



Network Traffic Analysis,  
Aggregation, Anomaly  
Detection and Prediction With  
Apache Kafka, Apache Spark  
and InfluxDB

/September 2017/

# CONTENT

- 3 Problem**
  - 3 Initial description
  - 3 Properties of the data flow
  - 4 Data processing requirements
- 5 Solution**
  - 5 High-level design
  - 6 Working prototype design
  - 8 Implementation
- 9 Results**
  - 9 Possible applications
  - 10 Application guidelines

At Bitworks we design and develop comprehensive software systems. Given a sophisticated app domain, we elaborate the most appropriate architecture for the problem by building a feasible proof of concept using reliable open-source components. The resulting solution offers maximum compliance with industry standards and meets reliability, performance, and security requirements.

# PROBLEM

## Initial description

One of our customers operates a large computer network. In order to maintain its healthy infrastructure, it is necessary to properly monitor all network activity, analyze traffic flows, and predict eventual problems.

However, even having been condensed by standardized statistical techniques ( [sFlow](#) ), the flow of the data is still too big and raw to store and analyze without preliminary processing.

We were assigned the task to design a scalable distributed system for aggregating, enriching, and analysing large streams of network data.

Patterns discovered in packet streams can form the basis for future network redesign and hardware upgrades, while abnormal changes in packet distribution may indicate hardware failure or malicious behaviour and thus demand a timely response

## Properties of the data flow

**While designing the system, we accepted the following assumptions about the data:**

- Data flow is continuous.
- Data is time-based. Each sample has a timestamp and samples are produced in the order corresponding to the timestamps.
- Data is raw. Part of its fields can be safely discarded for some analysis problems, while some should be enriched with external data.
- Data has many fields, both dimensional and metric. This allows the data to be partitioned and subsequently aggregated by some of its field values.
- Data is sampled. Statistical nature of the data and of the target metrics makes occasional reprocessing of the same values not an issue in the long run.

## Data processing requirements

**Given the data as described above, the system needs to process it in a time window as follows:**

- Discard unnecessary data fields.
- Rename data fields as needed.
- Create new data fields by enriching, performing basic operations (addition, subtraction, multiplication, division) on other fields, and naming the result.
- Aggregate metric data fields by some key dimensions with one of the aggregation operations (max/min, sum, count, unique count etc).

The resulting data is usually more compact and so it is easier to store. Nevertheless, it is still time-based and multidimensional, so it needs a specific type of database for storing and querying.

**The database needs to be optimized for the following operations:**

- Inserting records for the recent period of time (the order of the inserts might not necessarily correspond to the order of the record timestamps).
- Reading ordered records for a continuous period of time, filtered by one or more dimension fields.
- Deleting records for a continuous period of time after the expiration of the retention period.

**The processed data needs to be analysed in order to:**

- Detect and report anomalies and patterns.
- Predict trends.

Analysis should be performed alongside the data flow by making use of the historical data from the database. Specific analysis techniques to be performed are not known in advance, so the system needs to support user-defined analysis modules.

In addition, a visual analysis tool is needed for graphical display of the historical data.

The resulting system must be scalable and easily deployable.

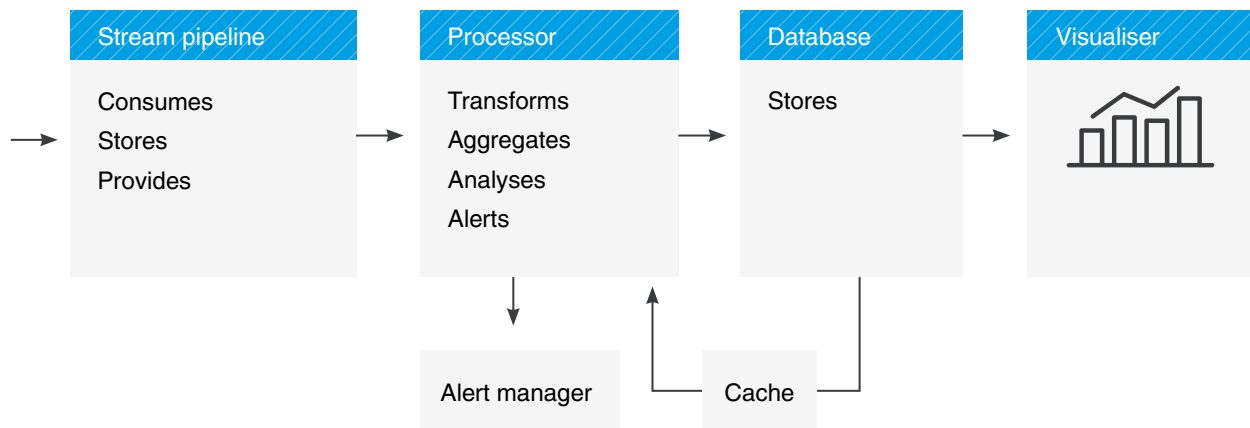
# SOLUTION

## High-level design

In accordance with the requirements we needed the following components for our system:

- Streaming platform for sequencing and partitioning incoming data.
- Stream processing platform for performing transformation and analysis.
- Time series database for storing resulting data and providing data access for analysis.
- Visual monitoring tool.
- Alert delivery system.

We designed the following high level architecture.

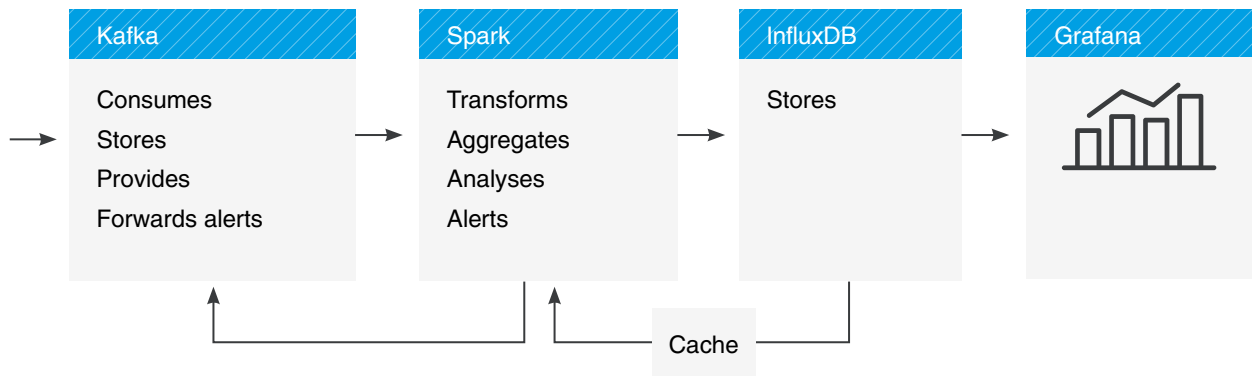


## Working prototype design







**We decided to build a working prototype to test if the architecture would meet our client's needs.**

**For the individual components we have chosen as follows:**

- **Python 3 as the development language.**  
🔗 [Python 3](#) is an open-source programming language with large contribution from data science community. It is one of the few languages supported by Spark, along with Java and Scala. This choice was determined by customer's preferences.
- **Kafka for pipelining initial streams of data.**  
🔗 [Apache Kafka](#) is an open-source distributed streaming platform often used in tandem with Spark. It collects events from multiple sources, stores them in a fault-tolerant way and guarantees at-least-once processing of the data. It supports multiple independent consumers and provides means for flexible scalable configuration of data partitions in relation to the structure of possible data sources and data consumers.
- **Spark for processing data.**  
🔗 [Apache Spark](#) is an open-source distributed data processing platform which has become de facto standard for solving big data problems. It provides a unified model for processing data sets and data streams in a similar manner. Its API is available in the Python programming language and has methods for transforming and aggregating data as well as a large library of machine learning algorithms for data analysis.
- **InfluxDB for storing data.** 🔗 [InfluxDB](#) is an open-source distributed database for storing time series. It is highly optimized for storing and accessing multidimensional time-based data, it surpasses many of its competitors in terms of performance, and is currently ranked #1 by popularity among other time series databases according to 🔗 [DB-Engines Ranking](#).
- **Grafana for visualizing data.** 🔗 [Grafana](#) is an open-source monitoring and analytics platform which provides a variety of ways to visually present time-based data, including many third-party plugins. It has built-in support for InfluxDB (with interactive query builder and other features), as well as for many other data sources.
- **Docker for the application deployment.**  
🔗 [Docker](#) is an open-source platform for building, shipping and running distributed applications. Docker allows flexible configuration for multicomponent systems, and guarantees correct execution regardless of the deployment target. InfluxDB and Grafana have official docker images, while docker image for Kafka is supported by the community.
- **Kafka for alert forwarding.**



We already had experience working with these components in different combinations, so we decided to test their feasibility for the architecture above.

Component	License
 Apache Kafka – streaming platform	<a href="#">Apache License 2.0</a>
 Apache Spark – distributed data processing platform	<a href="#">Apache License 2.0</a>
 InfluxDB – time series database	<a href="#">MIT License</a> for manual deployment <a href="#">ToS</a> for SaaS <a href="#">Software License Subscription Agreement</a> for SaaS and/or scalability
 Grafana – monitoring and analysis platform	<a href="#">Apache License 2.0</a> for manual deployment <a href="#">ToS</a> for SaaS
 Docker – deploying platform	<a href="#">Apache License 2.0</a>
 Python – programming language	<a href="#">License</a>

## Implementation

**We developed a Python application which runs on a Spark cluster.**

First and foremost it connects to the Kafka server as a consumer using options provided by a configuration file.

**Then it processes every batch of the data sent by Kafka in two steps:**

1. It transforms the data rows:

- Selects data fields relevant to the problem at hand.
- Renames data fields if necessary.
- Performs basic arithmetic operations (addition, subtraction, multiplication, division).
- Performs GeoIP transformations (country, city, ASN).

Nested transformations and user-defined transformation modules are not supported in the current prototype.

2. It aggregates data fields using one of the following aggregation functions:

- Sum, returns sum of all values.
- Mult, returns product of all values.
- Max, returns the largest value.
- Min, returns the smallest value.

Aggregation is possible after grouping data rows by some key field (or a combination of fields), as well as for the whole data batch.

Specifics of the transformation and aggregation steps are not predefined and can be specified in the configuration file by the DSL.

The processed data is then stored in InfluxDB.

The data is analyzed by built-in as well as by user-defined analysis modules. The modules query historical data from InfluxDB, analyze it for deviations, and send out notifications when anomalies are detected.

The current prototype allows sending alerts to a separate Kafka instance and/or topic.

### GitHub

An elaborate description of the prototype design is available at [GitHub repository](#).

Every user-defined module is a Python 3 package containing a class implementing the module interface. The package should be supplied alongside the application and its name and initialization parameters should be specified in the configuration file. A single module can be specified more than once.



# RESULTS

Guided by the high-level architecture and our choice of components, we developed a working prototype, which has proved the approach to be viable for quantitative and statistical analysis of network traffic in sFlow format. The prototype is open-source under [Apache License v2](#) and is available for everyone at [GitHub](#). It can be used as is for analyzing any kinds of CSV data streams in terms of the operations described above.

## Possible applications

**We believe that a similar approach might be applied for other problems where stream-oriented aggregation and analysis are needed. Possible fields of application include but are not limited to:**



### Telecommunications

- Network load monitoring
- DDoS attack detection and prediction



### Devops

- Hardware monitoring
- Log monitoring



### Finances

- Stock market monitoring
- Portfolio tracking



### Transport

- Traffic control
- Taxi order service



### Internet of things

- Climate control
- Smart city
- Manufacturing and production operations



### Software development

- Application usage
- Malware detection

Viability of the architecture for specific problems in the aforementioned domains should be verified on a case-by-case basis. If you want to try it for yourself, we would really appreciate your feedback. Following are some guidelines to help you decide if it is worth your effort.

## Application guidelines

### The proposed architecture will probably be viable for big data problems where:

- There are many automated events in time (counter readings, data packages, messages, etc.) that require to be monitored and analyzed.
- The data is time-based and has a distinct partitioning by certain criteria.
- The data needs to be transformed before being processed and stored.
- It is necessary to detect unexpected behaviour, to discover trends, and to predict future states of the data.
- The data needs to be aggregated, in order for the data storage to avoid additional scaling.
- Apache Spark is already a part of the solution.
- The computations to be performed are tolerant to occasional duplicate values in the case of a hardware failure.

### The architecture will not be viable if:

- Real-time monitoring with low latency is required. The use of the Spark Streaming framework enforces batching of the events, so that the response time cannot be less than the batch window.
- The problem needs an exact numerical solution. In the case of hardware failure some data may be processed more than once, which would lead to incorrect results. Additional measures need to be taken to ensure idempotence of the data operations, both at the consuming and the producing ends of the application. [Kafka 0.11](#) provides means to ensure exactly-once processing for both its producers and consumers. The system will also need to ensure idempotence of inserting the transformed data into the data storage.
- The data is not big. It might not need aggregating and scaling capabilities provided by the architecture, and a simpler solution might suffice.
- Complicated analysis is not needed. InfluxDB provides basic facilities for transforming and aggregating data, while [Kapacitor](#) can be used for running simple anomaly detection checks. Grafana can still be used for online data monitoring.

Additional measures need to be taken to ensure idempotence of the data operations, both at the consuming and the producing ends of the application. [Kafka 0.11](#) provides means to ensure exactly-once processing for both its producers and consumers. The system will also need to ensure idempotence of inserting the transformed data into the data storage.

InfluxDB with Grafana might be a better combination for this case.

# CONTRIBUTORS

## **Sergey Krickiy**

Development team leader

## **Denis Ryabokon**

Developer

## **Nikolai Bogoslovskiy**

Developer

## **Vyacheslav Podberezhnny**

Project manager

## **Valentin Rakhimov**

System analyst

13A Shishkova St,  
Tomsk, 634050, Russia

+7 (382) 2 70 54 77

30 Regent St,  
Jersey City, NJ 07302, USA

+1 929 402 8251

Hamburg, Germany

+49 (040) 5 48 91 029

[info@bw-sw.com](mailto:info@bw-sw.com)

<https://bitworks.software>