Lambda+, the Renewal of the Lambda Architecture: Category Theory to the Rescue

Annabelle Gillet, Éric Leclercq, Nadine Cullot

Lambda+, the renewal of the Lambda Architecture: Category Theory to the rescue

Annabelle Gillet[1], Éric Leclercq[1], and Nadine Cullot[1]

Laboratoire d'Informatique de Bourgogne - EA 7534
Univ. Bourgogne Franche-Comté
annabelle.gillet@depinfo.u-bourgogne.fr
eric.leclercq@u-bourgogne.fr & nadine.cullot@u-bourgogne.fr

Abstract. Designing software architectures for Big Data is a complex task that has to take into consideration multiple parameters, such as the expected functionalities, the properties that are untradeable, or the suitable technologies. Patterns are abstractions that guide the design of architectures to reach the requirements. One of the famous patterns is the Lambda Architecture, which proposes real-time computations with correctness and fault-tolerance guarantees. But the Lambda has also been highly criticized, mostly because of its complexity and because the real-time and correctness properties are each effective in a different layer but not in the overall architecture. Furthermore, its use cases are limited, whereas Big Data need an adaptive and flexible environment to fully reveal the value of data. Nevertheless, it proposes some interesting mechanisms. We present a renewal of the Lambda Architecture: the Lambda+ Architecture, supporting both exploratory and real-time analyzes on data. We propose to study the conservation of properties in composition of components in an architecture using the category theory. We relate a real implementation of our approach to architecture a social network observatory platform.

Keywords: Architecture pattern, Category theory, Lambda Architecture

1 Introduction and motivations

All information systems have a common point: they need an architectural design before being developed and deployed. The architecture must guarantee some properties and guide the consistency of the overall structure of the information system. In this context, architectural styles and patterns are used to build a system having the expected characteristics for each of its part as well as for its entirety, and to state the requirements of the technologies and programming techniques needed to achieve the goal sought. Thus, global requirements such as scalability, performance, reliability must be clearly identified to select the style of architecture, the different components and the interactions among them [23], and then choose technologies with properties (such as ACID for databases or micro

batch capabilities for stream processing) that fit all of the previous choices. The absence of coherence in a definition of an architecture can lead to the dreaded Big Ball of Mud [15], that reduces greatly the maintenance and evolutivity capabilities of the system. To help avoiding this situation, there are two major elements among architectural design artifacts: styles and patterns.

Styles are coarse grained specifications of the organisation of the architecture, that guide the interactions among components [1]. Each style brings naturally some architectural characteristics, while also imposing trade-offs on others. So, there is not a better style than the others, but solely situations where a style would be more suited to fulfill the expected characteristics. Some examples of architecture styles are the layered architecture [28], the microservices architecture [32] and the event-driven architecture [9].

An architecture pattern is a specific abstraction of a fixed architecture style for a particular set of essential characteristics [17]. It helps to identify within a style which combination of components will be more suited for a given context, but still provides enough freedom to adapt the implementation of the pattern to specificities of each situation. The level of detail can vary, as well as the restrictions of the application of the pattern. The Blackboard pattern [11], the Model-View-Controller [12] and the Lambda Architecture [29] are examples of architecture patterns. The Lambda Architecture is well-known in a Big Data context. It was introduced at the beginning of stream processing systems, and thus is oriented to compensate for the flaws of an emerging technology rather that taking advantage of the capabilities of such a technology.

Recent researches in software architecture try to formally define styles and patterns, to anticipate effects of the composition of components, and thus knowing beforehand the result of the evolution of a part of the architecture [6, 20]. When architectures evolve and grow, they can combine several smaller parts of architectures developed separately. When building a large scale, complex and distributed architecture, its parts can embed architecture styles on their own. These different cases can result in compositions of smaller architecture parts with their proper styles and patterns, so formalization should be able to express and control these compositions. Category theory [13] is a promising approach for formalization, due to its ease to represent compositions as it considers morphims and functors as first class citizens, and to its already existing proximity to the engineering software world, particularly with functional programming. Moreover, its graphical representation is a visual help to understand the formalization, and leads to a better comprehension of the system [35].

In this article we propose the Lambda+ Architecture pattern, an update of the Lambda Architecture, and a formalization to study the conservation of properties in compositions of components using the category theory. The rest of the article is organized as follows: section 2 describes the original Lambda Architecture pattern, as well as its uses and its flaws ; Section 3 uses the category theory to prove the loss of properties in the Lambda Architecture ; Section 4 shows our improved Lambda+ Architecture pattern ; Section 5 describes a real applied example of our pattern and ; Section 6 concludes this article.

2 Lambda Architecture and related work

The properties of correctness, low latency and fault-tolerance have always been a major concern when designing architectures. In [23], Lampson sketches some suggestions that are still relevant today, and that can be found, among others, in the Lambda Architecture, introduced by Marz in 2011 [29, 30]. The objective is simple: to compute predetermined queries with a very low latency and to ensure the correctness of the processing (figure 1). To do so, the Lambda is composed of three layers: the batch layer, the serving layer, and the speed layer.

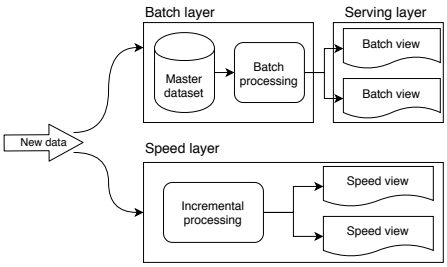

Fig. 1: Overview of the Lambda Architecture

The **batch layer** takes care of storing raw data in the master dataset and of executing the computations on the batch of data. The correctness of the results is the main concern of this layer. With the master dataset, it is possible to recompute the whole set of data if the batch processing appears to be erroneous, or if the needs have evolved. As the batch processing is only computed once in a while, the **speed layer** has to compensate this downside by a low latency capability. The computations are the same as in the batch layer, but the processing has to be incremental. However, this layer lacks the correctness property. The batch processing puts to disposal its results in the **serving layer**, to facilitate their access by users. The serving layer has often a role of indexing and presenting data, to enable a fast access to the views created by the batch layer. To keep results up-to-date, the serving also appends the speed views to the batch views.

With these specifications, the advantages of the Lambda Architecture are a strong fault-tolerance for machine and human faults, a guarantee of a correct result with the batch layer and a low latency with the speed layer. It has inspired many Big Data architectures, such as RADStack [38], which is an open-source plateform used to produce interactive analyzes. They developed Druid to use it as the serving layer, that supports real-time and batch data ingestion. It is also used to allow guided exploratory analyzes concerning pre-determined insights. In [31], the authors apply the Lambda Architecture to process data of smart grids. They add a querying and an analytics layers, in order to propose more flexibility in the querying capabilities that overcomes the fixed precomputed views of the Lambda. The system developed in [25] uses the Lambda

Architecture to develop a recommendation system for restaurants. It alters the original pattern of the Lambda, by having different computations in the batch and in the speed layers. The batch layer is responsible for executing heavy machine learning algorithms, while the speed layer exploits those results to propose recommendations to users. LinkedIn develops Pinot [18], a real-time distributed datastore designed for processing OLAP queries with low latency. They use the Lambda Architecture pattern to provide LinkedIn users with near real-time analytics functionalities, such as who viewed their profile. They compute the speed views in real-time, and when the views are complete (i.e., when they represent a day or a hour depending on the granularity) they become the batch views. The incoming data are then computed in a new speed view.

However, the Lambda has also been criticized a lot, due to its complexity to maintain and to evolve both the speed and the batch layers, that have to perform the same computation, but with different paradigms. It also lacks in flexibility, as its goal is to answer only predetermined queries. Thus, as saw in the examples of implementation of the Lambda Architecture, alternative use cases such as exploratory analyzes require to modify the pattern. Furthermore, by delegating the correctness property only to the batch layer and not to the speed layer, the low latency and the correctness properties cannot be obtained simultaneously. To clarify this statement, the Lambda has to be replaced in the context of its creation. At this time, streaming systems were only at their early stages, and thus did not have all the capabilities that they have today.

2.1 The evolution of stream processing systems

Streaming systems [4] had begun to emerge approximately at the same time as the conception of the Lambda Architecture. Marz had proposed Apache Storm in 2011 [36], the first stream processing system to encounter success in the industry field. It was born from the ascertainment that to produce a system with a real-time component, it took more time to create workers and queues and to ensure that their interactions are as expected than to develop domain logic. In 2015, Twitter had proposed Apache Heron [22]. They used previously Apache Storm, but needed a system more suited to their needs, with better performance, scaling capabilities and easier to manage. Spark Streaming [39] in 2012 took a different approach and proposes a micro-batch streaming system, relying on the Spark engine dedicated to the batch processing. MillWheel [3] and Apache Flink [8] also joined the world of streaming systems during those years.

The efficiency of streaming systems, on top of the low latency, comes with the guarantee of the processing [4]. There are three main levels of guarantee: 1) *at-most-once*, where elements are never processed more than once, but can be never processed, 2) *at-least-once*, where elements are never processed less than once, but can be processed multiple times, and 3) *exactly-once*, that surpasses the at-most-once and at-least-once guarantees, and thus each element is processed once and only once, allowing to compute correct results. However, the exactly-once guarantee is utopian, and in practice it is closer to effectively-once, where elements can be processed several times, but the effect on the state of the stream

is only counted once. This adds a strong constraint: the processing must not have side effects that are not idempotent. It means that in case of the reprocessing of a message, the global result must not be altered. When the Lambda Architecture came out, stream processing systems had mostly only the at-most-once or at-least-once guarantee, and could not offer more. So, the Lambda can be seen as a mean to compensate flaws of an emerging technology, rather than a pattern that fully exploits it.

2.2 Toward the end of the Lambda Architecture

The Kappa Architecture [21] proposes to get rid of the complexity, by keeping only the speed layer, arguing that it is enough to reach the goal of the Lambda. While this is a correct statement regarding the evolution of stream processing systems, it also discards the master dataset and the fault-tolerance property, one of the strengths of the Lambda. Another flaw in the Lambda Architecture is that it is loosely defined. Several interpretations of the serving layer can be found, that include or not the speed views. The aggregation of the speed and the batch views is not clearly defined, especially for unordered streams of data. The lack of precise definition extends to the style of the architecture. Each part is called layer, whereas layered architectures are a stacking of multiple components, where each component can interact with the components directly above or below it [33]. The event-driven architecture would be a more suited style. This leads to a need of a stricter definition of the architecture, as well as an adaptation of the role of its components, updated following the gain in maturity of the stream processing systems.

3 Using the category theory to study conservation of properties

In the research field of software engineering for architecture design, the need for proper theory and formalization has raised importance in the last decade [6, 20]. Designing, specifying and implementing software architectures are complex tasks, that require careful specifications to link and preserve characteristics through all the steps of creation. The development of theory in this field requests both practical and theoretical skills, in order to propose a model suited to the expectations, that takes into consideration the imperfections of the real-world of engineering.

ADLs [10] (Architectural Description Language) have an important role to formalize architectures. Boxes and lines ADL as well as ADL based on UML [7] cannot easily verify properties, due to their weak formalization. We focus on those ADL having well-established theoretical foundations. In [1], Abowd et al. provide a formal framework in Z to achieve a description of architectural styles. They argue that diagrams are not sufficient to impose only one meaning to represent an architecture, and that they can lead to misunderstanding. Malkis and Marmsoler in [27, 28] work on a formalization of architecture styles. They use the theory of sets and first order logic to build a model with ports and services

to represent interactions among components. They apply their proposal on two styles: the layered and the service-oriented architecture. In [24], Le Métayer uses the graph theory to propose a formalization of architectures. Nodes are entities of an architecture (client, server, or object entity depending on the level of abstraction), and links are communications between those entities. Mabrok in [26] tries a different approach, and uses the category theory to formalize the requirements and the attributes of an architecture. Ologs, a particular application of category theory thought to represent the study of the ontology of a subject, is used to organize the architecture.

Existing ADLs are based on set theory, graph theory and use first or high order logic to check properties or consistency of architectures. However, they do not study the conservation of those properties in compositions of components. To fill this need, category theory [13] is a promising approach: it allows to switch from a model to another or to navigate among abstraction levels, and thus to express various problems from different science fields, such as mathematics, physics or computer science [35]. By focusing on relations (the morphisms) and compositions, it proposes powerful mechanisms that can be applied to architectures. In this paper, **we focus on studying the conservation or the discarding of properties in compositions of components, by relying on the behaviour of functors combined to preorders**. This section only introduces some notions of category theory useful to understand this formalization, and cannot relate all the subtleties and the depth of it. We refer the reader to [35] for a more complete explanation of the category theory, and to the supplementary material available[1].

A **category** C is composed of four basic elements: 1) $\mathrm{Ob}(C)$, a collection of objects ; 2) for each pair $x, y \in \mathrm{Ob}(C)$, a set $\mathrm{Hom}_C(x, y)$ representing **morphisms** from x to y, namely a mean to get an object y (the codomain) from an object x (the domain). A morphism f from x to y is noted $f : x \to y$; 3) for each $x \in \mathrm{Ob}(C)$, a particular morphism id_x known as the identity morphism on x ; 4) for each triplet $x, y, z \in \mathrm{Ob}(C)$, a **composition** $\circ : \mathrm{Hom}_C(y, z) \times \mathrm{Hom}_C(x, y) \to \mathrm{Hom}_C(x, z)$. For two morphisms $f : x \to y$ and $g : y \to z$, the composition is noted $g \circ f : x \to z$. And of two laws: 1) for a morphism $f : x \to y$ with $x, y \in \mathrm{Ob}(C)$, we have $f \circ \mathrm{id}_x = f$ and $\mathrm{id}_y \circ f = f$; 2) for $f : w \to x$, $g : x \to y$ and $h : y \to z$ with $w, x, y, z \in \mathrm{Ob}(C)$, we have $(h \circ g) \circ f = h \circ (g \circ f) \in \mathrm{Hom}_C(w, z)$.

A **product** of two categories $C1$ and $C2$ produces a new category which objects are all the possible pairs (x, y) with $x \in \mathrm{Ob}(C1)$ and $y \in \mathrm{Ob}(C2)$ and morphisms $(x, y) \to (x', y')$ are pairs (f, g) where $f : x \to x' \in \mathrm{Hom}_{C1}(x, x')$ and $g : y \to y' \in \mathrm{Hom}_{C2}(y, y')$.

Two particular cases of categories are of special interest to formalize architectures. The **preorders**, in which between each pair $x, y \in \mathrm{Ob}(C)$, there exists a unique morphism $f : x \to y$. If there exist $f : x \to y$ and $g : x \to y$, then $f = g$. The **power sets**, that are sets which contain all the subsets of a given set. In a category, power sets can be organized as preorder, where morphisms link two subsets if the first subset is integrally included in the second.

[1] https://github.com/AnnabelleGillet/CategoryTheoryForArchitectures

To formalize architectures, we define three core categories: 1) the *Components*-category, in which objects are all the components of the architecture, without any morphisms ; 2) the *Architecture*-category which contains all the components and with morphisms representing the interactions between components, $f : x \to y$ means that the component x sends data to the component y ; 3) the *ComponentsPS*-category containing all the objects of the power set of the components, that will be used to connect the components to properties. To link those categories, we use functors.

A **functor** F maps a category C to a category C'. It is noted $F : C \to C'$, and affects both objects ($F : \mathrm{Ob}(C) \to \mathrm{Ob}(C')$) and morphisms (for each pair $x, y \in \mathrm{Ob}(C)$, we have $F : \mathrm{Hom}_C(x, y) \to \mathrm{Hom}_{C'}(F(x), F(y))$). To be valid, a functor must observe two laws: 1) the preservation of identities: $\forall x \in \mathrm{Ob}(C)$, $F(\mathrm{id}_x) = \mathrm{id}_{F(x)}$; 2) the preservation of composition: for any triplet $x, y, z \in \mathrm{Ob}(C)$ with morphisms $g : x \to y$, $h : y \to z$, we have $F(h \circ g) = F(h) \circ F(g)$.

To link the categories we have previously defined, we use functors: 1) $CA : Components \to Architecture$ to integrate components in the architecture ; and 2) $CCPS : Components \to ComponentsPS$ to study the behaviour of components in a set of different components.

Properties are represented with preorders, and each value of a property is an object. Morphisms go from the most satisfying value of the property to the least satisfying value. The symbol $\top$ is used as the top value to neutralize a component when it is not concerned by the property. Properties can be simple (only with *true* and *false* objects), multivalued, or more complex, and resulting of the composition of several other properties: in this case, properties are associated with a product of categories, and this product is linked to the next property with a functor that maps each combination to its signification in the next level property. The category *ComponentsPS* is connected to every property of level one (those than are not the result of a product of categories).

This formalization is applied on the Lambda Architecture, to formally prove its weaknesses. A composition exists with the batch, the serving and the speed layers because the serving layer merges the batch and the speed views to provide users with results. Using the category theory, we can extract some high-level knowledge from the known facts, given below (figure 2). The morphisms inside categories *ComponentsPS* and *Correctness* are defined as follows (with the notation $B =$ Batch, $Se =$ Serving and $Sp =$ Speed):

CPS (*ComponentsPS*)	morphisms with $\emptyset$ as domain are omitted
	$B - BSe : B \to BSe$
	$Se - BSe : Se \to BSe$
	$BSe - BSeSp : BSe \to BSeSp$
	$Sp - BSeSp : Sp \to BSeSp$

C (*Correctness*)
$\top - t : \top \to True$
$t - f : True \to False$
$id_\top : \top \to \top$
$id_t : True \to True$
$id_f : False \to False$

The morphisms inside the category $Real - time$ are the same as those of the *Correctness*-category. The effect on the objects of functors that link the *ComponentsPS*-category to the categories of the properties are given below:

$CPS - C$ (*Correctness*)
$B \to True$
$Se \to \top$
$Sp \to False$

$CPS - RT$ ($Real - time$)
$B \to False$
$Se \to \top$
$Sp \to True$

From these given facts, we want to deduce the value taken by the first composition of components $\{Batch, Serving\}$ for the correctness property. For this, we have to resolve the effect of the functor $CPS - C$ on the morphisms:

$$CPS - C : \underline{\mathrm{Hom}_{CPS}(B, BSe)} \rightarrow \mathrm{Hom}_C(\underline{CPS - C(B)}, CPS - C(BSe))$$
$$CPS - C : \underline{\mathrm{Hom}_{CPS}(Se, BSe)} \rightarrow \mathrm{Hom}_C(\underline{CPS - C(Se)}, CPS - C(BSe))$$

where the underlined elements are known. To establish the value of $CPS - C(BSe)$, we have to find two morphisms in the category C that would have the same codomain: one with the domain $True$ ($CPS - C(B)$), the other with the domain $\top$ ($CPS - C(Se)$). Only the pair $(\top - t, id_t)$ satisfies the requirements. As the codomain is $True$, it allows us to deduce that $CPS - C : BSe \rightarrow True$. Thus, the composition $\{Batch, Serving\}$ yields $True$ for the correctness property.

We use the same mechanism to deduce the value taken by the overall architecture for the same property:

$$CPS - C : \underline{\mathrm{Hom}_{CPS}(Sp, BSeSp)} \rightarrow \mathrm{Hom}_C(\underline{CPS - C(Sp)}, CPS - C(BSeSp))$$
$$CPS - C : \underline{\mathrm{Hom}_{CPS}(BSe, BSeSp)} \rightarrow \mathrm{Hom}_C(\underline{CPS - C(BSe)}, CPS - C(BSeSp))$$

This time, the pair of morphisms that meets the requirements is $(t - f, id_f)$. As the common codomain is $False$, it allows us to deduce that the overall architecture yields $False$ for the correctness property.

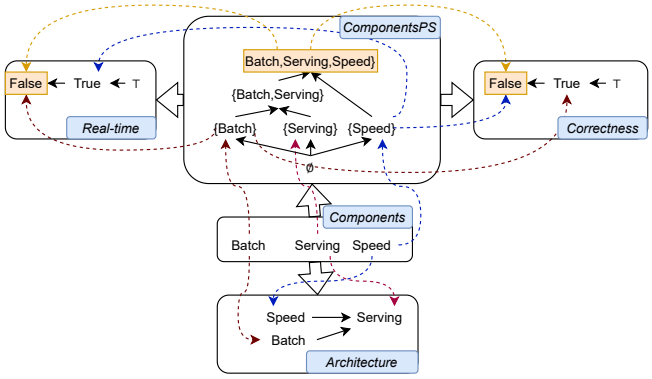

Fig. 2: Study of the preservation of properties in the Lambda Architecture[2]

The same reasoning can be applied for the real-time property, that yields $False$ for the overall architecture. With the category theory, we proved that the real-time and correctness properties are effective in a different layer, but that they do not hold in the whole Lambda Architecture. We can conclude with a fact about compositions of components in architectures: **if an individual component does not support a property, it will cause its loss in a composition of components**.

[2] To simplify the schema, the $ComponentsPS$-category has only individual components, as well as possible compositions of the architecture, and only the links from components to properties that does not lead to $\top$ are represented.

4 The Lambda+ Architecture pattern

To improve the Lambda Architecture, the correctness property should hold for all the components. Furthermore, the fault-tolerance should be kept, but as the reprocessing of data in a batch fashion is often incompatible with the real-time property, it should be integrated as an alternative running composition, activated only in case of a technical failure or to satisfy new needs. Use cases should also gain in flexibility, and the complexity induced by the development of the same process with different paradigms in the speed and in the batch layers should be avoided.

The Lambda+ Architecture is meant to be a renewal of the Lambda Architecture, by improving the support of the correctness property and by leveraging two main functionalities: 1) storing data for allowing flexible and exploratory data analyzes, and 2) computing in real-time predefined queries on data streams in order to have insights on well-known and identified needs. The duality between exploratory analyzes and predefined queries is of primary importance in a Big Data context, where the combination of volume and variety of data overcomes the capability of finding all the insights hidden in data. The fault-tolerance mechanism of the Lambda is kept, but is only activated when needed.

The adopted layer model of the Lambda Architecture, as stated in section 2, does not match the style of the architecture. Instead, the Lambda+ is composed of a set of components interacting together asynchronously with messages. This pattern borrows its principles from the Event-Driven Architecture style, which is well-suited for achieving performance, scalability and evolutivity. The trade-offs of this architecture style is a lack of simplicity and the difficulty of testing the whole architecture, due to the dynamic nature of the messaging workflow and the chaining of various processing components. Figure 3 shows an overview of the Lambda+ Architecture which includes five main components.

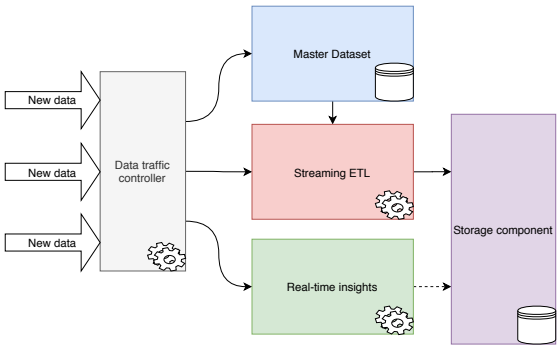

Fig. 3: Overview of the Lambda+ Architecture

The **data traffic controller component** is the entry point of the Lambda+ pattern. Data sources can have very different natures, such as an extraction from an existing store or a connection to an endpoint API that provides data. This

diversity requires a structuring of the stream of data, that can be achieved in different ways depending of the functional needs behind the architecture. The data traffic controller organizes data into streams. The filter and pipe architecture style is a suitable fit for this component: only lightweight processing are applied, such as removing duplicates or adding a timestamp, before sending data in a communication system that allows other components to have access to them. This mechanism yields a great independence among components, and is the basement for the fault-tolerance characteristic.

The batch layer has no reason to be anymore, and it can be replaced by a component more suited to the situation: the **streaming ETL component**. ETL processes have been part of analytics since a long time, mainly used to populate a data warehouse. However, these last years the need for more freshness in data has become a major preoccupation, and the batch behaviour of ETL — often run once per day at night — cannot fulfill this need. So, several works focus on executing ETL in real-time in order to perform low latency analysis [37, 14]. Streaming ETL transforms data continuously, rather than periodically as it is the case in classic batch ETLs. Stream processing systems are often used to achieve this goal. The role of the streaming ETL component is to populate the storage component, and to transform data if needed for doing so. It works in real-time when data arrive from the data traffic controller, and also in deferred time when data arrive from the master dataset. Data are extracted from the master dataset for example when a schema modification occurs in the storage component.

The aim of the **master dataset component** is the same as in the Lambda Architecture. It stores all the raw data, in case of an evolution of the streaming ETL component or of a failure that requires the re-processing of the data. This component contributes to the fault-tolerance property. To avoid the complexity blamed on the Lambda Architecture with the duplication of processing in the batch and speed layers, data are not directly processed in this component, but rather only extracted from the master dataset, to then be sent to the streaming ETL component. Thus, to maintain or evolve the architecture, only processes in the streaming ETL component have to be modified, avoiding at the same time one of the most criticized pitfall of the Lambda Architecture.

The **real-time insights component** is dedicated to compute predetermined queries or algorithms directly on the stream of data, in real-time, essentially using stream processing systems. The latency and the correctness are critical, but the technical advances of stream processing systems can handle these requirements. Computations can be simple, such as an aggregation (count or sum), or more complex, such as anomaly detection in time series [2]. Operations in this component are well identified and defined, and procure useful insights about data being processed. The storage component can be used to explore data and to find the value they can offer. This knowledge can then be exploited to automate the extraction of value in the real-time insights component. The result of these processing can be stored in the storage component, but due to the be-

haviour of stream processing systems detailed in section 2, this storage step has to be idempotent.

The **storage component** can be of different nature following the needs. It can be a standard data warehouse, a polystore or a data lake. This storage system is fed with data from the streaming ETL component and eventually by the real-time insights component. It puts to disposal processed data in a more suitable format, used for offline and exploratory analyzes. *Data warehouses* are a mature technology, that have been around before the era of Big Data [19]. They often gather data from different sources among an organisation with the help of ETL processes, in order to format and clean data for a business intelligence use. Data warehouses are built to help business analysts by structuring data according to a static schema for a given subject. By doing so, the analyst must only know the schema of his subject to extract value from data. However, this structuring induces a lack of flexibility. A *polystore* refers to a system that integrates heterogeneous database engines, storage systems and multiple data manipulation or programming languages using different paradigms [16]. The use of polystore brings several advantages: it allows to organize data according to particular use cases (e.g. graph DBMS well support linked data and graph traversal or path queries) and it enables parallel processing among several datastores according to the specificities of each kind of system in the polystore [5]. *Data lakes* are less well-defined than data warehouses or polystores [34]. They are often a solution when all data available are harvested, but their use from an analytical point of view is not yet defined. They have emerged to compensate the lack of flexibility of data warehouses. In these lakes, all data are gathered without a common schema, often in a unstructured or semi-structured form. Yet, with this freedom comes a high heterogeneity among data of the lake (in their source, their format, their content, their veracity, etc.), which can turn the data lake into a data swamp if it is not correctly organized.

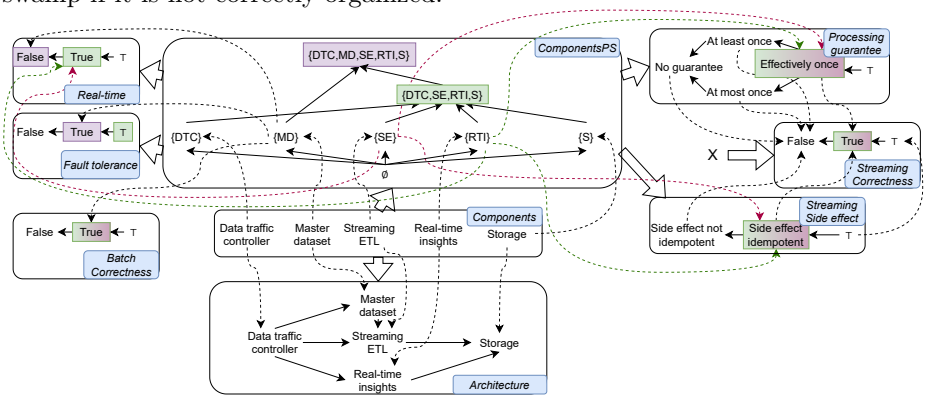

Fig. 4: The formalization of the Lambda+ Architecture pattern

The application of the formalization on the Lambda+ Architecture pattern (figure 4) is straightforward and follows the same mechanism that the one explained in section 3. A more detailed proof is given in the supplementary mate-

rial, including the one for products of categories. Two running compositions of components are possible: with or without the master dataset, as it is only linked to the streaming ETL when data have to be reprocessed. **There is no component that invalidates the correctness property, so it cannot be lost in a composition, and, contrary to the Lambda Architecture, this property holds for the overall architecture.** Concerning the real-time property, only the master dataset induces its loss, but in exchange the Lambda+ activates the fault-tolerance property. It is an acceptable trade-off, as the master dataset is only used in emergency cases, to reprocess data after a failure or when the needs change.

5 An example: Hydre

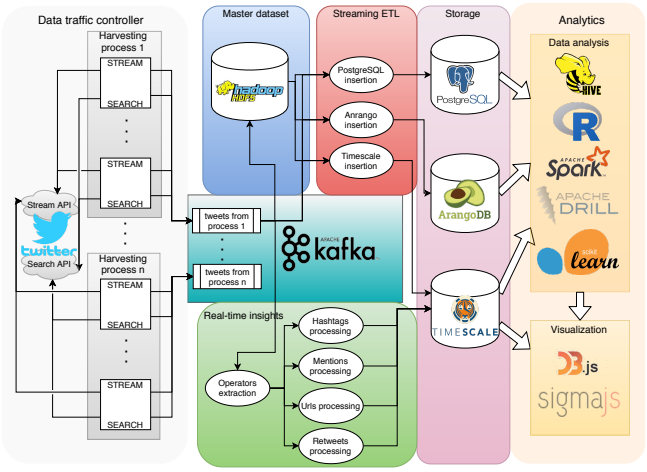

Fig. 5: The Hydre architecture

We have applied the Lambda+ pattern in an interdisciplinary research project (Cocktail) which aims at studying the discourses in the domains of health and food, as well as identifying weak signals in real time using social network data. We needed an architecture able of supporting a continuous gathering of data, while allowing real-time and flexible off-line analytics. It should also have a strong fault-tolerance property, as the use of data could change depending on the results of our researches. It resulted in the Hydre architecture (figure 5), that has been in production since April 2019, to harvest data from Twitter, compute real-time insights and store data for exploratory analyzes. It uses a 20 nodes Hadoop cluster, a 5 nodes Kafka cluster and 4 other servers to host a polystore and applications for exploratory analyzes, including Jupyter notebooks and Spark/Scala analytics processes. The major components of the Lambda+ Architecture are implemented as follows.

The **data traffic controller** is composed of harvesting sub-components, that use Twitter Search and Stream API with specific criteria (hashtags, ac-

counts, etc.) in order to gather tweets in JSON format about a chosen subject. We have only one datasource in this case. The harvesting sub-components add only lightweight information such as the timestamp of the harvesting. They are developed following the actor model with Akka, and send data to Kafka topics, the backbone of the architecture, that allows loosely coupled communications with the other components. The **master dataset** is implemented with Hadoop HDFS. Raw tweets are stored as lines of files, and the re-proccessing of data can be done by reading these files and sending each line as is, in another Kafka topic. The **streaming ETL** uses Kafka consumers to insert data in micro batch. The streaming ETL applies some transformations, and then stores tweets in the **storage** component, a polystore in our case. The polystore includes a relational, a graph, and a time series DBMSs. These databases are used for exploratory analyzes, mainly performed with Jupyter notebooks. Alongside, the **real-time insights** component extracts and aggregates several information about the harvesting, such as popular hashtags or users, using Kafka Streams. It stores the results in the time series database of the polystore. Although this insertion is a side-effect of the stream processing, it is an idempotent action, because the count of the elements will always yield the same result with an effectively once guarantee, and this result is stored for each element, replacing the old value if it already exists.

We have three main harvests continuously running, two global on food and health, and one more specific on COVID-19 vaccines. With the architecture active for 20 months in production, we have gathered 8.3To of raw data. We have already leveraged the fault-tolerance property of the master dataset several times, to apply a new processing as the needs had changed. It was done without impacting the real-time insertions of the streaming ETL. We have also realized some maintenance operations on the streaming ETL without having to stop the harvesting thanks to the message retention of Kafka, and by resuming the processing where it was paused once the maintenance was over. From a user point of view, the polystore is used to cover different needs: 1) to help social science researchers to find new keywords by displaying some query results and macroscopique indicators computed by the real-time insights component on an application server and on Jupyter notebooks ; 2) during research meetings, exploratory analyzes are done in live to guide the formulation of a social hypothesis based on available data ; and 3) once the hypothesis is well-defined, to extract a specific corpus fixed in time to perform more deeper analyzes.

6 Conclusion

We showed the obsolescence of the Lambda Architecture, mainly due to its limited use cases and to the evolution of stream processing systems. We proposed the Lambda+ Architecture pattern, the successor of the Lambda Architecture, that gets rid of its flaws. The Lambda+ defines a more flexible architecture, capable of handling both exploratory and real-time analyzes, and that fits more various use cases than the Lambda Architecture. We uses the category theory to

study the conservation of properties in compositions of components, and applied it on the Lambda and on the Lambda+ Architecture.

For future work, we plan to develop our formalization to study more various aspects of architectures: 1) to navigate among abstraction levels (i.e., the level of detail of the representation of the architecture) ; 2) to verify if an architecture follows a given style or pattern by using full functors (i.e., surjective functors) ; and 3) to extend the property description, including numerical values (e.g., to measure the execution time and deduce if it can be considered as real-time).

Acknowledgment: This work is supported by ISITE-BFC (ANR-15-IDEX-0003) coordinated by G. Brachotte, CIMEOS Laboratory (EA 4177), University of Burgundy.

References

1. Abowd, G.D., Allen, R., Garlan, D.: Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering and Methodology (TOSEM) **4**(4), 319–364 (1995)
2. Ahmad, S., Lavin, A., Purdy, S., Agha, Z.: Unsupervised real-time anomaly detection for streaming data. Neurocomputing **262**, 134–147 (2017)
3. Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: Millwheel: fault-tolerant stream processing at internet scale. VLDB Endowment **6**(11), 1033–1044 (2013)
4. Akidau, T., Chernyak, S., Lax, R.: Streaming Systems: The What, Where, When, and how of Large-scale Data Processing. " O'Reilly Media, Inc." (2018)
5. Alotaibi, R., Bursztyn, D., Deutsch, A., Manolescu, I., Zampetakis, S.: Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In: Proceedings of the 2019 International Conference on Management of Data. pp. 1660–1677 (2019)
6. Broy, M.: Can practitioners neglect theory and theoreticians neglect practice? Computer **44**(10), 19–24 (2011)
7. Broy, M., Cengarle, M.V.: Uml formal semantics: lessons learned. Software & Systems Modeling **10**(4), 441–446 (2011)
8. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36**(4) (2015)
9. Clark, T., Barn, B.S.: Event driven architecture modelling and simulation. In: International Symposium on Service Oriented System. pp. 43–54. IEEE (2011)
10. Clements, P.C.: A survey of architecture description languages. In: International Workshop on Software Specification and Design. pp. 16–25. IEEE (1996)
11. Craig, I.D.: Blackboard systems. Artificial Intelligence Review **2**(2), 103–118 (1988)
12. Deacon, J.: Model-view-controller (MVC) architecture (2009)
13. Eilenberg, S., MacLane, S.: General theory of natural equivalences. Transactions of the American Mathematical Society **58**(2), 231–294 (1945)
14. Fernandez, R.C., Pietzuch, P.R., Kreps, J., Narkhede, N., Rao, J., Koshy, J., Lin, D., Riccomini, C., Wang, G.: Liquid: Unifying Nearline and Offline Big Data Integration. In: Conference on Innovative Data System Research (CIDR'15) (2015)
15. Foote, B., Yoder, J.: Big ball of mud. Pattern languages of program design **4**, 654–692 (1997)
16. Gadepally, V., Chen, P., Duggan, J., Elmore, A., Haynes, B., Kepner, J., Madden, S., Mattson, T., Stonebraker, M.: The BigDAWG Polystore System and Architecture. In: High Performance Extreme Computing Conference. pp. 1–6. IEEE (2016)

17. Harrison, N.B., Avgeriou, P.: Leveraging architecture patterns to satisfy quality attributes. In: European conference on software architecture. pp. 263–270. Springer (2007)
18. Im, J.F., Gopalakrishna, K., Subramaniam, S., Shrivastava, M., Tumbde, A., Jiang, X., Dai, J., Lee, S., Pawar, N., Li, J., et al.: Pinot: Realtime olap for 530 million users. In: ACM SIGMOD. pp. 583–594 (2018)
19. Inmon, W.H.: Building the data warehouse. John wiley & sons (2005)
20. Johnson, P., Ekstedt, M., Jacobson, I.: Where's the theory for software engineering? IEEE software **29**(5), 96–96 (2012)
21. Kreps, J.: Questioning the Lambda Architecture. O'Reilly RADAR,online article, July (2014), https://www.oreilly.com/ideas/questioning-the-lambda-architecture
22. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: Stream processing at scale. In: ACM SIGMOD. pp. 239–250 (2015)
23. Lampson, B.W.: Hints for computer system design. In: Proceedings of the ninth ACM symposium on Operating systems principles. pp. 33–48 (1983)
24. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE Transactions on software engineering **24**(7), 521–533 (1998)
25. Lee, C.H., Lin, C.Y.: Implementation of Lambda Architecture: A Restaurant Recommender System over Apache Mesos. In: Int. Conf. on Advanced Information Networking and Applications (AINA). pp. 979–985. IEEE (2017)
26. Mabrok, M.A., Ryan, M.J.: Category theory as a formal mathematical foundation for model-based systems engineering. Appl. Math. Inf. Sci **11**, 43–51 (2017)
27. Malkis, A., Marmsoler, D.: A model of service-oriented architectures. In: Brazilian Symposium on Components, Architectures and Reuse Software. pp. 110–119. IEEE (2015)
28. Marmsoler, D., Malkis, A., Eckhardt, J.: A model of layered architectures. arXiv preprint arXiv:1503.04916 **178**, 47–61 (2015)
29. Marz, N.: How to beat the cap theorem (2011), http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
30. Marz, N., Warren, J.: Big Data: Principles and best practices of scalable real-time data systems. Manning (2015)
31. Munshi, A.A., Mohamed, Y.A.R.I.: Data lake lambda architecture for smart grids big data analytics. IEEE Access **6**, 40463–40471 (2018)
32. Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. International Journal of Open Information Technologies **2**(9), 24–27 (2014)
33. Richards, M., Ford, N.: Fundamentals of Software Architecture. O'Reilly (2020)
34. Sawadogo, P., Darmont, J.: On data lake architectures and metadata management. Journal of Intelligent Information Systems pp. 1–24 (2020)
35. Spivak, D.I.: Category theory for the sciences. MIT Press (2014)
36. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al.: Storm@ twitter. In: ACM SIGMOD. pp. 147–156 (2014)
37. Vassiliadis, P., Simitsis, A.: Near real time ETL. In: New trends in data warehousing and data analysis, pp. 1–31. Springer (2009)
38. Yang, F., Merlino, G., Ray, N., Léauté, X., Gupta, H., Tschetter, E.: The RAD-Stack: Open source lambda architecture for interactive analytics. In: Proceedings of the 50th Hawaii International Conference on System Sciences (2017)
39. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: USENIX Hot Topics in Cloud Computing (2012)