

**Project:   
The MEB-POC manufacturing system**

|  |  |
| --- | --- |
|  |  |

Deliverable D2 Template (v2.0)

|  |  |
| --- | --- |
| **Date** | 16/01/2019 |
| **Team ID** | TFLS |

|  |  |  |
| --- | --- | --- |
| **Team Members** | | |
| **Name and Surname** | **Matriculation number** | **E-mail address** |
| Tiziano Santilli | *261502* | *tizianosantilli@gmail.com* |
| Fabio Di Silvestro | *260651* | *fabio.disilv@gmail.com* |
| Sahil Babel | *261315* | *sahilbabel3@gmail.com* |
| Leonardo Marrancone | *260789* | *leonardo.marrancone@student.univaq.it* |

Table of Contents

[Challenges/Risk Analysis 3](#_Toc367493)

[Requirements Refinement 4](#_Toc367494)

[Functional Requirements 4](#_Toc367495)

[Non-Functional Requirements 4](#_Toc367496)

[Use Case Diagram 5](#_Toc367497)

[Informal Description of the system and its Software/System Architecture 6](#_Toc367498)

[Informal System Diagram 6](#_Toc367499)

[System Behavior and Considerations 7](#_Toc367500)

[System Space Diagram 8](#_Toc367501)

[Requirements Achievement 8](#_Toc367502)

[Design Decisions 9](#_Toc367503)

[Views and Viewpoints 14](#_Toc367504)

[UML Static and Dynamic Architecture View 15](#_Toc367505)

[CAPS Architecture View 19](#_Toc367506)

[CAPS Model 20](#_Toc367507)

[From Architecture to Code 21](#_Toc367508)

[Functional Requirements 21](#_Toc367509)

[Non-Functional Requirements 22](#_Toc367510)

[Summary 23](#_Toc367511)

# Challenges/Risk Analysis

In the table below, we wrote the most challenging, discussed and risky decision we faced during the development of the project. Thanks to the studies performed to write the code, we finally solve some open challenges (e.g. the use of the analytic engine). We found a way to let Apache Spark and Solace communicate (using the MQTT protocol) so we achieve a high level of performance and fault tolerance.

|  |  |  |  |
| --- | --- | --- | --- |
| **Risk** | **Date the risk is identified** | **Date the risk is resolved** | **Explanation on how the risk has been managed** |
| **Use or not use DB** | 29/11/2018 | 03/12/2018 | We decided to use a message broker and not to pull data from DB. The broker is faster because the events are directly intercepted from the tools (we don’t have to wait that the messages are stored in the DB). Using a broker, we don’t rely on a DB (so we avoid a point of possible failure). |
| **Architectural choice** | 30/11/2018 | 03/12/2018 | We decided to design a publish/subscribe system so that we can exploit all the benefits of this architecture. |
| **~~Use of Kafka as a message broker~~** | 30/11/2018 | 03/12/2018 | ~~We decided to use Apache Kafka instead of other message broker because is highly scalable, reliable and high performance.~~ |
| **Use of Solace as a message broker** | 17/12/2018 | 20/12/2018 | After the meeting we realize that for this project Solace is the better option like message broker. By the way we identify the risk about the communication between Solace and Apache Spark (analytics engine) as written above. The free version of Solace has limited capacity, but anyway it fits for our data amount. |
| **~~Use an~~ *~~analytics engine~~***  **~~(Apache Spark)~~** | ~~05/12/2018~~ | ~~07/12/2018~~ | ~~We decided to use an~~ *~~analytics engine~~* ~~as processor to make the operations faster. We will use Apache Spark. We identify a risk about the communication between Solace (message broker) and Apache Spark (\*).~~ |
| **Use an *analytics engine***  **(Apache Spark)** | 05/12/2018 | 21/01/2019 | We decided to use an *analytics engine* as processor to make the operations faster. We will use Apache Spark. We identify a risk about the communication between Solace (message broker) and Apache Spark. The risk is resolved because we wrote a custom receiver that allows to connect Spark on a durable queue of Solace by using the MQTT Protocol. |
| **Use of ElasticSearch as NoSQL Database** | 05/12/2018 | 10/12/2018 | We decided to use a NoSQL database since we think that the natively features of this technology fit better for the problem as described in the QoC Template. |

# Requirements Refinement

Starting from the specifications paper, we analyzed the requests and refined them as follow. In order to keep well separate the functional and non-functional requirements, we made two different sections.

After the requirements section, there is the Use case diagram that summarize the functionalities and the interactions among the system and the actors.

## Functional Requirements

The requirements are ordered by priority descending:

1. The system has to manage events intercepted from tools.
2. The system has to compute these events and gets additional information from the raw\_data database.
3. The system has to provide a user-friendly dashboard.
4. The system has to store the computed data for querying.
5. The dashboard needs to provide a complete system of filtering.

## Non-Functional Requirements

The requirements are ordered by priority descending:

1. The system must be without a single point of failure.
2. The system must manage peaks of 160000 messages every 30 minutes and must scale.
3. The system must be up 99.999%
4. The system needs to take no more than 5 minutes from the time the message is broadcasted from the tool to the time it is stored in the analytics database. However, the lower the delay the better.

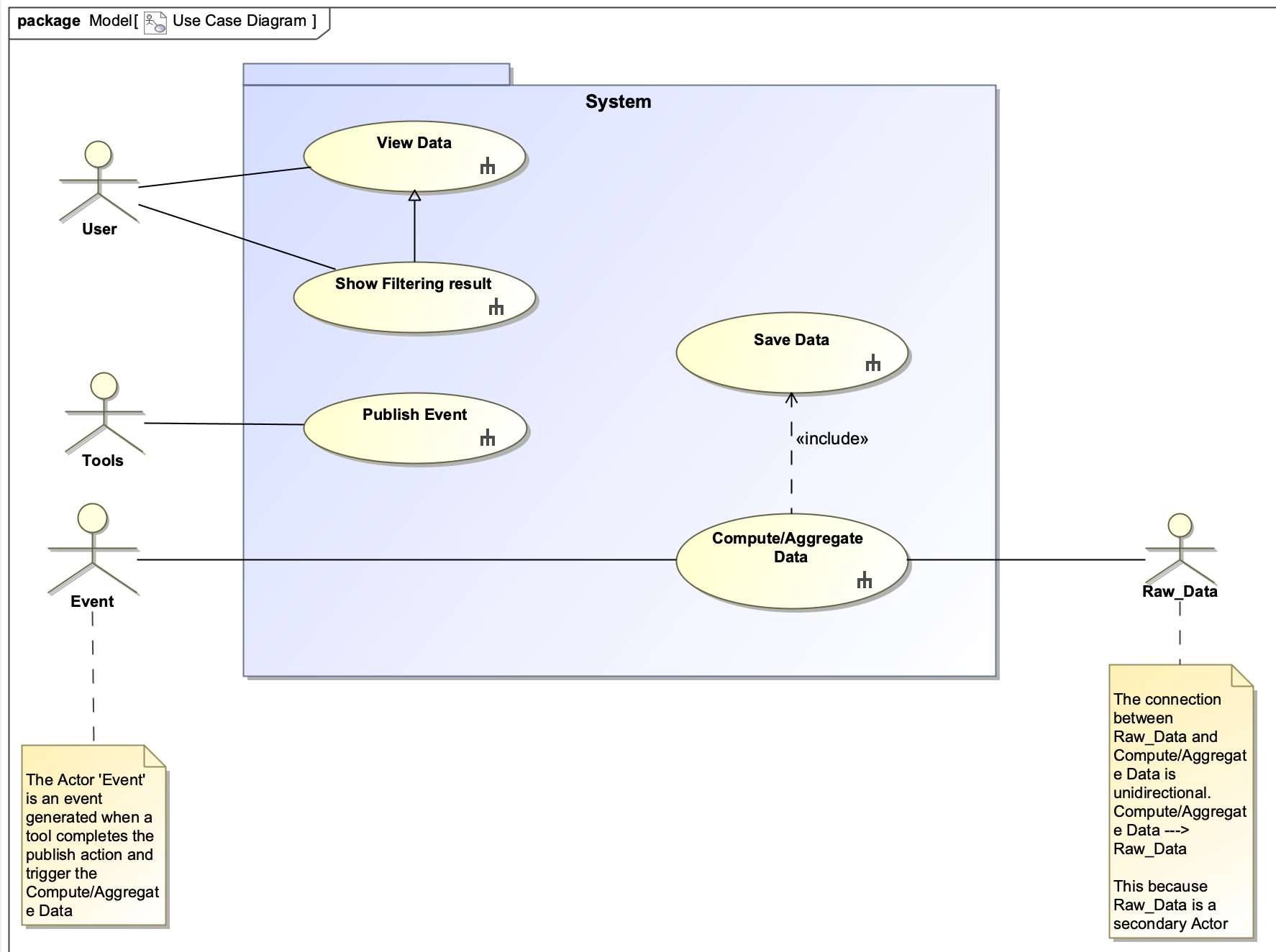
## Use Case Diagram

In our use case diagram, we can see that the user interacts with the use case “View Data” and its specialization “Show Filtering result”.

We also have an actor called “Tools” that interacts with the “Publish Event” Use Case.

The actor “Event” triggers the functionality “Compute/Aggregate Data” involving “Raw\_Data” as a secondary actor. We placed the actor “Event” in order to show how the computation of the final data is triggered. We decided this option because in our opinion is the clearest way to explain the functionality.

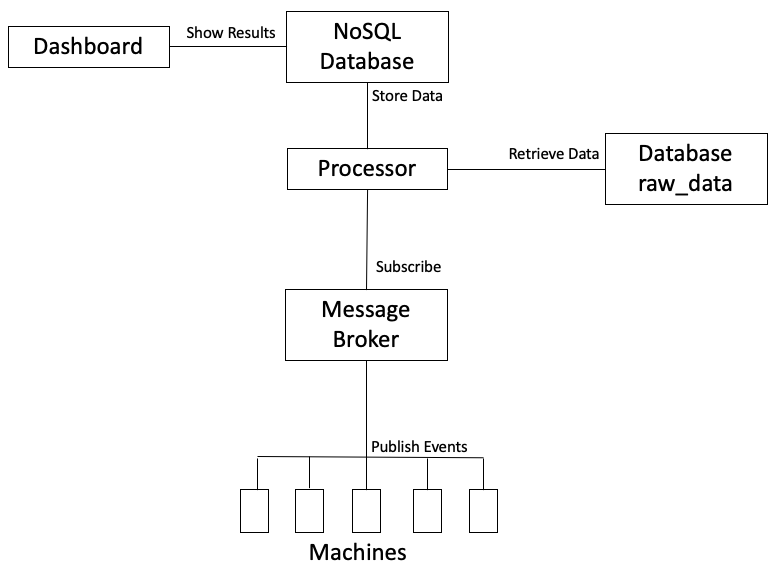
The “Compute/Aggregate Data” include the use case “Save Data”.



# Informal Description of the system and its Software/System Architecture

## Informal System Diagram

Below, an informal draw of the system. The goal of the diagram is to explain the software topology and the cohesion between the chosen technologies. There is a brief description of the interactions too.



Connection managed by HAProxy

## System Behavior and Considerations

Our system intercepts the machines XML events by using a message broker

(~~Apache Kafka~~) (Solace) and the MQTT protocol. The processor (Analytics engine: Apache Spark) subscribe the events topic from the message broker, at 1 second fixed intervals joins the arrived events with the data coming from the raw\_data database (opportunely queried) and finally stores the result data in a NoSQL (ElasticSearch) database. A dashboard (Kibana) is attached to the NoSQL database and used to show the data to the end user. Kibana offers the possibility to query the NoSQL index in a user-friendly and graphical way, so that the user can easily build its own search form. The technologies used ensures a near real time performance.

We used Solace in HA (high availability) mode, it means that we have three instances of solace running at the same time:

* Primary node: the active instance of Solace that manage the data.
* Backup node: idle instance of Solace that becomes primary in case the primary node fails.
* Monitoring node: instance of Solace in charge of coordinate the switch between primary and backup node in case of faults.

The connection between the tools and the message broker and between Spark and Solace are managed by HAProxy, the use of a proxy software is transparent to the clients and offers the possibility to connect to a single IP Address. HAProxy automatically switches between the primary and the backup instance of Solace in case of faults.

Since we use a publish/subscribe pattern, our system has these two features directly inherited from the architectural pattern:

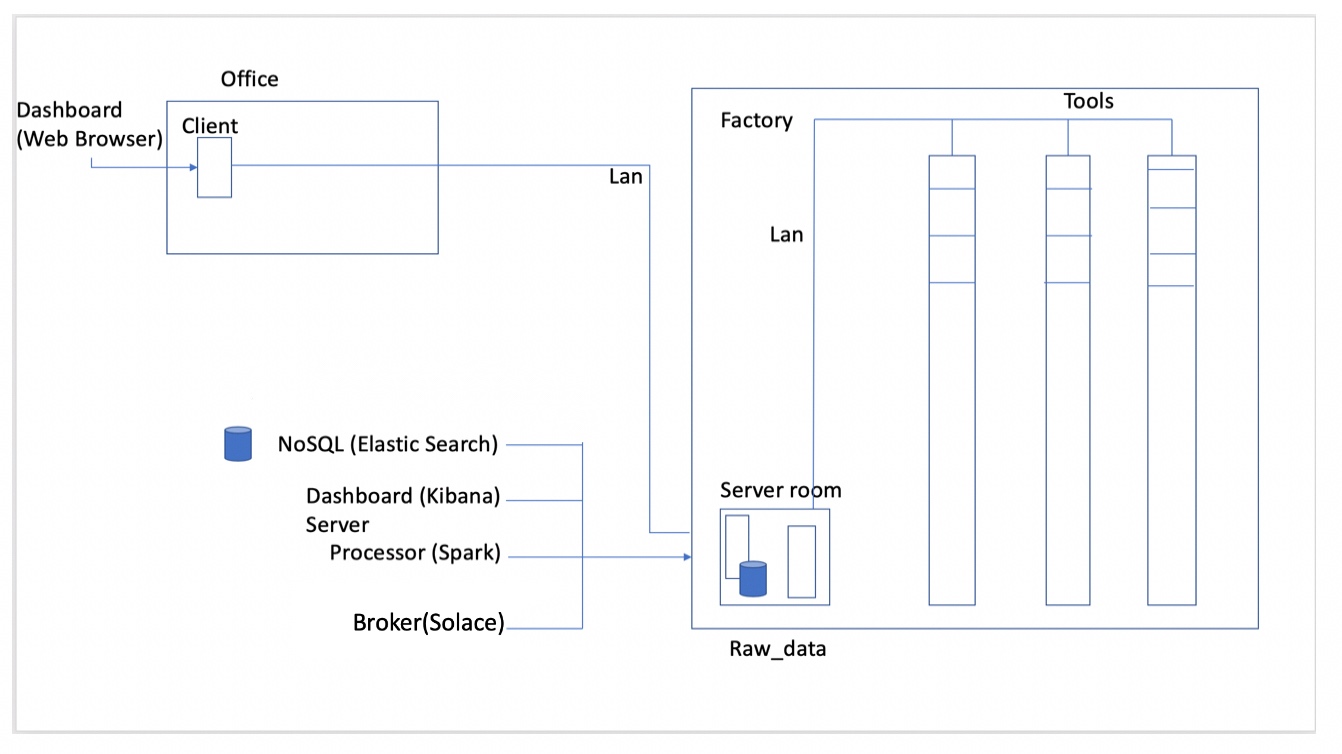
* Scalability.
* Possibility to change dynamically the network topology.

Our system can be divided in three main parts:

* External Part: includes the tools that publishes data into the broker and the provided database raw\_data.
* Processor: Apache Spark is in charge to join the data coming from the tools with the raw\_data database. The processor must also store the computed data in the NoSQL database.
* Dashboard: retrieve data from the database and shows a user-friendly dashboard.

## System Space Diagram

This figure depicts the environment where the system is going to be deployed.

**

## Requirements Achievement

Here below the list of how the functional and non-functional requirements are satisfied:

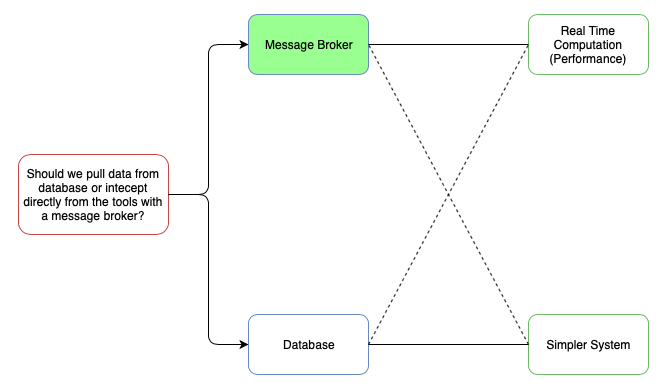
* ~~To Manage events intercepted from tools we use Apache Kafka~~
* To Manage events intercepted from tools we use Solace.
* To compute these events and get additional information from the raw\_data database we use the analytic engine Apache Spark.
* To provide a complete and user-friendly dashboard we use Kibana.
* The filtering and searching operation are managed by Kibana too.
* The system does not have a single point of failure because all the technologies that we are going to use provide distributed replicas.
* The system can easily manage the amount of data because all the tools are built for handling big data (note that we are going to use the limited free version of Solace - [https://solace.com/products/software](https://solace.com/prodicts/software)).
* To be up at 99.999% we use technologies that provide fault tolerant features.
* The use of technologies like NoSQL (Elasticsearch) and a message broker (Solace) (~~Kafka~~), ensure great performance and scalability. Data are managed in near real time.

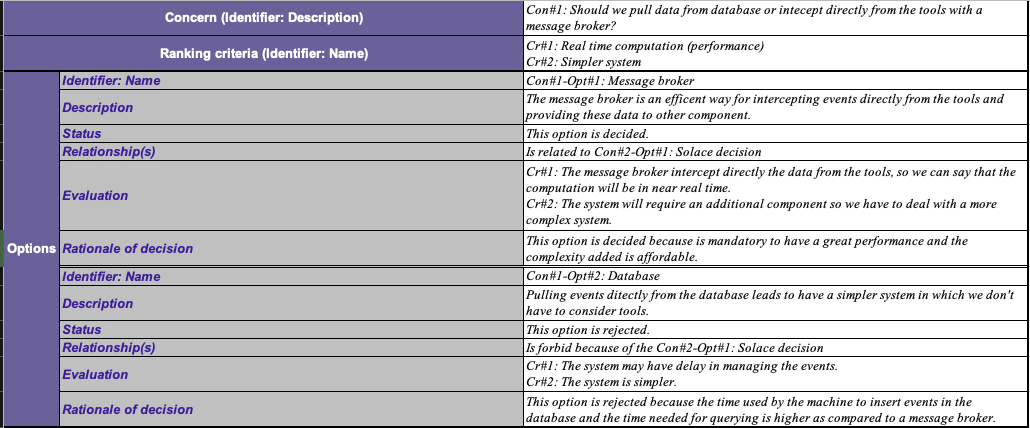
# Design Decisions

In this section there is the list of the most important design decisions taken by the team.

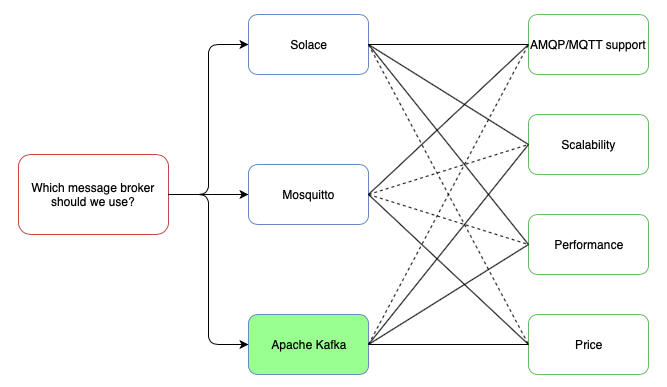
For a better visualization of the tables, our advice is to refer to the AK\_SPAM.xls.

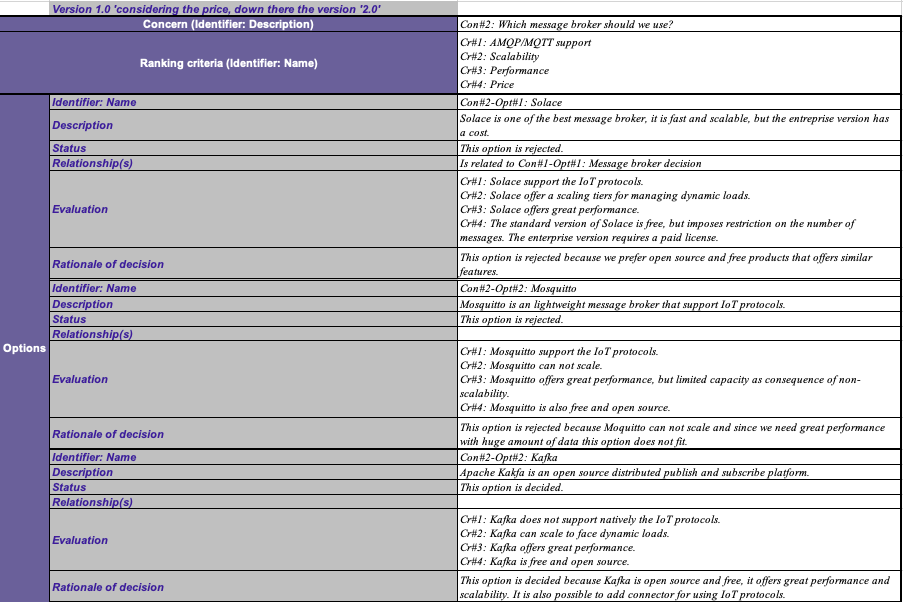
1. Database VS. Message Broker



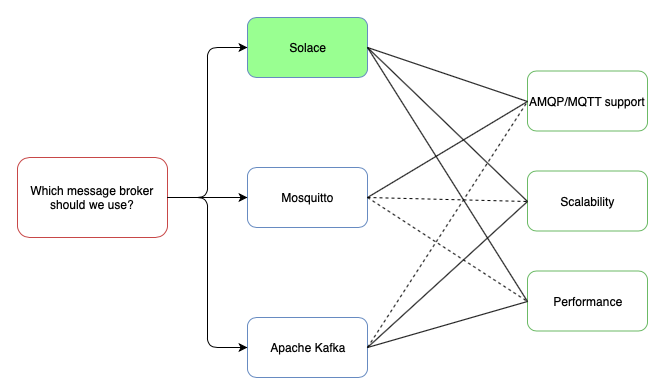


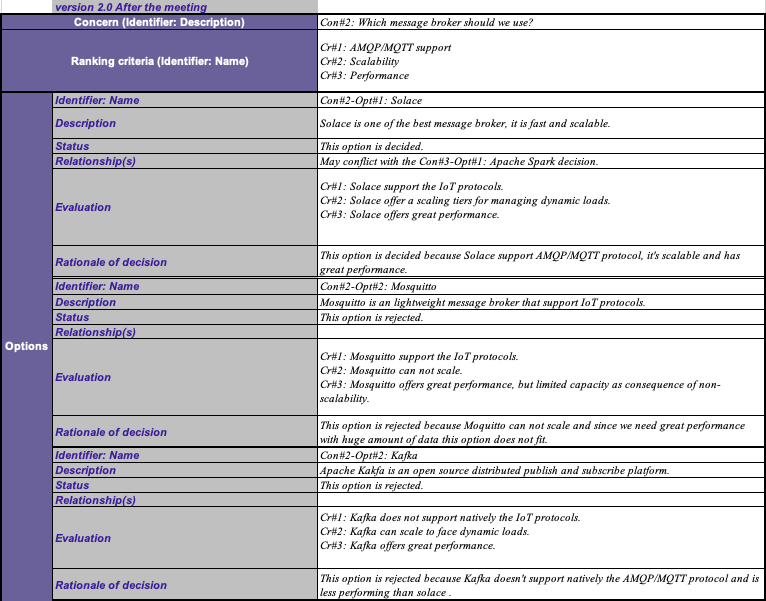
1. Solace VS. Mosquitto VS. Apache Kafka



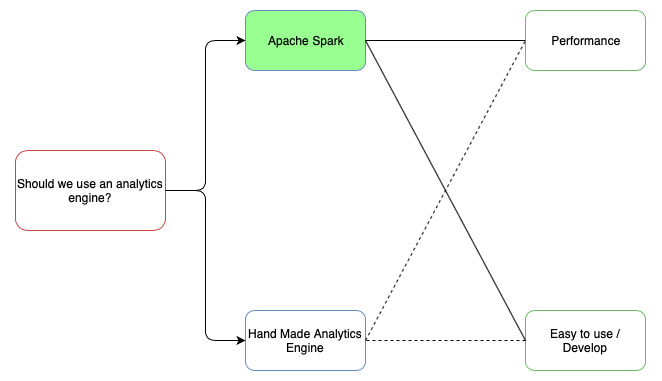


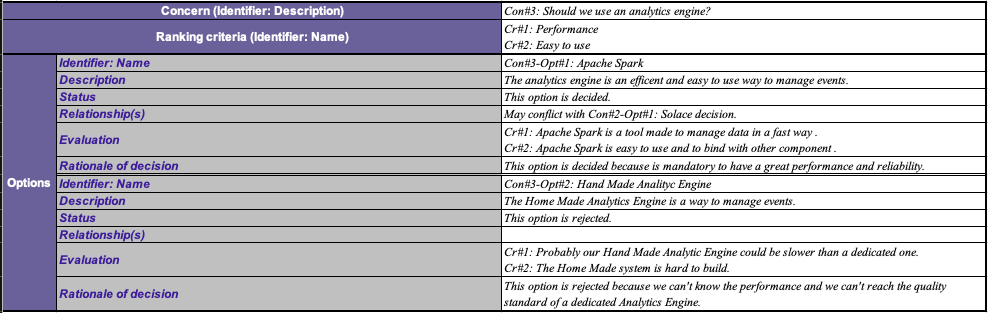
1. Solace VS. Mosquitto VS. Apache Kafka (After Meeting)

****

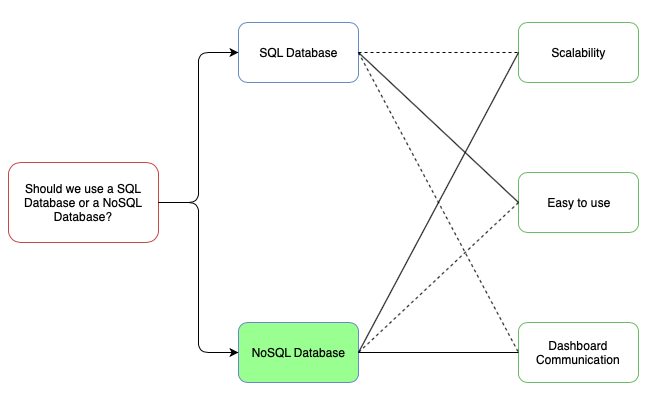


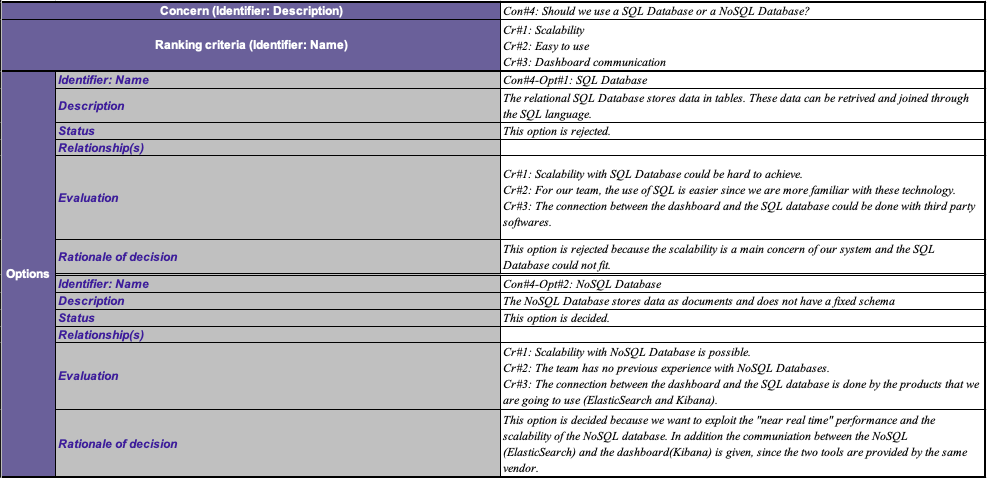
1. Apache Spark VS Homemade Processor





1. SQL Database VS. NoSQL Database





# Views and Viewpoints

The following table shows the links between the stakeholders and the different aspects/concerns of the system.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | End User | Network Administrator | Hardware Responsible | Database Administrator | System Developer | Architect | Administrative Office |
| Dependability |  | X | X | X | X | X |  |
| Scalability |  |  |  | X | X | X |  |
| Fault Tolerance |  | X | X | X | X | X |  |
| Networking & Communication |  | X | X |  |  |  |  |
| Usability | X |  |  |  | X | X |  |
| Performance |  | X | X | X | X | X |  |
| Security |  | X |  | X | X | X |  |
| Cost |  |  |  |  |  | X | X |

## UML Static and Dynamic Architecture View

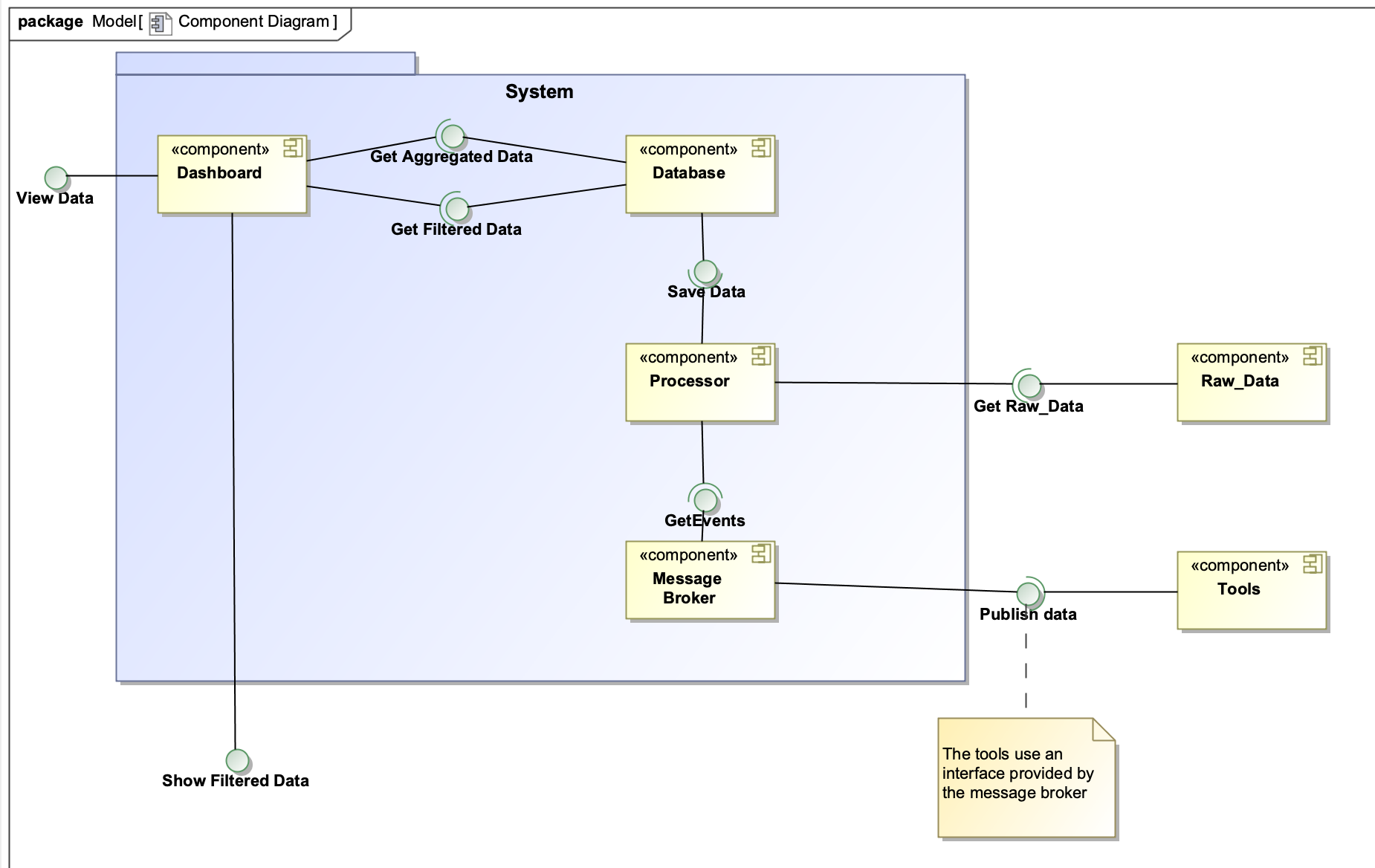
**Component Diagram**

The Component Diagram that we designed has two main part: the first is our system and the second is External (the pre-existent system).

The external part is composed by the component ‘Tools’ that use an interface provided by the message broker for publishing events and the component ‘Raw\_Data’ that provide an interface used by the processor to aggregate the data.

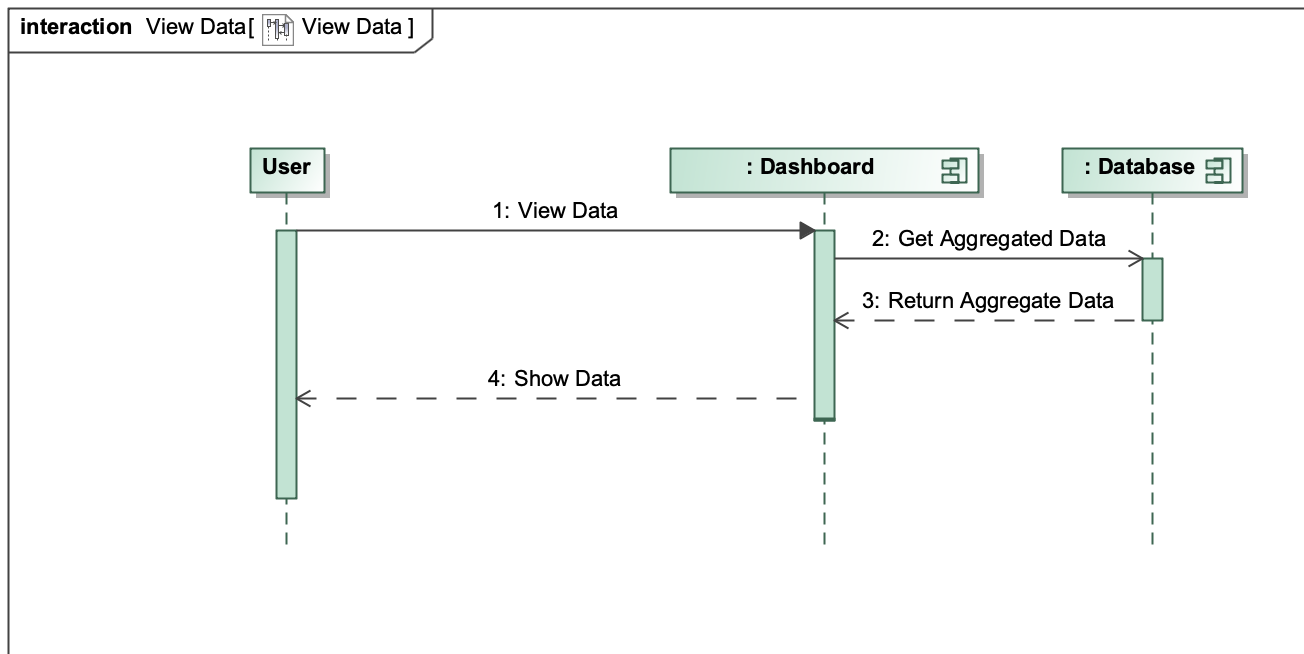
The part of our system has four components:

* Dashboard: used by the end-user, provides two interfaces to show the data and to allow the filtering of the data.
* Database: this component is used to store our aggregate data. It provides the interfaces “Get Filtered Data” and “Get Aggregate Data” that are used by the Dashboard Component to retrieve the data and build the user interface.
* Processor: this component is responsible to merge the data dispatched by the message broker with the data provided by the Raw\_Data database.
* Message Broker: this component provides an interface where the tools can send (publish) their data. Another interface is provided to allow the processor the gathering of the events.

**

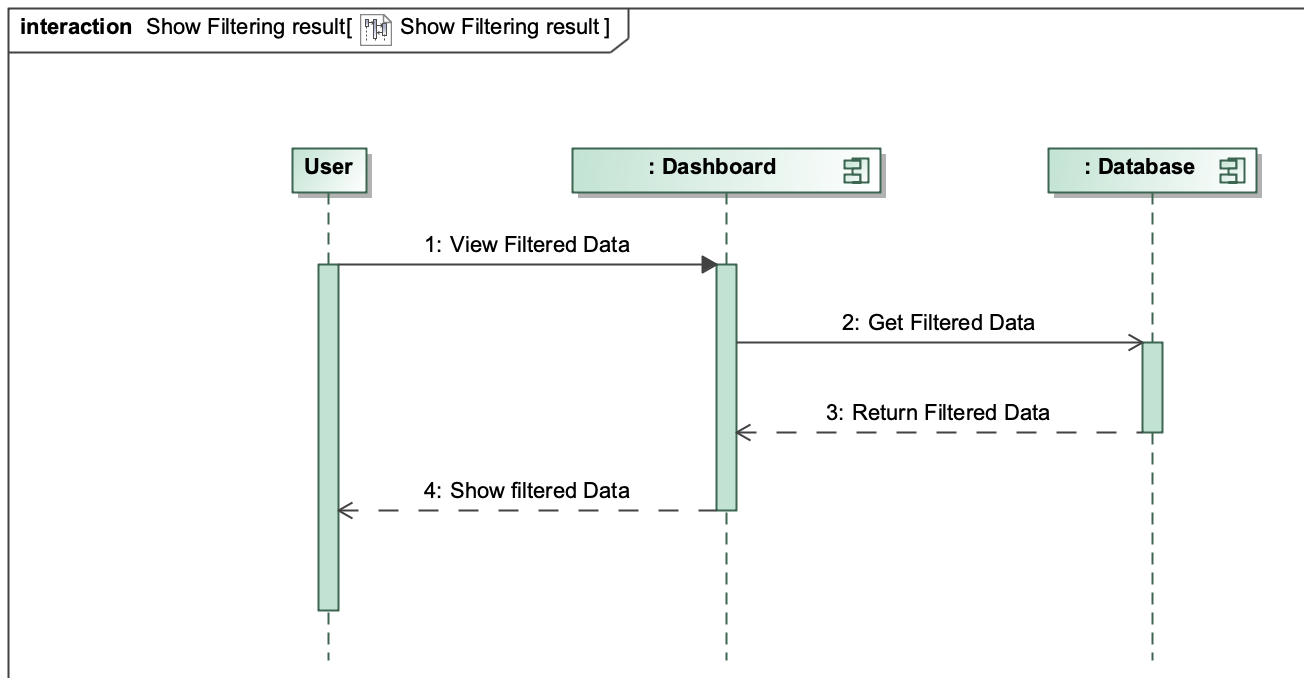
**Sequence Diagram: View Data**

This sequence diagram describes the View Data functionality, the user asks for the data to the dashboard that retrieve information from the Database component and then shows the results.

**

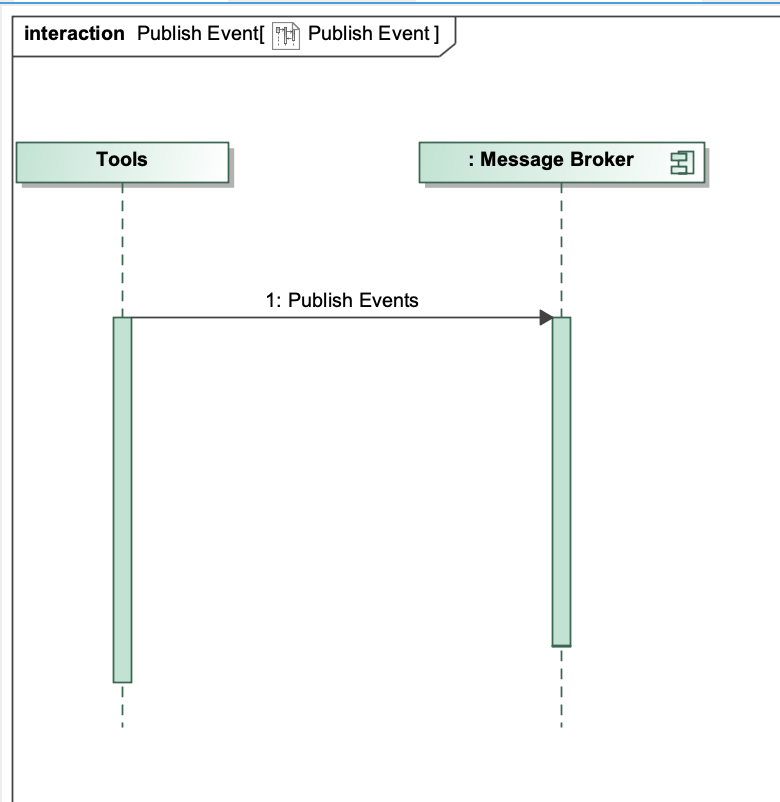
**Sequence Diagram: Show Filtering Result**

This sequence diagram describes the Show Filtering Result functionality, as before the user asks for specific data to the dashboard that retrieve information from the Database and then show the result.

**

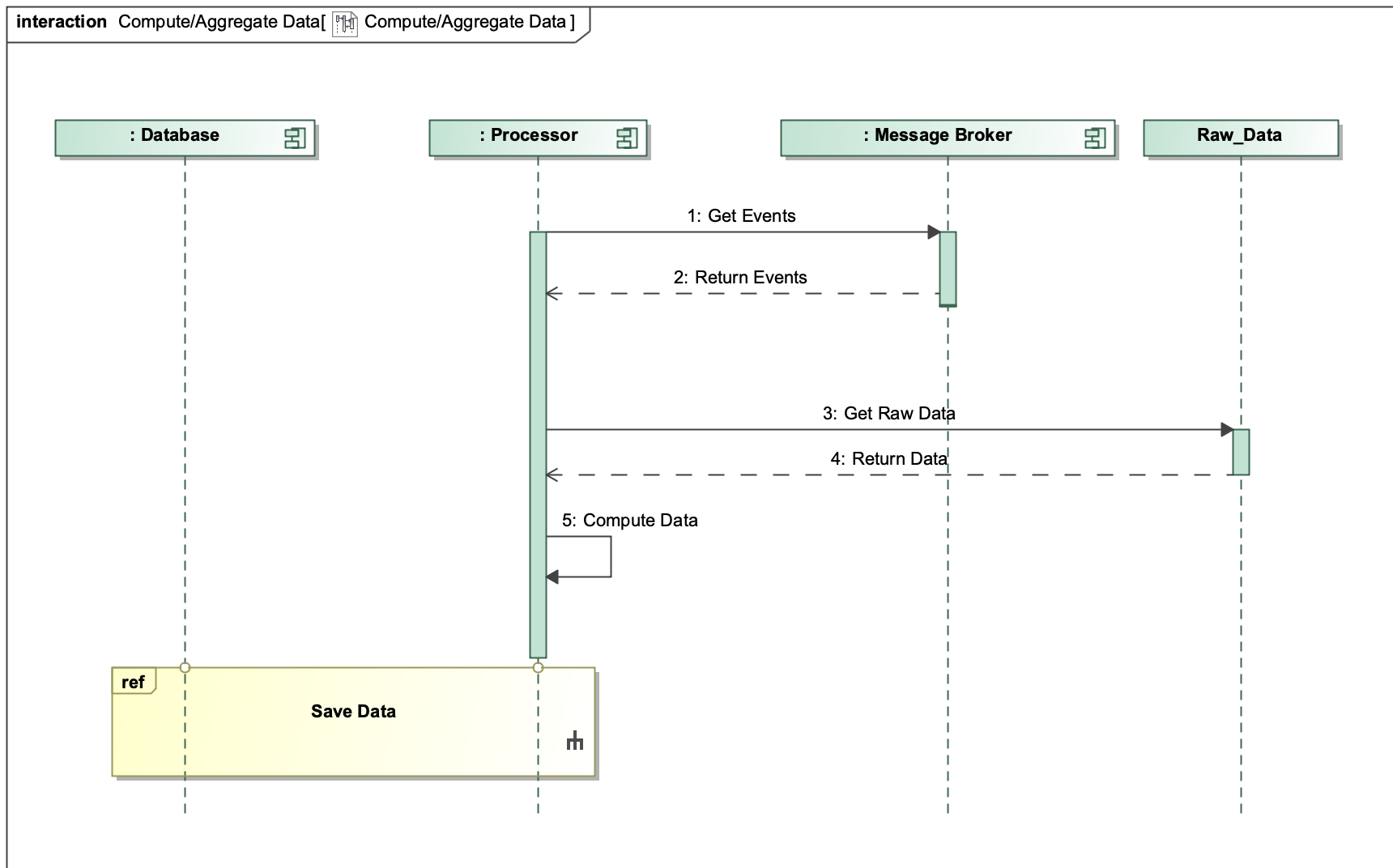
**Sequence Diagram: Publish Event**

This simple sequence diagram shows how the tools interact with the message broker.

**

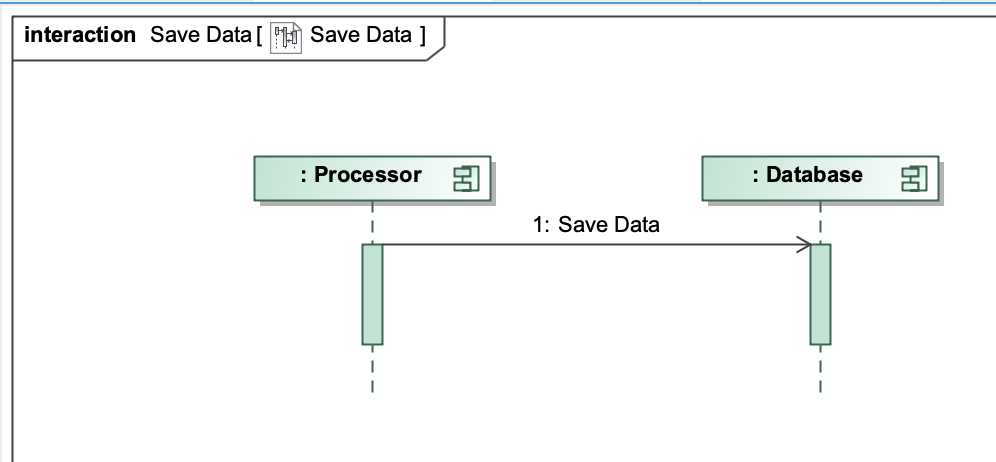
**Sequence Diagram: Compute/Aggregate Data**

The Compute/Aggregate Data sequence diagram shows how the system aggregates the data coming from the tools with the raw\_data information. The processor gets the event from the message broker, then the additional information from the raw\_data database. Finally, we have a reference to the Save Data sequence diagram to store the final data.

**

**Sequence Diagram: Save Data**

The Save Data sequence diagram is called every time that the Compute/Aggregate Data works, it saves the complete event information to the database.

**

## CAPS Architecture View

**CAPS SAML**

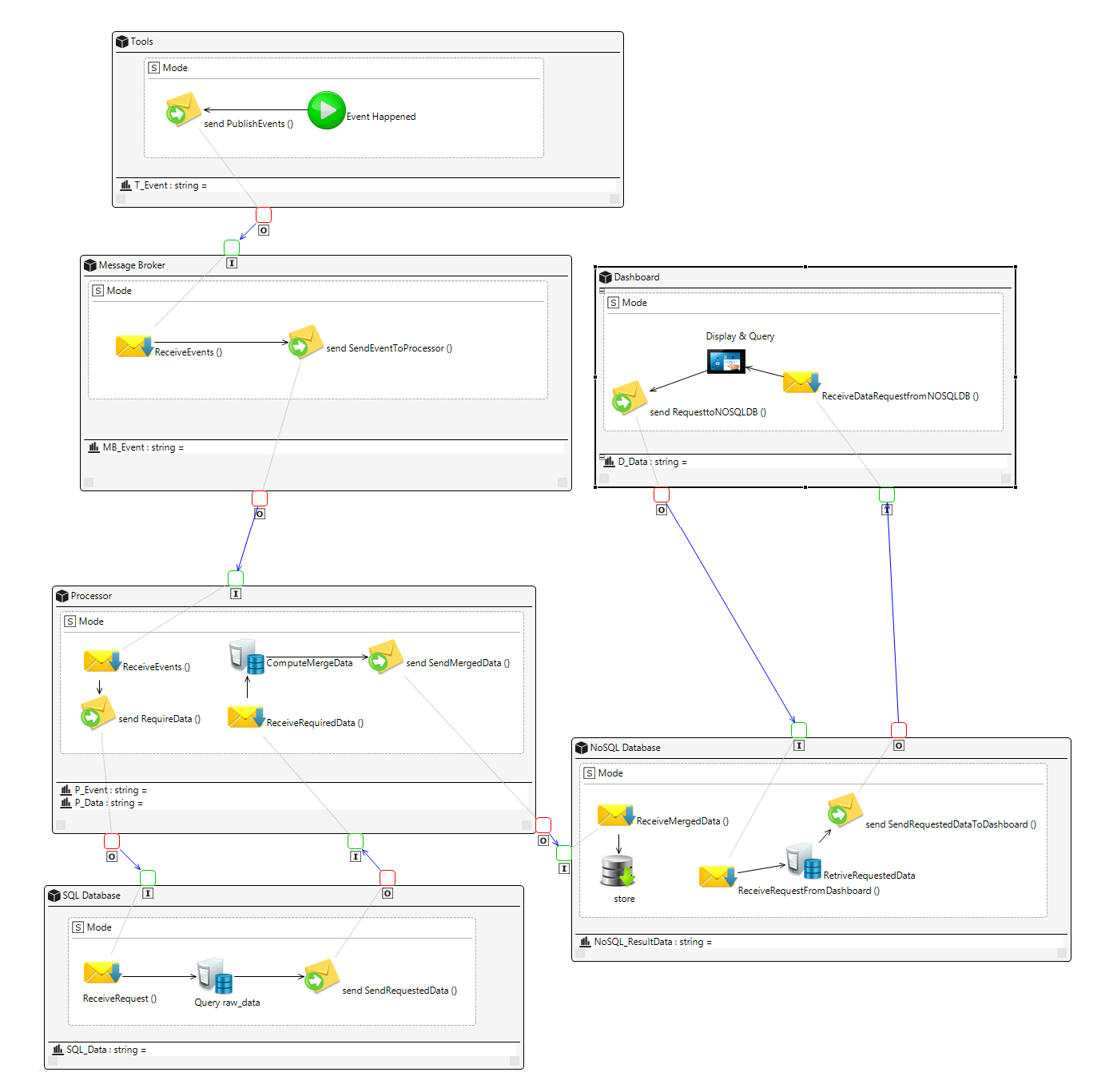
**CAPS Design Decisions**

The CAPS model reflects the system architecture described also through the UML Diagrams. In particular we have six components:

* Tools: single mode, the “Event Happened” trigger the publish of the events, represented by the unicast send message.
* Message Broker: single mode, it receives the message from the tool and alerts/forwards it to the processor.
* Processor: single mode, it receives the message from the message broker and immediately sends a request for the additional data to the SQL Database. Once the response arrives, the processor merges the data for computing the final results. Finally, the computed data are sent to the NoSQL Database.
* SQL Database: single mode, is in charge to receive the requests from the processor and provide the additional information.
* NoSQL Database: single mode, it receives the final data and stores them. The NoSQL Database remains ready for serving the requests coming from the dashboard.
* Dashboard: single mode, sends requests to the NoSQL Database who replies with the data that later are showed in the Dashboard display for the user.

Here below the CAPS Model:

## CAPS Model



# From Architecture to Code

We implemented all the decision that we took in the first part, that means that we have all these components and services:

* Solace as message broker
* HAProxy
* IOT protocol to send messages (MQTT)
* Apache Spark to manage the data
* Elasticsearch as NoSQL Database
* Kibana as Dashboard

In our system we simulated the tools messages with a handmade multithreading .xml generator in order to achieve (and overtake) the amount required.

We built a SQL database to simulate the raw\_data database, containing the decoded tools, recipes and steps names.

Here we have the list of the functional/non-functional requirement and how we achieve them:

## Functional Requirements

* The system has to manage events intercepted from tools:
  + Our system collects the events via Solace and manages them using Apache Spark.
* The system has to compute these events and gets additional information from the raw\_data database.
  + Our system computes the events and gets additional information from raw\_data using Apache Spark.
* The system has to provide a user-friendly dashboard.
  + Our system integrates Kibana dashboard, it means that the interface offered is user-friendly and allows to query, set default query and visualize the data in an easy and immediate way also for people that are not from IT. The dashboard is reachable within the LAN through a web browser avoiding installation/upgrade times and problems.
* The system has to store the computed data for querying.
  + Our system stores the data in Elasticsearch NoSQL database.
* The dashboard needs to provide a complete system of filtering.
  + With Kibana dashboard the end user can set all the filter that needs. Anyway, we saved a bunch of filters ready to use.

## Non-Functional Requirements

* The system must be without a single point of failure.
  + All the technologies that we used are without point of failure. Solace, Elasticsearch and Spark lie on the concept of Cluster and Node. It means that there are distributed nodes and replicas.

* The system must manage peaks of 160000 messages every 30 minutes and must scale.
  + Our System can manage 216000 messages in almost real time (< 2.5 sec) and both Elasticsearch and Spark are scalable on multiple nodes.
* The system must be up 99.999%
  + We can’t prove mathematically that our system can be up 99.999% but we tried to manage the whole system with about 1 million of messages without errors, so we think that we achieved the 99.999% target.
* The system needs to take no more than 5 minutes from the time the message is broadcasted from the tool to the time it is stored in the analytics database. However, the lower the delay the better.
  + Our system takes at most 1.5 seconds from the time the message is broadcasted from the tools to the time it is stored in the Elasticsearch database, we have to add another second to make the data searchable from the Kibana dashboard.

# Summary

In the first part of the document we focused on understanding the project specifications. We identified all the possible risks and we started to work and study in order to find the best solutions.

Then, we refined the project requirements starting from given paper, we analyzed more in detail both functional and non-functional requirements to have a better guideline for developing properly the project.

We used the Use Case diagram in order to have a clear view on the functionalities of the system.

After the refinement section, we start to draw informal diagrams representing the system topology and the external environment. We also start thinking about a first sketch of system behavior and technologies, always considering the requirements.

One of the toughest and discussed part was the Design Decisions chapter. There we studied technologies, architectures and designs in order to evaluate the strength and weaknesses features of each possibility.

In View and Viewpoints, we showed the interactions between the stakeholders and the concerns. In this section there are all the UML diagrams describing the static view (Component diagram) and dynamic view/behavior (Sequence diagrams).

The system architecture has been also modeled and described with CAPS, SWML, in order to have an IoT point of view of the system’ software part.

In the last part of this documentation, we explained how our choices allow the developed system to achieve all the functional requirements with an overall great performance and reliability.