# Towards an architecture for real-time event processing

Juan Martín Pampliega

Jampp
juan@jampp.com

**Abstract.** The purpose of this paper is to illustrate how and why we have evolved our data processing systems; from an initial version that relied on traditional RDBMSs and batch processing towards a system that processes a constant stream of data.

## 1 Introduction

Jampp is a mobile app marketing and retargeting company. Our platform helps mobile app advertisers to acquire and re-engage their users globally. This enables brands to go beyond simple installs and re-targeting clicks, as it optimizes for in-app activity and conversions, thus maximizing lifetime value.

Jampp markets apps by effectively buying programmatic ads. Our platform processes more than 200,000 Real Time Bidding (RTB) ad bid requests per second, which amounts to about 300 MB/s or 25 TB of data per day. Additionally, Jampp tracks more than 6 billion in-app events (app installs, actions taken inside the app, and contextual information used for user segmentation) per month.

The advertising technology (Ad Tech) market is technologically intensive, with low margins and high volume. Being able to effectively and promptly deal with the constant stream of data our platform receives is imperative for us to thrive in this ecosystem.

As big as the general trend is for Big Data systems of moving to real-time processing, this is even more important in the market we are competing in. Real-time processing is often used as a buzzword with no clear meaning since the notion of what is real-time latency might vary by milliseconds, seconds or minutes, according to the rules of each certain market. Therefore, we generally prefer to talk

about **stream processing** which refers to the continuous processing of infinite or unbounded data sets.

In the following sections, we will describe our journey from using more traditional techniques for processing our data to stream processing technologies.

## 2 Relevant business background

RTB allows display inventory to be purchased by the individual impression through a bidding system that unfolds in milliseconds before a web-page (or app screen) is loaded by a consumer.

Our platform at Jampp is what is referred to as a Demand Side Platform (or DSP). Basically, a system that automates the purchasing of online advertising on behalf of advertisers. Our bidding system is integrated with all of the most important Ad Exchanges in the world. An Ad Exchange is a system that connects DSPs with ad publishers (places where the actual ads are displayed) by running auctions of these ad spaces.

When talking about the RTB process, we have 3 main event types that appear in the following order:

1. **Auction:** notification of the possibility of buying an impression. The Ad Exchange notifies all bidding systems belonging to the DSPs that have integrated to it, including ours.
2. **Bid:** offer that our systems make for the impression that is currently being auctioned. It needs to include how much we are willing to offer and the creative's (i.e. advertisement) HTML markup. The bid needs to be sent in less than 80ms right after we have received the auction.
3. **Impression:** notification that the ad has been shown to the user.

After the impression, we have conversion events similar to any other programmatic ads system. Our platform receives some of these events from a different kind of systems called tracking platforms. These systems are integrated into each app with an SDK and track and record every in-app event completed by a user. Our clients, the owners of the apps, configure these tracking platforms so that we receive notifications about installs and in-app events.

There are three main types of conversion events:

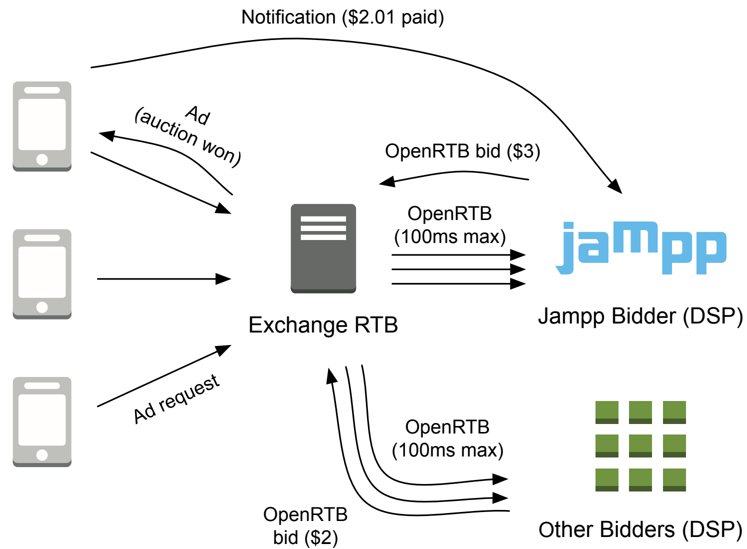


**Fig. 1.** RTB auction example where 3 opportunities of impressions generate 3 calls to the Ad Exchange and only one of them produces a winning bid.

1. **Click:** it occurs when the user clicks on the ad and is redirected either to the app store to install the app or to a landing screen inside the app if it is already installed.
2. **Install:** it occurs when the user installs the app on the device. Our platform receives this event the first time the user opens the app, so there might be a considerable delay after the click.
3. **In-app event:** it occurs when a user uses the app, completing actions such as viewing a listing, purchasing a product, booking a taxi, etc.

When referring to installs and in-app events, we can classify them into two categories. **Attributed events** are events that were originated by a click on one of Jampp's ads. These are the events we charge our clients for. The second category is **organic events**. These are events that are not originated by an ad from Jampp. Usually, the volume/number of organic events is much higher than the number of attributed events, and we use them to optimize our retargeting techniques.

In summary, our clients (the app owners) pay per install or in-app event. We use our machine learning algorithms to predict which impressions are most likely to generate a click and install or in-app event from the user. Our algorithms also try to estimate the price we need to bid to win that impression. Being able to obtain the "best impressions" is the fundamental way of making our business profitable.

## 3 Initial event processing system

As previously mentioned, we have two systems that receive events. On one side, a bidding system that generates events like auctions, bids, wins, loses and impressions. This system is written in a combination of Python and Cython and runs on approximately 200 instances (virtual machines) with 8 cores and 70 GB of memory in the Amazon Web Services (AWS) Cloud. Initially, since the volume of these events was orders of magnitude larger than the ones from the tracking platform, we were only publishing them through a PUB/SUB channel using ZMQ. The events were being published by a set of processes called Bid Loggers, which are also in charge of

filtering, aggregating, and sampling the data to persist it in the *PostgreSQL* database. The stream of events from the bidder is designed as PUB/SUB message stream without persistence. This means only messages that are actually being published are the ones which belong to topics that have actual consumers subscribed. When events are published, they are partitioned by a transaction id that is maintained throughout the whole chain of events: auction, bid, impression, click, in app event. This enables the consumers to do consistent sampling by subscribing to a particular partition and receiving all the associated events for one transaction. This capability is vital when systems want to analyze the events' data without being forced to handle 100% of the volume, avoiding skewed results.

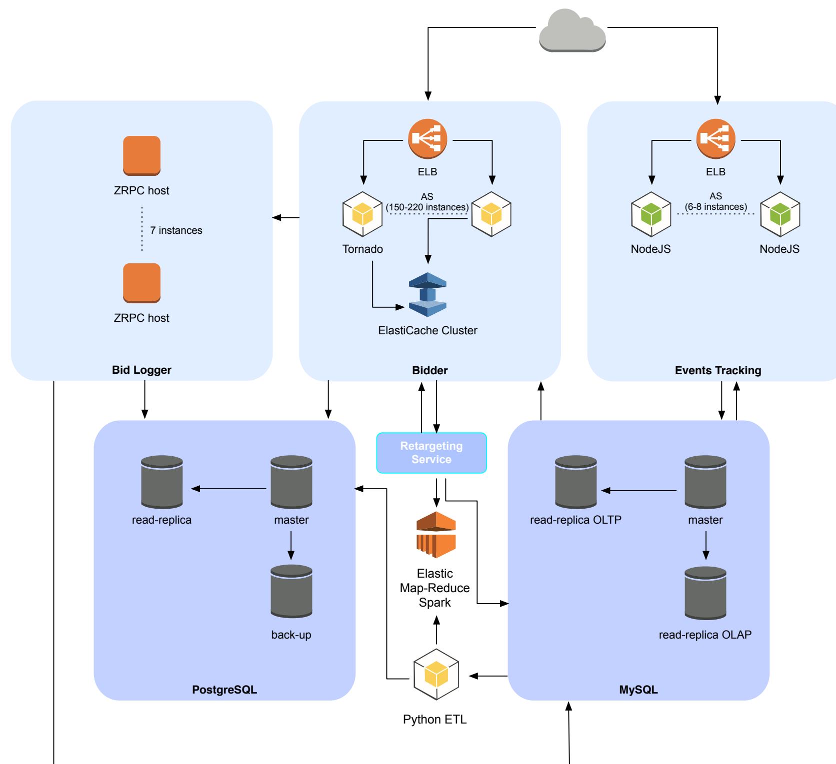


**Fig. 2.** Approximate initial architecture of Jampp's platforms.

The second system is an application written in *Node.js*, which runs on 10 instances. This tracking system receives clicks, installs and in-app events; both attributed and organic. Since in the beginning the volume of these events was much smaller and we were receiving organic events from just a couple of apps, we were able to use *MySQL* as the data store to keep all the raw events.

Raw events are copied from *MySQL* to the *PostgreSQL* database by a *Python* process. As the data is copied, events are aggregated to form a warehouse according to different dimensions like country, city, device OS, campaign, app category, advertiser, etc. This single *Python* process publishes into the *ZMQ* stream all events coming from *MySQL* after correctly inserting the data in *Postgres*. This might mean that, when the *Postgres* database is under heavy load, the insertions are delayed and there is a replication lag of events to the *ZMQ* stream.

## 4 Limitations of our initial architecture

Our initial approach was very good for dealing with the volume of events we received during the early stages of the company, and providing some analytics capabilities to the company in general. As the company continued growing, so did the need for data to be available in a faster, more detailed and scalable way. There were several factors that fostered these needs:

- Our specialized analytics and data science team needed raw event data at its most granular level to carry out their tasks.
- Several of the new systems we developed relied on Machine Learning models that needed granular and very enriched data.
- New systems like the ones implemented for fraud detection would greatly benefit from the data being available for consumption with the lowest latency possible.
- Several new systems needed access to this data and interfacing with traditional SQL databases was not an ideal solution for them in regards to scalable performance or querying capabilities.
- We needed a single and scalable source of truth that contained all the raw data from which we could derive consistent aggregated views for different systems.

– Replication lag between the MySQL database and the warehouse in the PostgreSQL would sometimes increase until the data was delayed for hours. We needed some way that could be easily scaled to reduce this lag just by adding more processing power.

# 5 First stream processing experience with organic events

As our first experience with building a stream processing pipeline, we migrated organic installs and in-app events from MySQL to a combination of Amazon Kinesis, Amazon Lambda, Amazon DynamoDB and Amazon S3.

The central processing technology here was Lambda, which is a product from Amazon that executes arbitrary code in Node.js, Python or Java. This code is executed every time a source event occurs. In our use case, this event consisted of messages being available for consumption in Kinesis. The good thing about Lambda is that Amazon completely manages the infrastructure where the code is executed, the user only supplies the code and sets how much memory it needs.

MySQL was a great tool to start with, but, as we began onboarding more and more clients (and bigger clients with significantly higher volumes of organic events), the growth in the raw data simply became too much for MySQL to handle efficiently.

Additionally, having MySQL as the only place where this data was available made it even harder to access it efficiently. As the needs of the company and other data systems evolved, the data team was under increasing pressure to make this data available in a scalable and efficient way.

Thinking about the problem in detail, it was clear that we needed some kind of scalable event queue (or log). The best two options we found were Apache Kafka and Amazon Kinesis. Kafka seemed to be a wonderful, efficient, and very robust technology to fulfill this use case. The problem is that we are a small team and we are big users of the Amazon Web Services (AWS) Cloud. When you are a small team that needs to move fast, nothing beats a product that can decently fulfill the use case *and* is also hosted and maintained by your cloud provider. That's why we decided to go with Kinesis.

An Amazon Kinesis stream is an ordered sequence of data records. Each record in the stream has a sequence number that is assigned by Kinesis. The data records in the stream are distributed into shards which determine the read and write throughput the stream supports. We only needed to set the amount of streams according to the needed throughput and reshard it (divide the shards) in case we need to scale and need additional throughput.

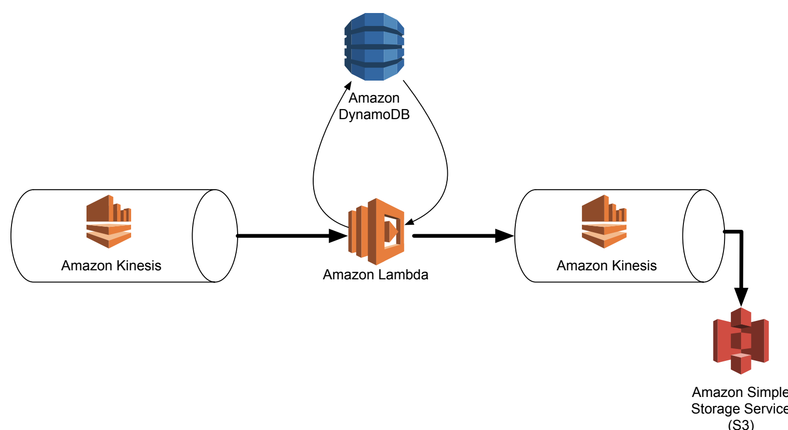


**Fig. 3.** Architecture of the first lambda pipeline.

As the logic needed to process each event was as simple as enriching each message separately, we decided to try Amazon Lambda and DynamoDB to implement it. These tools are all automatically scalable, and created and managed by Amazon. So we just needed to implement the logic and deploy the code without worrying about the infrastructure. Finally, we went about implementing this system and having the enriched data available for real time applications through additional Kinesis streams and to batch systems through S3.

## 6 Options for stream processing

We had a pretty straightforward experience developing our first stream processing pipeline with AWS Lambda, Kinesis and Dy-

namoDB. But moving the rest of our event processing infrastructure to a stream processing pipeline would mean having pipelines with much larger volumes and more complex logic (including for example: out of order data and missing data).

It hard to explain all the characteristics needed in a robust stream processing engine, but fortunately we can reference the *Dataflow paper*[4] and *Stream Processing 101*[1] and *Stream processing 102*[3] blog posts.

Keeping this in mind, we turned into investigating different options, aside from the mix of tools we used for our first pipeline.

### 6.1 Apache Spark

Spark is the go-to tool for Big Data processing nowadays. It was created at Berkeley's AmpLab, spawned as an Apache Foundation project that generated a company driving it forward, named Databricks. When looking into Spark we found several aspects we liked about it:

- You can code the pipelines in Scala, Java or Python. We use mostly Python at Jampp, but also have experience in the other two main languages.
- It uses the same business logic code with some small tweaks to do batch or stream processing.
- Great coverage of Machine Learning algorithms and facilities for running and combining them.
- Interactive console when writing in Scala or Python to quickly test code or look through data.
- Configured and installed by Amazon in their Hadoop cluster product (Elastic Map Reduce)
- Compatible with most possible data sources S3, Kinesis and Kafka.
- Big community, which makes the tool evolve faster and provides lots of examples and support.

In our initial assessment Spark seemed like a great fit, but digging deeper we found some worrying aspects for our use case:

- Spark's main way of maintaining state for streaming applications is keeping maps of key-values in memory which is very inefficient when dealing with a large volume of data.

- Spark does not handle the notion of event time processing, so it can't efficiently handle out of order data or very long windows of time.
- Spark does micro-batching, which is not true stream processing. This introduces unnecessary latency to the pipeline, which might not be a big deal, but it had to be considered.
- Even though we can use Amazon EMR for installing Spark, there is still a lot of work in maintaining it and tuning it, which Lambda mostly obviates.

As these items show, when considering Spark specifically for a stream processing use case, it lacks many much needed characteristics.

### 6.2 Apache Flink

Flink was started as a research project between different universities in Germany in 2010. Afterwards, it moved to the Apache foundation in 2014. Prominent members of the Flink community founded the company Data Artisans, where they commercialize solutions and offer consulting. As we looked into Flink we were very pleasantly surprised. As mentioned in *this recent post*[2], Flink fulfills most of the features of Google's Dataflow model, which is a model that describes all the capabilities needed for robust, modern, stream processing.

Some of Flink's capabilities really stood out:

- Enables processing of data by event time so as to handle out of order data.
- Enables early and late firing of event time windows so as to provide partial results early on or after a timeout.
- Supports local state for data enrichment by using RocksDB for out of memory data.
- Allows periodic checkpoints of the application state to HDFS, to enable simple restart of running applications when deploying new versions of the pipeline.

Keeping these items in mind and comparing them to Spark's, we concluded that Flink was a much better solution to fulfill our use case. However, we also found some issues worth considering:

- Flink's state is checkpointed as a whole to HDFS each time, there are no incremental checkpoints just yet.
- Flink only supports Java and Scala, not Python.
- Flink's source connector for reading events from Kinesis is still under development.
- There are no facilities offered by Amazon to automatically install, configure and update Flink in Amazon EMR.
- Flink is a much younger project than Spark, and there is a lot less material to use as reference when developing complex applications.

## 7 Implementation

In sum, although we had a very good experience implementing a stream processing pipeline with AWS Lambda, since the newer pipelines needed much more refined capabilities, we investigated other options.

Both Spark and Flink have aspects where they shine in regards to Lambda, but neither of them ultimately convinced us as a better alternative. Spark does not really do event time processing for out-of-order events. Flink does not integrate well with AWS products and seemed somewhat immature. Finally, both of them needed much more infrastructure and maintenance work than Lambda.

Therefore, we decided to continue doing our stream processing with Lambda, and develop a custom implementation for handling out-of-order data, and event time windows.

The main logic that this processing entailed was enriching the data of each event with all the data from the previous events of the same transaction chain. For example, if we received a click, we needed to enrich the content of this click with the data from the corresponding impression and bid. Fortunately, each event already contains a transaction identifier field which uniquely identifies the transaction it belongs to.

Since *each* event was enriched with all this data, it could grow up to 8-10 KB in size; when adding up the millions of events we receive each minute, this translated to a very large volume of information. This meant that using *DynamoDB* as a single database to store all the events would be very costly, since the pipeline involved a very large throughput.

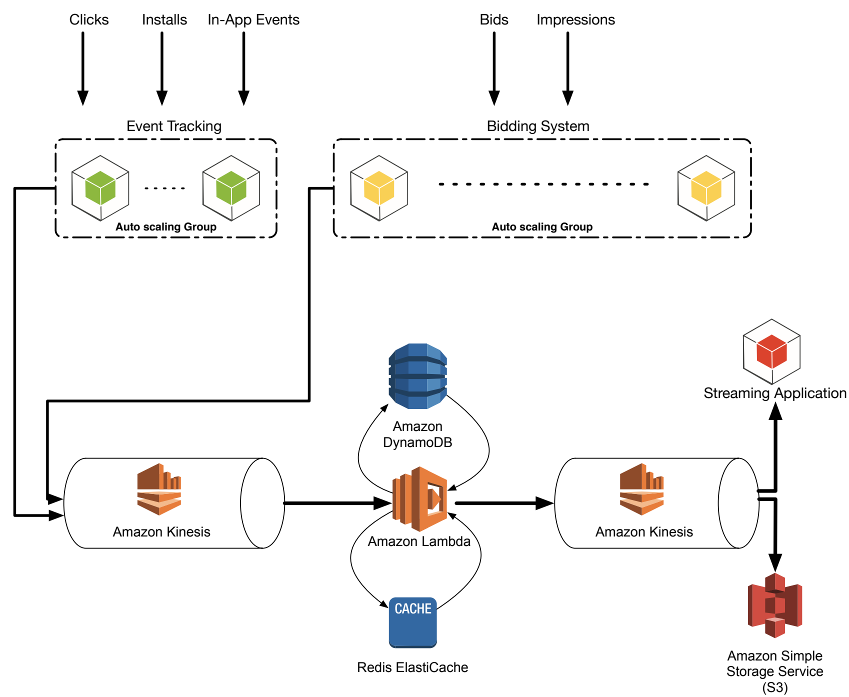


**Fig. 4.** Final architecture for event stream processing.

Looking at the problem closely, we decided to separate the storage according to the event type. Bids and impressions are much bigger in numbers and size, but have a much shorter life span. We usually just need about 10 minutes of bids and 4-5 hours of impressions. We decided to store them in a *Redis* cache that we provisioned using AWS Elastic Cache product. Aside from a lower fixed cost, *Redis* lets us set expiration for keys, which was ideal for dealing efficiently with bids and impressions. As clicks, installs and in-app events have a much longer lifetime we decided to continue using *DynamoDB* for them.

The final piece of the puzzle was how to handle out-of-order and missing events. We implemented an index of transactions in *Redis* which contained the events they were missing in the chain. There is an additional Lambda function that periodically scans this index and emits transactions that are expired. When events arrive out of order, this index helps us keep track of which later events in the chain are waiting for the previous ones, and fire an enrichment and emission process of them.

Since at the moment we are prioritizing having complete data over having data as early as possible, this logic was all we needed to implement.

## 8 Future Improvements

Even though we finally chose Lambda to implement our stream processing pipelines, we see Flink as a very promising technology. Most the reasons that made us choose another technology are set to be fixed this year with the inclusion of a Kinesis connector, incremental snapshots and some much needed maturing of the tool. Using Flink would give us the capabilities to do some early emission of a window's data so as to better serve applications that prefer low latency over correctness. Moving to Flink would mean more work maintaining and tuning the infrastructure but, since we already have a working pipeline and have the data available for other use cases, we can spare time doing it.

# 9 Conclusion

The evolution of the Big Data landscape has brought about the proliferation of data sources that produce unbounded data sets. Being able to process these streams continuously rather than with batch methods, has introduced numerous benefits that are invaluable to our business. Much progress has been done in the last couple of years, but there is still need for further development before having a mature tool that fulfills all the aforementioned needs. From our point of view, Flink seems to have a head start since it brings to the table a very complete feature set.

We think that this overview of our journey implementing these stream processing pipelines can help shed some light on best practices and important things to consider when building this type of data processing systems.

# References

1. Tyler Akidau. The world beyond batch: Streaming 101, August 2015. [Online; posted 5-August-2015].
2. Tyler Akidau. Why apache beam? a google perspective, May 2016. [Online; posted 03-May-2016].
3. Tyler Akidau. The world beyond batch: Streaming 102, January 2016. [Online; posted 20-January-2016].
4. Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.