University of L'Aquila

Department of Information Engineering, Computer Science and Mathematics

Course of Software Architectures

The MEB-POC Manufacturing System

Team Name		**LinuxFellows**
Name-Surname	Matriculation Number	Email Address
Luigi Cerone	261212	luigi.cerone1@student.univaq.it
Andrea D'Ascenzo	261123	andrea.dascenzo@student.univaq.it
Aly Shmahell	258912	aly.shmahell@student.univaq.it

February 19, 2019

Contents

List of Figures

Chapter 1

First chapter

1.1 Challenges and Risk Analysis

Table 1.1: Risk Management Table

Risks	**Date of Identification**	**Date of Resolution**	**Method of Resolution**
Finding a DBMS technology that is able to process the specified number of operations	2/12/18	17/12/18	Decided on a hybrid embedded database (using an interactive query and a high throughput DBMS e.g. RocksDB or something of similar calibre).
Message broker support for the AMQP protocol	2/12/18	17/12/18	Swapped AMQP for the Kafka messaging protocol.
Finding a Network protocol that can handle the 10Mb message load	2/12/18	17/12/18	Decided on the Kafka binary protocol over TCP.

1.2 Requirements Refinement

1.2.1 Functional Requirements

1.2.1.1 Dashboard requirements

- **Dashboard-Queries**: The dashboard allows the user to perform queries on the data provided by the system's tools, after they have been stored in the analytics database.
- **Dashboard-Filters**: The user can use filters that apply constraints on the queries.
- **Dashboard-Default**: The dashboard has an initial mode in which it performs a default query upon initialization.

1.2.1.2 Database Requirements

- **Information-Structure**: The database system should allow for storage of structured information about a specific tool.
- **Information-Concurrency**: The database system should allow for concurrent operations like selection, update and insertion.
- **Information-Processing**: The database system stores information about tools after they have been processed with additional predefined data provided by another system.

1.2.1.3 Input/Output Requirements

- **IO-Input**: The system should be able to capture information from the tools in real time.
- **IO-Output**: The system should provide the processed information in real time.

1.2.2 Non-Functional Requirements

- **Scalability:**
  - The system needs to be able to support a volume of data with peaks of 80000 messages per 30 minutes.
  - The system needs to be able to scale up to a volume of data with peaks of 160000 messages per 30 minutes.
- **Performance**:
  - The system should be able to handle an amount of data of up to 10Mb per message.
  - The system should guarantee that the time period from a message broadcast until the storage of the message in the analytics database is less or equal to 5 minutes.
- **Availability**: The system should guarantee High Availability of class 5, which means Uptime should be 99,999% and downtime should be less or equal to 5.25 minutes per year.
- **Reliability**: The system needs to be fault tolerant, the system architecture should be without Single Points of Failure.

1.2.3 Requirements Prioritization

Each requirement is given a priority on a scale of 1 to 5, with value of 1 indicating a minimum priority and a value of 5 indicating a maximum priority.

1.2.3.1 Priorities of Functional Requirements

- Dashboard-Queries: 4
- Dashboard-Filters: 3
- Dashboard-Default: 5
- Information-Structure: 5
- Information-Concurrency: 5
- Information-Processing: 5
- IO-Input: 5
- IO-Output: 5

1.2.4 Use-Case Diagrams

The team has identified the following actors in the system:

- User
- Tool
- System

1.2.4.1 User use case diagram

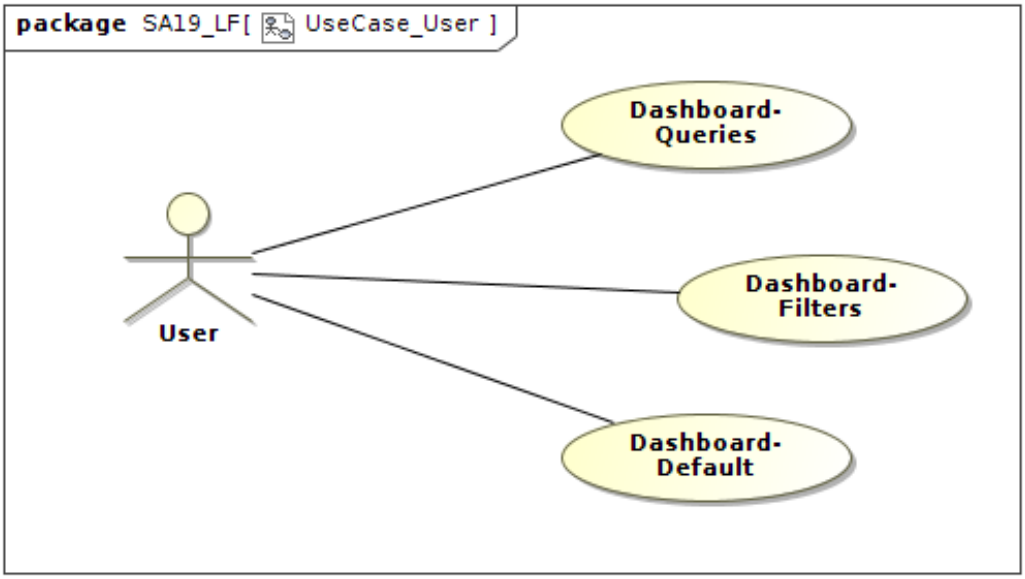

Figure 1.1: Use case diagram of the actor **User**.

1.2.4.2 Tool use case diagram

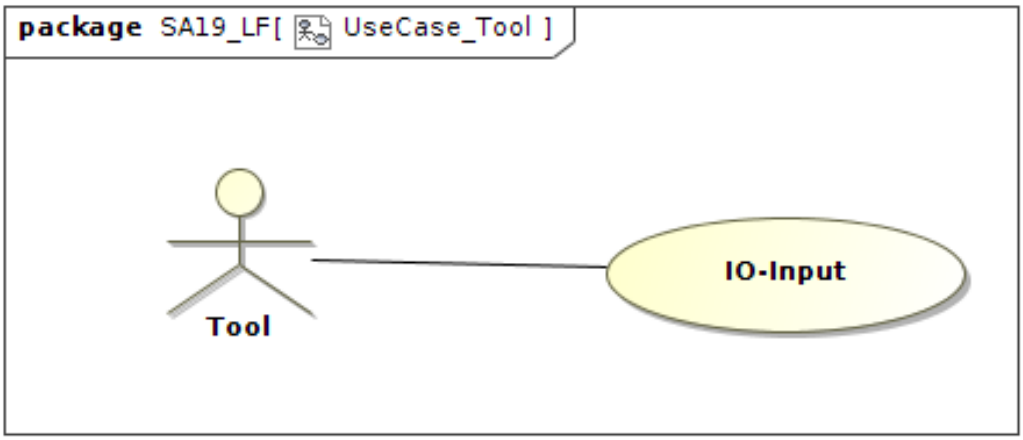

Figure 1.2: Use case diagram of the actor **Tool**.

1.2.4.3 System use case diagram

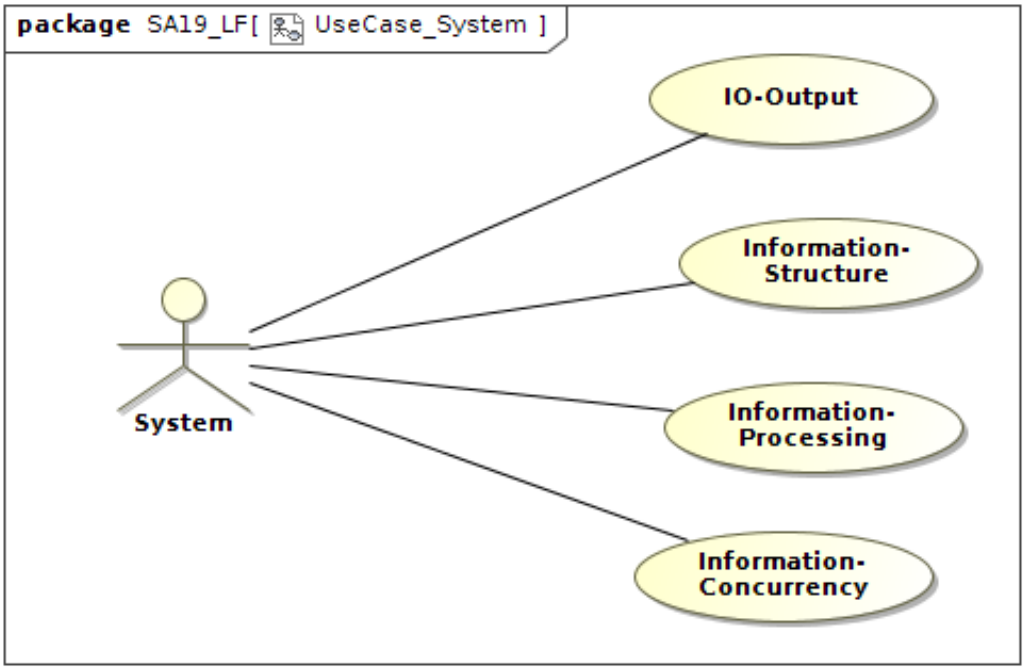

Figure 1.3: Use case diagram of the actor **System**.

1.2.5 Tabular description of the most important requirements

Table 1.2: Information-Processing table

Use case name	Information-Processing
Participating actors	Tools, External system, system
Description	After that the system has collected the information sent by a tool, it processes it adding data, and stores it into the database system
Triggering event	Information receiving
Priority	High, level 5

Table 1.3: IO-Input table

Use case name	IO-Input
Participating actors	Tools, system
Description	Whenever a tool sends information, the system has to capture it in any case
Triggering event	Information receiving
Priority	High, level 5

Table 1.4: Dashboard-Default table

Use case name	Dashboard-Default
Participating actors	User, system
Description	Whenever the user execute the dashboard, the system should prints a default query
Triggering event	Dashboard request
Priority	Medium-High, level 4

1.3 Informal Description of the System & its Software Architecture

1.3.1 Description of the System

The system is composed of different high level components:

- **fab_data connector**: Under the assumptions described in the section 1.4.1, we get the events from the database fab_data. So the purpose of this component is to constantly query the external database in order to insert these information into our system. The method used to stream from the fab_data database is described in 1.4.3.2.

- **raw_data connector**: The purpose of this component is to query external raw_data database in order to retrieve the 10Mb files that contain the recipe regarding a specific tool. The method used to stream from the raw_data database is described in 1.4.3.2.

- **Message Broker**: This is the main component of our system: it takes the information from both the **fab_data connector** component and the **raw_data connector**, manipulates them and performs message translation & processing. After the computation, this component stores its results in the *analytics_database*
  It also manages the information that the final users can monitor in the dashboard after applying the selected filters, The result information are retrieved from the *analytics database*.

- **Analytics Database**: This the component responsible for the storage of the information. It contains the database itself but also DBMS and a controller used to perform storage operations (insertion,selection, etc). When the user apply specific filters in the dashboard these information has to be retrieved in this component's database.

1.3.2 Architectural Pattern

The team has decided to use **Publish/Subscribe** pattern, major information are in 1.4.2.

1.3.3 System boundaries

External Components:

- The database raw_data is provided by the customer and so are the information regarding the recipes. The database is assumed to be reliable and fault tolerant. Also the information stored in it are assumed to be consistent and updated in real time. This system will be simulated in our prototype but without their non-functional requirements.

- The database fab_data is provided by the customer and so are the information regarding the tools event. The database is assumed to

be reliable and fault tolerant. Also the information stored in it are assumed to be consistent and updated in real time. This system will be simulated in our system prototype but without their non-functional requirements.

- A **network** with an architecture that is able to support the needed ratio of information.

1.3.4 Informal Diagram of the System

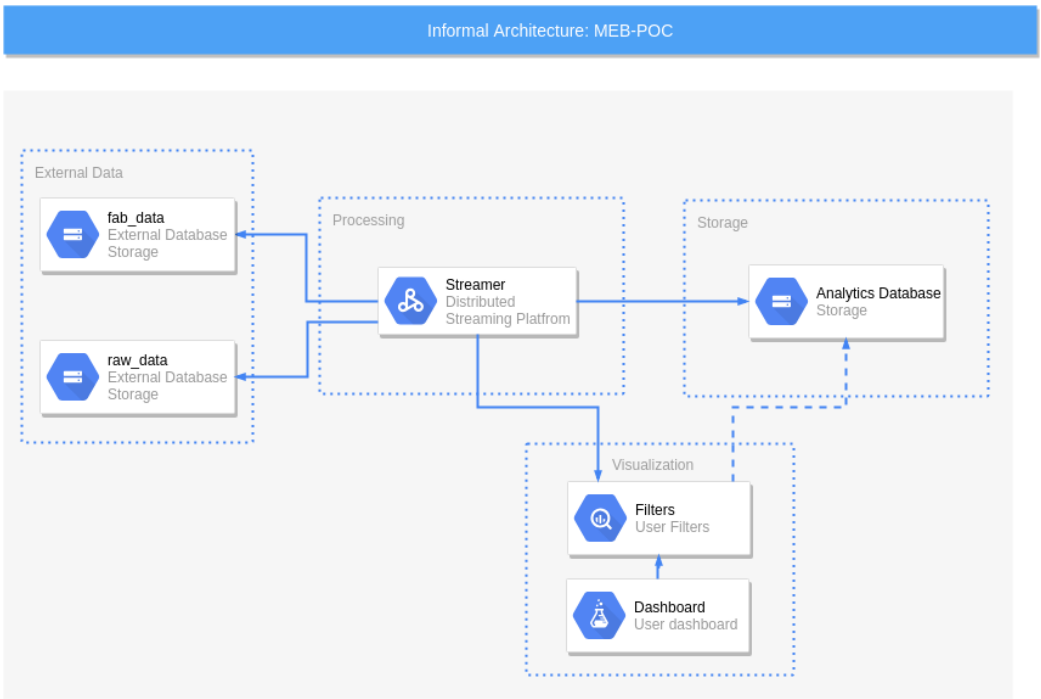

Figure 1.4: Informal architecture of the system.

1.3.5 Requirements' Fulfillment

The requirements have been fulfilled in the following way:

- **Dashboard-Queries**: The dashboard allows the user to perform queries on the data provided by the system's tools. There are two major REST endpoint, one */category/n* allows the user to get all the information regarding the category number n, the other */tool/eqipID* allows the user to get all the information about a specific tool identified by equipID.

- **Dashboard-Filters**: The user has the possibility to apply filters.
- **Dashboard-Default**: The dashboard has an initial mode in which it will perform a default query.
- **Information-Structure**: The database system allows the storage of structured information about a specific tool into the Analytics database by using both a DBMS and the Kafka Embedded Database.
- **Information-Concurrency**: The database system allows concurrent operations like selection, update and insertion by using a JDBC specific driver.
- **Information-Processing**: The database system stores information about tools after they have been processed with additional predefined data provided by another system.
- **IO-Input**: The system is able to capture information from the tools in real time by using the interface to fab_data and raw_data databases which will publish the required information to their specific topics.
- **IO-Output**: The system provides the processed information in real time.
- **Scalability**: This is provided by the message broker.
- **Performance**: The system is able to handle both the amount of requests and data as stated in the requirements.
- **Availability**: The developed system does respect this requirement.
- **Reliability**: The system architecture and the prototype are without Single Points of Failure. One tool that helps us in fulfilling this goal is the message broker that provides redundancy and fault tolerance.

1.4 Design decisions

1.4.1 Tools' events (fab_data capture) - fetching from DBMS vs sensor interception

Concern (Identifier: Description)		*Con#1: How does the system take the events from the tools?*
Ranking criteria (Identifier: Name)		*Cr#1: number of messages & network traffic* *Cr#2: fault-tolerance* *Cr#3: loss of data* *Cr#4: ease of simulation.*
Options	***Identifier: Name***	*Con#1-Opt#2: Pull events from the provided database*
	Description	*The customer provides us a fab_data database where all the tools' events are stored as soon as possible.*
	Status	*This option is decided.*
	Relationship(s)	-
	Evaluation	*Cr#1: We don't need to intercept each meassage, we only need to pull the provided database. There aren't messages that are sent to our system, it's our connectors that pulls the data with fewer messages than the other approach.* *Cr#2: We assume that the provided database is fault-tolerant.* *Cr#3: We assume that each tool' event is stored in the database in real time. Our system, when pulls information from it, takes all the rows.* *Cr#4: We only need to fill a database with specific data in order to simulate the external system.*
	Rationale of decision	*This option is decided because satisfy our requirements and solve the criteria in optimal manner.*
	Identifier: Name	*Con#1-Opt#2: Messages interceptions*
	Description	*Each tool individually sends the event to our system that receives and handles it.*
	Status	*This option is rejected.*
	Relationship(s)	-
	Evaluation	*Cr#1: Each tool sends messages when an event occurs, so in the networks could travel a lot of data.* *Cr#2: We need to make the reception of the messages fault-tolerant.* *Cr#3: We need a way to detect when a message is lost and maybe some backup strategy.* *Cr#4: We need to implement different script that each individually simulate a tool.*
	Rationale of decision	*This option is rejected because it doesn't satisfy the requirements.*

Figure 1.5: First design decision.

In order to prevent an high amount of messages towards our system which would be hard and time-consuming to manage individually, the team has decided to obtain the tools' data from the *fab_data* database, which is provided externally.

The team assumes that the provided *fab_data* database is fault-tolerant and that the information are stored in it as soon as they are made available by their generating tools (in real-time).

This choice prevents loss of data in case of problems in the system's connection, guarantees that every message will be retrieved by the system, makes it possible to eventually rebuild the history of messages.

1.4.2 Architectural Pattern

Concern (Identifier: Description)		*Con#2: Which architectural pattern should the system use?*
Ranking criteria (Identifier: Name)		*Cr#1: scalability* *Cr#2: decoupling*
Options	***Identifier: Name***	*Con#2-Opt#2: Publish/Subscribe*
	Description	*PubSub architecture where the publishers are the external databases (fab_data & raw_data), the topics are related to event's messages architecture and subscribers are the end users.*
	Status	*This option is decided.*
	Relationship(s)	-
	Evaluation	*Cr#1: Pub/Sub architecture provides scalability, if one topic is under strain we can add other computational power to it.* *Cr#2: Pub/Sub architecture provides de-coupling.*
	Rationale of decision	*This option is decided because it satisfy our requirements and solves the criteria in optimal manner.*
	Identifier: Name	*Con#2-Opt#2: Client/Server*
	Description	*Communication in this architectue among the components is based on client/server pattern.*
	Status	*This option is rejected.*
	Relationship(s)	-
	Evaluation	*Cr#1: In order to achieve scalability we need to set up load balancing tecniques.* *Cr#2: Clients are stricty coupled with server's architecture.*
	Rationale of decision	*This option is rejected because it doesn't satisfy the requirements.*

Figure 1.6: Second design decision.

The team has decided to opt for a Publish/Subscribe architecture. In the context of our application the publisher of the messages (i.e. the **producer**) is the component that gets the information from the *fab_data* database. So the messages of the system are the tools event (whose structure is described in the project specification). The **consumers** of the system are the users and the topics are the different categories. Components may subscribe to a set of events. It is the job of the publish-subscribe run-time infrastructure to make sure that each published event is delivered to all subscribers on that category.

We have decided to use this architectural pattern also because Publish/-Subscribe has great **scalability**, if we see that the current system is unable to satisfy our system constraints we can identify the most computationally loaded topics and subscribe other servers to these topics in order to parallelize their work (**horizontal scalability**). Another advantage is the **decoupling** of the system, publishers are loosely coupled to subscribers and so they are allowed to remain ignorant of system topology.

1.4.3 From Message Broker to Streaming Platform

Concern (Identifier: Description)		*Con#3: Which message broker/streaming platform should the system use?*
Ranking criteria (Identifier: Name)		*Cr#1: real time* *Cr#2: storage (persistence)* *Cr#3: history* *Cr#4: team knowledge about the technology and documentation of it.* *Cr#5: support for AMQP protocol*
Options	***Identifier: Name***	*Con#3-Opt#1: Apache Kafka*
	Description	*Implementation of Apache Kafka as streaming platform*
	Status	*Decided*
	Relationship(s)	-
	Evaluation	*Cr#1: it guarantees a message throughput up to 800k/sec* *Cr#2: it is a durable message store, meaning that it is able to store the message in an undefined period of time* *Cr#3: this point strictly follows from the one above: it is able to provide an history about the messages passed over the system* *Cr#4: the team has studied the technology during the univerisity course and the kafka is very well documented* *Cr#5: Kafka doesn't support AMQP protocol*
	Rationale of decision	*This option permits us to reach the highest message speed on the network; it permits us to query any time of the history of the messages; it guarantees redundancy of data, thanks to its storing*
	Identifier: Name	*Con#3-Opt#2: RabbitMQ*
	Description	*Implementation of RabbitMQ as message broker*
	Status	*Rejected*
	Relationship(s)	-
	Evaluation	*Cr#1: it guarantees a message throughput up to 100k/sec* *Cr#2: it does not provide message storing, since every message is erased after the first consume request.* *Cr#3: this point strictly follows from the one above: since the messages can be accessed once, history can not be obtained* *Cr#4: the team hasn't studied/used the technology and the tool is quite documented online.* *Cr#5: RabbitMQ supports the AMQP protocol.*
	Rationale of decision	*This option does not achieve our goals*
	Identifier: Name	*Con#3-Opt#3: Solace*
	Description	*Implementation of Solace as message broker*
	Status	*Rejected*
	Relationship(s)	-
	Evaluation	*Cr#1: it guarantees a message throughput up to 750k/sec* *Cr#2: it provides message storing, meaning that it is able to store the message in a persistent manner* *Cr#3: this point strictly follows from the one above: it is able to provide an history about the messages passed over the system* *Cr#4: The team hasn't used this technology and it is not well documented online.* *Cr#5: Solace supports AMQP protocol*
	Rationale of decision	*This option is rejected beacuse the team isn't familiar with this tool and it is not well documented. Also it doesn't have something like "Kafka Connect" that allows our system to stream contents of external databases.*

Figure 1.7: Third design decision.

1.4.3.1 Apache Kafka

In order to manage the messages, the team has decided to implement the Apache Kafka **Streaming Platform** instead of a conventional message bro-

ker. Kafka perfectly complies with our needs[1]. It guarantees a very high throughput in terms of messages digested (peak of 800k/sec [2]), which accomplishes our goal to afford the real time message processing. In this scenario, Kafka provides the fastest speed, compared to other message brokers like RabbitMQ, which could handle up to 100k/sec messages - due to its routing complexity. Regarding storage, Kafka is better than RabbitMQ: Kafka is a durable message store, in which clients can get a *replay* of the event stream on demand, as opposed to more traditional message brokers where once a message has been delivered, it is removed from the queue. In our case this feature is useful since it permits to achieve a history of the messages which have passed through the system, and a recovery plan in case of malfunction, since Kafka permits the storage of the data.

1.4.3.2 Kafka Connect

Another reason that favors Kafka over other Message Borker is Kafka Connect API[3], more specifically a **Source Connector** which could "ingest entire databases and stream table updates to Kafka topics.". This is exactly the way in which the team has decides to pull information from the external *fab_data* and *raw_data* databases.

There are also two different ways that could be used in order to implement this external database "streaming" as states in this article[4], either by using a JDBC plugin for the database's driver or by using **CDC** (i.e. *Log-based Change-Data-Capture*). The team has opted for the CDC, in particular by using the Debezium[5] platform.

[1] https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka
[2] https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-ma
[3] https://docs.confluent.io/current/connect/index.html
[4] https://www.confluent.io/blog/no-more-silos-how-to-integrate-your-databases-with-apache-kafka-and-cdc
[5] https://debezium.io/

1.4.4 Recipes' Retrieval - raw_data caching

Concern (Identifier: Description)		*Con#4: How does the system handle raw_data?*
Ranking criteria (Identifier: Name)		*Cr#1: memory usage* *Cr#2: performance* *Cr#3: scalability*
Options	***Identifier: Name***	*Con#4-Opt#1: Internal Caching*
	Description	*Given the fact that the data flow from raw_data DBMS to the message stream is not continuous, but rather a flow-on-demand, as in data flows according to the OID from the latest fab_data capture, the flow needs to be discrete and the data needs to be cached before being passed through the message stream. The discritization of the flow and the caching of the data is done using the kafka streams application API (inside the streaming platform).*
	Status	*This option is decided.*
	Relationship(s)	-
	Evaluation	*Cr#1: Memory usage becomes minimal as we make the flow discrete (as opposed to a continuous flow), and the memory of the raw_data topic can be set to recycle with the memory of fab_data topic.* *Cr#2: Aside from simple checks to see which OID needs to be fetched to the raw_data topic, the performance can be evaluated as O(n), with n being the number of fab_data instances being streamed. However, O(n) represents the worst case scenario in which all OIDs are new, but if an OID is in the cache then it doesn't have to be fetched again, which enhances the performance.* *Cr#3: Since the API used for this option is the same as the one used by the bulk of the system, then this option scales up with the system.*
	Rationale of decision	*This option satisfies the requirements, enhances the performance, and scales up with the system.*
	Identifier: Name	*Con#4-Opt#1: External Caching*
	Description	*Given the fact that the data flow from raw_data DBMS to the message stream is not continuous, but rather a flow-on-demand, as in data flows according to the OID from the latest fab_data capture, the flow needs to be discrete and the data needs to be cached before being passed to the message stream. The discritization of the flow and the caching of the data is done by providing an external application (outside the streaming platform) that takes the latest OID from the message stream, then fetches the recipe from raw_data DBMS then provides a dataflow to the raw_data topic.*
	Status	*This option is rejected.*
	Relationship(s)	-
	Evaluation	*Cr#1: Memory usage increases as we need to route the OID to the external caching application, then the provided recipes will be consumed cintinuously by raw_data topic, which increases the load inside the message stream.* *Cr#2: Performance decreases as we route the OID to the external caching application.* *Cr#3: Since we are using an external application for this option, then it requires special maintainance and specific development cycles to guarantee its scalability.*
	Rationale of decision	*The option is rejected because it impacts performance negatively, increases memory usage, and doesn't scale up with the system.*

Figure 1.8: Forth design decision.

In order to integrate the process of recipe retrieval from the raw_data DBMS, the team has decided to build an integrated discrete cashing methodology

within the streaming platform using its own API.
This methodology allows for a dataflow-on-demand from the raw_data DBMS only when needed, robust and interlocked development cycle, and scalability out of the box.

1.4.5 Storage

Concern (Identifier: Description)		*Con#5: Where are the information stored?*
Ranking criteria (Identifier: Name)		*Cr#1: Fault tolerance* *Cr#2: Relations among the data* *Cr#3: Simplicity of the architecture*
	Identifier: Name	*Con#5-Opt#2: Kafka State & External Database*
	Description	*The final processed information is stored both in the Kafka State and in an external database.*
	Status	*This option is decided.*
	Relationship(s)	-
	Evaluation	*Cr#1: Given the fact that all the data are stored in the kafka state our system inherits the redundancy and fault tolerance of the Kafka platform. The information are also stored in an external database.* *Cr#2: The data are stored in Kafka State as key-value logs and in the external database with relational/not relational (according to the customer needs).* *Cr#3: The usage of kafka Stream and interactive queries simplify the system architecture.* *Cr#4: The data could in case be indexed by third parts dashboard systems.*
	Rationale of decision	*Thi option gives the system fault tolerance inheriting it from Kafka (so without additional work) and uses also an external database that could be used with search indexing engine and third parts dashboard system.*
	Identifier: Name	*Con#5-Opt#2: Kafka State*
	Description	*The final processed information is stored only in the Kafka State.*
	Status	*This option is rejected.*
	Relationship(s)	-
	Evaluation	*Cr#1: The data are stored into the Kafka state so this option gives our system fault tolerance.* *Cr#2: The data are stored only as key-value based lightweight database.* *Cr#3: The usage of Kafka stream and Interactive Queries simplify the architecture.* *Cr#4: The data couldn't in case be indexed by third parts dashboard systems.*
	Rationale of decision	*This option is rejected because it doesn't allow to integrate our system in the future with a dashboard provided by third parts dashboard based on an indexing enigne.*
	Identifier: Name	*Con#5-Opt#3: External Database*
	Description	*The final processed information is stored only in an external database (relational or not based on the consumer's request).*
	Status	*This option is rejected.*
	Relationship(s)	-
	Evaluation	*Cr#1: The data are stored in an external database and this option is not fault tolerance. We should design our sysem to be fault tolerance so we need to introduce technologies that allows the database to act in such way.* *Cr#2: The data could be stored in a relational or not relational way according to customer needs.* *Cr#3: The architecture is not simplified.* *Cr#4: The data could in case be indexed by third parts dashboard systems.*
	Rationale of decision	*This option is rejected because it requires additional technologies and work in order to make it fault tolerant.*

Figure 1.9: Fifth design decision.

In our system we need to store the analytics information in a database.

In the project specification there are two constraints, our system needs to handle 80.000 messages every 30 minutes (Constraint 1) and should be

able to scale up to 160.000 messages every 30 minutes (Constraint 2). So the messages flow is the following:

Table 1.5: Messages ratio table

Number of messages		**Time**
Constraint 1	**Constraint 2**	
80.000	160.000	30 min
$\approx$ 2.700	$\approx$ 5.200	60 sec
$\approx$ 45	$\approx$ 90	1 sec

Also the user, by using dashboard, could request both real-time and past information. In order to provide the user with the real-time information the team has decided that the user could subscribe to the requested topics (i.e. the topics where the translated information are published). As for the queries of past information the team has decided to use Interactive Queries provided by Apache Kafka, which makes the interrogation on kafka's logs, so they use the streaming platform also as permanent storage. This approach is discussed in the official documentation [6]. The team has decided to store the information also in an external database (which could be relational or not relational but in our prototype we used a MongoDB's one) because in this way we could extend our system with real-time and non-real-time dashboard provided by third parts software which require traditional databases (like Kibana).

1.5 Views and Viewpoints

Stakeholders:

- **Sensor Engineer**: Responsible for designing and implementing the sensor networks.
- **Core Developer**: Designs the communication system and the streams application (built using the streaming platform) that process our data.
- **Database Developer**: Designs and administrates the database where our information are stored.
- **UI Designer**: Responsible for developing the dashboard where the information will be displayed.

[6] https://docs.confluent.io/current/streams/concepts.html#interactive-queries

- **System Integrator**: Responsible for system integration, emergency response and system migration.
- **End Users**: The final user of our system.

Table 1.6: Concern-Stakeholder Traceability

	Sensor Engineer	**Core Developer**	**Database Developer**	**UI Designer**	**System Integrator**	**End User**
Security		X	X		X	
Cost	X	X	X	X	X	
Networking & Communication		X			X	
Data Analysis	X	X				X
Response Time		X	X		X	X
Dependability	X	X	X	X	X	
Usability						X
Scalability		X			X	
Energy Consumption	X	X				

Chapter 2

Project Architecture

2.1 Component Diagram

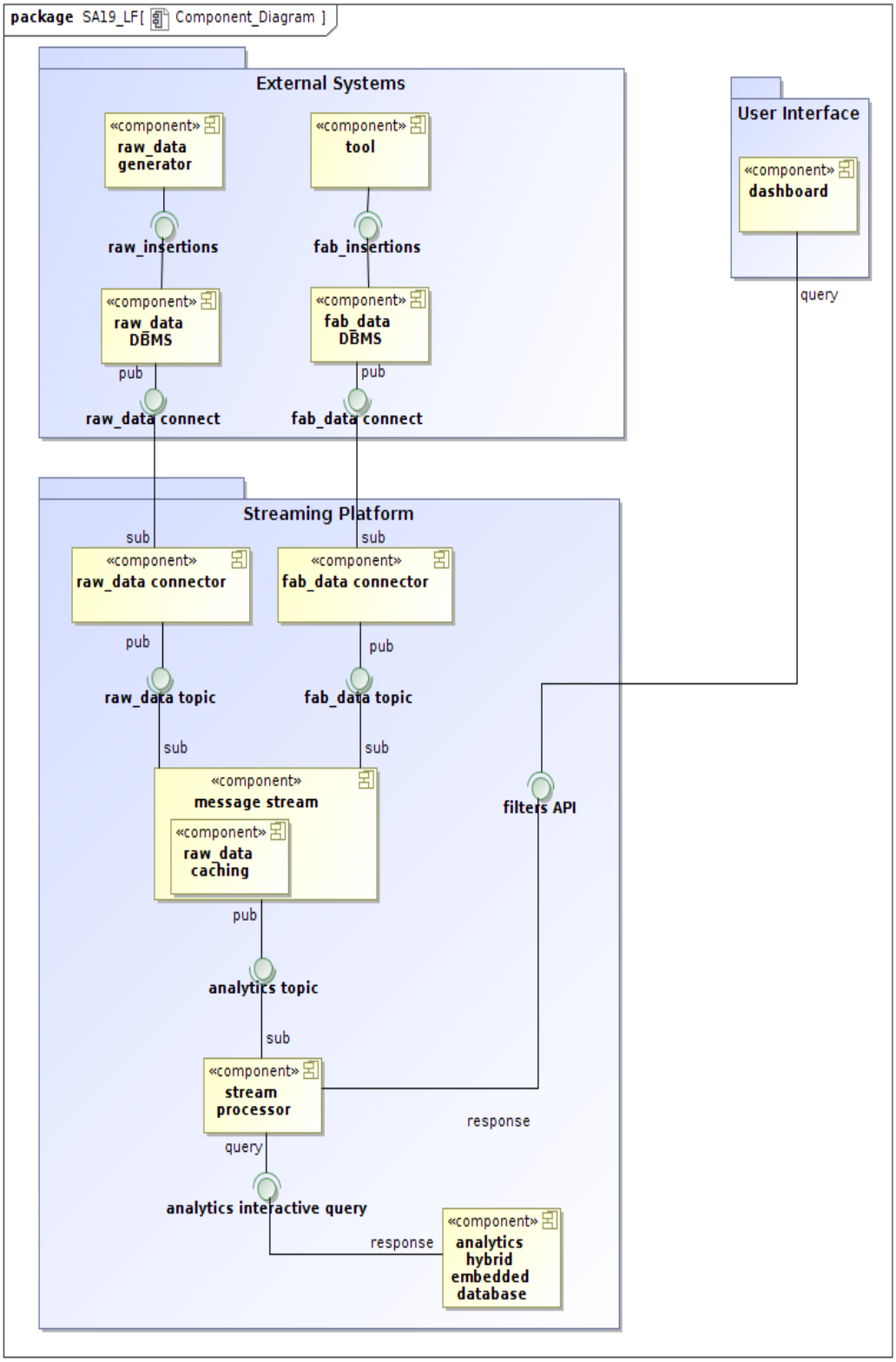

Figure 2.1: Component diagram.

2.1.1 Components Description

- External Systems Package: This package contains the external databases and systems which our system depends upon. This package represents everything that is provided by the customer.
  - **raw_data DBMS**: This component refers to the external database raw_data.
  - **fab_data DBMS**: This component refers to the external database fab_data.
  - **raw_data generator**: This component refers to an external system that inserts data into the external database raw_data.
  - **tool**: This component refers to the external tool that generates events and inserts them into the fab_data database.
- Streaming Platform Package:
  - **fab_data connector**: This component is responsible for streaming the event messages (received by the external fab_data database via a specific tool interface) into a Kafka topic called fab_data topic.
  - **raw_data connector**: This component is responsible for streaming the translation messages (received by the external raw_data database via a specific tool interface) into a Kafka topic called raw_data topic.
  - **message stream**: This component is subscribed to two topics: fab_data topic & raw_data topic. It is responsible for the translation of the fab_data events. In order to do this, this component uses internal caching.
  - **raw_data caching**: This component is used to cache the information taken from the raw_data database by using raw_data topic. It caches the latest pair (OID, nameTranslation) received for each id (which could be eqipOID, recipeOID, stepOID). Therefore it allows for a raw_data:fab_data relationship in the system of 1:n.
  - **stream processor**: This component does the following:
    - consumes translated information from the specific topics and stores them into the analytics DBMS, both the Kafka one and the MongoDB one.

    - applies both default and user-specified filters to the analytics information.
    - sends requests to the analytics DBMS according to some of the filters (for getting past information).
    - provides the analytics information (past and current) to the dashboard.
  - **analytics hybrid embedded database**: This component is responsible for the storage of the tools' information.

- User Interface Package:
  - **dashboard**: This component is responsible for the visualization of the stored information. Also it allows the final user to apply specific filters on the requested data.

2.1.2 Interfaces Description

- External Database Package:
  - **raw insertions**: This interface represent the communication between the raw_data generator and the raw_data DBMS.
  - **fab insertions**: This interface represent the tool(s) and fab_data DBMS.
  - **raw_data connect**: This interface is responsible for the communication between the external database raw_data and our system. This communication is discrete, and happens on demand by our system.
  - **fab_data connect**: This interface is responsible for the communication between the external database fab_data and our system. This communication has to be in real-time, as soon as a new entry is inserted into the database this interface has to provide it to our system.

- Streaming Platform Package:
  - **fab_data topic**: This interface refers to all the topics that are used by the component fab_data connector to stream the external fab_data database.
  - **raw_data topic**: This interface refers to all the topics that are used by the component raw_data connector to stream the external raw_data database.

  - **analytics topic**: This interface refers to all the topics in which the translated information are written so that the component *message stream* can store them.
  - **analytics interactive queries**: This interface performs CRUD operations with the analytics DBMS.

- Analytics Package:
  - **filters API**: This interface is responsible for the communication between the user's dashboard and our system.

2.2 Sequence Diagram

2.2.1 Dashboard

This diagram represents the actions' flow of a query made by an end user on the dashboard. This component will send a message to the stream processor component, which is responsible of the analytics message flow. If the time concerning the query is the present, it means that the end user is interested in some part of the flow which is passing through the stream processor at the current moment, and the processor will consume the interested data; otherwise, it will send a MakeQuery() message to the hybrid embedded analytics database, located in our system. In this database all the analytics data are stored, and it's where the system will search to answer queries about past information. In both cases, the dashboard will receive a return.

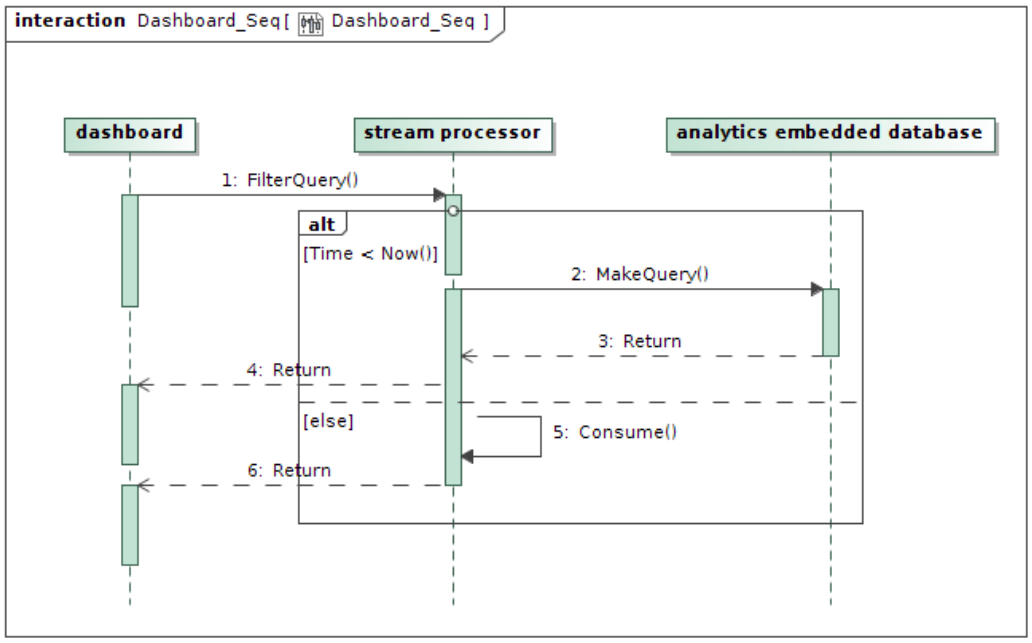

Figure 2.2: Sequence diagram from the **dashboard** point of view.

2.2.2 Raw_data

This diagram represents the events that occur when the raw_data generator inserts new data into the database. Of course, this means that it will send an Insert message to the raw_data DBMS, which in turn will publish this value on its Kafka topic, namely the raw_data connect, at which the raw_data connector is subscribed and in turn will send a consume operation. After that the raw_data connector will publish the processed message in the raw_data topic.

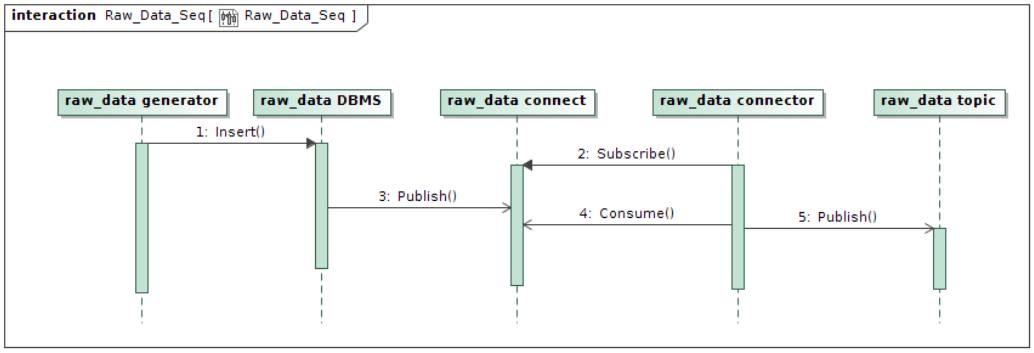

Figure 2.3: Sequence diagram from the **raw_data** point of view.

2.2.3 Fab_data

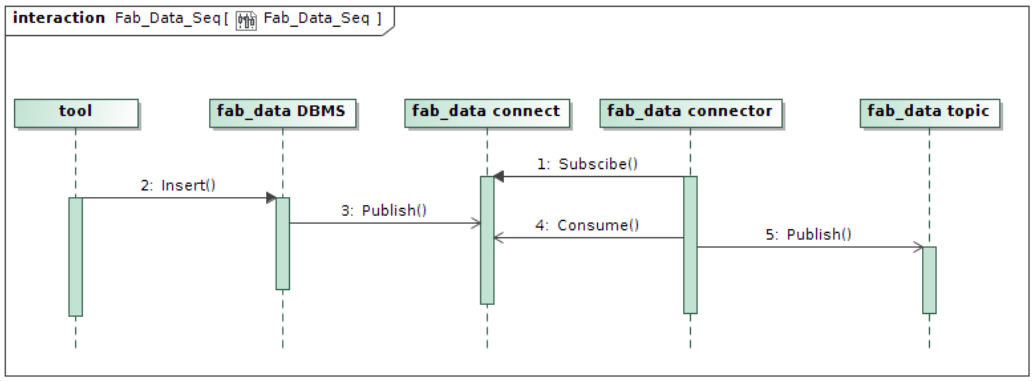

Figure 2.4: Sequence diagram from the **fab_data** point of view.

This diagram represents the events that occur when a tool inserts an event into the external database. When this happens the fab_data connector (which was subscribed to that topic) receives the new event and, after some computation, publishes it in the new fab_data topics.

2.2.4 Streaming

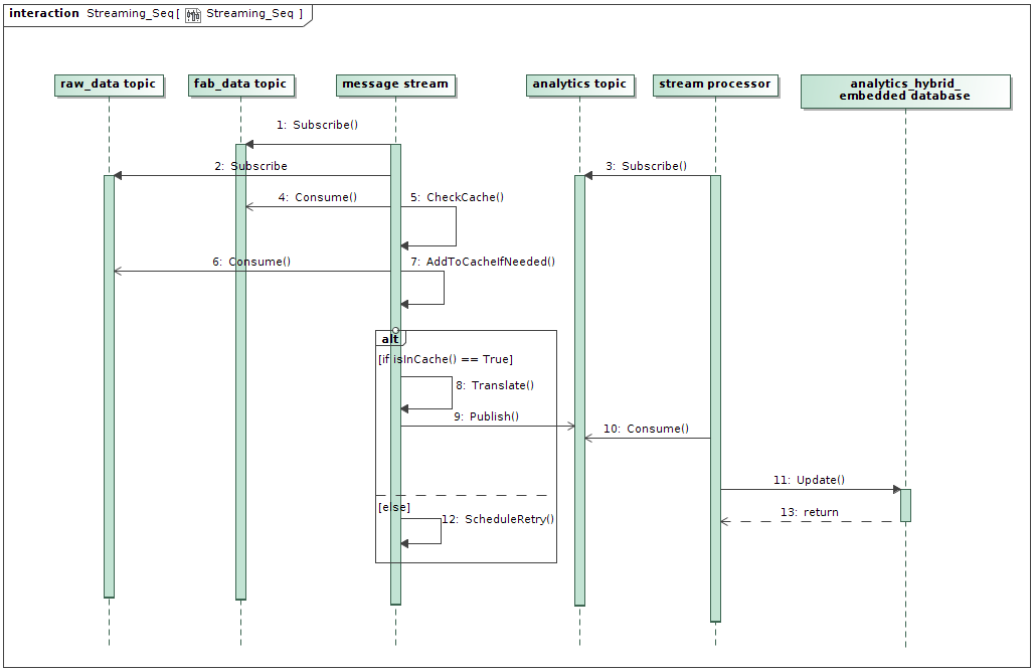

Figure 2.5: Sequence diagram of the **Streaming platform** flow of events.

This diagram represent the actions taken by the streaming platfrom package. the *Message Stream* subscribes to both fab_data and raw_data topics then when there is a publication of a new message on the fab_data topic, the message is consumed by the message stream. Then the message stream compares the OID in the message to its cache, if a recipe with that OID is not present, it stores this failed translation into a specific kafka topic and it delays the translation so that it can retry later (every 30 sec). Else if the information are present, this component performs the translation and republishes the result into a specific output topics in the analytics_topics interface. This message is consumed my the stream processor that stores it both in the kafka DB and, by using kafka connect, also in the external database.

Chapter 3

Second deliverable

3.1 CAPS

3.1.1 CAPS Diagrams

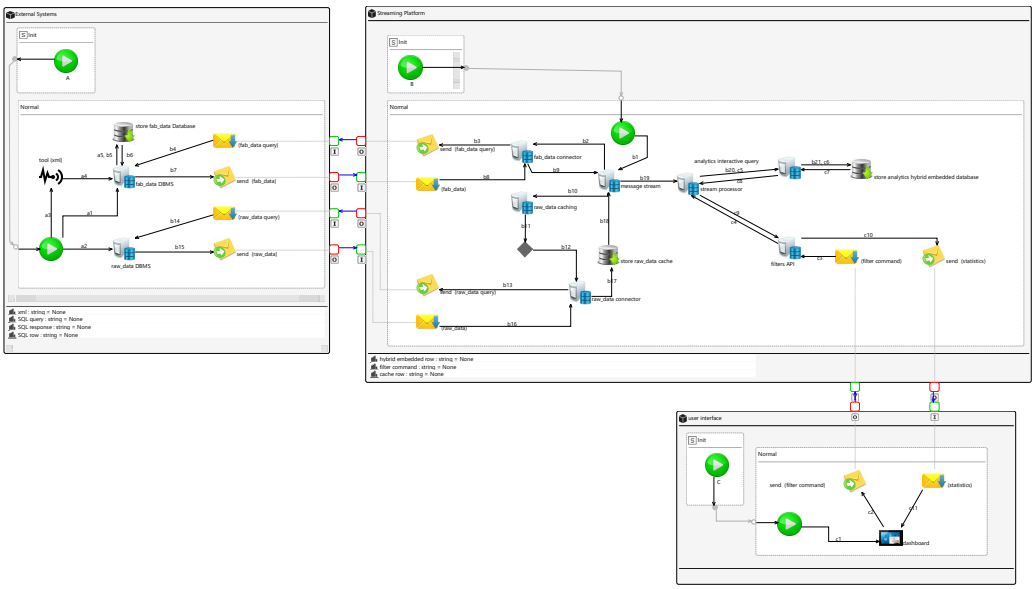

Figure 3.1: SAML Diagram

The MEB-POC SAML Diagram source-code is downloadable from Google Drive and also available at the "CAPS" branch of the projectś github repository (at the moment the repository is private).

3.1.2 CAPS Design Decisions

3.1.2.1 Design Decision 1

Concern (Identifier: Description)		*Con#1: How to illustrate the flow sequence of commands and data in the SAML diagram?*
Ranking criteria (Identifier: Name)		*Cr#1: Human Readable.* *Cr#2: Easily Expandable.* *Cr#3: Easy to Follow.*
Options	***Identifier: Name***	*Con#1-Opt#1 : AlphaNumeric Tagging with Alphabetic Sources.*
	Description	*We tag the Start symbol in each Init mode with an alphabetic character, selected based on the order of activation of the component, then we tag the connections inside the whole diagram with one of the designated characters followed by a number based on the order of the connection in the flow sequence.*
	Status	*decided.*
	Relationship(s)	-
	Evaluation	*Cr#1: Alphanumeric tags are human readable.* *Cr#2: Alphanumeric tags are expandable up to 24 components according to the scheme we described, which is more than enough for the number of components we currently have (3), therefor we have room for improvement.* *Also, each Alphabetic character can be duplicated, thus if the need arises it can be infinitely expandable.* *Cr#3: Having numeric characters as a tail in this scheme allows for easily understanding the flow.*
	Rationale of decision	*This option is decided because it satisfies the criteria.*

Figure 3.2: CAPS design decision

3.1.2.2 Design Decision 2

In the SAML diagram, we have decided to illustrate the tool output as XML, as indicated in the System Specification, it is therefore assumed that the fab_data DBMS has a plugin or an extention capable of understanding that format and performing any necessary conversion of XML into a suitable format for the fab_data database.

3.2 From architecture to code

3.2.1 Implemented service

The system proposed in the previous chapters has been implemented in all its parts by using Kafka framework and by simulating the external databases with an ad-hoc tool developed by the team. The code for the MEB-POC system is downloadable from Google Drive and is also available in the "master" branch of the projectś github repository (at the moment the repository is private). The simulator's code is downloadable from Google Drive and, as above, available at github(now the repository is private). The language used is Java and there is an appendix chapter that tries to describe the steps needed to run the system on another laptop.
A short video that show the execution of the prototype is available at https://youtu.be/ARS0Be7CP28.

Overall all the external dependencies of the system and the tools used by the team to develop it are:

- Kafka Stream and Kafka Processor API,
- Jackson library for POJO serialization/deserialization,
- Jersey library for the RESTful interface used to implement the dashboard,
- MongoDB driver used to access an external backup database,
- Maven as dependencies manager,
- Git as version control manager.

The final translated information are stored both in the Kafka Stream Local State in a persistent way and also in an external MongoDB database. The following is a diagram that represents the topics used in the system and that tries to explain the flow of information:

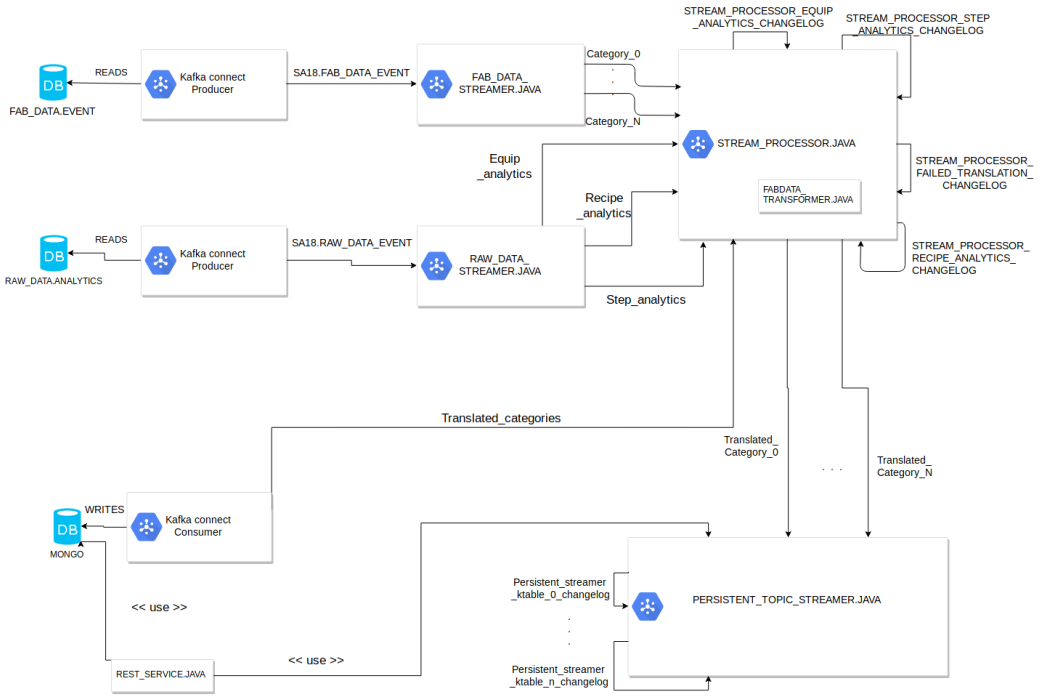

Figure 3.3: Used topics for the communcation

The text written above the lines represents the topic used to communicate between the classes. The topics: streamer-processor-equip_analytics-changelog, streamer-processor-recipe_analytics-changelog, streamer-processor-step_analytics-changelog are **persistent** topics i.e. when the system is shut down the data they contains is not lost because it is written into the disk of the kafka's cluster computers. These topics contains the mappings used for translation in the form OID –> Translation. The topic streamer-processor-failed_translations-changelog is another **persistent** topic where there are written the FabEvent that our system couldn't translate (major details about the translation topic is in the following section). All the topics of the form persistent-streamer-ktable_NUMBER-changelog are used to store the result of the translation for each category in a **permanent** way, in this way this information benefits of kafka fault tolerance and replication. Also, by using the Interactive Query API [1], these data could be queried with an SQL-like syntax[2].

The prototype has been implemented in package whose names respect the component diagram's one. The following is the class diagram relative to

[1] https://kafka.apache.org/10/documentation/streams/developer-guide/interactive-queries.html

[2] https://www.confluent.io/product/ksql/

the Message Stream component:

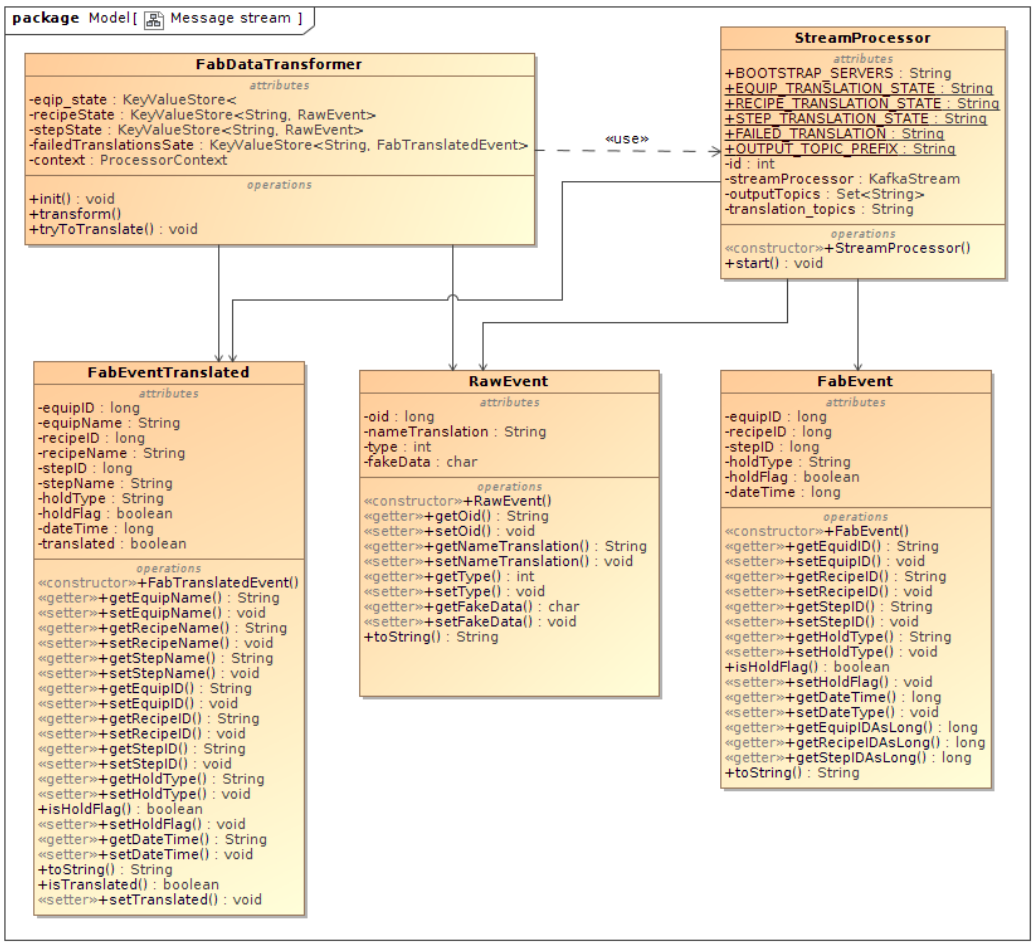

Figure 3.4: Class diagram for package **message_stream**

This package contains all the logic about the translation from a FabConnectEvent's object into a FabTranslatedEvent one. In the following section there is an activity diagram that describes the algorithm used to perform this operation.

3.2.2 Translation logic

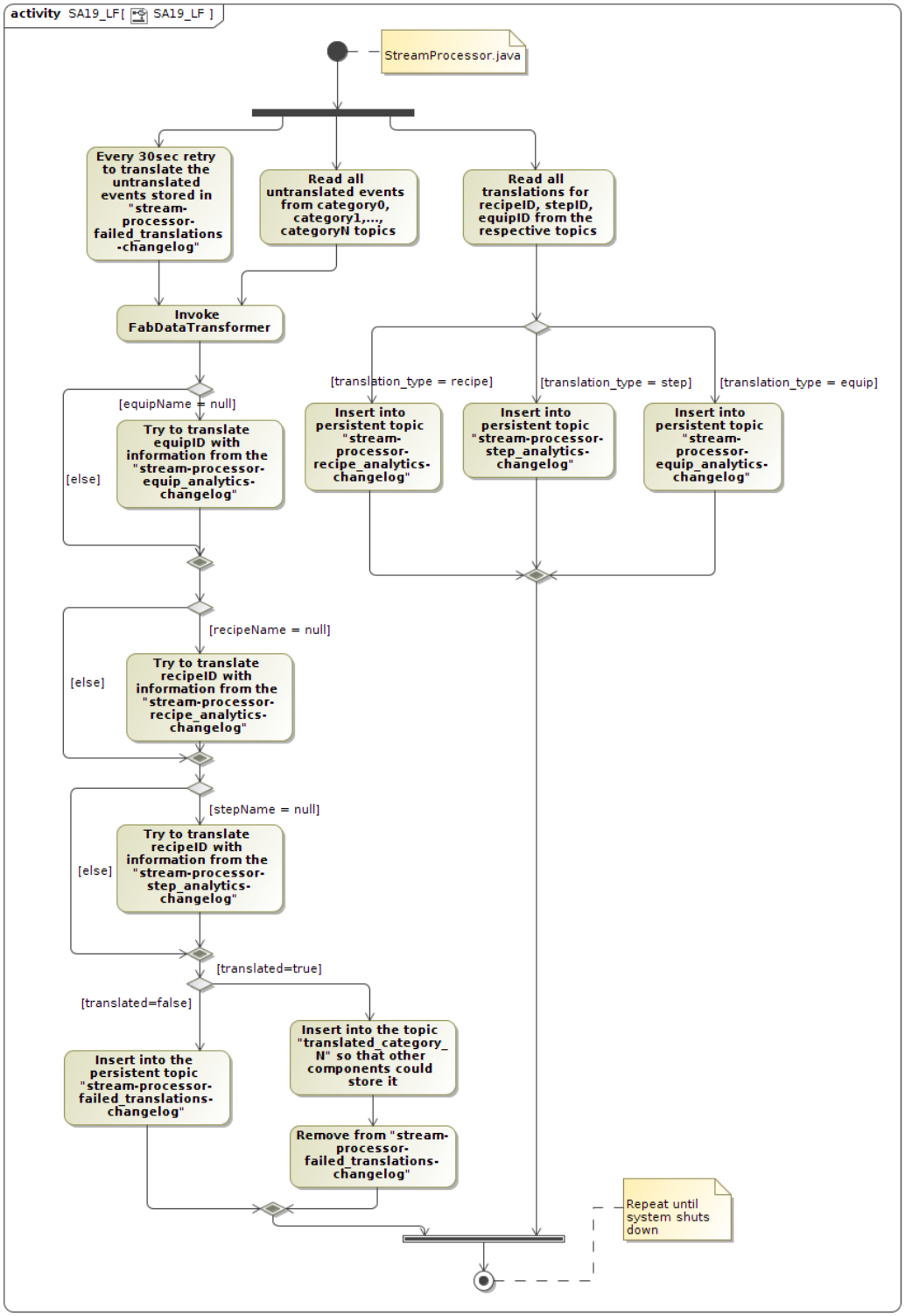

Figure 3.5: Activity diagram for translation logic.

3.2.3 Tests

In the requirements section we said that our system should be albe to scale up to 160.000 msg every 30 mins. This means almost 5300 msg every minute, so one msg every 11ms.

Table 3.1: Tests

Number of messages sent from simulator	**Time**
6.600	75837ms $\approx 1msg/11ms$

We did our tests on a single laptop, meaning that the pc was both running the simulator and the prototype of our system. So the kafka cluster was composed by only one pc. This means that in a production environment where there is a cluster of brokers among which the operations could be separated by using publish/subscribe architecture, our system could increase its performance. Also another thing to note about our tests is that the simulation of the external databases is computational expensive.

In order to test the size of the messages in the raw_data database we added a fake field called fakeData used to grow the size of the message. We tried our system with it but, given the fact that the simulation requires high computational power, team thinks that the final stats could be further improved.

3.3 Conclusion

The system architecture proposed in the first chapter is based on a publish/subscribe pattern, implemented using the Kafka framework. All the translated information are stored in the Kafka Streams local state and also in an external database (in our case a MongoDB's one). The architecture (and so the prototype) has a series of advantages provided by the Kafka framework, such as:

- If a producer loses the connection to the cluster it will wait for the connection to come back online without crashing or losing messages.
- If a consuming client loses the connection it will simply wait for the cluster to come back online and it will resume the consumption from where it left
- The information stored in the local state are replicated in the kafka cluster, in this way the final information are stored in a reliable and fault-tolerant manner.

- If there is a particular category in which the load of work is heavier with respect to the others, we can add more computational power to the kafka cluster thanks to its **horizontal scalability**.
- The components are decoupled from each other, this is a typical advantage of publish/subscribe architecture.

So, to summarize, we can say that the chosen architecture has proven us to be able to scale up to the requested messages ratio.

Appendices

Appendix A

Installation notes

Install mysql server and create an user with username *sa18* and password *software_architectures_18*. Grant to this user all the permission to perform CRUD operation.

In order to install kafka

```
cd /home/USER_NAME/Downloads (or "Scaricati")
sudo wget http://it.apache.contactlab.it/kafka/2.1.0/kafka_2.11-2.1.0.
    tgz

tar xvzf kafka_2.11-2.1.0.tgz
mkdir /opt/kafka
sudo mv kafka_2.11-2.1.0/ /opt/kafka
```

In order to use Kafka Connect you need to install mysql on the computer and the configure the log of the DBMS. Edit the file */etc/mysql/my.cnf* by adding the line log_bin = mysql-bin :

Example:

```
[mysqld]
server-id = 42
log_bin = mysql-bin
binlog_format = row
binlog_row_image = full
expire_logs_days = 10
```

and restart the mysql server.

Then we need to create two services, one for zookeeper and one for kafka.

```
sudo nano /etc/systemd/system/zookeeper.service
```

with code:

```
 [Unit]
Description=Apache Zookeeper server (Kafka)
```

```
Documentation=http://zookeeper.apache.org
Requires=network.target remote-fs.target
After=network.target remote-fs.target

[Service]
Type=simple
User=YUOR_USER
Environment=JAVA_HOME=/usr/java/java1.8.0
ExecStart=/opt/kafka/bin/zookeeper-server-start.sh /opt/kafka/config/
    zookeeper.properties
ExecStop=/opt/kafka/bin/zookeeper-server-stop.sh

[Install]
WantedBy=multi-user.target
```

Note that you need to set JAVA_HOME and profile name according to your system and that the used configuration file (i.e. config/zookeeper.properties) is the default one. Then start it with:

```
sudo systemctl start zookeeper.service
```

For the kafka service:

```
sudo nano /etc/systemd/system/kafka.service
```

with code:

```
[Unit]
Description=Apache Kafka server (broker)
Documentation=http://kafka.apache.org/documentation.html
Requires=network.target remote-fs.target
After=network.target remote-fs.target kafka-zookeeper.service

[Service]
Type=simple
User=YOUR_USER

Environment=JAVA_HOME=/usr/java/java1.8.0
ExecStart=/opt/kafka/bin/kafka-server-start.sh /opt/kafka/config/server.
    properties
ExecStop=/opt/kafka/bin/kafka-server-stop.sh
```

Note that you need to set JAVA_HOME and profile name according to your system and that the used configuration file (i.e. config/server.properties) is the default one. Then start it with:

```
sudo systemctl start kafka.service
```

For the kafka connect:

```
sudo nano /etc/systemd/system/connect.service
```

with code:

```
  [Unit]
Description=Apache Kafka Connect
Documentation=http://kafka.apache.org/documentation.html
Requires=network.target remote-fs.target
After=network.target remote-fs.target kafka-zookeeper.service

[Service]
Type=simple
User=YOUR_PROFILE

Environment=JAVA_HOME=/usr/java/java1.8.0
ExecStart=/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/
    connect-distributed.properties
```

Note that you need to set JAVA_HOME and profile name according to your system and that the used configuration file (i.e. config/connect-distributed.properties) is the default one **plus** the following line:

```
plugin.path=/opt/kafka/connect/
```

In this folder you need to put the debezium CDC plugin for your specific DBMS server and the debezium sink for the sink database, downloadable at https://debezium.io/docs/install/ (unzipped) and at https://www.confluent.io/connector/kafka-connect-mongodb-sink/.

Start it with:

```
sudo systemctl start connect.service
```

Then we need to import the fab_data database from its fab_data.sql file and the raw_data database from raw_data.sql. Subscribe our CDC connector by making a POST request to our system at the port in which kafka connect is running.

```
curl -i -X POST -H "Accept:application/json" \
   -H "Content-Type:application/json" http://localhost:8083/connectors/
       \
   -d '{
     "name": "mysql-connector",
     "config": {
           "connector.class": "io.debezium.connector.mysql.MySqlConnector
               ",
           "database.hostname": "localhost",
           "database.port": "3306",
           "database.user": "sa18",
           "database.password": "software_architectures_18",
           "database.server.id": "42",
           "database.server.name": "sa18",
```

```
            "database.history.kafka.bootstrap.servers": "localhost:9092",
            "database.history.kafka.topic": "dbhistory.sa18",
            "include.schema.changes": "true",
            "max.request.size":"104857600"
        }
    }'
```

Note this last curl operation needs to be done each time you restart the connect.service service.

The following is used to register the connector used to insert all the translated information into the MongoDB database. Note that the database, in this case "connect", needs to be created upfront.

```
curl -i -X POST -H "Accept:application/json" \
    -H "Content-Type:application/json" http://localhost:8083/connectors/
        \
    -d ' {
    "name": "mongodb-sink",
    "config": {
        "connector.class": "at.grahsl.kafka.connect.mongodb.
            MongoDbSinkConnector",
        "tasks.max": "1",
        "topics": "translated_categories",
        "mongodb.connection.uri": "mongodb://localhost:27017/connect",
        "mongodb.document.id.strategy":"at.grahsl.kafka.connect.mongodb.
            processor.id.strategy.BsonOidStrategy",
        "mongodb.collection": "events",
        "key.converter": "org.apache.kafka.connect.json.JsonConverter",
        "key.converter.schemas.enable": false,
        "value.converter": "org.apache.kafka.connect.json.JsonConverter",
        "value.converter.schemas.enable": false
    }
}'
```

In order to run the prototype there is also the need to use an application JavaEE server for the web application, in fact given the fact that the dashboard is based on a RESTful service, we used TomEEplus container.

Download it from http://tomee.apache.org/download-ng.html and configure it (set CATALINA_HOME and JAVA_HOME).

Finally we need to create some topics that our system needs (the majority of them are auto created by it).

```
./kafka-topics.sh --create --zookeeper localhost:2181 --replication-
    factor 1 --partitions 1 --topic translated_categories
./kafka-topics.sh --create --zookeeper localhost:2181 --replication-
    factor 1 --partitions 1 --topic sa18.raw_data.analytics
./kafka-topics.sh --create --zookeeper localhost:2181 --replication-
    factor 1 --partitions 1 --topic step_analytics
```

```
./kafka-topics.sh --create --zookeeper localhost:2181 --replication-
    factor 1 --partitions 1 --topic recipe_analytics
./kafka-topics.sh --create --zookeeper localhost:2181 --replication-
    factor 1 --partitions 1 --topic equip_analytics
./kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --
    entity-name sa18.raw_data.analytics --alter --add-config max.message.
    bytes=104857600
```

The last command is very important, because it sets the topic size to a number able to process the 10Mb information requested from the raw_data database.