Big Data Architectures and Introduction to Hadoop

Dr. Sonali Agarwal
Associate Professor, Department of IT
IIIT Allahabad

Common Features of a Big Data Systems with-in Data Science Domain

- Most of the Big Data Systems work in distributed environment.
- **HOT-SPaCe:** Characteristics of Distributed Systems
  - Heterogeneity
  - Openness
  - Transparency
  - Scalability
  - Programing Ease
  - Concurrency
  - Error (Failure) Handling

Heterogeneity

H – Heterogeneity

- Distributed systems run across different hardware, operating systems, networks, and programming languages.
- Without abstraction, integration is painful. Middleware solves this.
- **Example:**
  - A banking system may have Linux servers, Windows desktops, Android/iOS mobile apps.
  - **Java Virtual Machine (JVM)** hides OS differences, letting the same Java program run anywhere.

Heterogeneity

- Variety and differences in
  - Networks
  - Computer hardware
  - Operating systems
  - Programming languages
  - Implementations by different developers
- *Middleware* as software layers to provide a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, OS, and programming languages.
- *Mobile Code* to refer to code that can be sent from one computer to another and run at the destination (e.g., Java applets and Java *virtual machine*).

Openness

- **O– Openness**
- Openness is concerned with extensions and improvements of distributed systems.
- Openness = ability to extend, integrate, and interoperate using published, standard interfaces.
- New components can be added without breaking the system.
- Detailed interfaces of components need to be published.
- Differences in data representation of interface types on different processors (of different vendors) have to be resolved.

Openness

Openness in Distributed Systems

- **Openness** = Ability to extend, integrate, and interoperate using published, standard interfaces
- POSIX standard: Apps run on Linux/Mac/ UNIX without changes
- **Hadoop ecosystem**: Plug-and-play with Hive, Spark, Flume, Sqoop
- Healthcare (FHIR standard): Hospitals share patient data across different systems

Analogy:
USB Port – *Any device (mouse, keyboard, pen drive) works as long as it follows the standard*

Openness = Standards make integration seamless

Transparency

T – Transparency

- The system should look like one unified whole, not scattered components.
- **Types:** Access, Location, Replication, Concurrency, Mobility, Performance, Scaling.
- **Example:**
  - **Network File System (NFS):** Opening a remote file feels the same as opening a local file.
  - **ATM network:** Any ATM shows the same account balance, no matter where you are.
- Transparency has different aspects. These represent various properties that distributed systems should have.

Access Transparency

- Enables local and remote information objects to be accessed using identical operations.
- Example:
- **Network File System (NFS):**
  - Opening a remote file looks identical to opening a local file (open("file.txt")).
- **SQL Queries:**
  - Same query works whether the database is local or distributed.

Location Transparency

- Enables information objects to be accessed without knowledge of their location.

Example

- Web browsing: You just type a URL; you don’t care which physical server delivers the page.
- Cloud storage: Retrieving a file from Google Drive doesn’t require knowing which datacenter it’s stored in.

Concurrency Transparency

- Enables several processes to operate concurrently using shared information objects without interference between them.
- ATM network: Many users can withdraw from different ATMs simultaneously, and balances remain consistent.
- Database transactions: Two students registering for the last seat in a course → only one succeeds (handled by concurrency control).

Replication Transparency

- Enables multiple instances of information objects to be used to increase reliability and performance without knowledge of the replicas by users or application programs
- Example: Distributed DBMS
- Example: Mirroring Web Pages.
- Watching a Netflix movie → content may come from the nearest replica, invisible to the user.

Failure Transparency

- Enables the concealment of faults
- Allows users and applications to complete their tasks despite the failure of other components.
- Example: Database Management System
  - **Google File System (GFS):** If a server holding data fails, another replica serves the data without the user noticing.
  - **Amazon AWS:** If one server fails, load balancers redirect traffic to healthy servers.

Mobility Transparency

- Allows the movement of information objects within a system without affecting the operations of users or application programs
- Mobile IP: User can move from Wi-Fi to mobile data, but their session (e.g., a Zoom call) continues.

Performance Transparency

- Allows the system to be reconfigured to improve performance as loads vary.
- Example: **Distributed Make:**
- Tasks automatically shift to faster/less loaded nodes to speed up compilation.

Scaling Transparency

- Allows the system and applications to expand in scale without change to the system structure or the application algorithms.
- World Wide Web: Adding millions of websites/users didn’t require changing the browser.
- Distributed Databases: Adding new nodes to Cassandra increases capacity, but applications still query it the same way.

Scalability

One of the important characteristics to be considered in the distributed case:

Scalability

Performance

Latency

Availability

Fault tolerance

is the ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

Scalability

- Adaptation of distributed systems to
  - accommodate more users
  - respond faster (this is the hard one)
- Usually done by adding more and/or faster processors.
- Components should not need to be changed when scale of a system increases.
- Design components to be scalable!

Scalability

- **System Size**: Higher performance when adding more machines
- **Software**: Can framework and middleware work with larger systems?
- **Technology:** Impact of scaling on time, space and diversity
- **Application**: As problem size grows (compute, data), can the system keep up?
- **Vertical vs Horizontal:** ?
- …

Scalability

- Strong vs. Weak Scaling
- **Strong Scaling**: How the performance varies with the # of processors for a *fixed total problem size*
- **Weak Scaling**: How the performance varies with the # of processors for a *fixed problem size per processor*
  - MapReduce is intended for "Weak Scaling"

Scalability

Strong Scaling

- Performance improvement when the **problem size is fixed** but we **increase the number of processors**.
- *“If I keep the workload the same, and add more processors, do I finish faster?”*
  - A 100 GB dataset is processed.
  - On 1 processor → 10 hours.
  - On 10 processors → ideally 1 hour (but usually a bit more due to overhead).
- HPC (High Performance Computing), simulations, weather modeling.

Scalability

Weak Scaling

- Performance change when the problem size per processor is fixed but the number of processors increases.
- If each processor keeps the same workload, and I add more processors (thus a bigger total workload), can the system handle it in the same time?”
- Example: Each processor handles 10 GB.
- 1 processor → 10 GB → 1 hour.
- 10 processors → 100 GB total (10 GB each) → ideally still 1 hour.
- Big Data frameworks like Hadoop/MapReduce, where adding nodes allows handling more data, but the runtime doesn’t decrease for a fixed dataset.

Programming Ease

- Programming distributed systems is difficult
  - Divide a job into multiple tasks
  - Understand dependencies between tasks: Control, Data
  - Coordinate and synchronize execution of tasks
  - Pass information between tasks
  - Avoid race conditions, deadlocks
- Parallel and distributed programming models/languages/abstractions/platforms try to make these easy
  - Example:
  - In MapReduce, you just write map() and reduce() — the framework handles splitting data, distributing tasks, and recovering from failures.
  - In Apache Spark, a simple line rdd.map().reduce() runs across hundreds of nodes.

Concurrency

- Multiple processes may run in parallel and access shared resources. Proper coordination avoids conflicts. Components in distributed systems are executed in concurrent processes.
- Components access and update shared resources (e.g. variables, databases, device drivers).
- Integrity of the system may be violated if concurrent updates are not coordinated.
  - Lost updates
  - Inconsistent analysis

Example:

- Online shopping cart: Two users buying the last item — concurrency control ensures only one succeeds.
- Databases: Use locks/transactions to prevent inconsistent updates.

Error (Fault) Handling

- Failures are inevitable in distributed systems; design must tolerate them.
- **Techniques:** Redundancy, replication, checkpointing, automatic failover.
- **Example:**
  - **Google File System (GFS):** Each file is stored in 3 replicas across different servers. If one fails, data is still available.
  - **Amazon AWS:** Zones & regions replicate services for high availability.

Fault Tolerance and System Availability

- High availability (HA) is desired in all clusters, grids, P2P networks, and cloud systems. A system is highly available if it has a long Mean Time to Failure (MTTF) and a short Mean Time to Repair (MTTR).
- System Availability = MTTF / (MTTF + MTTR)
- All hardware, software, and network components may fail. Single points of failure that bring down the entire system must be avoided when designing distributed systems.

Fault Tolerance and System Availability

- For clouds, availability relates to the time that the datacenter is accessible or delivers the intended IT service as a proportion of the duration for which the service is purchased.
- Adding hardware redundancy, increasing component reliability, designing for testability all help to enhance system availability and dependability.
- In general, as a distributed system increases in size, availability decreases due to a higher chance of failure and a difficulty in isolating failures.

Things we now have to consider in the distributed case:

Scalability

Performance

Latency

Availability

Fault tolerance

ability of a system to behave in a well-defined manner once faults occur

Failure Handling (Fault Tolerance)

- Hardware, software and networks fail!
- Distributed systems must maintain *availability* even at low levels of hardware/software/network *reliability*.
- Fault tolerance is achieved by
  - recovery
  - redundancy

Things we now have to consider in the distributed case:

Scalability

Performance

Latency

Availability

Fault tolerance

the proportion of time a system is in a functioning condition. If a user cannot access the system, it is said to be unavailable.

Things we now have to consider in the distributed case:

Scalability

Performance

Latency

Availability

Fault tolerance

Availability =
uptime / (uptime + downtime)

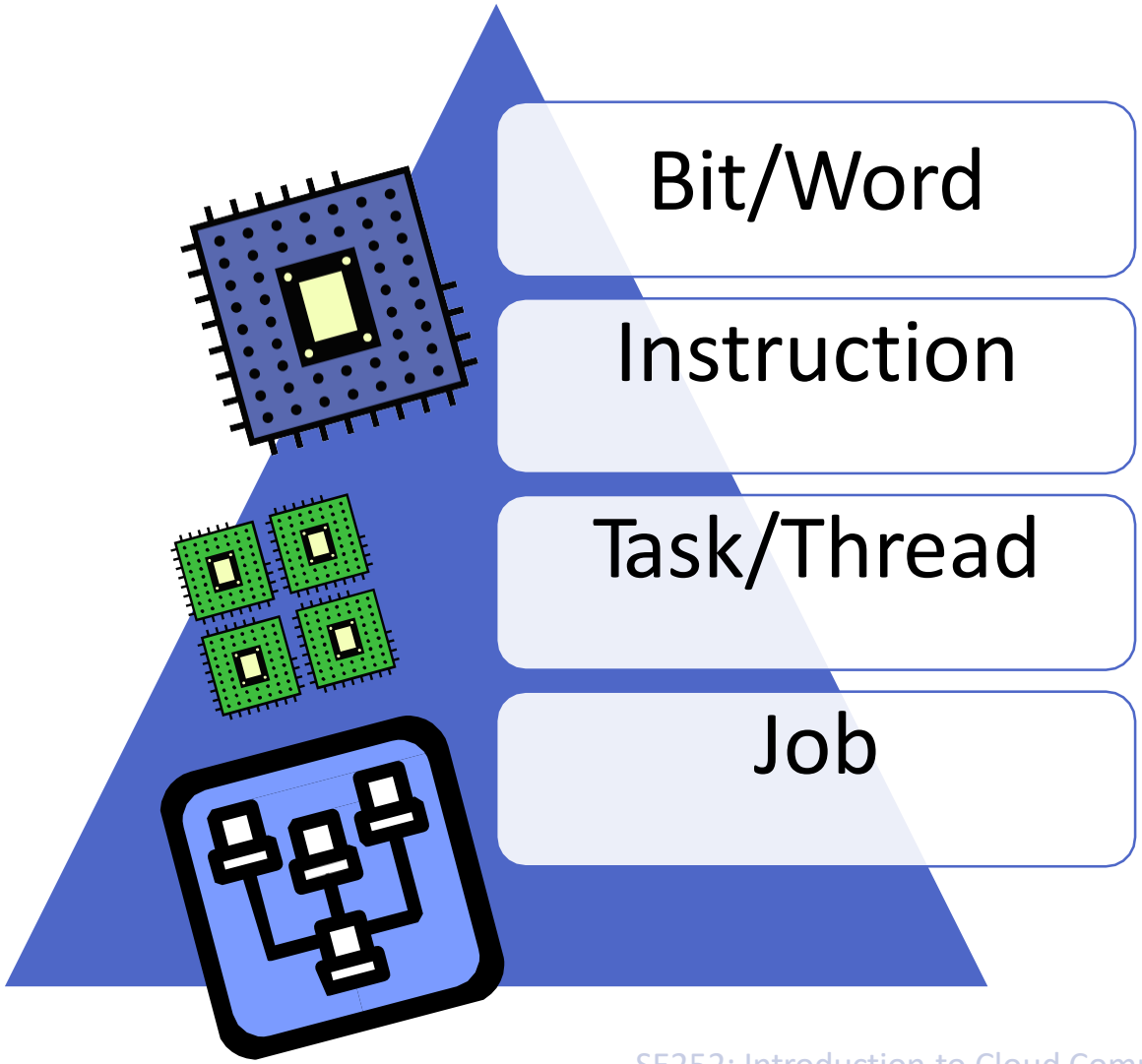
Availability % *How much downtime is allowed per year?*

90% ("one nine") More than a month 99% ("two nines")
Less than 4 days 99.9% ("three nines")
Less than 9 hours 99.99% ("four nines")
Less than an hour 99.999% ("five 9s) ~ 5 min
99.9999% ("six nines") ~ 31 seconds

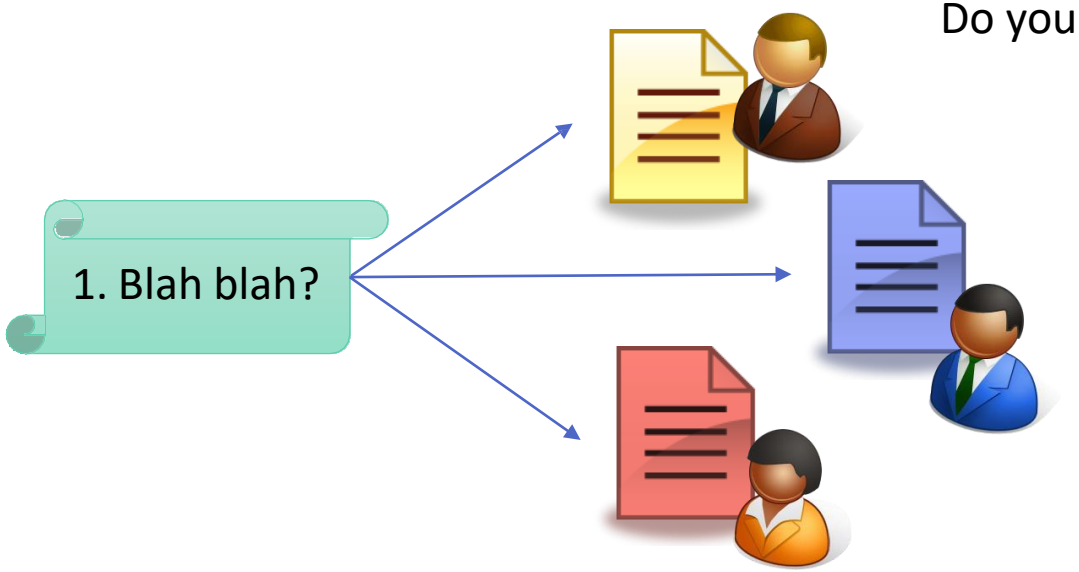
Security

- In a distributed system, clients send requests to access data managed by servers, resources in the networks:
  - Doctors requesting records from hospitals
  - Users purchase products through electronic commerce
- Security is required for:
  - Concealing the contents of messages: security and privacy
  - Identifying a remote user or other agent correctly (authentication)
- New challenges:
  - Denial of service attack
  - Security of mobile code

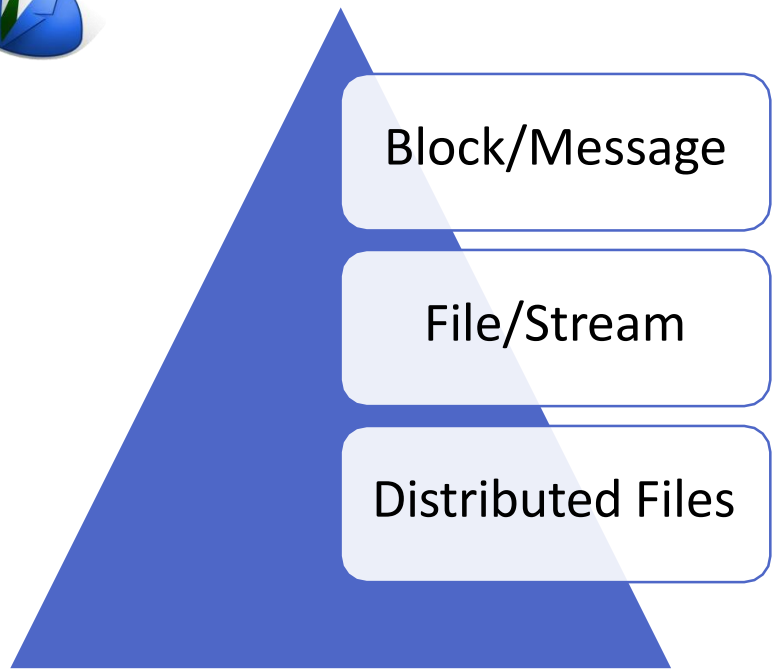
Degrees of parallelism



Degrees of Parallelism



- Data parallel vs. Task Parallel
  - Independent processes
  - Independent data dependency
- Tight vs. Loose Coupling



How Big Data systems have been designed in view of all of these potential distribution-specific issues?

Heterogeneity: Embracing Diversity

The Challenge

Clusters contain diverse hardware, OSes, programming languages, and network technologies that must work together seamlessly.

Design Principle

Use middleware and abstraction layers so applications don't need to handle underlying differences.

Real-World Implementations

HDFS

Works across commodity servers running different OSes (Linux, Windows) with various storage types.

Apache Spark

Supports multiple languages (Scala, Java, Python, R) — developers use their preferred language while the system handles integration.

Kubernetes

Abstracts away whether applications run on on-premises servers, AWS, or Azure cloud infrastructure.

Openness

The Challenge

Systems must be extensible, interoperable with other systems, and adaptable to future technologies.

Design Principle

Publish **open APIs,** follow industry standards, and provide extension points through plug-in architectures.

Implementation Examples

- The **Hadoop ecosystem** integrates specialized tools like Hive (SQL), Pig (scripting), and Mahout (ML) through open interfaces
- **Cloud storage services** (S3, Azure Blob) expose REST APIs enabling any tool to integrate
- **POSIX compliance** allows Spark and Hadoop jobs to access compatible storage without modifications

Key Insight: Open systems avoid vendor lock-in and allow organizations to adapt as technology evolves, protecting long-term investments in distributed architecture.

Transparency: Hiding Complexity

"Make simple things simple, make complex things possible."

The Challenge

Users and developers shouldn't need to understand internal details like data location, replication strategies, or job distribution.

Design Principle

Hide implementation complexity behind clean abstractions that make distributed systems appear as single, unified entities.

Types of Transparency

- **Access Transparency:** SQL queries look identical whether the database is local or distributed
- **Location Transparency:** Google Drive users access files without knowing which datacenter they're in
- **Replication Transparency:** Netflix streams from the nearest CDN without user awareness
- **Failure Transparency:** S3 automatically serves data from another copy if a server fails
- **Scaling Transparency:** Adding 100 new Cassandra nodes requires no query changes

Security: Protecting Distributed Assets

The Challenge

Distributed systems present broader attack surfaces—more nodes, more network traffic, and more points of vulnerability.

Design Principle

Implement multi-level security through authentication, encryption, and fine-grained access control throughout the system.

Security Implementation Patterns

Authentication

Hadoop clusters use **Kerberos** for secure identity verification

Encryption

Spark, Kafka, and Cassandra implement TLS/SSL to prevent data interception

Access Control

MongoDB and Elasticsearch implement RBAC with granular permissions

Example: Healthcare systems storing patient records in Hadoop must implement encryption and audit trails for HIPAA compliance

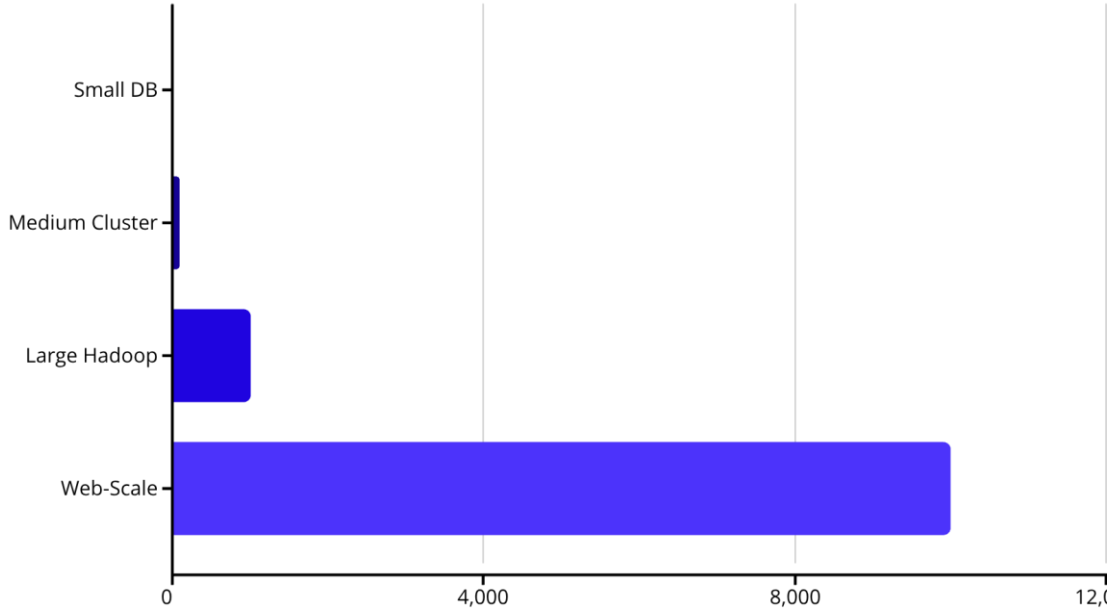
Scalability: Growing Without Limits

The Challenge

Big Data workloads grow rapidly from terabytes to petabytes, often with unpredictable scaling requirements.

Design Principle

Prioritize **horizontal scaling** (adding more machines) over vertical scaling (bigger servers) to create elastic, cost-effective architectures.



Real-World Examples

- **Hadoop MapReduce:** Processes petabytes by splitting data into blocks for parallel execution
- **Cassandra Database:** Linearly scalable—Amazon runs thousands of nodes for shopping cart data
- **Netflix:** Dynamically spins up thousands of additional AWS instances during peak viewing hours

Fault Tolerance & Availability

> "In a large enough distributed system, something is always failing."

The Challenge

Hardware, software, and network failures are inevitable at scale. The larger the system, the more frequent the failures.

Design Principle

Build redundancy, replication, checkpointing, and automatic failover into every layer of the architecture.

Implementation Examples

- **HDFS**

  Default replication factor = 3, ensuring data availability even when nodes fail

- **Spark Checkpointing**

  Saves computation progress; jobs can restart from checkpoints after failure

- **AWS S3**

  Data automatically replicates across multiple a vailability zones

Concurrency: Coordinating Shared Resources

The Challenge

Multiple processes or users may access and update shared resources simultaneously, creating race conditions and inconsistent states.

Design Principle

Implement coordination services, distributed locks, and transaction mechanisms to maintain consistency across components.

Importance

Without proper concurrency control, distributed systems can produce incorrect results, lose data, or deadlock.

Concurrency Solutions

Coordination Services

Apache ZooKeeper provides centralized coordination for distributed systems:

- Leader election in Hadoop & Kafka
- Configuration management
- Distributed locking primitives

Database Approaches

- **MongoDB:** Replica set elections maintain consistency
- **MySQL Cluster:** ACID transactions for strong
- concurrency control
- **Banking Example:** Two users withdrawing from the same account must use atomic transactions to prevent overdrafts

Principles in Action: Case Study

E-Commerce Platform During Holiday Shopping Season

Heterogeneity

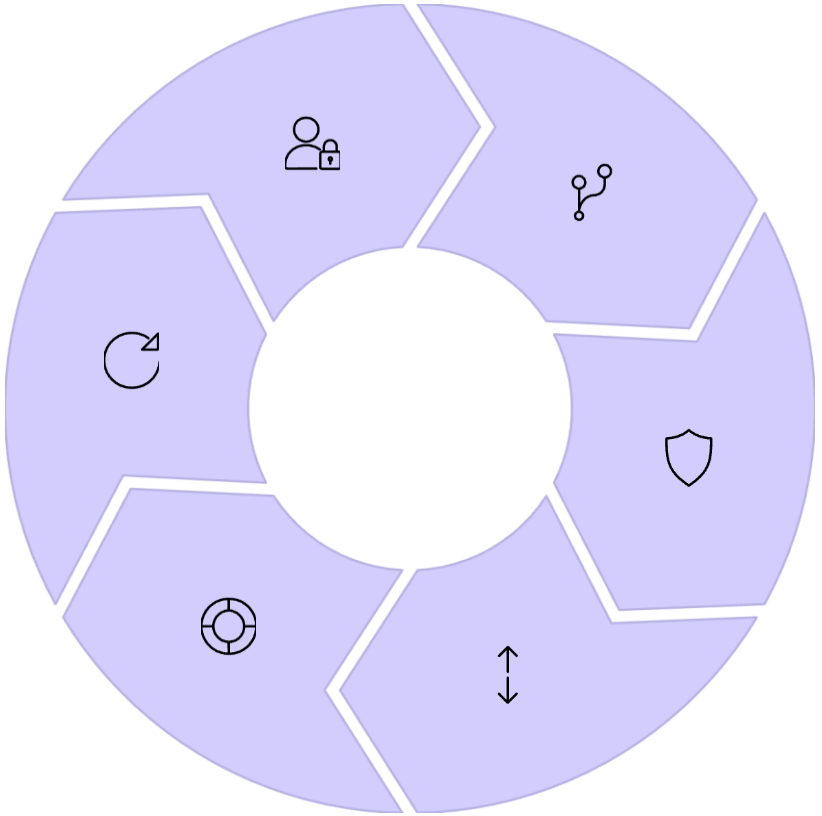
Mix of on-prem and cloud resources with microservices in different languages

Concurrency

Inventory system uses distributed locks to prevent overselling

Fault Tolerance

Multi-region deployment ensures availability despite regional outages



Openness

REST APIs enable seamless integration between services and third-party systems

Security

Encryption of payment data, tokenization, and Role-Based Access Control for sensitive operations

Scalability

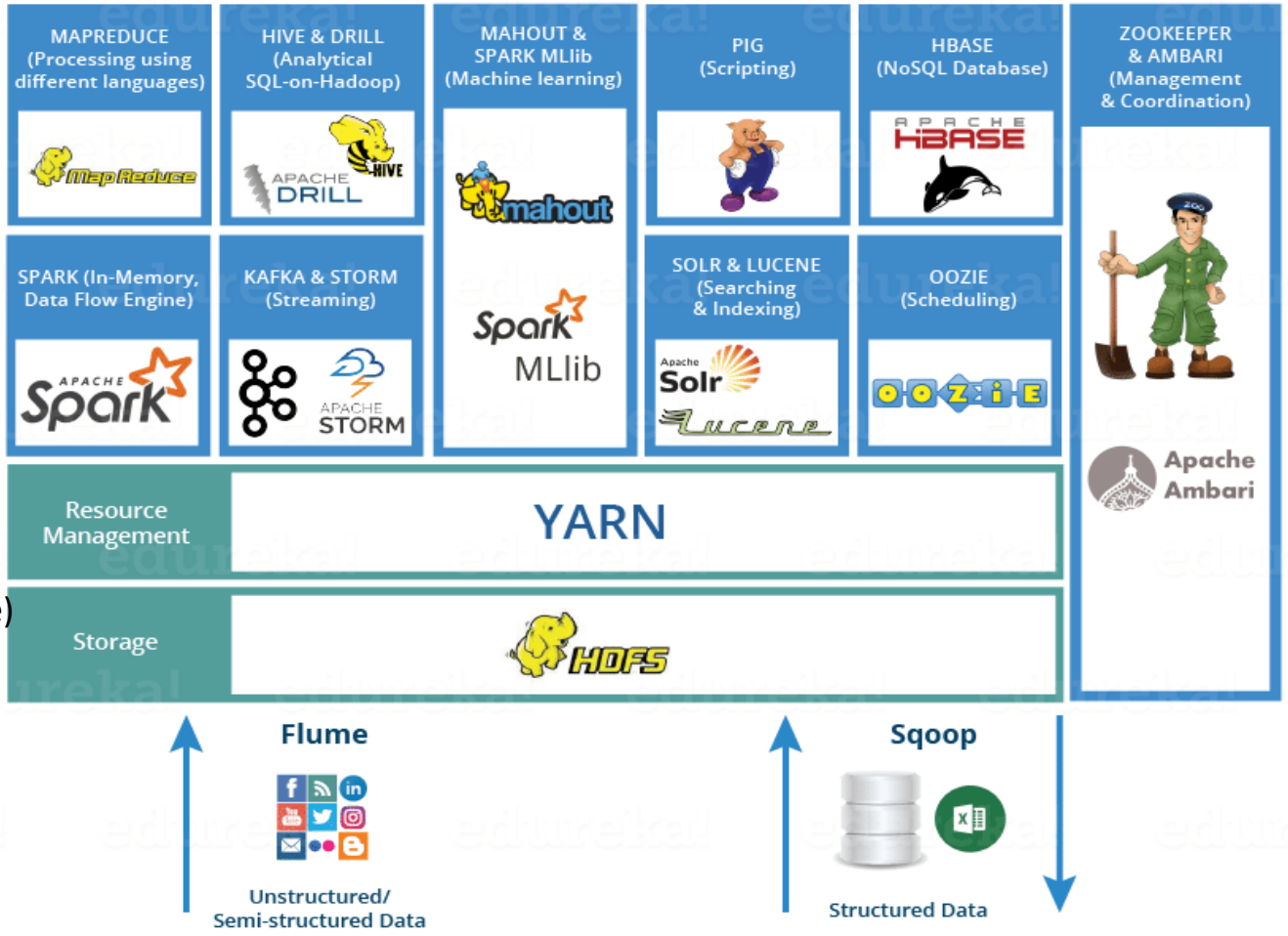
Auto-scaling during Black Friday traffic spikes (10x normal volume)

By applying all seven principles, the platform handled a 500% increase in traffic with 99.99% uptime during peak holiday shopping, processing over 25,000 transactions per minute.

Hadoop Ecosystem

HDFS -> Hadoop Distributed File System
YARN -> Yet Another Resource Negotiator
MapReduce -> Data processing using programming
Spark -> In-memory Data Processing
PIG, HIVE-> Data Processing Services using Query (SQL-like)
HBase -> NoSQL Database

Mahout, Spark MLlib -> Machine Learning
Apache Drill -> SQL on Hadoop
Zookeeper -> Managing Cluster
Oozie -> Job Scheduling
Flume, Sqoop -> Data Ingesting Services
Solr & Lucene -> Searching & Indexing
Ambari -> Provision, Monitor and Maintain cluster



Why Architecture Matters in Big Data

Big Data is defined by the 5 Vs:

- **Volume**: Petabytes of information
- **Velocity**: Millions of events per second
- **Variety**: Structured, semi-structured, unstructured
- **Veracity**: Uncertainty, incompleteness
- **Value**: Extracting business intelligence

Without proper architecture, systems bec

⊗ **Slow**: Queries on terabytes taking hours instead of seconds

⚠ **Inflexible**: Unable to scale with growing data volumes

⚠ **Unreliable**: Single points of failure causing outages

Reference Architectures Evolved to Solve These Challenges

Each pattern optimizes for different priorities in the data processing pipeline

Big Data Architectures

Big Data is still **“work in progress”**

- Choosing the right architecture is key for any (big data) project.
- Big Data is still quite a young field and therefore there are no standard architectures available which have been used for years.
- In the past few years, a few architectures have evolved and have been discussed online.
- Know the use cases before choosing your architecture.
- To have one/a few reference architectures can help in choosing the right components.

Big Data Architectures

Data Processing architectures

- Lambda architecture
- Kappa architecture

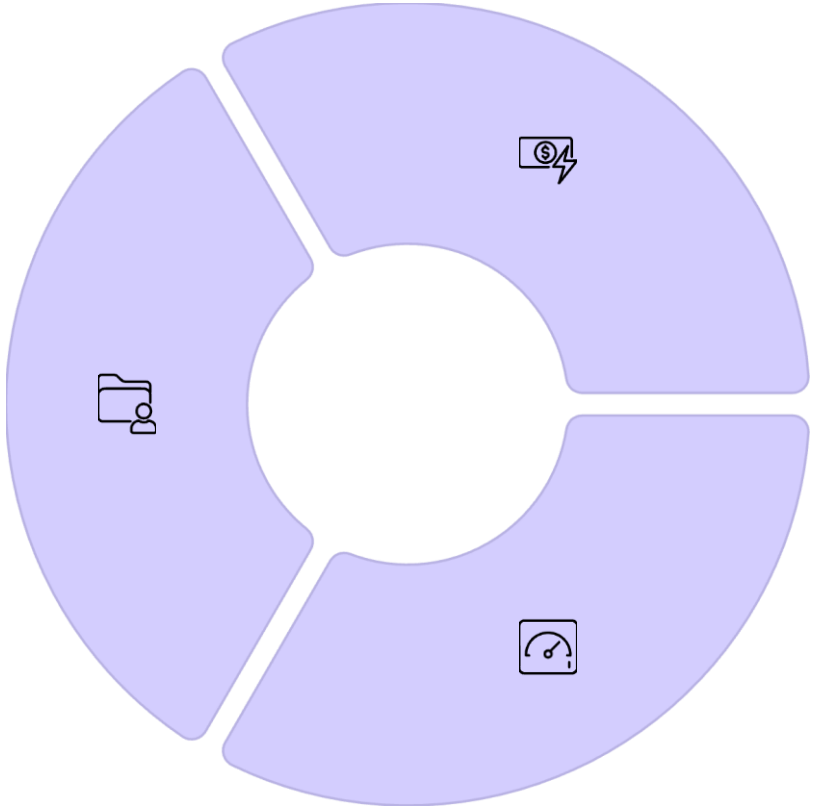
Lambda Architecture

Concept: Combining batch and streaming for both accuracy and speed

Batch Layer

"Cold Path"

- Stores all data immutably
- Processes with Hadoop, Spark Batch
- Produces complete but delayed views



Speed Layer

"Hot Path"

- Real-time event processing
- Uses Storm, Spark Streaming, Flink
- Creates fast but approximate views

Serving Layer

Unified Access

- Merges batch and real-time views
- Optimizes for query performance
- Provides complete + fresh results

Lambda Architecture Example: Twitter Analytics

Data Sources

Billions of tweets, user actions, and engagement metrics

Batch Layer

Trains ML models on historical tweets to identify patterns and sentiment baselines

Speed Layer

Real-time sentiment analysis on trending hashtags and breaking topics

Serving Layer

Dashboards showing both long-term trends and live activity

Data Processing Architectures

- **Lambda architecture**
  - When working with very large data sets, it can take a long time to run the sort of queries that clients need.
  - These queries can't be performed in real time, and often require algorithms such as MapReduce that operate in parallel across the entire data set.
  - The results are then stored separately from the raw data and used for querying.

Data Processing Architectures

- **Lambda architecture**
- One drawback to this approach is that it introduces latency — if processing takes a few hours, a query may return results that are several hours old.
- Ideally, you would like to get some results in real time (perhaps with some loss of accuracy), and combine these results with the results from the batch analytics.

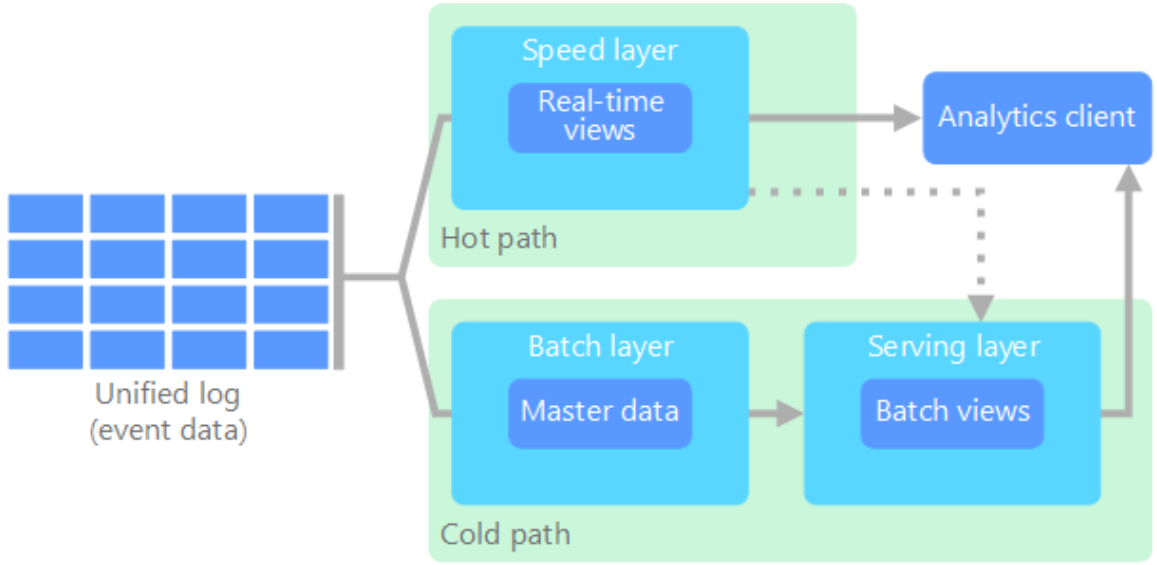
Data Processing Architectures

Lambda architecture

- The **lambda architecture**, first proposed by Nathan Marz, addresses this problem by creating two paths for data flow. All data coming into the system goes through these two paths:
- A **batch layer** (cold path) stores all of the incoming data in its raw form and performs batch processing on the data. The result of this processing is stored as a **batch view**.
- A **speed layer** (hot path) analyzes data in real time. This layer is designed for low latency, at the expense of accuracy.

Data Processing Architectures

- **Lambda architecture**
- The batch layer feeds into a **serving layer** that indexes the batch view for efficient querying. The speed layer updates the serving layer with incremental updates based on the most recent data.



Data Processing Architectures

- **Lambda architecture**
  - Combines (Big) Data at Rest with (Fast) Data in Motion
  - Closes the gap from high-latency batch processing
  - Keeps the raw information forever
  - Makes it possible to rerun analytics operations on whole data set if necessary
  - Have to implement functionality twice
    - Once for batch
    - Once for real-time streaming

Data Processing Architectures

- **Software requirements of Lambda architecture**
  - Batch layer
    - HDFS (Storage) + MapReduce/PIG/Hive (Functions)
  - Speed Layer
    - Storm / S4 / Spark Streaming
  - Serving Layer
    - NoSQL database (Cassandra/ Hbase / CouchDB/ Voldemort/ MongoDB)
  - Queuing System
    - Apache Kafka / Flume

Data Processing Architectures

- **Hardware requirements for the lambda architecture**
- Batch layer
  - 1 replicated master node (6 cores CPU, 4 GB memory, RAID-1 storage, 64-bit operating system)
  - 2 worker nodes (12 cores CPU, 4 GB memory, 2 TB storage, 1 GbE NIC)
  - 1 dedicated resource manager (YARN) node (4 GB memory, and 4core)
- Speed layer Shares the Hadoop node
- Serving layer 2 nodes (1TB, 4 cores, 16 GB memory)

Data Processing Architectures

- **Kappa Architecture**
- A drawback to the lambda architecture is its complexity.
- Processing logic appears in two different places — the cold and hot paths — using different frameworks.
- This leads to duplicate computation logic and the complexity of managing the architecture for both paths.
- The **kappa architecture** was proposed by Jay Kreps as an alternative to the lambda architecture.
- It has the same basic goals as the lambda architecture, but with an important distinction: All data flows through a single path, using a stream processing system.

Kappa Architecture

Concept: Stream-first design that simplifies Lambda by eliminating the batch layer

Single Pipeline

All data flows through a unified streaming system

Recomputation via Replay

Historical processing done by replaying data from log (typically Kafka)

Modern Streaming Engines

Leverages Flink, Samza, or Spark Structured Streaming

Proposed by Jay Kreps (LinkedIn/Confluent) to address Lambda's complexity

Ideal Use Cases

- Fraud detection systems
- Real-time log monitoring
- Anomaly detection platforms
- Stock market analytics
- Real-time recommendation systems

Kappa Architecture Example: LinkedIn

User Interactions

Profile views, connection requests, content engagement, job applications

Kafka Streams

Billions of events per day stored in immutable logs with configurable retention

Stream Processing

Samza/Flink processes events for recommendations, analytics, and feature extraction

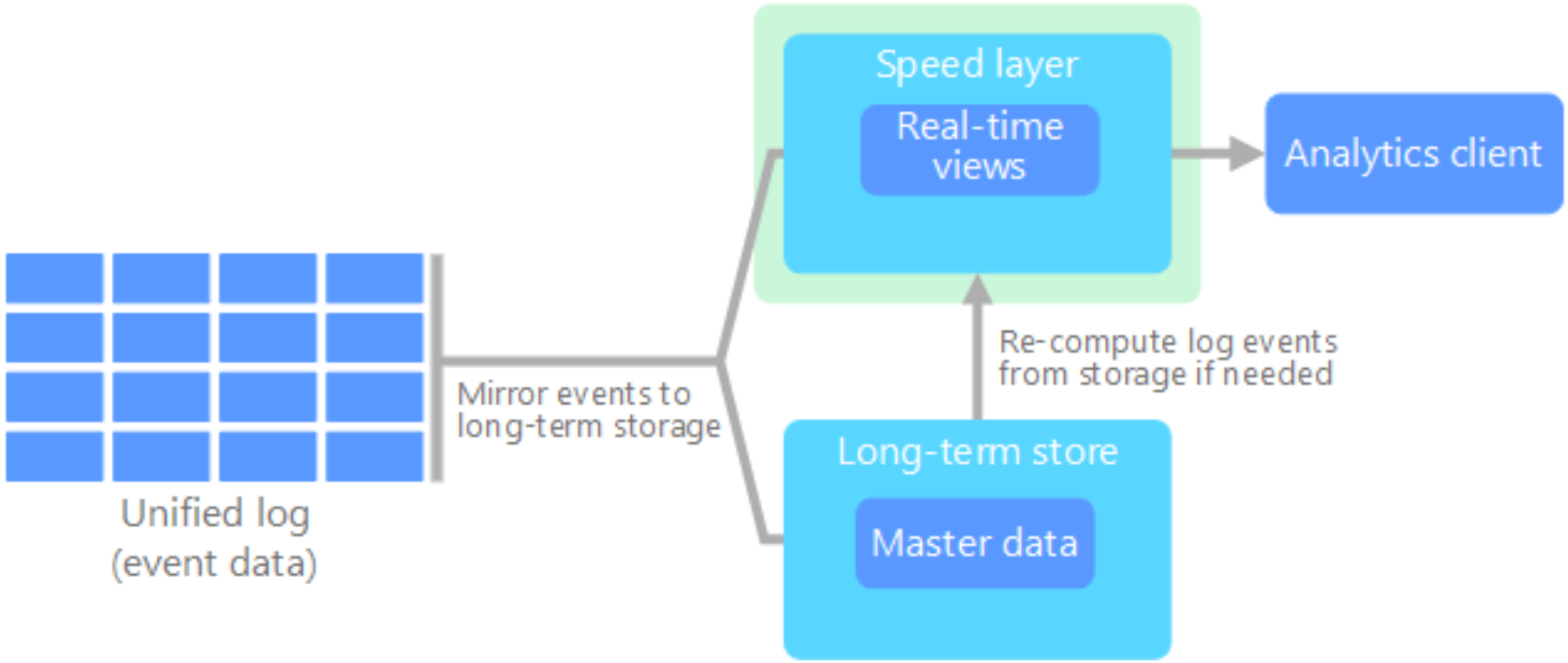
Real-time Models

Recommendation systems updated continuously based on user behavior patterns

When algorithm changes are needed, LinkedIn can replay the entire event history from Kafka to rebuild models without a separate batch system.

Big Data Architectures

- **Kappa Architecture**



Data Processing Architectures

- **Kappa Architecture**
- There are some similarities to the lambda architecture's batch layer, in that the event data is immutable and all of it is collected, instead of a subset.
- The data is ingested as a stream of events into a distributed and fault tolerant unified log.
- These events are ordered, and the current state of an event is changed only by a new event being appended.
- Similar to a lambda architecture's speed layer, all event processing is performed on the input stream and persisted as a real-time view.
- If you need to recompute the entire data set (equivalent to what the batch layer does in lambda), you simply replay the stream, typically using parallelism to complete the computation in a timely fashion.

Data Processing Architectures

- **Kappa Architecture**
  - The solution for low latency use cases
  - Process each event separately
  - => low latency Process events in micro-batches
  - => increases latency but offers better reliability Previously known as “Complex Event Processing”
  - Keep the data moving / Data in Motion instead of Data at Rest

Data Processing Architectures

- **Software requirements for the Kappa architecture**
- The software requirements for the Kappa architecture are quite similar to those of the Lambda Architecture minus the Hadoop platform used to implement the batch layer which is absent here.
- Because it can retain ordered data logs allowing for data reprocessing, Apache Kafka is preferred for ingestion.
- Apache Flink is particularly suitable for processing as it allows building time windows for computations.
- A popular alternative to it is Apache Samza

Data Processing Architectures

- **Hardware requirements for the Kappa architecture**
  - Ingestion tools
    - 10 servers having each :12 physical processors, 16 GB RAM
  - Speed layer (Storm)
    - Minimum one server having : 16 GB RAM, 6 core CPUs of 2 GHz (or more) each, 4 x 2 TB, 1 GB Ethernet
  - Serving layer
    - similar to lambda architecture

Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- From a practical viewpoint, Internet of Things (IoT) represents any device that is connected to the Internet.
- This includes your PC, mobile phone, smart watch, smart thermostat, smart refrigerator, connected automobile, heart monitoring implants, and anything else that connects to the Internet and sends or receives data.
- The number of connected devices grows every day, as does the amount of data collected from them. Often this data is being collected in highly constrained, sometimes high-latency environments.

IoT / Event-Driven Architecture

Concept: Designed specifically for billions of distributed devices and sensors

Field Gateway

Edge computing for near-device preprocessing, filtering, and aggregation

Cloud Gateway

Reliable message ingestion using Kafka and other protocols

Stream Processors

Real-time analytics engines for immediate insights and alerts

Cold Storage

Data lake for archiving historical data (HDFS, S3, Azure Data Lake)

IoT Architecture Example: Smart City Traffic

Components

- Traffic cameras
- Road sensors
- Vehicle counters
- Weather stations
- Signal control systems

Data Characteristics

- High frequency (readings every few seconds)
- Geographically distributed
- Mixed data types (video, numeric, text)

Edge Processing

Field gateways filter and aggregate sensor data to reduce bandwidth needs

Data Ingestion

Kafka ingests data streams from thousands of sensors across the city

Real-time Analytics

Spark Streaming detects congestion patterns and triggers signal adjustments

Historical Analysis

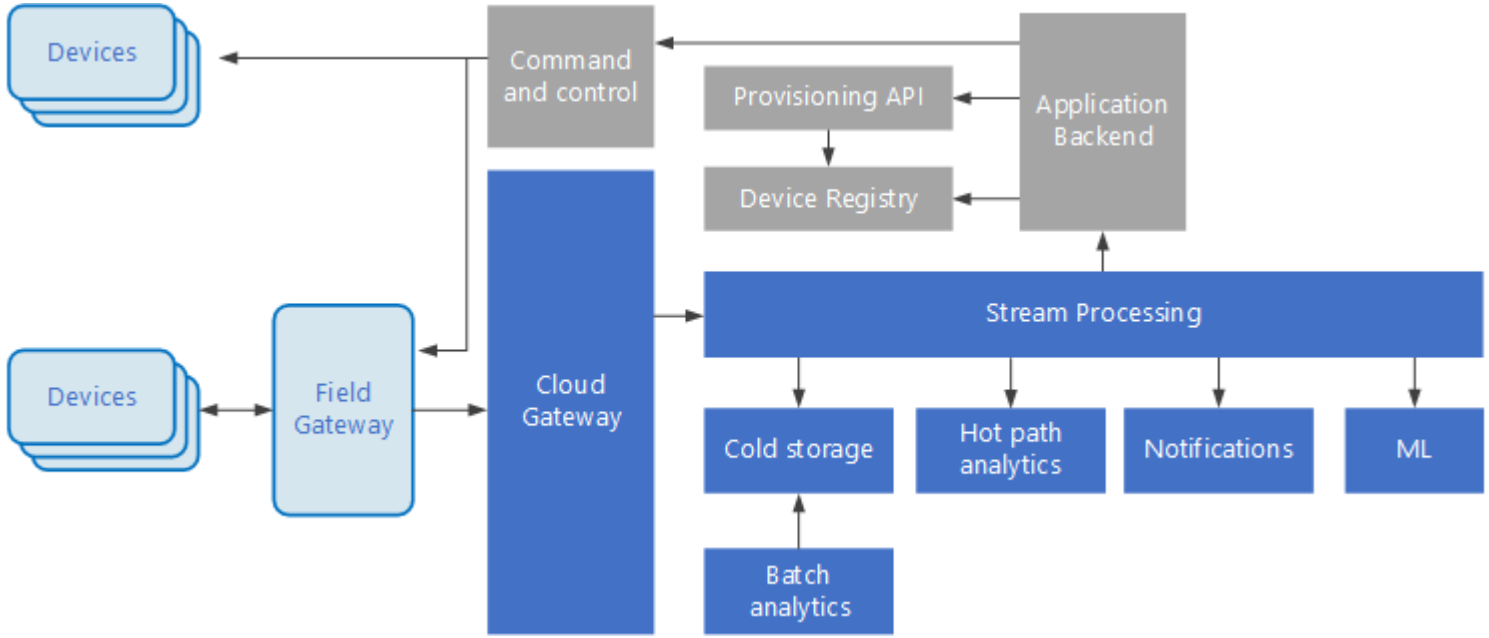
Cold storage retains 1+ year of traffic data for urban planning

Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- In other cases, data is sent from low-latency environments by thousands or millions of devices, requiring the ability to rapidly ingest the data and process accordingly.
- Therefore, proper planning is required to handle these constraints and unique requirements.

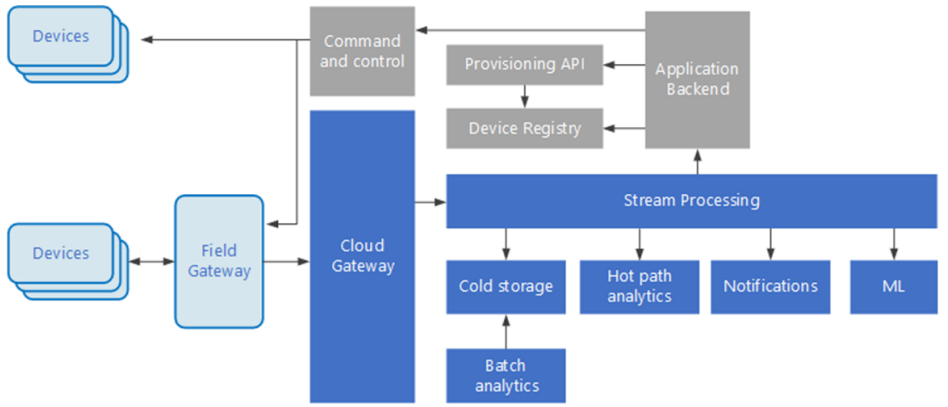
Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- Event-driven architectures are central to IoT solutions.
- The following diagram shows a possible logical architecture for IoT.
- The diagram emphasizes the event-streaming components of the architecture.



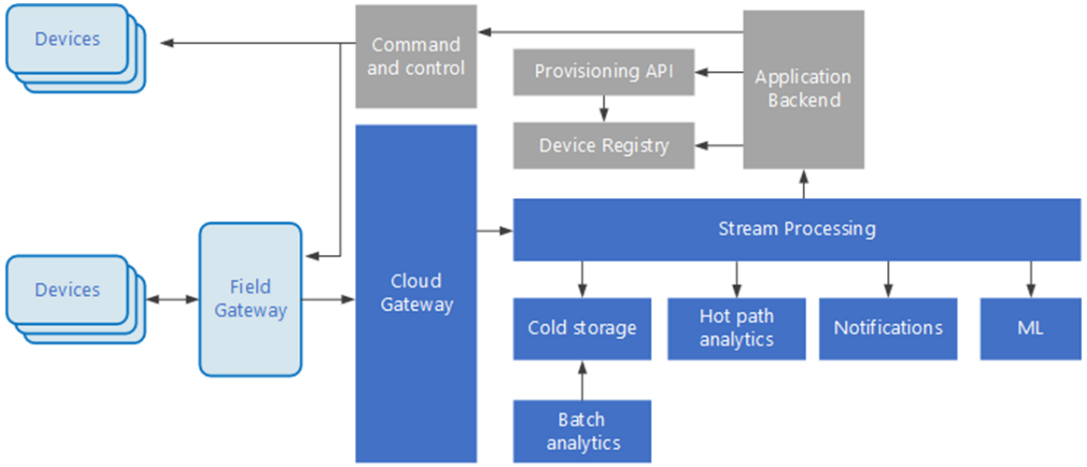
Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- The **cloud gateway** ingests device events at the cloud boundary, using a reliable, low latency messaging system.
- Devices might send events directly to the cloud gateway, or through a **field gateway**. A field gateway is a specialized device or software, usually collocated with the devices, that receives events and forwards them to the cloud gateway. The field gateway might also preprocess the raw device events, performing functions such as filtering, aggregation, or protocol transformation.
- After ingestion, events go through one or more **stream processors** that can route the data (for example, to storage) or perform analytics and other processing.



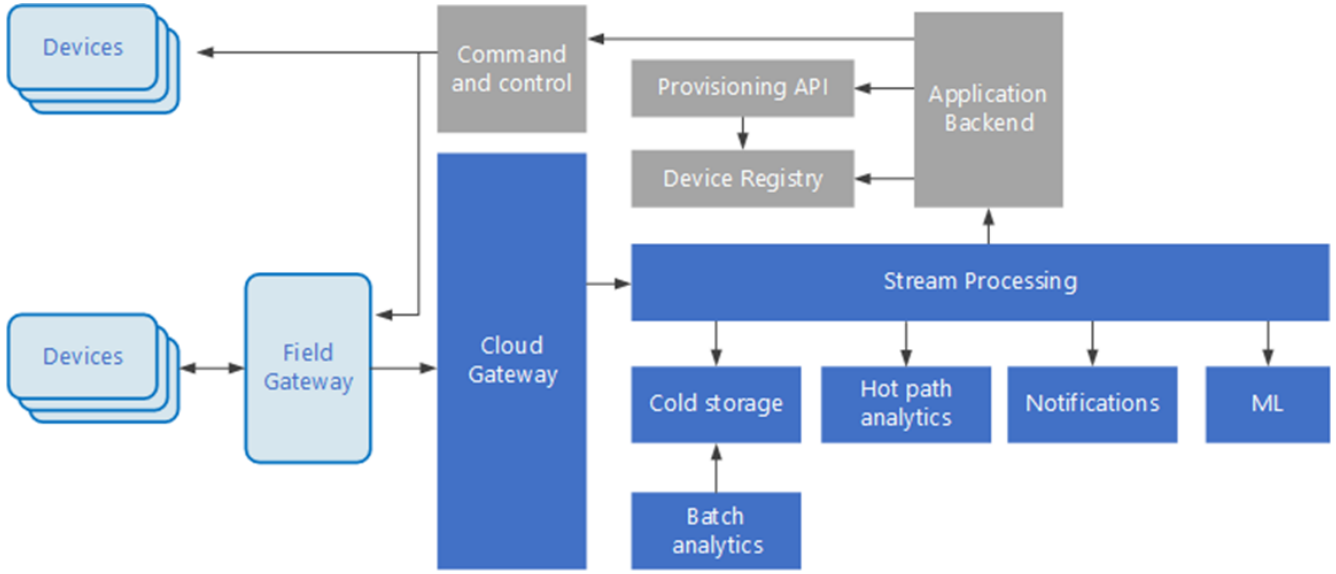
Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- The following are some common types of processing.
  - Writing event data to cold storage, for archiving or batch analytics.
  - Hot path analytics, analyzing the event stream in (near) real time, to detect anomalies, recognize patterns over rolling time windows, or trigger alerts when a specific condition occurs in the stream.
  - Handling special types of nontelemetric messages from devices, such as notifications and alarms.
  - Machine learning.

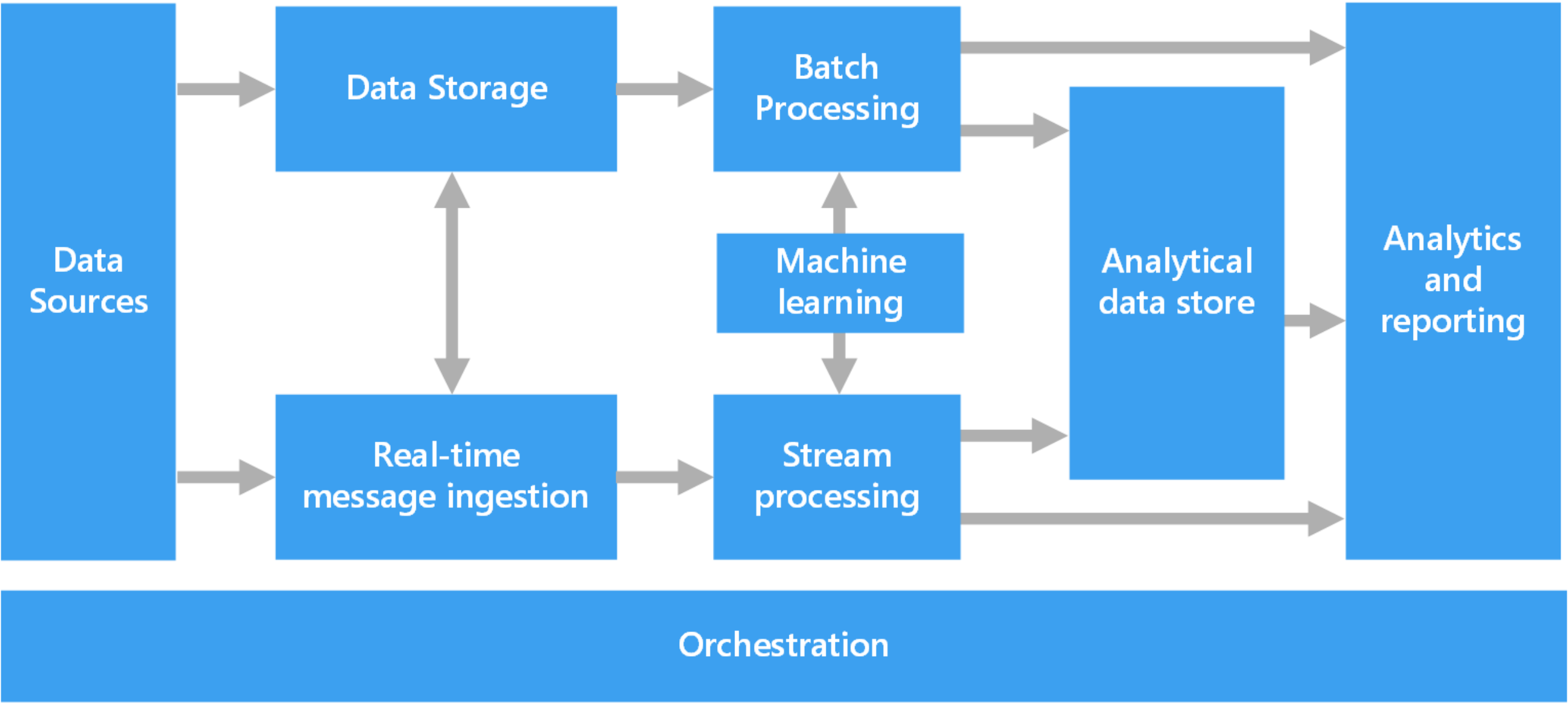


Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- The **device registry** is a database of the provisioned devices, including the device IDs and usually device metadata, such as location.
- The **provisioning API** is a common external interface for provisioning and registering new devices.
- Some IoT solutions allow **command and control messages** to be sent to devices.



Components of a big data architecture

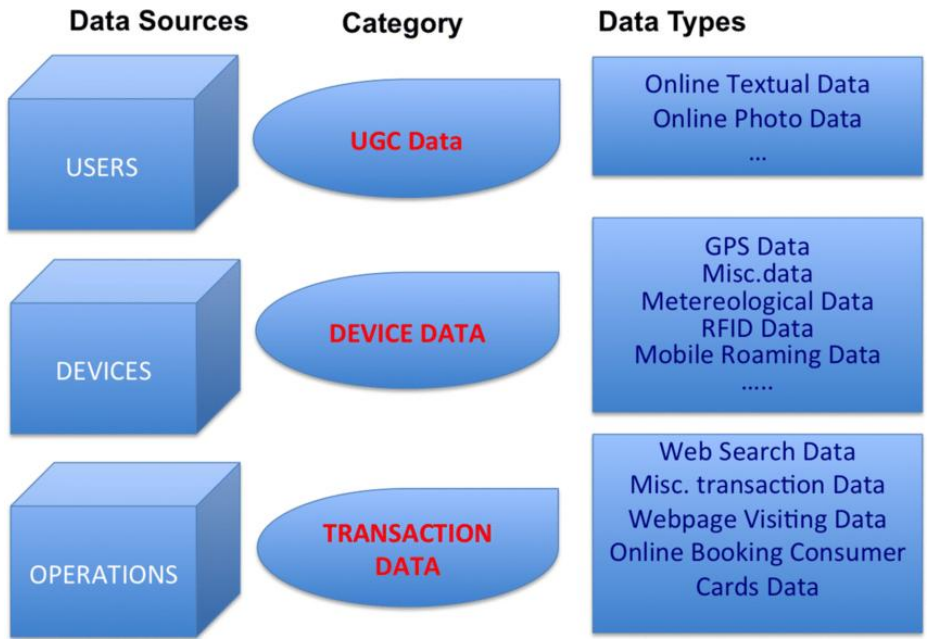


Data Source Layer

- The Big Data sources are the ones that govern the Big Data architecture.
- The designing of the architecture depends heavily on the data sources.
- The data is arriving from numerous sources that too in different formats.
- This data can be both batch data as well as real-time data.
- These sources pile up a huge amount of data in no time.
- The Big Data architecture is designed such that it is capable of handling this data.

Data Source Layer

- **Data sources**. All big data solutions start with one or more data sources.
- Examples include:
  - Application data stores, such as relational databases.
  - Static files produced by applications, such as web server log files.
  - Real-time data sources, such as IoT devices.



Data source layer

- Data from SQL, NoSQL stores (MySQL, Oracle, PostgreSQL, MongoDB, etc. – Mostly structured)
- (Semi/Un)-structured data (CRM, marketing, campaign, spend, revenue, leads, etc.)
- Web logs or other log files (weblogs, user clicks, user visits, activity, etc.)

Unstructured data

The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

```
<University>
 <Student ID="1">
   <Name>John</Name>
   <Age>18</Age>
   <Degree>B.Sc.</Degree>
 </Student>
 <Student ID="2">
   <Name>David</Name>
   <Age>31</Age>
   <Degree>Ph.D. </Degree>
 </Student>
....
</University>
```

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

Data storage Layer

- **Data storage**. Data for batch processing operations is typically stored in a distributed file store that can hold high volumes of large files in various formats.
- As the volume of data generated and stored by companies has started to explode, sophisticated but accessible systems and tools have been developed – such as Apache Hadoop DFS (distributed file system).
- A data lake can also be an example of data storage.

Data storage Layer

- **Direct-attached storage (DAS)**
- Direct-attached storage stands for all types of physical data storage devices you can connect to a computer.
- Portable and affordable — yet only accessible by one computer at a time — DAS is a standard solution for keeping small-scale records data backups or for transferring data between devices.
- Popular types of direct-attached storage include external hard drives or solid-state drives (SSD), flash drives (USB sticks) and, although now dwindling in popularity, CD/DVD disks and other older methods.

Data storage Layer

- **Network-attached storage (NAS)**
- Network-attached storage (NAS) is a special hardware unit, featuring file-level architecture that can be accessed by more than one device as long as all users are connected to the internal network.
- In essence, a NAS unit features several storage disks or hard drives, processors, RAM, and lightweight operating systems (OS) for managing access requests.

Data storage Layer

- **Storage area networks (SAN)**
- Storage area networks (SANs) help assemble an even more complex on-premises data management architecture that features two components:
- A dedicated network for data exchanges with network switches for load balancing
- Data storage system, consisting of on-premises hardware.
- The purpose of SAN is to act as a separate “highway” for transmitting data between servers and storage devices across the organization, in a bypass of local area networks (LANs) and wide-area networks (WANs). Featuring a management layer, SANs can be configured to speed up and strengthen server-to-server, storage-to-server and storage-to-storage connections.

Data storage Layer

- **Software-defined storage (SDS)**
- A software-defined storage (SDS) system is a hardware-independent, software-based storage architecture that can be used on any computing hardware platform.
- The two major benefits of SDS include:
  - Scalability:
  - Total cost of ownership (TCO)

Data storage Layer

- **Hyperconverged storage (HCS)**
- Hyperconverged storage allows you to virtualize on-premises storage resources to create a shared storage pool. Each hardware unit (node) gets virtualized and then connected in a cluster, which you then manage as a unified system.
- By using hyperconverged storage architecture and virtualizing some of your legacy servers, you can significantly reduce maintenance costs.

Data storage Layer

- **Cloud data storage**
- Unlike other options, cloud data storage assumes you will (primarily) use offsite storage of data in public, private, hybrid or multicloud environments that are managed and maintained by cloud services providers such as Amazon, Microsoft and Google, among others.
- Unlike NAS or SAN, public cloud data storage doesn’t require a separate internal network — all data is accessible via the internet. Also, there are virtually no limits on scalability since you are renting storage resources from a third party that effectively offers an endless supply of servers.

Data storage Layer

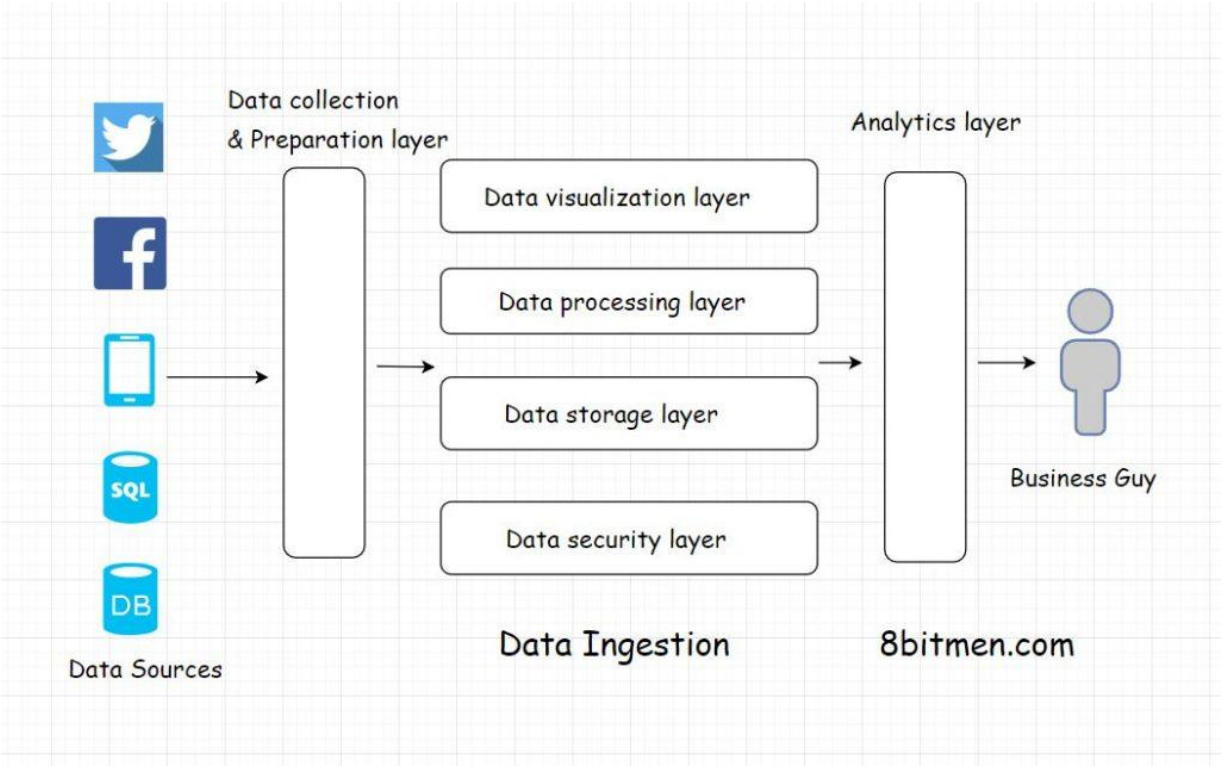
- **What is a data lake?**
- A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale.
- You can store your data as-is, without having to first structure the data, and run different types of analytics—from dashboards and visualizations to big data processing, real-time analytics, and machine learning to guide better decisions.

Data storage Layer

Characteristics	Data Warehouse	Data Lake
Data	Relational from transactional systems, operational databases, and line of business applications	Non-relational and relational from IoT devices, web sites, mobile apps, social media, and corporate applications
Schema	Designed prior to the DW implementation (schema-on-write)	Written at the time of analysis (schema-on-read)
Price/Performance	Fastest query results using higher cost storage	Query results getting faster using low-cost storage
Data Quality	Highly curated data that serves as the central version of the truth	Any data that may or may not be curated (ie. raw data)
Users	Business analysts	Data scientists, Data developers, and Business analysts (using curated data)
Analytics	Batch reporting, BI and visualizations	Machine Learning, Predictive analytics, data discovery and profiling

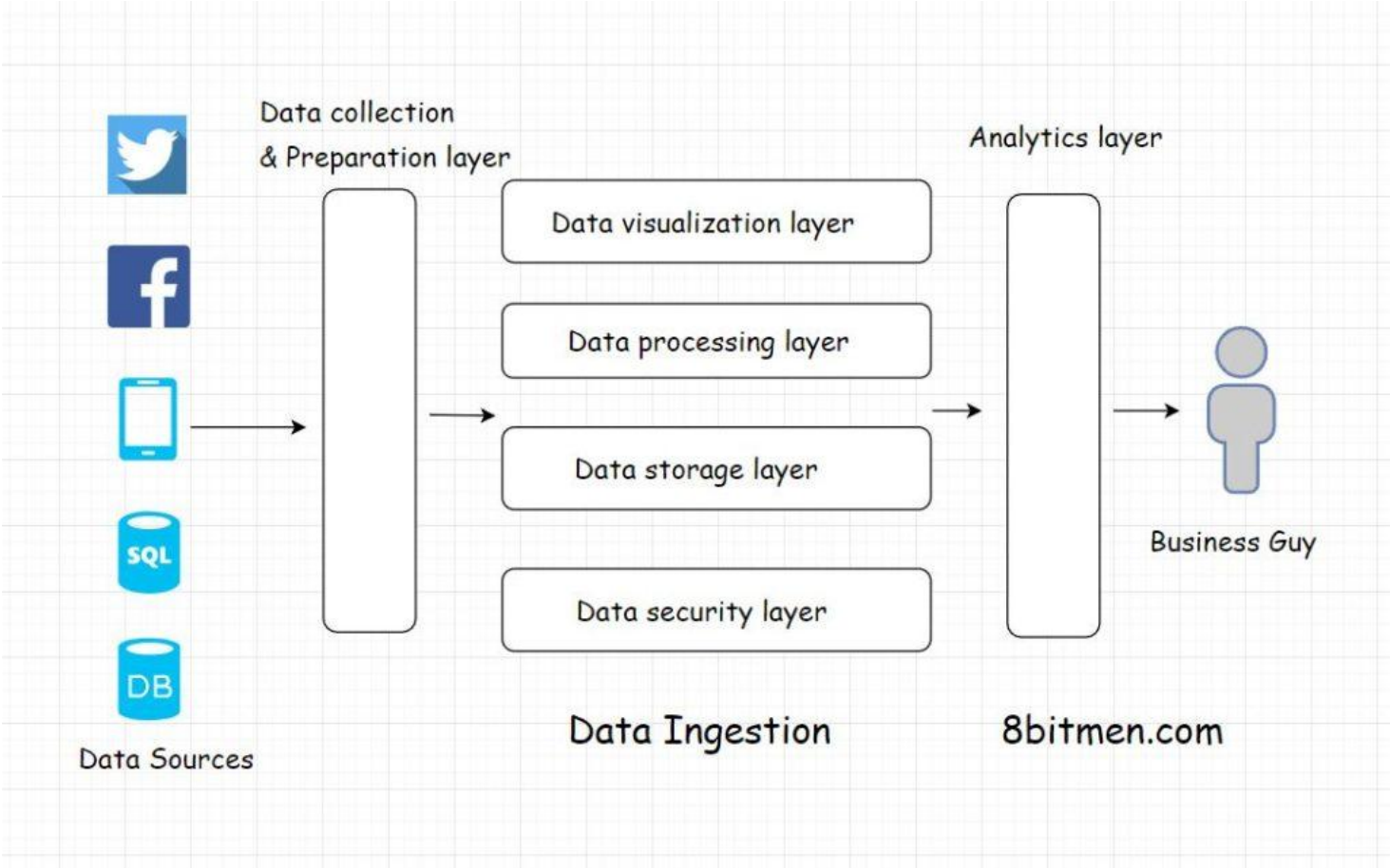
Data Ingestion Layer

- Data ingestion is the initial & the toughest part of the entire data processing architecture.
- The key parameters which are to be considered when designing a data ingestion solution are:
- Data Velocity, size & format: Data streams in through several different sources into the system at different speeds & size. Data streams from social networks, IoT devices, machines & what not. And every stream of data streaming in has different semantics. A stream might be structured, unstructured or semi-structured.



Data Ingestion Architecture

- The frequency of data streaming: Data can be streamed in continually in real-time or at regular batches. We would need weather data to stream in continually. On the other hand, to study trends social media data can be streamed in at regular intervals.



Orchestration

- Most big data solutions consist of repeated data processing operations, encapsulated in workflows, that transform source data, move data between multiple sources and sinks, load the processed data into an analytical data store, or push the results straight to a report or dashboard.
- To automate these workflows, you can use an orchestration technology such Apache Oozie and Sqoop.

Big Data Architecture – Layered Components

Data Source Layer

Databases, CRM systems, IoT sensors, web logs, clickstreams, third-party APIs

Data Ingestion Layer

Kafka, Flume, Sqoop, NiFi, Event Hubs - moving data reliably into the system

Data Storage Layer

HDFS, Cassandra, HBase, Cloud Data Lakes (S3, Azure Blob, GCS)

Processing Layer

Batch (MapReduce, Spark) and Streaming (Flink, Spark Streaming, Storm)

Big Data Architecture – Layered Components (cont.)

Orchestration Layer

Apache Oozie, Airflow, Control-M - managing complex workflows and dependencies

Analytics Layer

SQL engines (Hive, Presto, SparkSQL) and ML libraries (MLlib, TensorFlow)

Visualization Layer

Tableau, Power BI, Kibana, Superset - translating insights into actionable views

Security & Governance Layer

Ranger, Knox, Atlas - ensuring compliance and protecting sensitive data

Layer Implementation Examples

Company	Layer	Technologies	Scale/Impact
Netflix	Ingestion	Kafka for streaming events	500B+ events daily
Amazon	Storage	S3 for video logs	Exabytes of data
Uber	Processing	Spark Streaming	Near real-time matching
Airbnb	Orchestration	Airflow for ETL	1000s of daily workflows
Facebook	Analytics	Presto for SQL queries	Petabyte-scale analytics
Spotify	Visualization	Custom dashboards	Music streaming patterns

MapReduce can help in Distributed Computing

- Restricted key-value model
  - Same **fine-grained operation** (Map & Reduce) repeated on big data
  - Operations must be **deterministic**
  - Operations must be **idempotent/no side effects**
  - Only communication is through the shuffle
  - Operation (Map & Reduce) output saved (on disk)

MapReduce Pros

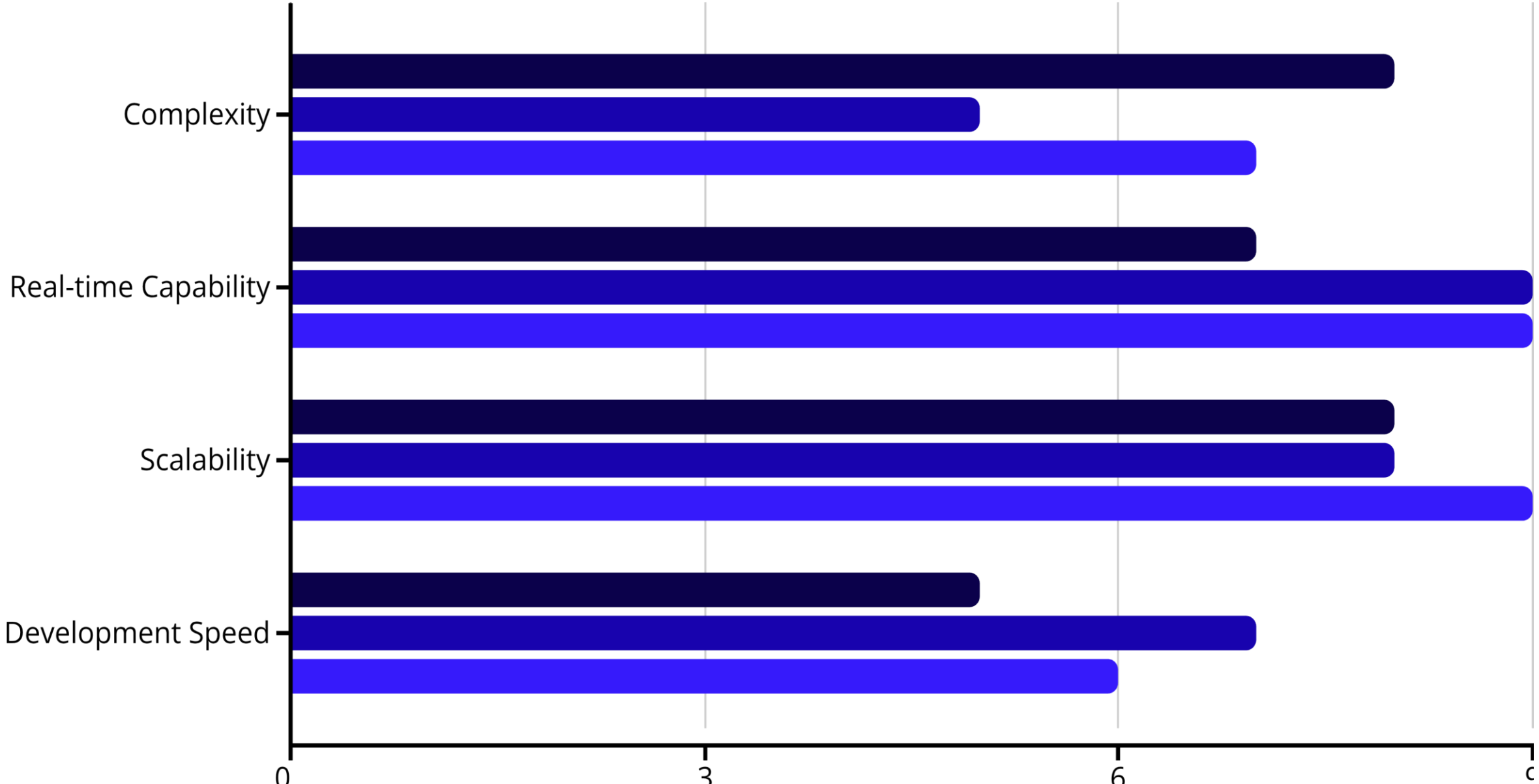
- Distribution is completely **transparent**
  - Not a single line of distributed programming (ease, correctness)
- Automatic **fault-tolerance**
  - Determinism enables running failed tasks somewhere else again
  - Saved intermediate data enables just re-running failed reducers
- Automatic **scaling**
  - As operations as side-effect free, they can be distributed to any number of machines dynamically
- Automatic **load-balancing**
  - Move tasks and speculatively execute duplicate copies of slow tasks (*stragglers)*

MapReduce Cons

- Restricted programming model
  - Not always natural to express problems in this model
  - Low-level coding necessary
  - Little support for iterative jobs (lots of disk access)
  - High-latency (batch processing)

- Addressed by follow-up research
  - **Pig** and **Hive** for high-level coding
  - **Spark** for iterative and low-latency jobs

Comparative Analysis: Choosing the Right Architecture



Architecture Feature Comparison

Feature	Lambda	Kappa	IoT/Event-Driven
Complexity	High (dual logic)	Lower (single pipeline)	Medium (device mgmt)
Batch Support	Native	Via stream replay	Sometimes
Streaming Support	Yes	Native	Yes
Recomputation	Batch layer	Replay stream	Replay events
Primary Use Cases	Social media, financial	Fraud detection, logs	Smart devices, industrial

Architecture Selection Decision Tree

Do you need both historical and real-time processing?

Yes → Consider Lambda first

No, just real-time → Consider Kappa

Is your data coming from distributed IoT devices?

Yes → IoT/Event-Driven architecture likely best

No → Continue with Lambda or Kappa evaluation

Do you have development resources for dual pipelines?

Yes → Lambda provides accuracy and performance advantages

No → Kappa offers simplicity with acceptable trade-offs

Are your batch processing windows very long (hours/days)?

Yes → Lambda separates these concerns efficiently

No → Kappa may provide sufficient performance

The Future: Serverless Big Data

Moving toward managed services and unified processing models

Emerging Trends: Serverless Big Data

Cloud-Native Solutions

AWS Lambda + Kinesis

Event-driven, serverless data pipelines with automatic scaling

Google Dataflow

Unified batch and streaming programming model with autoscaling

Azure Functions + Event Hubs

Serverless compute triggered by streaming data events

Unified Platforms

Databricks Delta Lake

ACID transactions on data lakes with unified batch/stream processing

Apache Beam

Single programming model for batch and streaming with multiple runners

Snowflake

Cloud data platform unifying storage, compute, and analytics

Key Takeaways

Architecture Selection Is Critical

The right architecture aligns with your specific data characteristics, processing needs, and technical resources

Layered Approach Provides Flexibility

Understanding the component layers allows you to mix and match technologies as requirements evolve

Convergence Is Happening

Modern platforms are increasingly unifying batch and streaming paradigms to simplify architecture

Many More things to learn……

Thank you