# DATALYZER: Streaming data applications made easy

Mario González-Jiménez, Juan de Lara

Modelling and Software Engineering Research Group
http://miso.es
Universidad Autónoma de Madrid (Spain)

**Abstract.** Nowadays, streaming data are continuously generated from thousands of sources, including social networks, mobile apps, sensors, e-commerce transactions, and many more. Hence, it becomes very useful to build applications able to process these data, with the purpose of filtering interesting parts, monitor their run-time evolution, persist valuable chunks, trigger events upon certain conditions are met and provide analytics. While several frameworks and systems have emerged to create this kind of applications, these systems tend to be low-level, based on complicated APIs, challenging to install and configure for end-users, and requiring from high performant hardware for their execution. Our goal is to lower the entry level to develop, deploy and run streaming applications.
To accomplish this goal, we propose DATALYZER, an approach to create streaming data applications on the cloud based on a visual language. This way, DATALYZER provides a facility to describe streaming data sources in an open way, and a visual language to describe the execution flow of the streaming application. DATALYZER is based on model-based development principles, where code is generated automatically, and then compiled, deployed and executed on the cloud. As a proof of concept, we describe a case study in enterprise systems, and how it can be built using our prototype tool.

**Keywords:** Streaming data; Data transformation; Data monitoring; Cloud-based development environments; Model-based development; Code generation

## 1 Introduction

We live in a hyper-connected society, where 2.5 quintillions bytes of data are created every day[1]. Millions of heterogeneous sources around the world produce continuous streams of data, including sensors, social networks, IoT devices, mobile and web applications, among many others. Therefore, there is an increasing need to observe, monitor, analyse, detect anomalies [7] and make apps out of these heterogeneous data. However, currently, building such applications is only possible by engineers with specialized technical knowledge, and having expensive hardware at their disposal. Moreover, the required technologies for the demanding requirements of this kind of applications tend to have a steep learning curve and be difficult to install, configure, and run [2].

[1] https://www.domo.com/learn/data-never-sleeps-5

In order to solve the previous issues, we propose DATALYZER, a cloud-based platform to build streaming data applications. Our approach follows model-based development principles [1] and is based on a domain-specific visual language (DSL) to facilitate programming by non-experts. DATALYZER is able to synthesize code – based on Apache Kafka[2] – compile it, configure the application and run it in an automated way. This way, the end user uses the web browser to create visually the model of the application to be built. As the approach is cloud-based, there is no need for difficult configurations by the end user. Moreover, the approach is extensible, and relies on a common format to describe live data sources. This permits the addition of new heterogeneous sources as needed, which then can be used in DATALYZER programs. In this paper, we describe the principles of DATALYZER, and show our prototype tool support.

In order to explain and discuss DATALYZER'S features and capabilities, we will be using as running example the creation of a streaming application to monitor the data flows generated by the traffic of a video on demand service. In such scenario, we would like to monitor at run-time, in an integrated way, the users logging into the system, the contents they are visualizing to produce a run-time ranking with the most viewed content and the bandwidth generated (considering the video resolution and bitrate being used). In addition, we would like to integrate in the monitoring dashboard flows originating outside the systems of the company, like the mentions to the company profile in social networks such as Twitter.

The rest of the paper is organized as follows. Sec. 2 overviews the main concepts behind our approach. Sec. 3 describes the architecture and tool support. Sec. 4 compares with related work, and Sec. 5 ends with the conclusions and future work.

## 2 DATALYZER

This section describes the main ingredients of DATALYZER: a uniform, extensible representation of streaming data sources (Sec. 2.1) and a visual DSL to describe execution workflows (Sec. 2.2).

### 2.1 The Data Sources

Dynamic data sources available on the web are heterogeneous with respect to the protocols and technologies used, and their access architectures, typically either pull-based (polling) or push-based (streaming) [4]. Our goal is to be able to create dynamic data streams out of any source type, and hence our approach permits abstracting away the technical details of the different sources (e.g., whether pull or push). This way, we can use them as dynamic live data channels in a uniform way. For this purpose, we use a declarative description model for data sources, which facilitates the incorporation of new sources. A scheme of the description model is shown in Fig. 1(a).

Our model distinguishes between data sources, and data source *types*. The latter refers to types of data transmission technologies like REST or TCP sockets, but also permits describing non-standard technologies. For example, live data platforms like Satori[3]

[2] `https://kafka.apache.org/`
[3] `https://www.satori.com/`

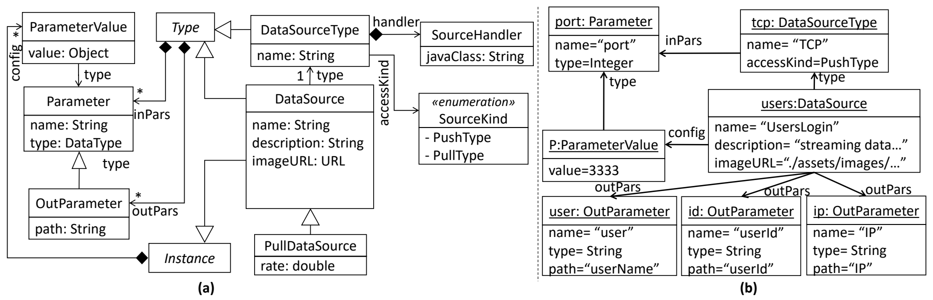


**Fig. 1.** (a) "Sources" conceptual model (excerpt) for dynamic data sources. (b) Description of one of the data sources types and a data source of the running example.

or social networks streams like Twitter[4]. Explicitly representing data source types in our meta-model, instead of hardcoding them, fosters extensibility. A DataSourceType is described by its name and an access kind (pull or push). DataSourceTypes can define input and output parameters (inherited from Type). For example, a TCP socket connection defines a port as input parameter, while a REST API defines a URL, and Twitter declares a private token key and secret parameters for authorization. Each source type is handled by a Java class, specified in SourceHandler.

Once a DataSourceType is described (e.g., REST), it is possible to create DataSources of such type (e.g., OpenWeatherMap[5]). A data source contains a name, a description, and an image to be displayed on the DATALYZER DSL editor. DataSources inherit from both Type and Instance. This is so as they need to provide values for the input parameter of its DataSourceType (e.g., the URL value for REST), and may declare parameters in their turn (e.g., OpenWeatherMap needs the city name). In the next subsection, we will see that our DSL supports an additional instantiation level, as when used for creating a specific streaming application, users need to specify values for the input parameters of DataSources (e.g., a particular name for the city in OpenWeatherMap).

Please note that OutParameters have an attribute called path, to describe how to retrieve their value from the incoming message. In case of XML, this would be an XPath expression [11], while for JSON we use a similar approach to JSONPath[6]. Finally, in case of pull-based sources, like for example those of type REST, our system polls for new data at customizable rates, which permits emulating a live data stream source.

**Example**. Fig. 1(b) describes one data source for the running example. In this case, we assume the information on logged users is obtained through a TCP socket connection. A TCP DataSourceType declares a port parameter, of type integer. Then, a DataSource of type TCP can be defined (object users, with name UsersLogin). This needs to configure the parameters of its type, which in this case is the port, set to 3333. In addition, the data source may declare input parameters to be filled when used in a DATALYZER model, and outputs that the source produces (three in this case, named user, userId and IP). A

[4] `https://developer.twitter.com/`
[5] `https://openweathermap.org/api`
[6] `https://github.com/json-path/JsonPath`

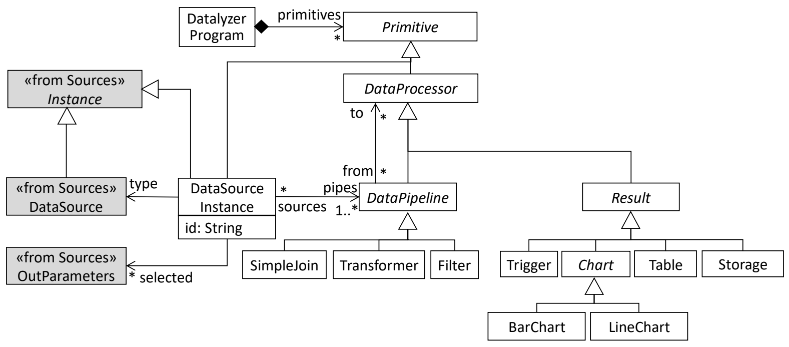


**Fig. 2.** DATALYZER meta-model excerpt.

data source of type TCP will not normally declare input parameters, while sources of other types like REST will normally do. When this data source is used in a DATALYZER application, a socket will be created in the specified port and will wait for incoming connections where the data is received, while the transmitted data will contain the three specified output fields.

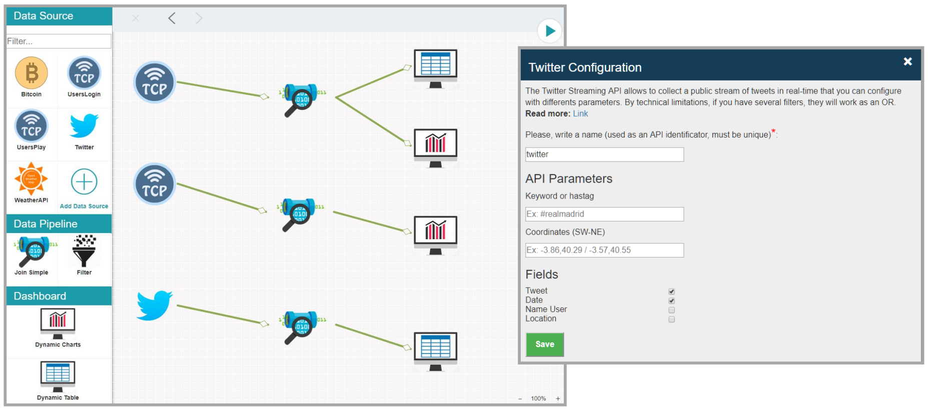
## 2.2 The DATALYZER DSL

Once the data sources of interest are defined, the user can create her application using a visual DSL we have designed. In practice, this creation process occurs within a web browser, using a drag and drop mechanism. After finishing the description of the application, a code generator we have built synthesizes the program code, compiles it, and finally executes the described streaming application.

A meta-model excerpt for the DATALYZER DSL is shown in Fig. 2. The DSL is made of three main primitives: DataSourceInstances, DataPipelines and Results. The DataSourceInstances can instantiate DataSources among those previously defined, as seen in Sec. 2.1. When a DataSourceInstance is included in a model, the user needs to configure the input parameters that the corresponding DataSource has defined, and select the outputs she wants to capture. Please note that, as DataSourceInstance inherits from Instance, it can define ParameterValues for the Parameters of the DataSource.

DataSourceInstances must be connected to some Data Pipeline. These are in charge of joining, filtering and transforming (e.g., aggregate operators) the data received from the data sources. Several data sources can be connected to the same data pipeline. While we foresee different kinds of join semantics (like using temporal windows), DATALYZER currently supports a simple join based on a single tuple window [3]. This pipeline is able to join data tuples from several sources, by only considering the last tuple obtained from every source. Pipelines can be connected to each other, but connections need to be cycle-free and to end on nodes of type Result.

Results are terminal nodes, which produce a visualization for the data, a trigger, or persist the data. Currently, DATALYZER supports three types of visualizations for data: based on tables, and using dynamic bar and line charts. Other visualization techniques, e.g., based on geographical maps are under development.



Twitter Configuration

The Twitter Streaming API allows to collect a public stream of tweets in real-time that you can configure with differents parameters. By technical limitations, if you have several filters, they will work as an OR.
**Read more:** Link

Please, write a name (used as an API identificator, must be unique)*:

twitter

API Parameters

Keyword or hastag

Ex: #realmadrid

Coordinates (SW-NE)

Ex: -3.86,40.29 / -3.57,40.55

Fields

Tweet ✓
Date ✓
Name User ☐
Location ☐

Save

**Fig. 3.** Constructing the running example with DATALYZER: the application described in the DSL editor (left), and the data source configuration view for a Twitter source (right).

**Example**. Fig. 3 depicts an excerpt of the DATALYZER model for the running example (created with the DATALYZER web editor). The left window shows the palette with the primitives for creating the streaming application model. The canvas contains flows describing how to handle the data produced by three different sources: two coming from a TCP connection (UsersLogin, defined in Fig 1(b), and UsersPlay), and the other one from Twitter. This Twitter data source declares two input parameters: keywords/hashtags, and geoposition coordinates. These parameters are used to retrieve only the tweets meeting such criteria. In addition, Twitter declares 4 output parameters: the tweet text, its date, the user name and the location. The DataSourceInstance for Twitter can then give values to the inputs and select the desired outputs. Please note that we can have several DataSourceInstance objects of the same data source in the same model (e.g., several sources taking data from Twitter).

## 3 Architecture and Tool Support

Fig. 4 shows DATALYZER's architecture, made of the following components:

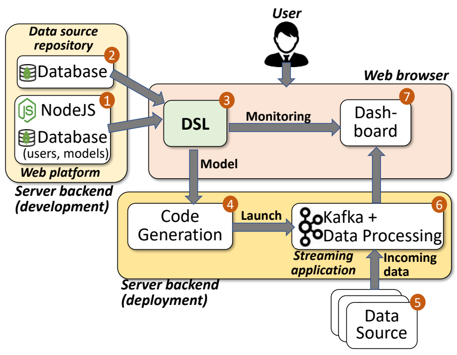


**Fig. 4.** DATALYZER architecture.

**Web Platform.** A web platform developed using NodeJS and MVC architecture (see label 1 in Fig. 4). It features a user registration system, using a MongoDB database, to identify and authenticate the connecting users. In this platform, users may create projects that contain the models of the streaming applications. The platform permits managing those projects, listing them, editing and executing the models.

**Data Source Repository.** The description of data source types and data sources (seen in Sec. 2.1) are stored in a MongoDB database (label 2 in the figure). This database is connected with the DSL, in order to show the different data sources in the editor palette. The database is read every time the DSL editor is opened, so that the list of defined data sources is available at each moment. Users can add new sources through the web browser using a form.

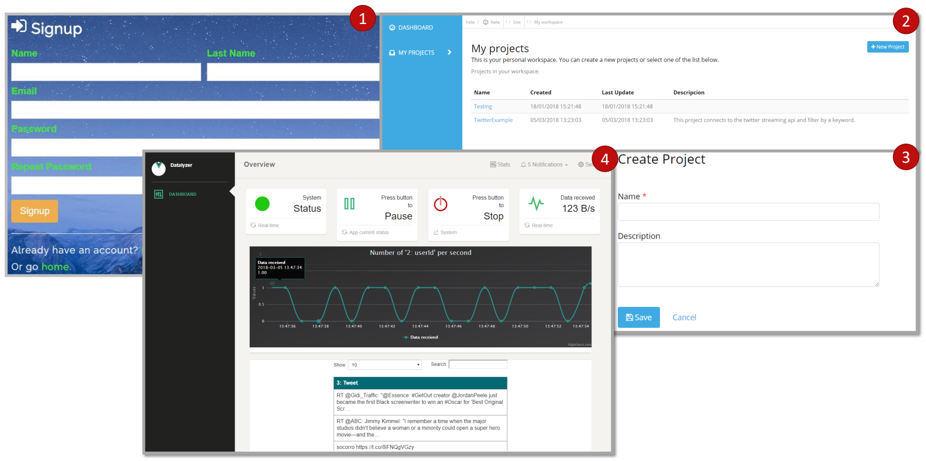
**DSL.** The DSL editor is integrated in the web platform (label 3 in the figure). This editor is accessed whenever a project is edited. The DSL editor is built in Javascript, and contains built-in validations to ensure (syntactical) model correctness at design time. If a model is correct, it can be stored in the database, so that it can be opened, modified or executed later.

**Code Generation.** We have built a code generator in Java that is invoked by the web server whenever the application is to be executed (label 4 in the figure). Our strategy for code generation creates a Java class for every object in the model. The end result is an independent, fully functional Java project (based on Apache Kafka as described below), which expects to receive and process the live data sources as specified (label 5).

**Streaming Application.** The generated application (label 6) is executed on the sever. It uses Apache Kafka as a basis to create the data channels. This technology offers fast data pipelines, which are fault tolerant, scalable and distributed. The generated code processes these data channels as specified using the DSL. Once a data channel is initialized and opened, it waits to receive information from the data sources. When data is received, these are inserted in the data channel. This process continues until the application is paused or stopped. All data channels are ultimately connected to a terminal Result node, which visualizes or persists the data as specified with the DSL.

**Dashboard.** The streaming applications are executed in a server, so that the user does not have direct access to them. To allow interaction, we have developed a graphical, browser-based dashboard, which is integrated in the web platform (see label 7). The developed interface (see also Fig. 5, label 4) contains controls to monitor the application status (running on/off), a counter with the received bytes in real time, and management controls to execute, pause and stop the application. The dashboard is specific to each project, and is configured dynamically according to the description made with the DSL. Given the model, the dashboard is able to search and identify the objects of type Result and add the corresponding widgets to the panel. For example, the available charts and tables in the DSL have a corresponding Javascript widget, which are added when the dashboard is opened. To receive information and communicate with the streaming applications, a socket is used.

**Example**. Fig. 5 shows several steps in defining and executing the running example. First, the user needs to create a personal account in our platform through the registration form (label 1). After logging into the platform, the user can access her personal workspace to list her projects (label 2) or to create a new one (label 3). They way of creating the running example project in the DSL editor was described in Secs. 2.1 and 2.2. Once the model is created and validated, the user can execute it by accessing the dashboard (label 4). Here, she can manage in real-time the streaming application generated, and monitor the data processed while the application is running. The application will continue to execute on the server even when the user browser is closed. The user can stop the application by accessing again the dashboard and pressing the “stop” button.



My projects

This is your personal workspace. You can create a new projects or select one of the list below.

Projects in your workspace:

| Name | Created | Last Update | Descripcion |
|---|---|---|---|
| Testing | 18/01/2018 15:21:48 | 18/01/2018 15:21:48 | |
| TwitterExample | 05/03/2018 13:23:03 | 05/03/2018 13:23:03 | This project connects to the twitter streaming api and filter by a keyword. |

| 3: Tweet |
|---|
| RT @Gidi_Traffic: "@Essence: #GetOut creator @JordanPeele just became the first Black screenwriter to win an #Oscar for 'Best Original Scr… |
| RT @ABC: Jimmy Kimmel: "I remember a time when the major studios didn't believe a woman or a minority could open a super hero movie—and the… |
| socorro https://t.co/8lFNQgVGzy |

**Fig. 5.** Steps followed to define and execute the running example.

## 4 Related Work

Stream processing has been around for decades (see [8] for a survey in the 90s), and stream programming languages, like StreamIt [10], already existed in the early 2000s. However, it is recently that we are witnessing the emergence of a plethora frameworks capable of stream processing, like Apache Kafka, Samza [12], Spark[7], Flink[8], Storm[9], or IBM Infosphere Streams [5]. This shows the relevance of stream processing nowadays. However, these frameworks rely on low-level programming, have generally a steep learning curve and are generally complex to configure and deploy.

Even though numerous systems have emerged, a recent survey on stream processing systems [2] concludes there is *“the need for high-level programming abstractions that enable developers to program and deploy stream processing applications on these emerging and highly distributed architecture more easily, while taking advantage of resource elasticity and fault tolerance”*. Hence, aiming for this goal we developed DATALYZER, so that streaming applications become easier to define, deploy and execute.

The integration of heterogeneous sources, both static and dynamic is also a concern to enable highly flexible, dynamic applications [9]. Classical Extract-Trasform-Load (ETL) systems have started to consider streaming data, as seen by commercial systems like Microsoft Data Factory[10] or MuleSoft[11]. DATALYZER is inspired by these approaches, but we explicitly focus on simplicity of use and extensibility. Our aim is to become open source, and by our compilation approach we are independent on the target execution platform, avoiding vendor lock-in. Compared to approaches to data integration like [4], our approach is novel in which it permits extra flexibility to define

[7] `https://spark.apache.org/`
[8] `https://flink.apache.org`
[9] `http://storm.apache.org/`
[10] `https://azure.microsoft.com/en-us/services/data-factory/`
[11] `https://www.mulesoft.com/`

data source types (like TCP or REST), data sources (like OpenWeatherMap) and data source instances (using OpenWeatherMap in an application) in an integrated way.

## 5 Conclusions and Future Work

In this paper, we have presented DATALYZER, a cloud-based approach for the creation of streaming data applications, based on a visual DSL. The approach permits a visual description of the application, which is then compiled, deployed and executed automatically. The approach is extensible, as new data sources can be added on the fly. Overall, the approach aims at lowering the entry barrier for the creation and execution of streaming data applications.

While DATALYZER is currently fully functional, it is still under development. First, we would like to extend its expressive power, with more advanced primitives like data matching conditions and triggers (in the style of Complex-Event Processing systems [6]), data transformers and analyser blocks, as well as predictive and machine learning primitives. We are currently performing stress and performance tests, which may lead to optimizations in the system architecture.

*Acknowledgements.* Work partially funded by the Spanish MINECO (TIN2014-52129-R) and the R&D programme of the Madrid Region (S2013/ICE-3006).

## References

1. M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Sw. Eng. Morgan & Claypool Publishers, 2017.
2. M. D. de Assunção, A. D. S. Veith, and R. Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *J. Network and Computer Applications*, 103:1–17, 2018.
3. N. Dindar, N. Tatbul, R. J. Miller, L. M. Haas, and I. Botan. Modeling the execution semantics of stream processing engines with SECRET. *VLDB J.*, 22(4):421–446, 2013.
4. A. Harth, C. A. Knoblock, S. Stadtmüller, R. Studer, and P. A. Szekely. On-the-fly integration of static and dynamic sources. In *COLD*, volume 1034 of *CEUR Workshop Procs.*, 2013.
5. M. Hirzel et al. IBM streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7, 2013.
6. D. C. Luckham. *The power of events - an introduction to complex event processing in distributed enterprise systems*. ACM, 2005.
7. L. Rettig, M. Khayati, P. Cudré-Mauroux, and M. Piórkowski. Online anomaly detection over big data streams. In *2015 IEEE Int. Conf. on Big Data*, pages 1113–1122. IEEE, 2015.
8. R. Stephens. A survey of stream processing. *Acta Inf.*, 34(7):491–541, 1997.
9. N. Tatbul. Streaming data integration: Challenges and opportunities. In *IEEE 26th Int. Conf. on Data Engineering Workshops (ICDEW 2010)*, pages 155–158, 2010.
10. W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proc. CC*, volume 2304 of *LNCS*, pages 179–196. Springer, 2002.
11. W3C. XML Path Language (XPath) 3.1. https://www.w3.org/TR/xpath-31/.
12. Z. Zhuang, T. Feng, Y. Pan, H. Ramachandra, and B. Sridharan. Effective multi-stream joining in Apache Samza framework. In *2016 IEEE Int. Con. on Big Data*, pages 267–274. IEEE Computer Society, 2016. See also `https://samza.apache.org/`.