your latency requirements are minutes, consider Spark batch jobs. Restarting every few minutes is generally easier than keeping a streaming process running for months! Spark's streaming and batch support is helpful for the Lambda Architecture (http://lambda-architecture.net/), which mixes streaming and batch data analytics into a unified view (although Lambda is falling out of favor).

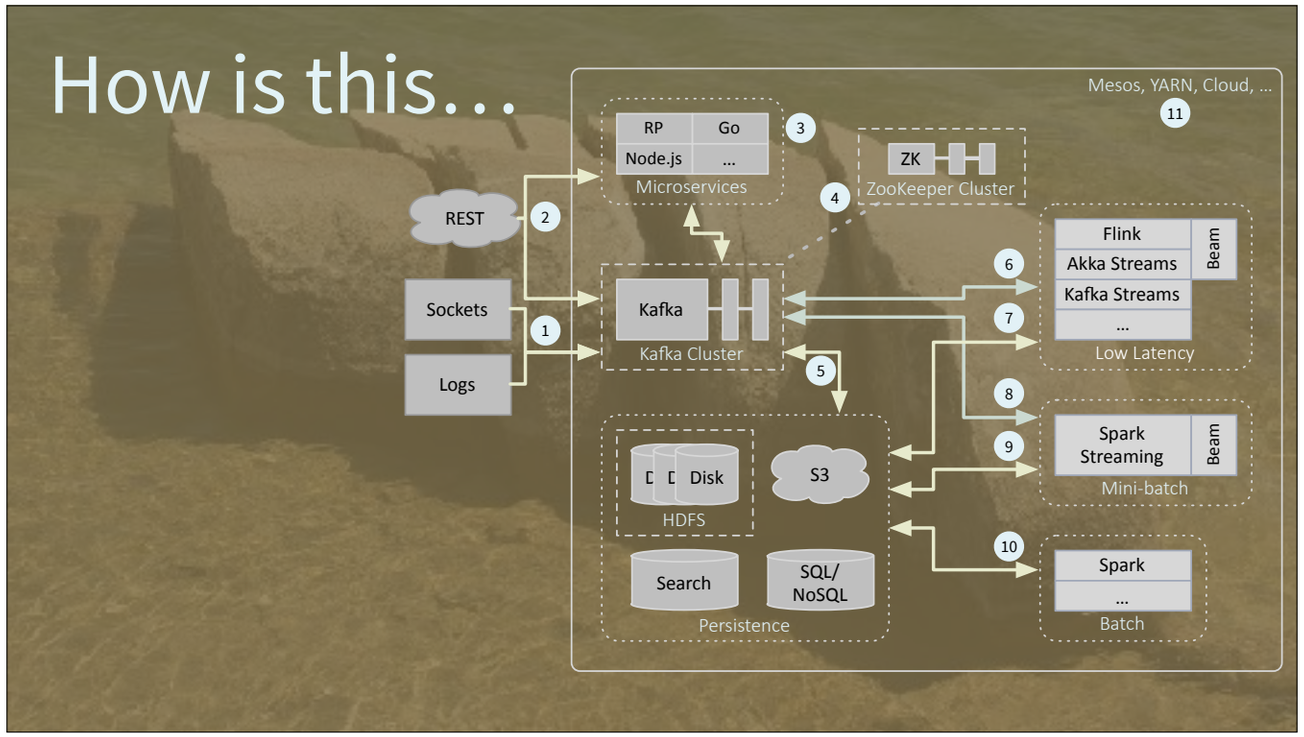
- What about the Lambda Architecture?
  - Implement analytics twice.
  - Ad hoc, custom view
- So, less popular now



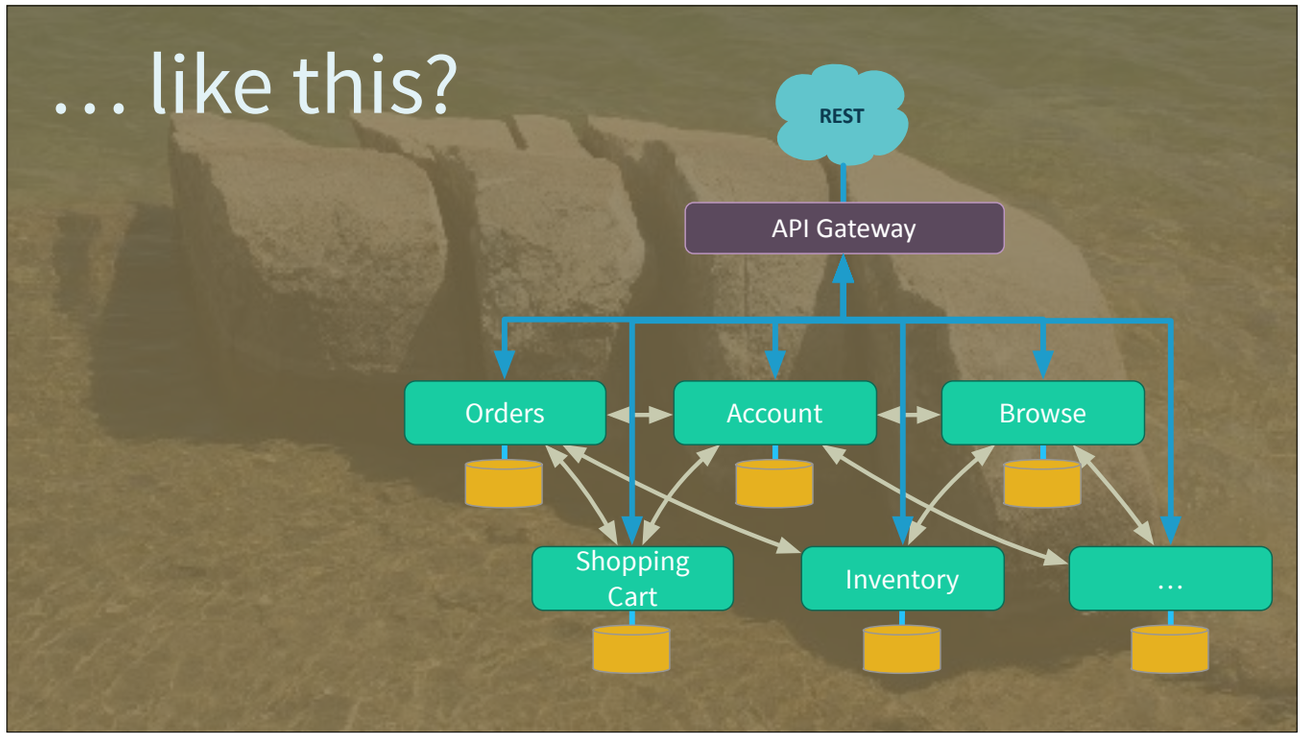
Lambda was popular for a while, but it's now seen as a transition architecture to the architecture I'm describing here, which is primarily stream based. Lambda was great when we had inefficient MapReduce jobs that we could only run periodically, but we wanted to account for up to date - if approximate - results. However, it meant implementing logic twice, once in the batch layer and once in the speed layer, then a custom view that joined results had to be implemented. Today, with much more efficient batch tools like Spark and richer streaming tools, like Beam, we can do far more with streaming without resorting to Lambda. Jay Kreps, the creating of Kafka and one of the first proponents of stream-based architectures, joking called his alternative the "Kappa Architecture".

I'm going to argue that service architectures (classically three tier, but evolving…) and data architectures (classically Big Data like Hadoop) are converging.

Photo: "Sliced" rocks, 1000 Island Lake, Ansel Adams Wilderness, California.

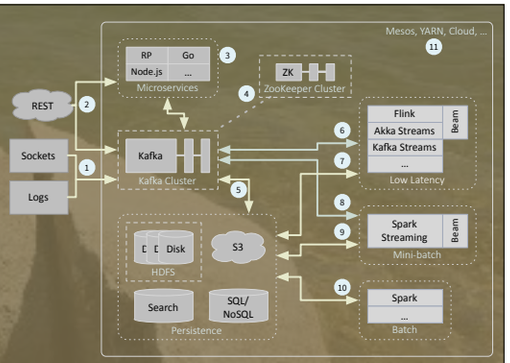


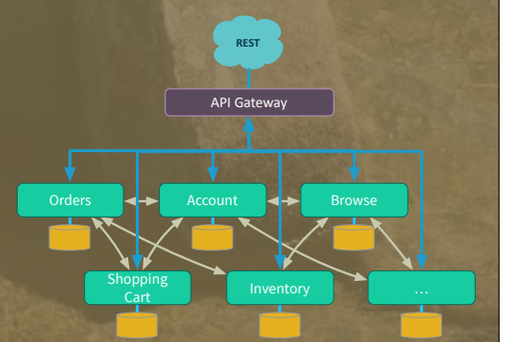
How are fast data architectures like …



… microservice architectures??

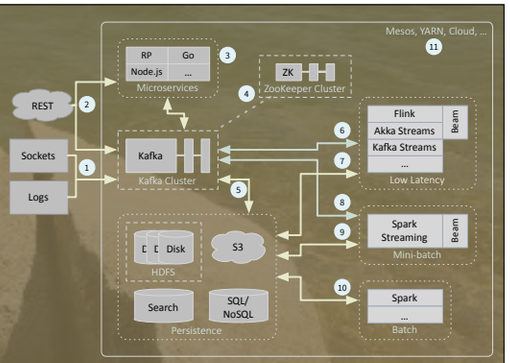
- A data app / microservice:
  - A single responsibility.
  - …

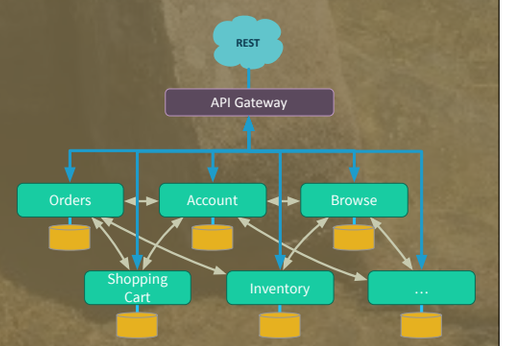




Each data app, streaming or batch, is typically doing one thing, like ETL this Kafka topic to Cassandra, or compute aggregates for a dashboard, etc. Similarly, services evolving now into microservices are also supposed to do one thing and communicate with other services for the “help” they need.

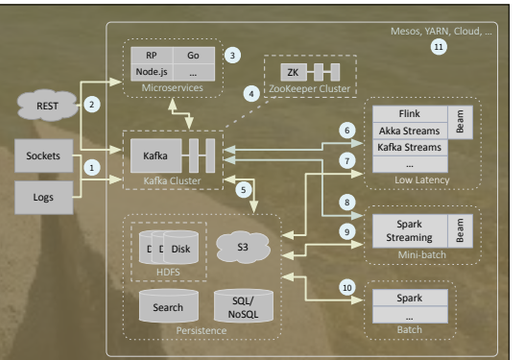
- A data app |microservice:
  - A single responsibility.
  - The input never ends.

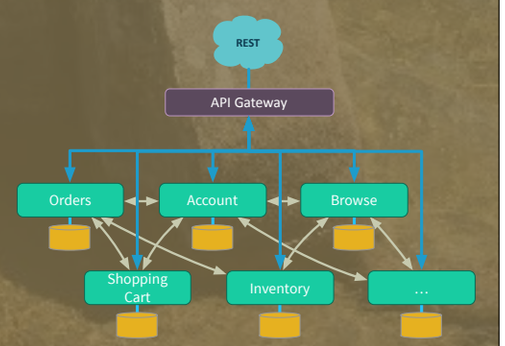




A streaming data app will have a never-ending stream of data to process. Similarly, requests for service will never stop coming to microservices.

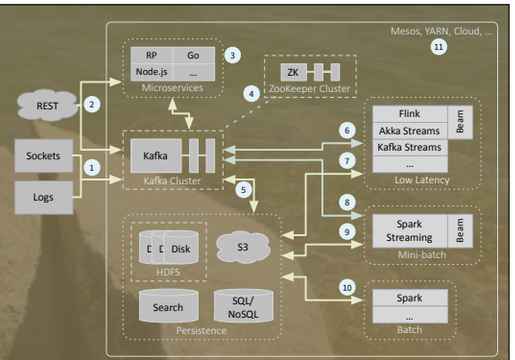
- A data app/microservice:
  - A single responsibility.
  - The input never ends.
  - So, both must be available, responsive, resilient, & scalable. I.e., reactive

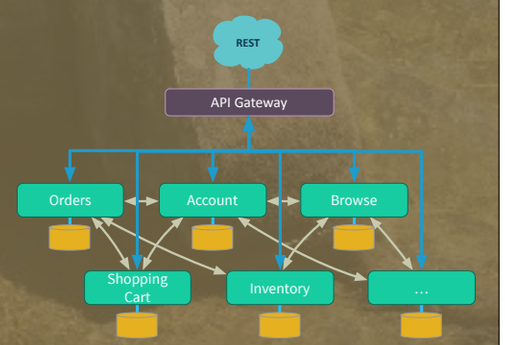




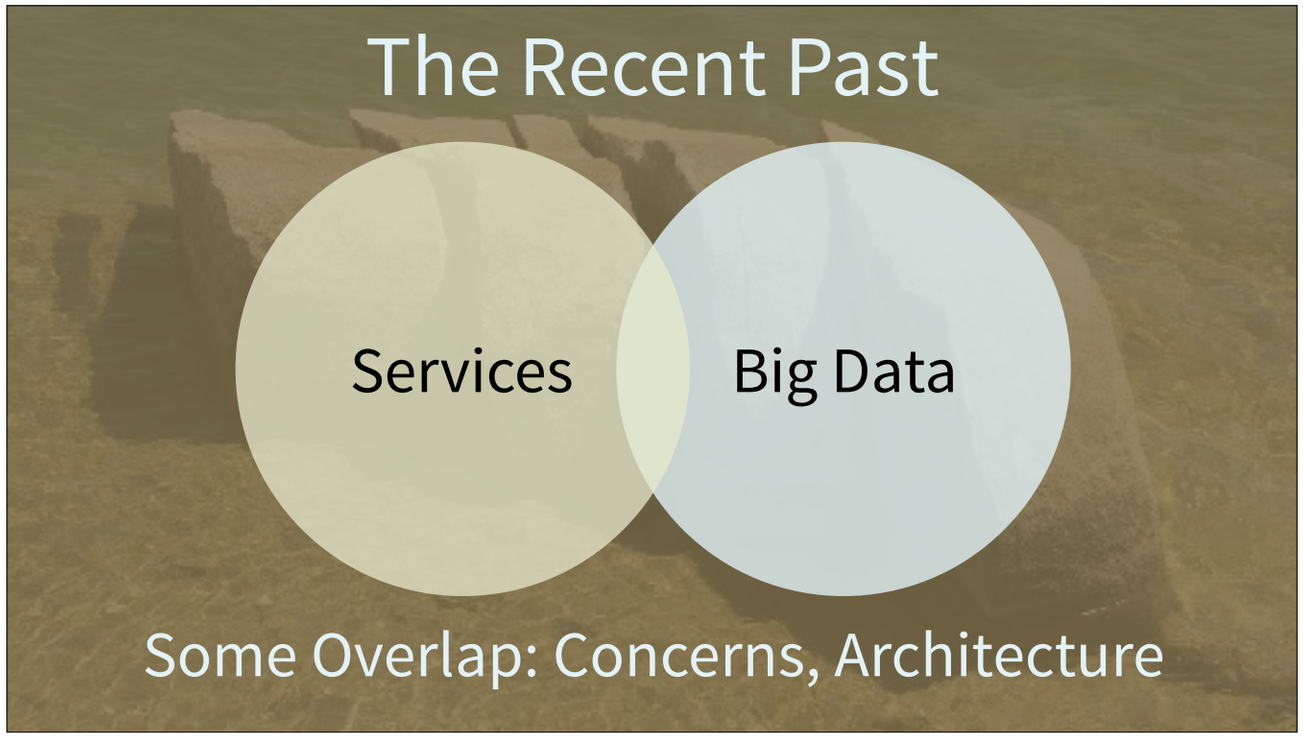
So, both kinds of systems have similar design problems, hence both should be implemented in similar ways.

- Also, even "non-data" microservice environments have ever growing data.
- So data becomes a dominant concern.

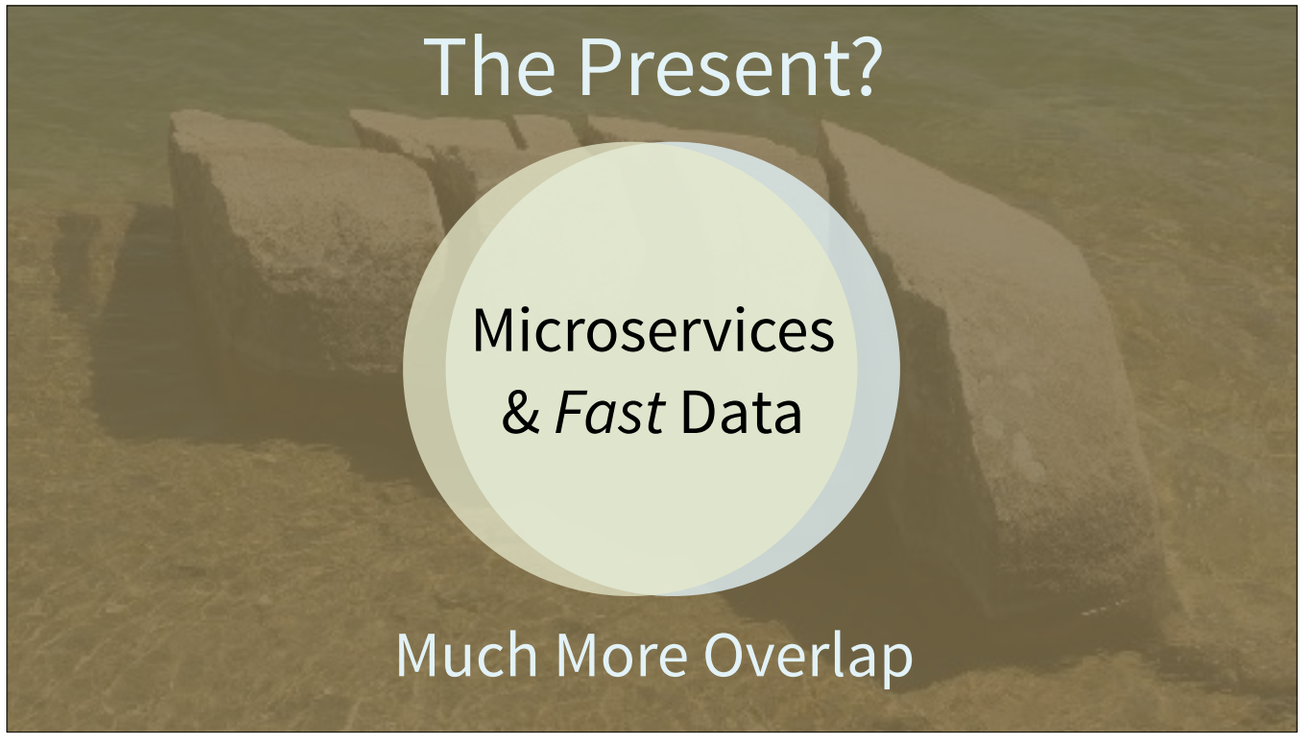




Also, organizations that aren't particularly data focused, especially when they are new endeavors, often find that data grows to be a dominant concern, a sign of success! For example, Twitter started as a classic three-tier application, but now most of their infrastructure looks like a petroleum refinery.



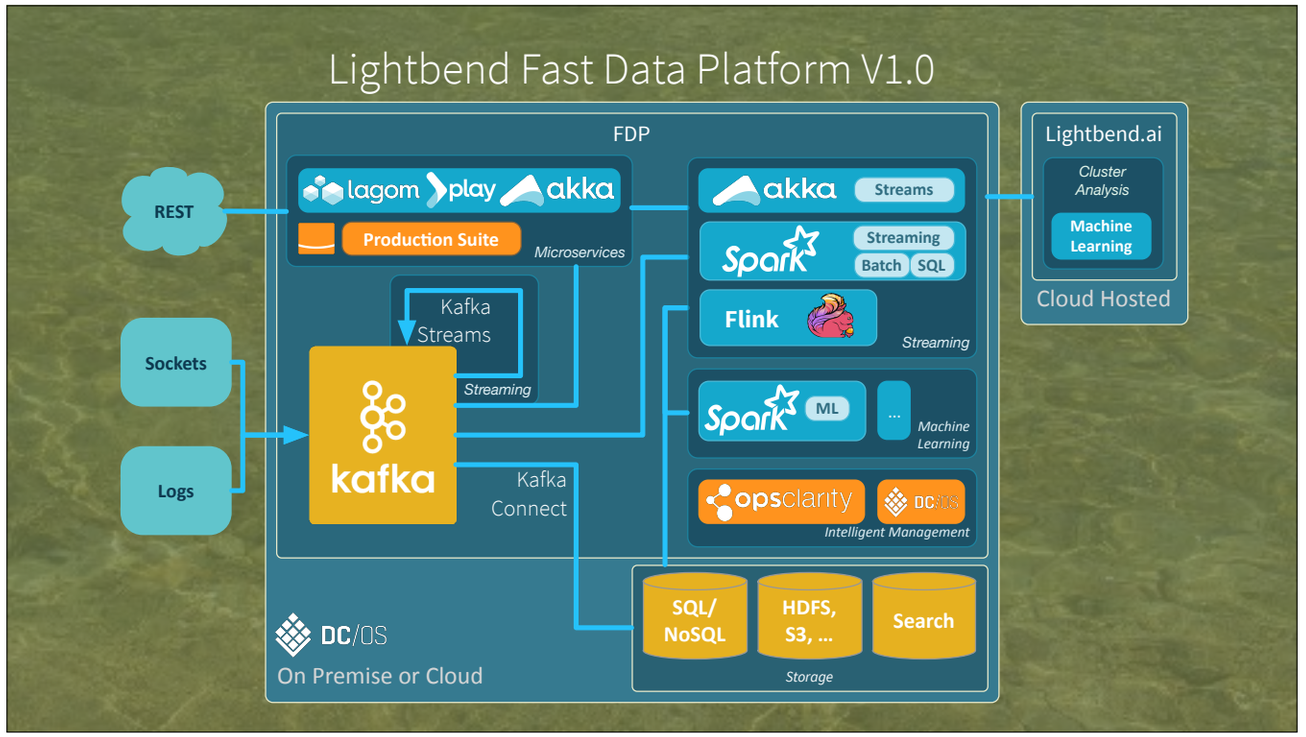
Until recently, people building “canonical” services for general-purpose IT apps have focused on high availability, scalability, resilience, etc. (i.e., the Reactive principles), recently moving to microservices to do this better.
The Big Data world has focused on data storage and cluster scalability, with less need to worry about the Reactive principles.
Of course these was some overlap, but they were different spheres…



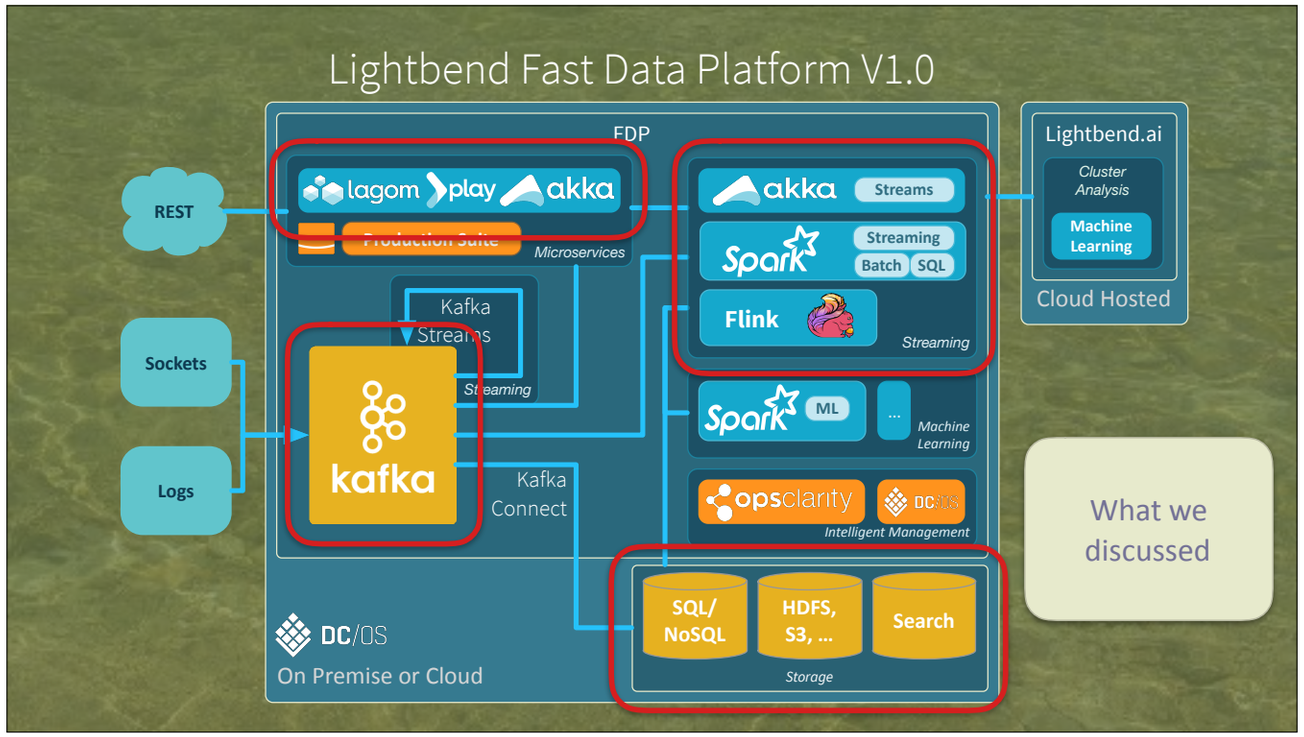
IT apps have been evolving towards finer-grained microservices and offline Big Data environments have been evolving towards online, stream processing or “Fast Data” environments. I’m going to argue that the architectures for these two spheres are converging towards the middle.

Lightbend is taking what we've learned about streaming and providing an integrated suite of tools for our customers, called Lightbend Fast Data Platform.

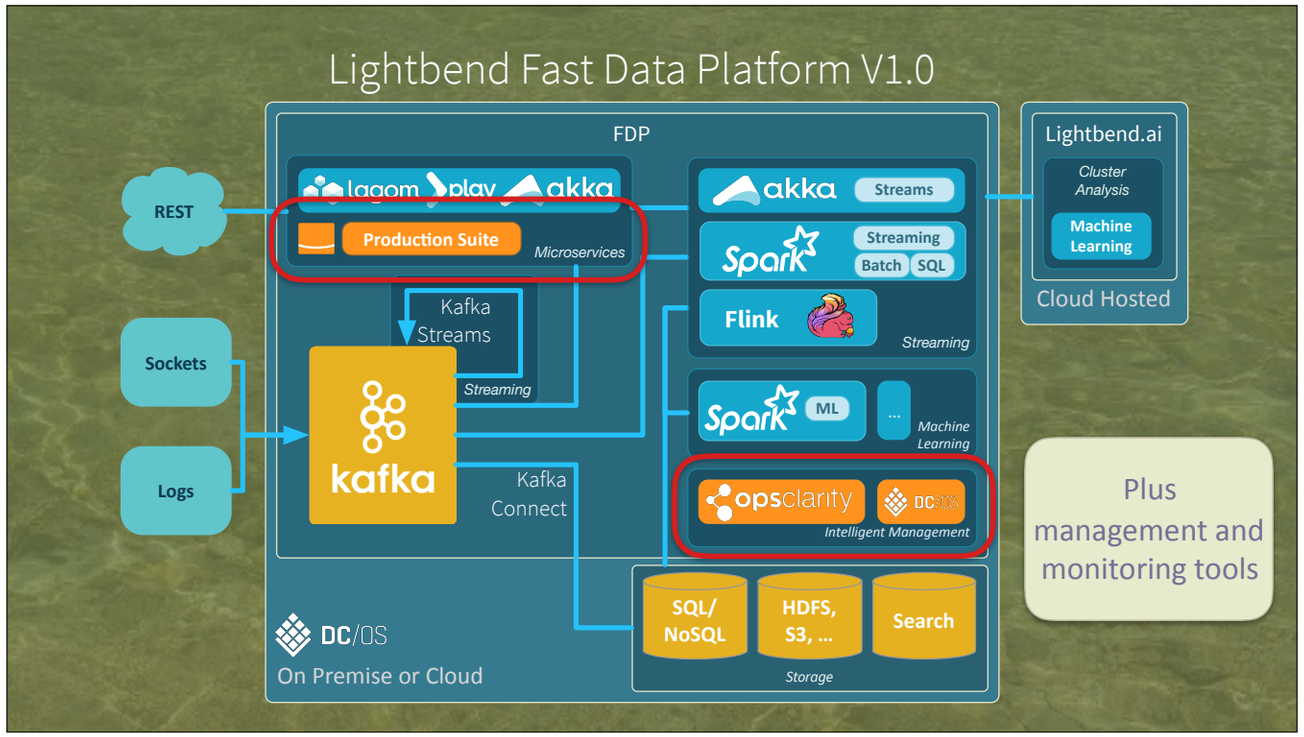
Photo: Surface of a mountain lake, Olympic National Park, Washington State.



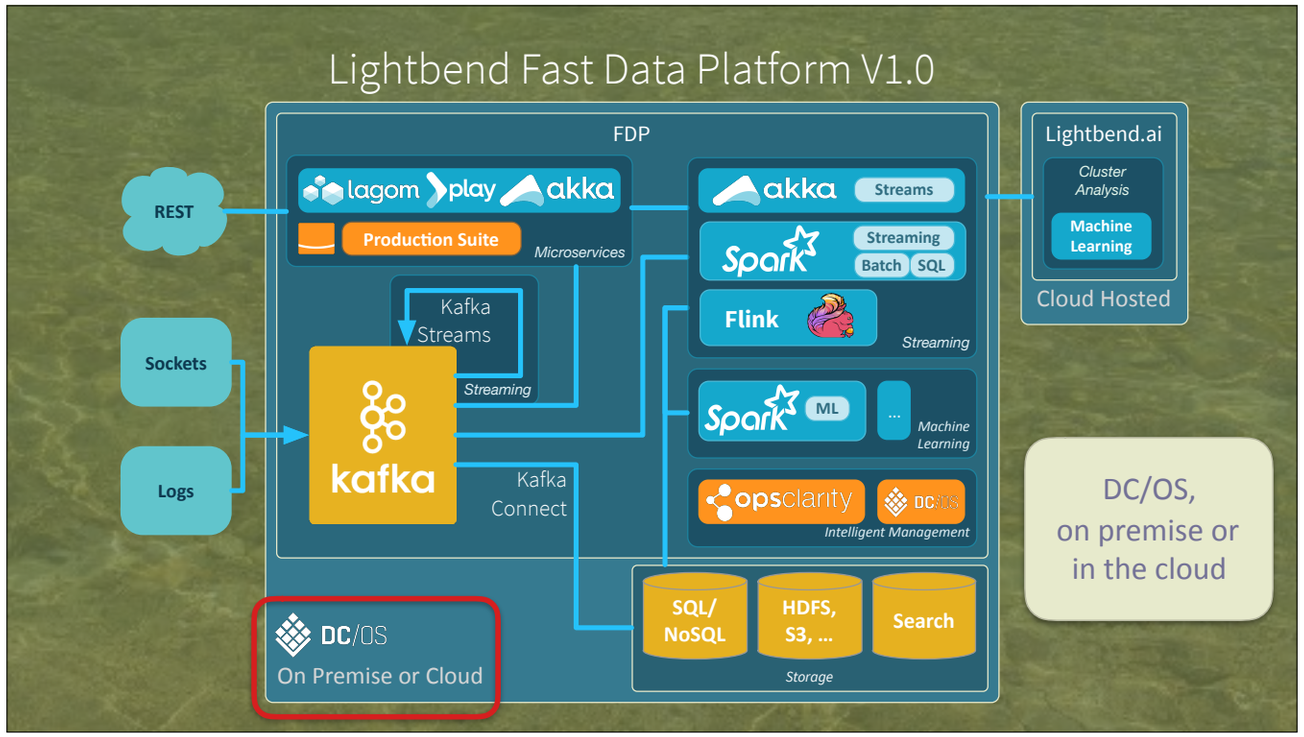
Lightbend is taking what we've learned about streaming and providing an integrated suite of tools for our customers, called Lightbend Fast Data Platform.



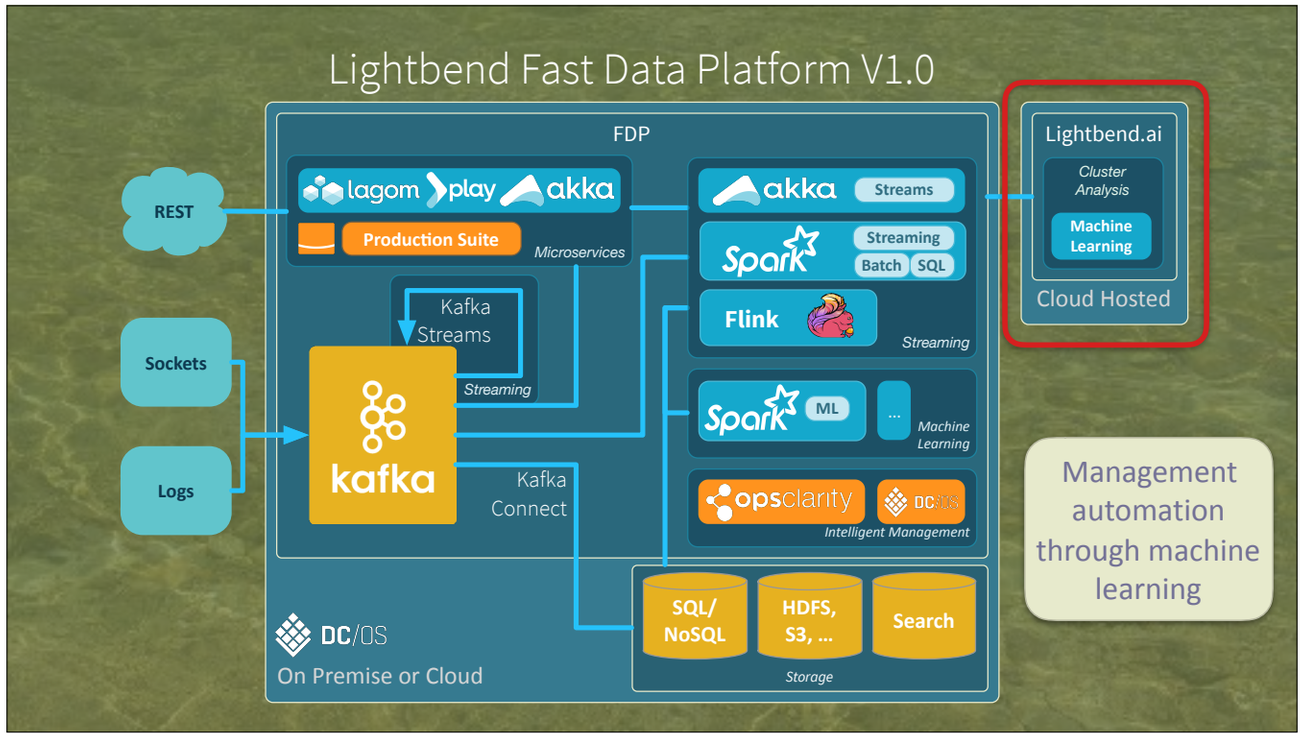
Here's what we discussed so far.



Lightbend is taking what we've learned about streaming and providing an integrated suite of tools for our customers, called Lightbend Fast Data Platform.

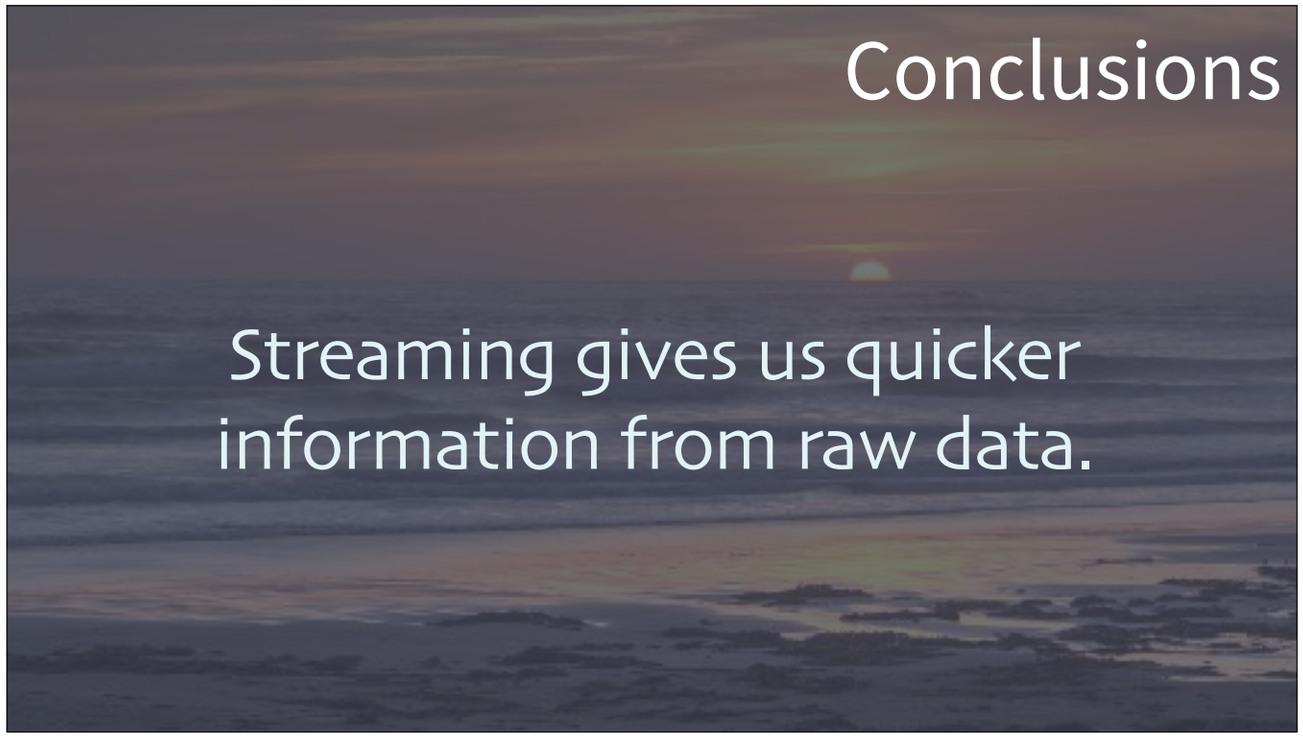


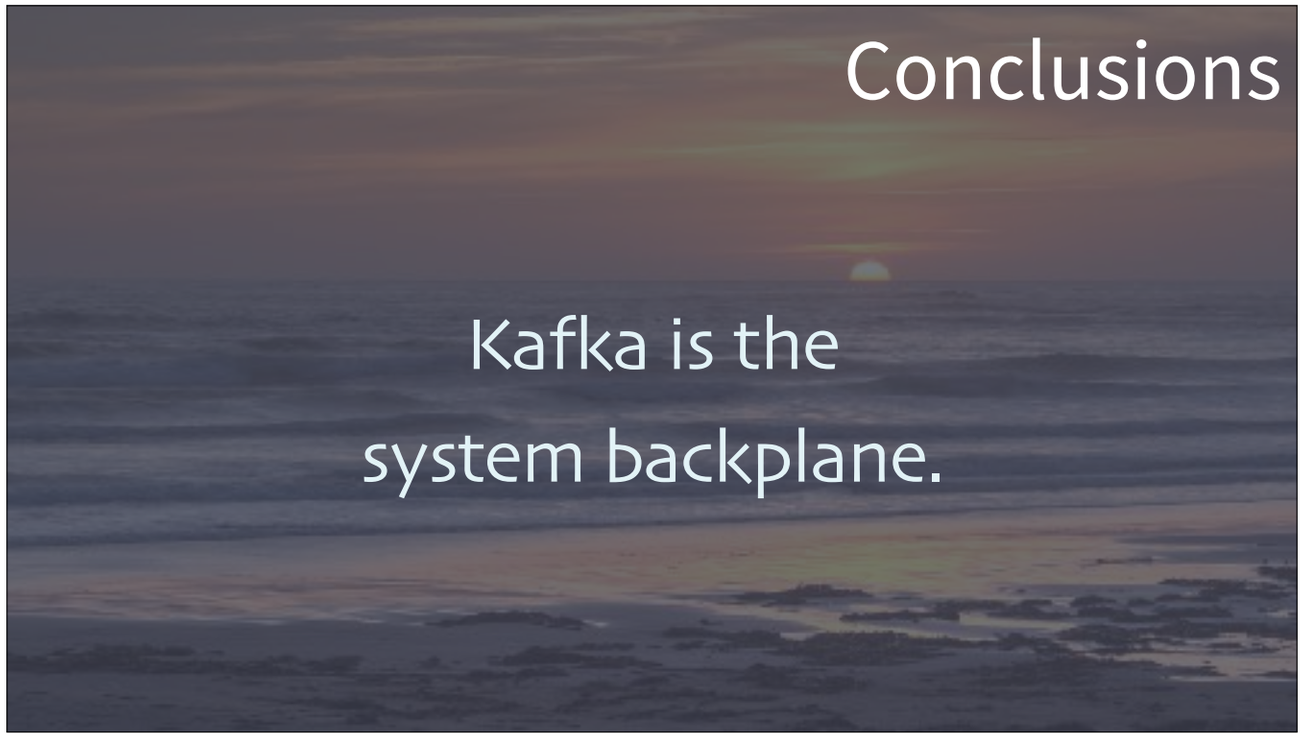
Lightbend is taking what we've learned about streaming and providing an integrated suite of tools for our customers, called Lightbend Fast Data Platform.



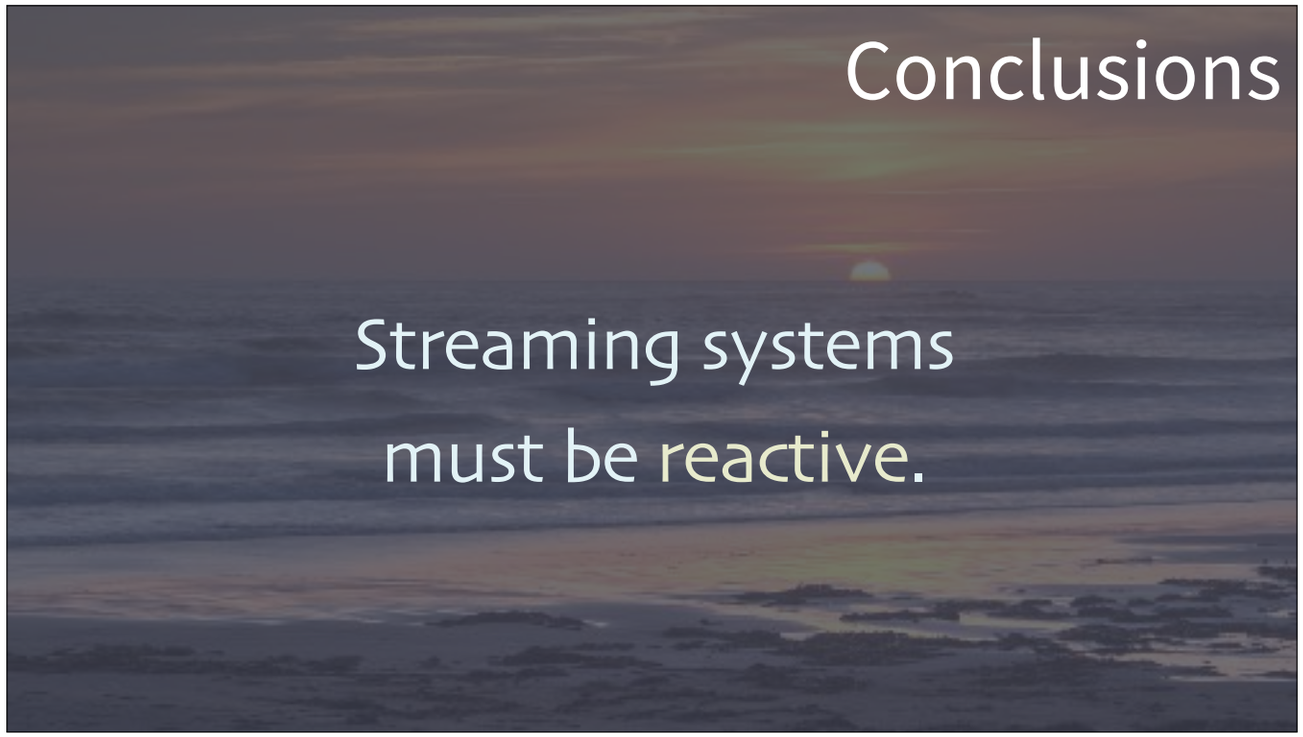
Lightbend is taking what we've learned about streaming and providing an integrated suite of tools for our customers, called Lightbend Fast Data Platform.

Photo: Sunset over the Pacific Ocean, Olympic National Park, Washington State.


Conclusions
Streaming gives us quicker
information from raw data.


Conclusions
Kafka is the
system backplane.


Conclusions
Several streaming engines cover
the spectrum of analysis needs.

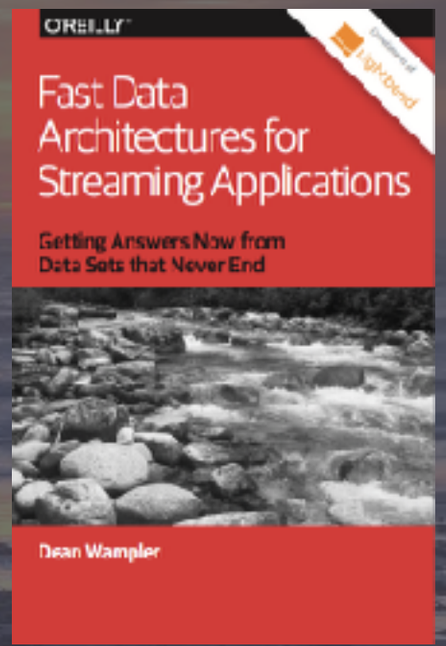

Conclusions
Streaming systems
must be reactive.


Conclusions
Microsystems grow
to be data-centric.


Conclusions
Therefore,
streaming and microservice
architectures are merging.

Free as in 🍺

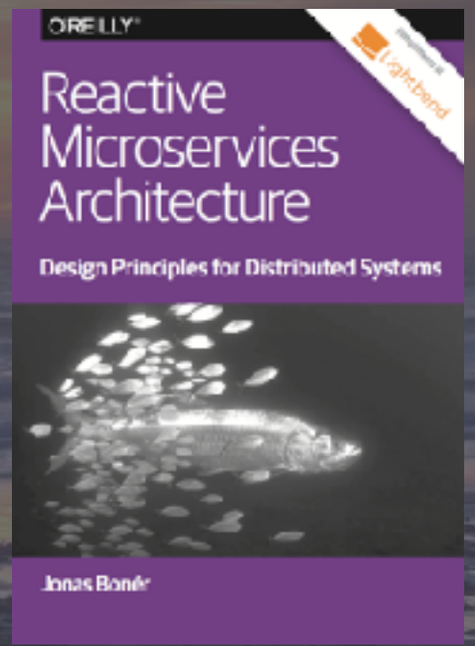
bit.ly/fastdata-ORbook



My book, published last fall that describes the points in this talk in greater depth. I've refined the talk a bit since this was published.

Free as in 🍺

lightbend.com/
reactive-microservices-architecture



For more on microservices,

For more on microservices,