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

Annabelle Gillet, Eric Leclercq, Nadine Cullot

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

Annabelle Gillet
LIB Univ. Bourgogne Franche Comté
Dijon, France
annabelle.gillet@depinfo.u-bourgogne.fr

Éric Leclercq
LIB Univ. Bourgogne Franche Comté
Dijon, France
eric.leclercq@u-bourgogne.fr

Nadine Cullot
LIB Univ. Bourgogne Franche Comté
Dijon, France
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.

## KEYWORDS

Architecture pattern, Category theory, Lambda Architecture

## 1 INTRODUCTION

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 [6], 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 [3], that reduces greatly the maintenance and evolutivity capabilities of the system.



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 [1, 5]. 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 [2] 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 [9].

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. For more details on this work, see [4].

## 2 THE LAMBDA ARCHITECTURE PATTERN

The properties of correctness, low latency and fault-tolerance have always been a major concern when designing architectures. In [6], 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 [7, 8]. The objective of the Lambda is simple: to compute predetermined queries with a very low latency and to ensure the correctness of the processing. To do so, the Lambda is composed of three layers: the batch layer, that takes care of storing raw data in the master dataset and of executing the computations on the batch of data while preserving the correctness property, the speed layer, that performs the same computations as the batch layer but with an incremental processing to support the low latency property but not the correctness one, and the serving layer, that puts the results to disposal.

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. 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 computations, but with different paradigms. It also lacks in flexibility, as its goal is to answer only predetermined queries. Thus, 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. It includes the correctness property, that have since been integrated into the stream processing systems. So, the Lambda can be seen as a mean to compensate flaws of an emerging technology, rather than a pattern that fully exploits it.

## 3 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 incompatible with the real-time property, it should be integrated as an alternative running composition of components, 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 different layers should be avoided.

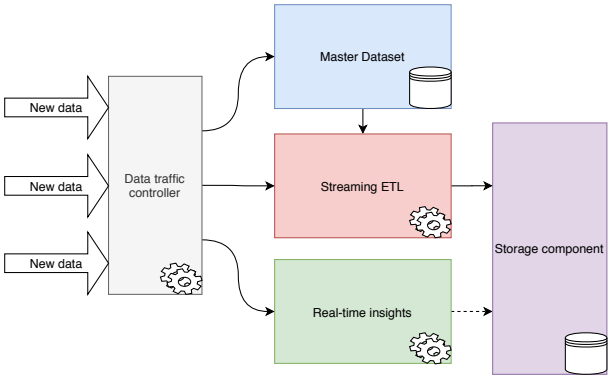


**Figure 1: Overview of the Lambda+ Architecture**

The Lambda+ Architecture (figure 1) 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 in a way that allows 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 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.

## 4 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 [1, 5]. 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.

To fill this need, category theory [2] is a promising approach: it allows to switch from a model to another or to navigate among abstraction levels [9]. By focusing on relations (the morphisms) and compositions, it proposes powerful mechanisms that can be applied to architectures: the behaviour of functors combined with preorders allows the study of the conservation or the discarding of properties in compositions of components.

## 5 CONCLUSION

We proposed the Lambda+ Architecture pattern, the successor of the Lambda Architecture, that gets rid of its flaws and fits more various use cases by handling both exploratory and real-time analyzes. We used the category theory to study the conservation of properties in compositions of components.

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., the execution time to deduce if it can be considered as real-time).

## ACKNOWLEDGMENTS

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

## REFERENCES

[1] Manfred Broy. 2011. Can practitioners neglect theory and theoreticians neglect practice? *Computer* 44, 10 (2011), 19–24.
[2] Samuel Eilenberg and Saunders MacLane. 1945. General theory of natural equivalences. *Trans. Amer. Math. Soc.* 58, 2 (1945), 231–294.
[3] Brian Foote and Joseph Yoder. 1997. Big ball of mud. *Pattern languages of program design* 4 (1997), 654–692.
[4] Annabelle Gillet, Éric Leclercq, and Nadine Cullot. 2021. Lambda+, the Renewal of the Lambda Architecture: Category Theory to the Rescue. In *International Conference on Advanced Information Systems Engineering*. Springer, 381–396.
[5] Pontus Johnson, Mathias Ekstedt, and Ivar Jacobson. 2012. Where's the theory for software engineering? *IEEE software* 29, 5 (2012), 96–96.
[6] Butler W Lampson. 1983. Hints for computer system design. In *Proceedings of the ninth ACM symposium on Operating systems principles*. 33–48.
[7] Nathan Marz. 2011. How to beat the CAP theorem. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
[8] Nathan Marz and James Warren. 2015. *Big Data: Principles and best practices of scalable real-time data systems*. Manning.
[9] David I Spivak. 2014. *Category theory for the sciences*. MIT Press.