Architectures for Big Data

Federico Bruzzone

6 ottobre 2022

Indice

1 Course presentation

The course aims at describing **big data processing framewokds**, both in terms of **methodologies** and **techonologies**.

Part of the lesson will focus on **Apache spark** and **distributed patterns**.

"May I ask..." a brave student voice break the presentation.

It is not a spurious correlation

- What an Architecture is?
- Why so I need to know this stuff?
- What is this "Hadoop"? Do I reallt need to know what a Name Node is?
- I would like to put a jBoss inside a Docker to allow Kubernetes load balancing it! (No! This is too much even for a joke)

1.1 You are going to learn

- How to **distribute computation** over clusters using Map Reduce model
- How to write **Apache Spark** code
- How **Hadoop works** and why it works that way
- What a **software architecture** is
- How to design batch architectures to manage **data workflows**
- Several **design patterns** that could be used in a **distributed** environment
- The **limit of traditional SQL** with Big Data

1.2 Topics Overview

1. Enterprise Architectures
2. Design Patterns
3. Hadoop
4. Distributed Algorithms
5. Big Data and SQL
6. Big Data Document
7. Containers

1.3 Technologies Overview

1. Python
2. Apache Spark - Resilient Distributed Dataset
3. ELK Stack: Elastic Search, Logstash, Kibana
4. Docker

1.4 Workshops Overview

1. Workshop 1 - R. Tommasi (Marelli)
2. Workshop 2 - F. Palladino (artea.com)
3. Workshop 3 - D. Malchiodi (Unimi)
4. Workshop 4 - D. Malagodi (Google)

2 Architecture 101

Architectures:

- The art or practice of **designing** and **building** structure and especially habitable ones.
- A unifying or coherent **from** or **structure**

Foundation for the study of Software Architecture / L. Wolf, 1992

Software architecure principles can be **inherited** by appealing to several well-established architectural disciplines.

While the subject matter for the two is quite different, there are a number of intresting **architectural points** in building architecture that are suggestive for software architecture

- multimple **views**
- architectural **styles** item style and **materials** +

2.1 Multiple Views

2.1.1 Building Architecture

Building Architecture uses MULTIPLE VIEWS

A building architect works with the customer by means of a number of different views in which sone **particular aspect of the building** is emphasized.

For exmaple, there are elevations and floor plans that give the **exterior views** and "**top-down**" views, respectively.

The elevation views may be supplemented by **contextual drawings** or even scale models to provide the customer with the look of the building in its context.

2.1.2 Different Stakeholders

Each perspective is not just a matter of different level or detail.

It is linked with **different natures** and **accountability**.

- The **Owner** needs the building for a specific purpose. He/she does not know how, but hw/she knows perfectly **why**
- The **Architect** needs to project and formalize something that fit completely with owner's needs, to plan the **what**
- The **Builder** needs to design **how** the what will be built matching with natural laws and techological costraints

2.1.3 ~~Building~~ Software Architecture

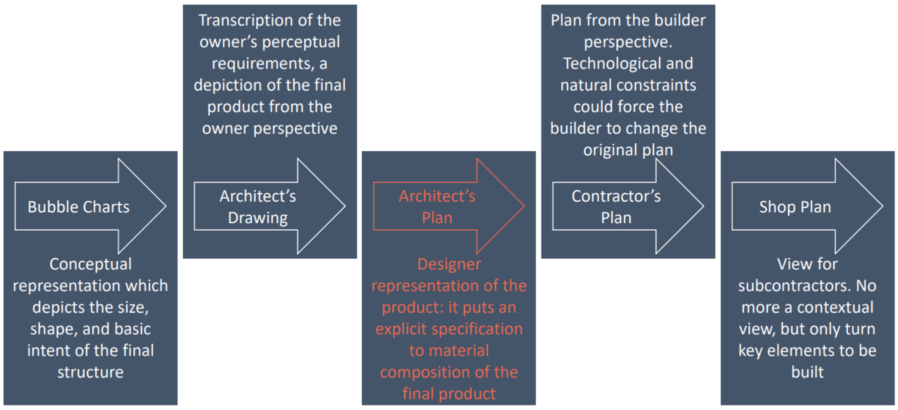
~~Building~~ Software Architecture uses MULTIPLE VIEWS

Different **type of users** will use Software Architecture: each of them will need a specific point of view.

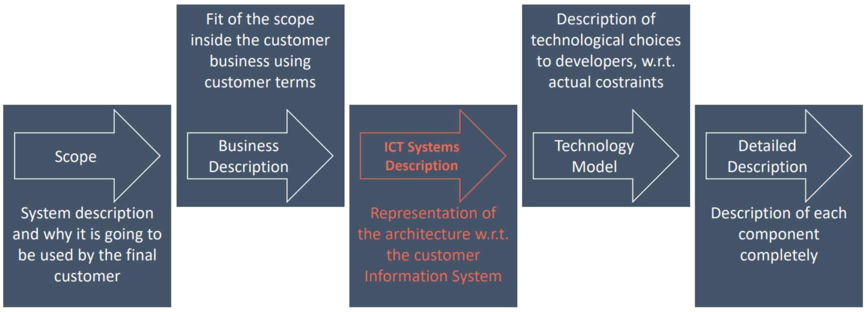
A **Full Stack** developer needs to know how to write code inside the Architecture while a **Data Scientist** where are data they need.

*Since the technology permits **destributing** large amounts of computing facilities in small packages to **remote location**, some kind of structure (or architecture) is imperative because **decentralization without structure is chaos**.*

2.1.4 Zachman Framework for Building



2.1.5 Zachman Framework for Information System



2.1.6 Different point of views

Each perspective is not just a matter of different level of detail.

It is linked with **different natures** and **accountability**.

- **Input-Process-Output**
  Product description in detail w.r.t. intended capabilities, appearance, and interactions with users
- **Entity-Relationship-Entity**
  «Stuff things is made of», description of data in each building blocks
- **Node-Line-Node**
  Flows between each component

2.2 Architectural Styles

Software Architecture A software architecture is a set of **architectural elements** that have a particular form.

[...]

The architectural form consists of weighted **properties** and **relationship**.

[...]

An underlying, but integral, part of an architecture is the rationale for the various choice made in defining an architecture.

2.2.1 Building Architecture

Building Architecture exploits different ARCHITECTURAL STYLES

Descriptively, architectural style defines a particular codification of **design elements** and formal arrangements.

Prescriptively, style limits the kinds of design elemetns and their **formal arragements**.

That is, an architectural style constrains both the **design elements** and the **formal relationship** among the degign elements.

2.2.2 ~~Building~~ Software Architecture

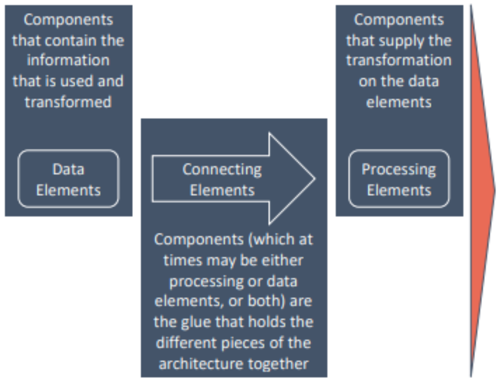
~~Building~~ Software Architecture exploits different ARCHITECTURAL STYLES

Architectural Style **encapsulates** important decision about elements and emphasizes important constraints on them and their relationships.

We can use Architecture Style both to **constrain** the architecture and to **coordinate** cooperating architects.

Moreover, style **embodies** those decision that suffer **erosion and drift**: an emphasis on it as a constraint on the architecture provides a visibility to certain aspects of the architecture so that violations of those aspects and insesnsitivity of them will be more obvious.

2.2.3 Elements



Properties are used **to constrain the choice of architectural elements**. They define the minimum desired constraints unless otherwise stated: by default on "what is not constrained by the architect may take any form desired by the designer or implementer"

Relationship are used **to constrain the "placement" of architectural element** - how the different elements may interact and how they are organized with respect to each other in the architecture

Rationale is an underlying. but integral, part of an architecture for the various choices mad in defining an acrchitexture. **It captures the motivation for the choice of architectural style, the choice of elements, and the form to satisfy the system constraints**

2.2.4 Enterprise Architecture Styles

1. ***1990 - Common Object Request Broker Architecture - COBRA*** *"Framework to allow objects hosted in different systems to make remote procedures call via a computer network using an Object Request Broker which marshals and serializes these requests"*
2. ***2003 - Service Oriented Architecture - SOA*** *"Framework for integrating business processes and supporting IT infrastructure as secure, standardized components - services - that can be reused and combined to address changing business priorities"* Bieberstein, Bose et al. 2005
   1. 2012 - Microservices
3. ***2004 - Message Oriented Architecture - MOM*** "Framework to allow objects hosted in different systems to send messages via a computer network using Message Broker to distribuite Application modules over heterogeneous platform"
4. ...

2.3 Style and Material

2.3.1 Building Architecture

Classical Architecture combines STYLE and MATERIALS

The materials have **certain properties** that are exploited in providing a particular style. One may combine structural with aesthetic uses of materials, such as that found in the post and beam construction of tudor-style houses.

However, **one does not build a skyscraper with wooden posts** and beams.

The **material aspects** of the design elements provide both aesthetic and engineering bases for an architecture.

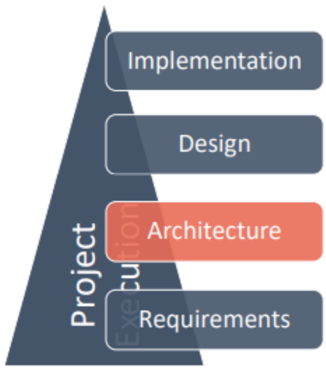
2.3.2 ~~Building~~ Software Architecture

~~Building~~ Software Architecture combines STYLE and MATERIALS

The same function can be obtained using **different subsystems**.

To train a **Neural Network Python** could be the best fit, while to put the trained Network in production using **FPGA** to physically build the network could be a better solution.

2.4 When an Architecture is designed



- Implementation: Representations of the algorithms and data types that satisfy the **design, architecture** and **requirements**
- Design: Modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the **architecture** and to satisfy the requirements.

- Architecture: Selection of **architectural elements**, thier **interactions**, and the **constraints** to provide a framework in which to satisfy the **requirements** and serve as a basis for the **design**
- Requirements: Determination of the information, processing, and the characteristics of that information and processing needed by the user of the system

There new problems involve the system-level design of software, in which the important decisions are concerned with the kinds of modules and subsystems to use and the may these modules and subsystems are organized.

This level of organization, the software architecutre level, requires new kinds of abstractions that capture essential properties of major subsystems and the ways thay interact.

2.5 Architecture as a framework for abstractions

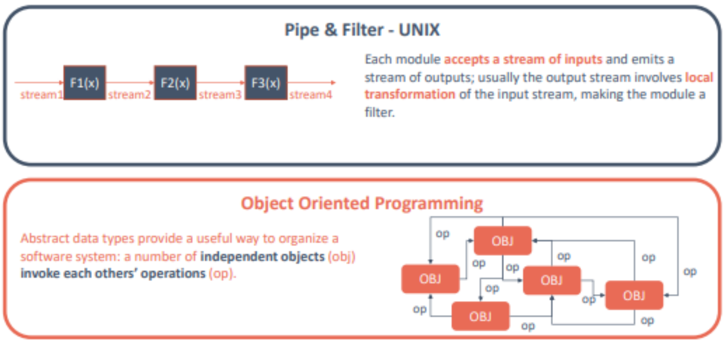
- The essence of **abstraction** is recognizing a pattern, naming and defining it, analyzing it, inding ways to specify it, and providing some way to invoke the pattern by its name without error-prone manual intervention
- This process **suppresses the detail of the pattern's implementation**, reduces the opportunity for clerical error, and simplifies understanding of the result
- In other words, good abstraction is **ignoring the right detail** at the right times

"The development of ***individual abstractions*** *often follows a common pattern:*

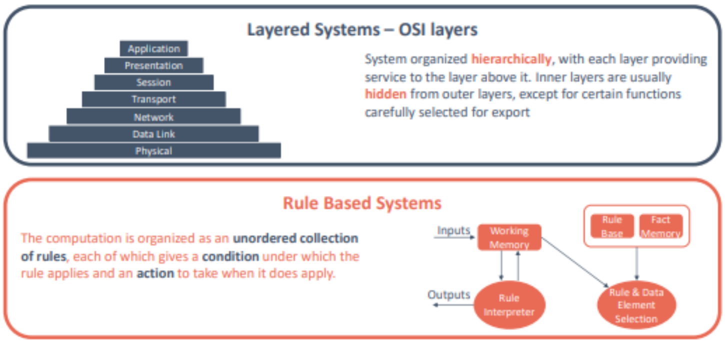
- *First, problems are* ***solved ad hoc***
- *As experience accumulates, some solutions turn out to work better than others, and* ***a sort of solklore is passed informally*** *from person to person*
- *Eventually the useful solutions are understood more systematically, and they are* ***codified*** *and analyzed*
- *This in turn enables* ***a more sphisticated level of*** *practice and allows us to tackle harder problems"*

2.5.1 Example of Abstraction

First example



Second example



2.5.2 System-subsystem Abstraction

System are constructed by combining **subsystems**:

- indipendently **compilable modules**, linked by shared data or procedure calls
- sets of **design decisions** and the **code** that implements them

  Subsystem **have an internal structure**. It is often useful to design that substructure at an architectural level before implementing it

Each subsystem may performs:

- a **specific function** to the system begin implemented
- a **more common function** such as communication or storage

Identifying and **classifying** the system functions that are common to many applications is a significant first step to the development of a software architecture.

2.6 BOC-App

2.6.1 Bully Operation Center

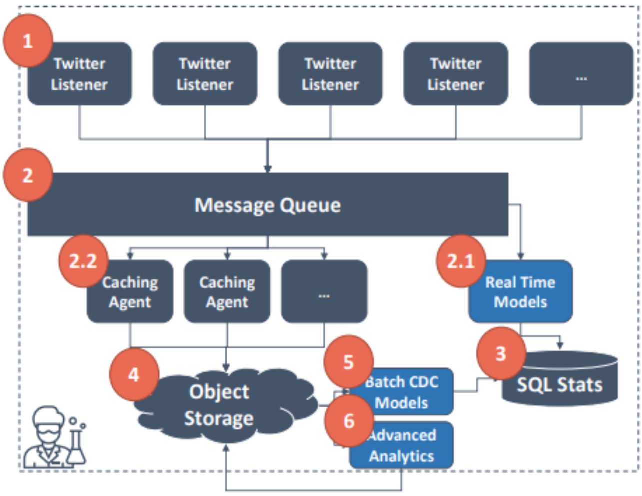
A WebApp supports a **group of Analysts** to spot **aggressive users**.

Natural Language Processing is used to **classify** each tweets.

When the **number of aggressive tweets** done by a given user go beyond a given threshold, this user is surfaced to an Analyst **who can decide to ban** it.

When a user is classified by the Analyst as bully, **the number of bullied users** iscomputed as the number of users after a bully user comment stop to use tweeter.

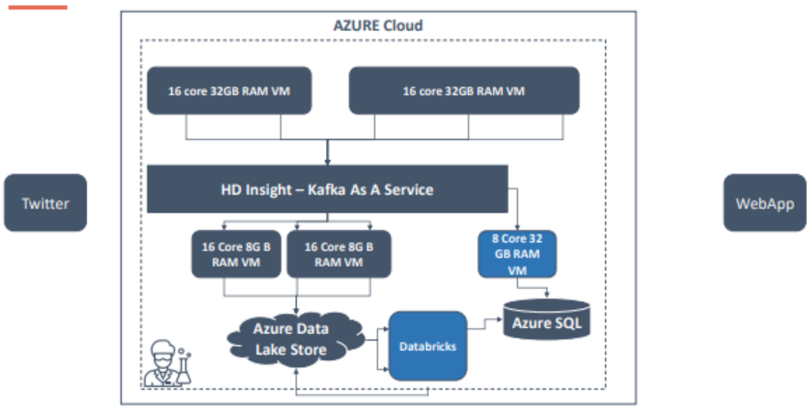
2.6.2 Problem Abstraction



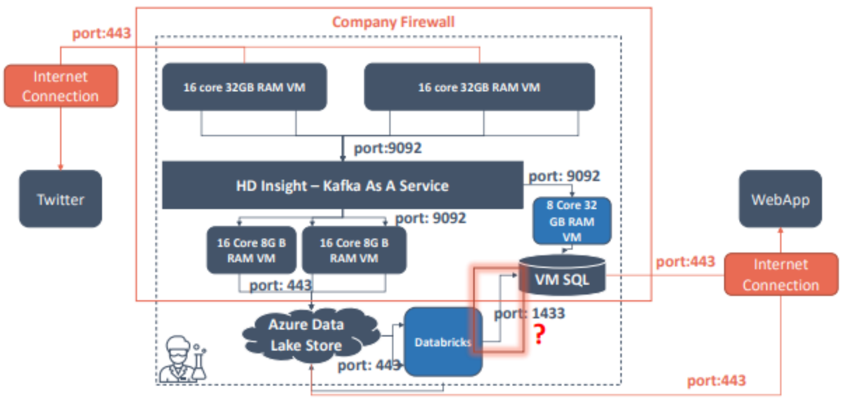
1. Different **Twitter Listeners** ensure:
   (a) **Scalability**: we can follow as many as hashtag we need
   (b) **Reliability**: we can re-distribute hashtegs to other node if a given Listener fail
2. A **Message Queue** allow serveral concurrent consumers on each hashtag
   (a) **Real Time Consumer**
   (b) **Caching Consumer**
3. Data **-as-is-** is saved using the **Object Storage** model for further analysis
4. Some models **cannot be done in real time** for several reason (e.g., training a classifier)

5. What is **I discover a given user is a bully**? I will be interested in counting a-posteriori number of users buillied by him

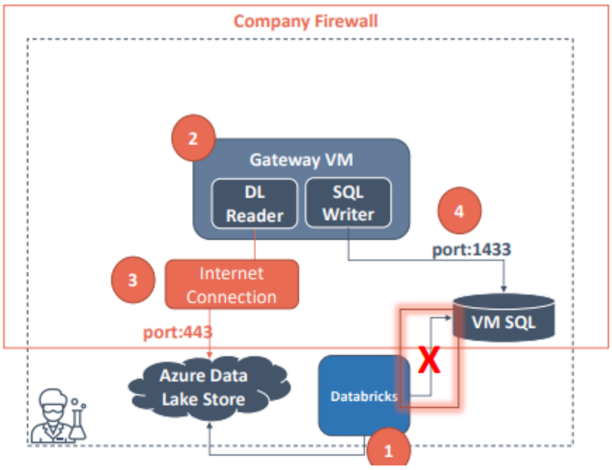
2.6.3 SubSystem Perspective



2.6.4 Network Perspective

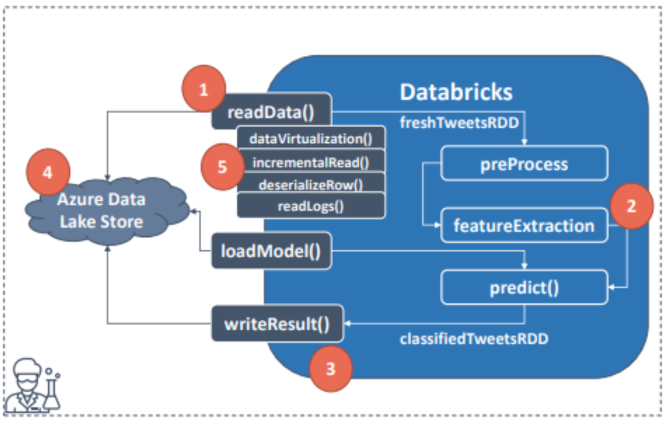


2.6.5 SQL through firewall needs care



1. Databricks writes directly on the **Data Lake**
2. Infrastructure team create a gateway Virtual Machine in the **same network** of the SQL machine
3. A **443 allow rule** for the Gateway is created and a Data Lake reader agent starts to read all fresh data
4. On the same Gateway, a second agent write down data inside the SQL VM

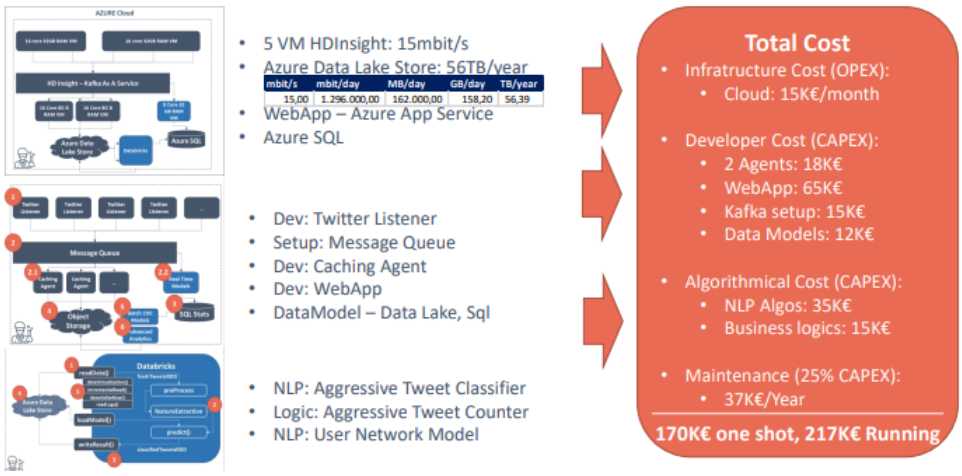
2.6.6 Data Scientist Perspective



1. Data Scientist knows using **BOCArc.readData()** will receive **e fresh tweets**

2. Then he/she can **focus** on the code needed to clean data, extract **features** and **predict**
3. At the end, he/she will call the **BOCArc.writeResult()** method which automagically writes result **somewhere**
4. He/she won't know all tweets are written on the **Data Lake**
5. He/she won't know readData() hides:
    (a) A **virtualization layer** to decouple physical data with needed one
    (b) a **CDC strategy implemented** to get only fresh tweets
    (c) Tecnhical features such as **deserialization** and **log tracing**

2.6.7 CFO Perspective



3 Architecture 102

Architectures:

- The art or practice of **designing** and **building** structure and especially habitable ones.
- A unifying or coherent **from** or **structure**

Foundation for the study of Software Architecture / L. Wolf, 1992

Software architecure principles can be **inherited** by appealing to several well-established architectural disciplines.

While the subject matter for the two is quite different, there are a number of intresting **architectural points** in building architecture that are suggestive for software architecture

- multimple **views**
- architectural **styles** item style and **materials** +

3.1 Preliminary Concept

Never take anything for granted

3.1.1 Apache Kafka and Pub/Sub

- Kafka is a **distributed system** consisting of servers and clients that communicate via a high-performance TCP network protocol.
- Kafka combines three key capabilities so tou can implement your use cases for **event streaming end-to-end**
  - To **To publish (write** and **subscribe to** (read) streams of events, including continuous import/export of your data from other system
  - To **store** streams of events durably and reliably for as long as you want
  - To **process** streams of events as they occur or retrospectively
- An **event** records the fact that "something happened" in the world or in your business [e.g., a user posts a tweet]
- **Producers** are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, **producers and consumers are fully decoupled and agnostic of each other**, which is a key design element to achieve the high scalability that Kafka is known for.
- Events are organized and durably stored in **topics**. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. Another way to see it, the **topic is like an INDEX** on a SQL table.

3.1.2 Extract Transform Load – ETL (*)

- In computing, extract, transform, load (ETL) is the general procedure of **copying data from one or more sources into a destination system**
  - Data extraction involves **extracting data from homogeneous or heterogeneous** sources
  - Data transformation processes data by **data cleaning and transforming them into a proper storage format/structure** for the purposes of querying and analysis
  - Data loading describes **the insertion of data into the final target database** such as an operational data store, a data mart, data lake or a data warehouse

- ETL can be used:
  - to increase data quality and consistency
  - to normalize data
  - to apply simple/complex logics such as id to string conversion through a lookup table
  - to prepare data for a presentation layer
- ETL can be one-shot or incremental

3.1.3 Object Storage Model

- Object storage is a **computer data storage architecture** that manages data asobjects. It is opposed to other storage architectures like file systems or block storage
- Each object typically contains **data, contextual information** (metadata), and **technical information** (header)
- It is based on a **shared naming convention**, which generates a unique id for each object, and a shared **serialization strategy** (e.g., json)
- It allows to **distribute data** over several nodes
- Object storage was created to allow **retention** of massive amounts of unstructured data

3.1.4 Data Lake

- Models you can build using data cannot **be known a priori**: if some pieces ofinformation are not saved when produced (e.g., under sampling a sensor), they **could not be re-acquired later**
- **Legacy Systems** integration is pretty complex and expensive: once a legacy system is integrated, there is no reason not to get all available information
- Other Data Architectures have problems dealing with **heterogeneous data** or data which format and content can **change over time**
- Data Lakes is based on **four pillars**:
  - Unprocessed data (only serialized in objects)
  - Data saved forever
  - Good reading/writing performances
  - Schema available on read

3.1.5 Concrete and Abstract Classes

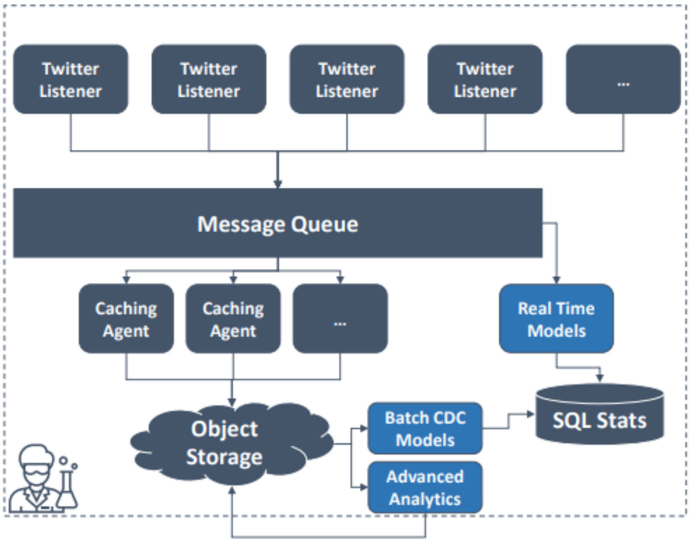
- Concrete Class
  - A concrete class is a class that can be instantiated
- Abstract Class
  - Cannot be instatied
  - Contains Abstract Methods
  - Can contain concrete methods
  - Specifies virual methods via signatures that are to be implemented
  - Before a class derived from an abstract class can be instantiated, all abstract methods of its parent classes must be implemented

3.1.6 Lock-in

- Vendor lock-in makes **a customer dependent on a vendor** for products and services
- A supplier **successfully locks in** a customer when:
  - **the cost of changing** supplier is higher that the cost of keeping it
  - without that cost, other **suppliers can outperform** the actual supplier
- An Enterprise Architecture can protect the company from Vendor lock-in
- Another kind of lock-in is the so called **knowledge lock-in**: this kind of lock-in happens when the **cost of knowledge transfer** is higher than the benefit to dismiss a person/team. Again, a Software Architecture can **mitigate this risk**

3.2 Software Architecture Pillars

3.2.1 8 reasons why



Bully Operation Center

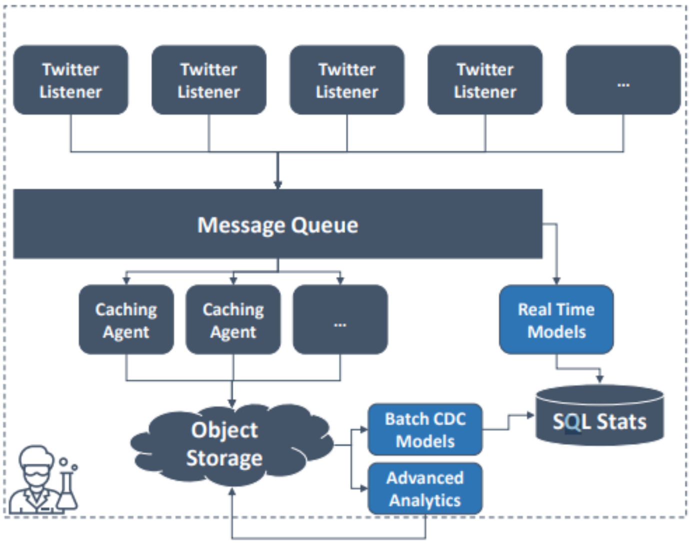
- A WebApp supports **a group of Analysts** to spot **aggressive users**
- Natural Language Processing is used to **classify** each tweets
- When the **number of aggressive tweets** done by a given user go beyond a given threshold, this user is surfaced to an Analyst **who can decide to ban** it
- When a user is classified by the Analyst as bully, **the number of mbullied users** is computed as the number of users after a bully user comment stop to use tweeter

3.2.2 Software Architecture Pillars

1. Being the framework for satisfying requirements
2. Being the technical basis for design
3. Being the managerial basis for cost estimation and process management
4. Enabling component reuse
5. Focus on centralization
6. Enhancing productivity and security

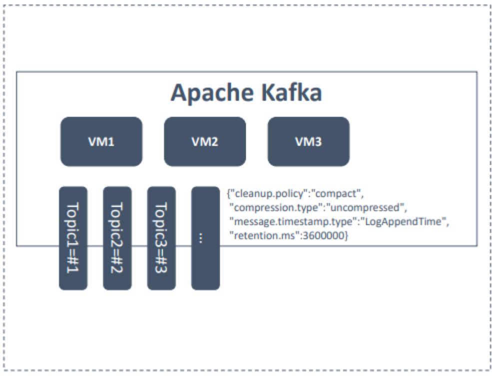
7. Enabling enterprise systems integration (Enterprise Application Integration)
8. Allowing a tidy scalability
9. Controlling software processes execution
10. Avoiding handover and people lock-in

3.2.3 Being the framework for satisfying requirements



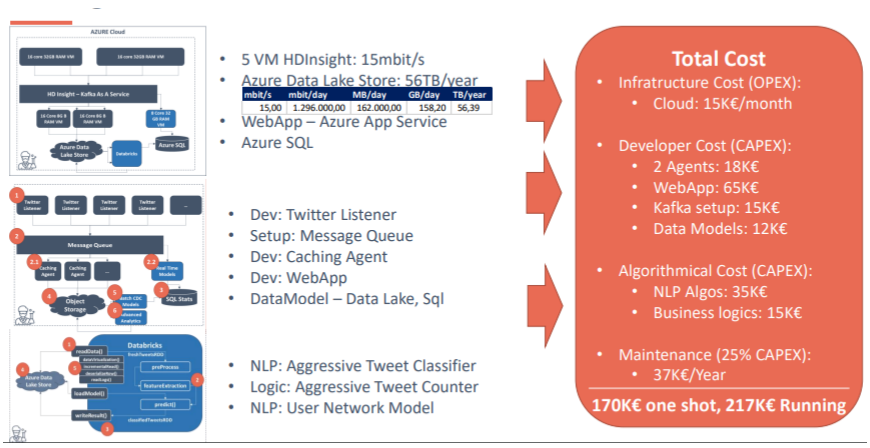
- Functional Requirement
  - Am I able to spot aggressive users?
  - Will the Analyst have all the needed information to decide to ban a user?
- Technical Requirement
  - Am I able to process all the tweets in time?
  - Am I able to check in the history bullied users?
- Security Requirement
  - Is it compliance with GDPR (data privacy rules)?

3.2.4 Being the technical basis for design

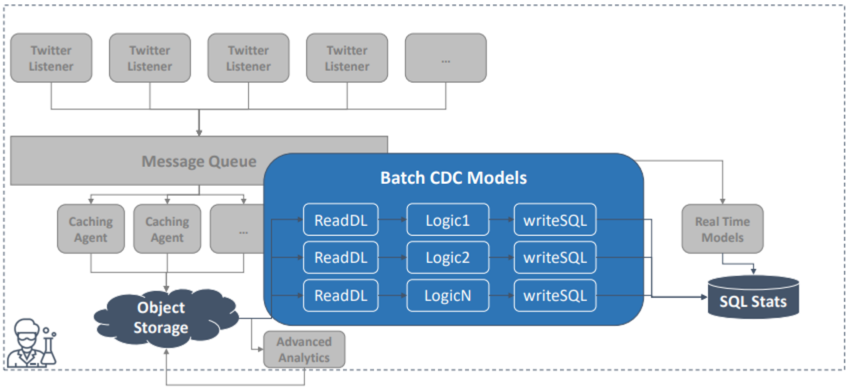


Modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements

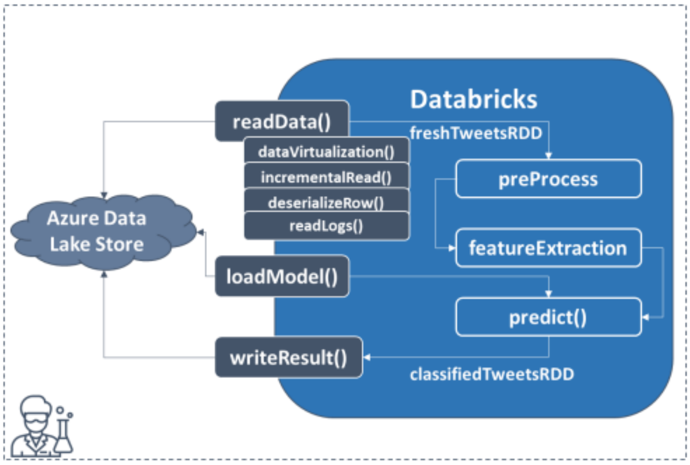
3.2.5 Being the managerial basis for cost estimation and process management



3.2.6 Enabling component reuse

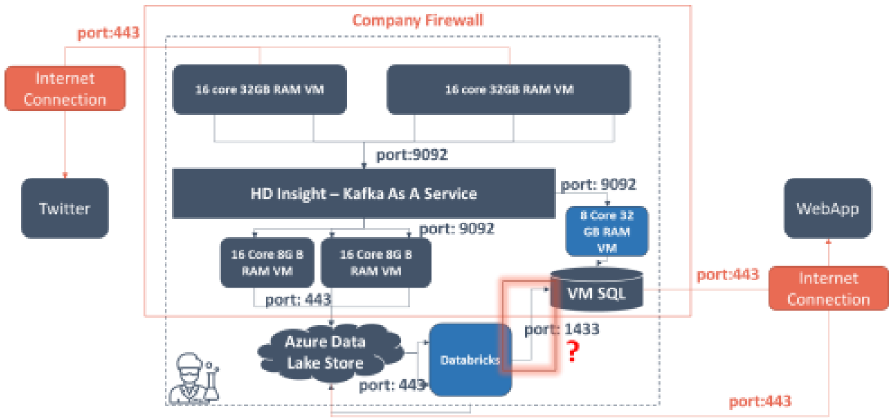


3.2.7 Focus on centralization



- Azure Data Lake Store in our Architecture is the Single Source of Truth
- Data are Centralized there

3.2.8 Enhancing productivity and security



- We know exactly which ports we need to open
- We can build a gateway to access SQL data from WebApp to avoid SQL Injection (access is centralized there)
- We can build the component to hide Data Lake and enforce ACL Rules

3.3 Design patterns

are formalized best practices found to solve common problems

3.3.1 Hundreds of Design Pattern

V · T · E Software design patterns [hide]	
Creational	Abstract factory · Builder · Dependency injection · Factory method · Lazy initialization · Multiton · Object pool · Prototype · RAII · Singleton
Structural	Adapter · Bridge · Composite · Decorator · Delegation · Facade · Flyweight · Front controller · Marker interface · Module · Proxy · Twin
Behavioral	Blackboard · Chain of responsibility · Command · Interpreter · Iterator · Mediator · Memento · Null object · Observer · Servant · Specification · State · Strategy · Template method · Visitor
Functional	Closure · Currying · Function composition · Functor · Monad · Generator
Concurrency	Active object · Actor · Balking · Barrier · Binding properties · Coroutine · Compute kernel · Double-checked locking · Event-based asynchronous · Fiber · Futex · Futures and promises · Guarded suspension · Immutable object · Join · Lock · Messaging · Monitor · Nuclear · Proactor · Reactor · Read write lock · Scheduler · Thread pool · Thread-local storage
Architectural	ADR · Active record · Broker · Client–server · CBD · DAO · DTO · DDD · ECB · ECS · EDA · Front controller · Identity map · Interceptor · Implicit invocation · Inversion of control · Model 2 · MOM · Microservices · MVA · MVC · MVP · MVVM · Monolithic · Multitier · Naked objects · ORB · P2P · Publish–subscribe · PAC · REST · SOA · Service locator · SN · SBA · Specification
Cloud Distributed	Ambassador · Anti-Corruption Layer · Bulkhead · Cache-Aside · Circuit Breaker · CQRS · Compensating Transaction · Competing Consumers · Compute Resource Consolidation · Event Sourcing · External Configuration Store · Federated Identity · Gatekeeper · Index Table · Leader Election · MapReduce · Materialized View · Pipes · Filters · Priority Queue · Publisher-Subscriber · Queue-Based Load Leveling · Retry · Scheduler Agent Supervisor · Sharding · Sidecar · Strangler · Throttling · Valet Key
Other	Business delegate · Composite entity · Intercepting filter · Lazy loading · Mangler · Mock object · Type tunnel · Method chaining
Books	*Design Patterns* · *Enterprise Integration Patterns*
People	Christopher Alexander · Erich Gamma · Ralph Johnson · John Vlissides · Grady Booch · Kent Beck · Ward Cunningham · Martin Fowler · Robert Martin · Jim Coplien · Douglas Schmidt · Linda Rising
Communities	The Hillside Group · The Portland Pattern Repository

This computer science article is a stub. You can help Wikipedia by expanding it.

3.3.2 The 23 Classical Patterns by Type

Creational

Creating an object rather than instantiate it directly

- **Abstract factory** groups object factories that have a common theme.
- **Builder** constructs complex objects by separating construction and representation.
- **Factory** method creates objects without specifying the exact class to create.
- **Prototype** creates objects by cloning an existing object.
- **Singleton** restricts object creation for a class to only one instance.

Structural

Using inheritance to compose new objects

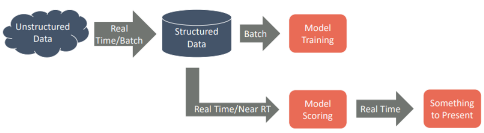
- **Adapter** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
- **Composite** composes zero-or-more similar objects so that they can be manipulated as one object.
- **Decorator** dynamically adds/overrides behavior in an existing method of an object.
- **Facade** provides a simplified interface to a large body of code.
- **Flyweight** reduces the cost of creating and manipulating a large number of similar objects.
- **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Behavioral

Defining how objects can communicate

- **Chain of responsibility** delegates commands to a chain of processing objects.
- **Command** creates objects which encapsulate actions and parameters.
- **Interpreter** implements a specialized language.
- **Iterator** accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Memento** provides the ability to restore an object to its previous state (undo).
- **Observer** is a publish/subscribe pattern which allows a number of observer objects to see an event.
- **State** allows an object to alter its behavior when its internal state changes.
- **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime.
- **Template** method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

3.3.3 A classic Big Data challenge



3.3.4 From ETL to ELT