# Sapienza University of Rome Laboratory of Advanced Programming

# EDUHUB

#### Shlok Bharuka, Domenico Cesare Lattari

Sommario

[Sapienza University of Rome 1](#_bookmark0)

[Laboratory of Advanced Programming 1](#_bookmark1)

[EDUHUB 1](#_bookmark2)

[Abstract 3](#_bookmark3)

[Introduction 3](#_bookmark4)

[User Stories 5](#_bookmark5)

1. [User stories: “As a user (superadmin or security administrator) I want to perform the login to](#_bookmark6) [the system so that I can use the software.” 5](#_bookmark6)
2. [User stories: “As a superadmin I want to insert a new user in the system so that I can](#_bookmark7) [authorize it to use the application with the right role.” 9](#_bookmark7)
3. [User stories: “As a superadmin I want to edit a user in the system so that I can authorize a](#_bookmark8) [new security administrator to use the software.” 14](#_bookmark8)
4. [User stories: “As a superadmin I want to view and filter the user's information so that I can](#_bookmark9) [perform my research.” 18](#_bookmark9)
5. [User stories: “As a superadmin I want to delete a user in the system so that I can remove the](#_bookmark10) [users that are not needed.” 21](#_bookmark10)
6. [User stories: “As a System Administrator I want to see the list of the anomalies detected by](#_bookmark11) [hosts applications so that I can have the control of the entire architecture.” 25](#_bookmark11)
7. [User stories: “As a System Administrator I want to run a background process on the host](#_bookmark12) [target to check a set of preestablished triggers, so that I can see if some anomalies occur.” 29](#_bookmark12)
8. [User stories: “As A system Administrator I want to perform the action "reset to green" if some](#_bookmark13) [anomalies occur (after that a manually intervent is done), so that I see only the anomalies that](#_bookmark13)

[aren't checked.” 32](#_bookmark13)

[Technologies 35](#_bookmark14)

[Scrum 36](#_bookmark15)

[Estimation (Function Points C COCOMO II) 38](#_bookmark16)

[Code 39](#_bookmark17)

# Abstract

In most scenarios, many operating systems are running, each with a particular task. We can consider strategic infrastructures such as hospitals, banks, companies, or systems vital for an entire country (such as water systems, energy systems, government, national defense, etc.).

In many cases, there exist generic solutions that offer common services which can be a good fit for medium to low emergencies but are insufficient for very critical infrastructure.

In the event of a cyber-attack, it is most important to react immediately to understand the gravity of the situation and respond with appropriate countermeasures. This is why we are presenting this project, which allows interaction with any type of target host to detect anomalies (potentially critical situations).

By leveraging Watchdog microservices within Docker containers, this solution improves the analysis of security logs, providing real-time anomaly detection. The purpose of this system is to offer a robust and scalable approach to security log analysis, ensuring prompt identification of potential threats and irregularities.

Security logs generated by Linux-based systems contain crucial information for identifying security threats. However, manual log inspection becomes impractical as system scale increases, necessitating automated and efficient anomaly detection mechanisms.

The proposed system overcomes challenges faced by traditional log viewing systems by utilizing a microservices architecture, Docker containerization, AMPQ messaging, and distributed databases, providing a comprehensive solution for security log analysis. This architecture excels in real-time anomaly detection by continuously monitoring security logs and rapidly identifying patterns indicative of potential security threats.

The proposed system represents a scalable, adaptable, and proactive approach to securing Linux-based systems, combining real-time anomaly detection with efficient log retrieval mechanisms. Our solution demonstrates how a distributed architecture can address maximum scalability challenges, allowing companies with distributed workforces across the globe to monitor potential anomalies in the system.

# Introduction

A distributed software architecture is an approach to designing and implementing software systems where the functionality and components of the software are distributed across multiple nodes or machines within a network. In this architecture, different software components can run on separate physical machines, virtual servers, or containers. These components communicate with each other over a network, often using standard communication protocols such as HTTP, TCP/IP, or other specific protocols.

The aim of distributed software architecture is to enhance scalability, availability, and reliability of the system. By breaking the system into distributed components, it's possible to reduce the load on individual nodes and balance the workload across multiple processing resources. Additionally, this architecture enables greater fault tolerance, as a failure on a single machine does not compromise the entire system.

One of the primary goals of distributed software architecture is to make the physical distribution of software components transparent to end-users. This means that users shouldn't have to worry about where the different components are running or how they are connected to each other. The distributed architecture should provide a unified interface for accessing the system, regardless of its physical distribution.

Distributed systems offer several advantages over monolithic, or single, systems:

* **Scalability & Flexibility:** Distributed systems make it easier to add computing power as the need for services grows. With the ability to spin up servers on demand, distributed systems can dynamically scale to handle increased workloads, thereby enhancing performance and reducing time to completion.
* **Fault Tolerance:** By distributing components across multiple nodes, distributed systems mitigate the risks associated with a single point of failure. This bolsters reliability and fault tolerance, ensuring that the system remains operational even if individual nodes fail.
* **Reliability:** A well-designed distributed system can withstand failures in one or more of its nodes without significantly impacting performance. In contrast, a monolithic system may experience complete application downtime if the server fails, leading to service disruptions.
* **Speed:** Heavy traffic can overwhelm single servers in monolithic systems, resulting in performance degradation for all users. Distributed systems, with their scalability and load balancing capabilities, can maintain high-performance levels even under heavy loads, thus ensuring optimal user experience.
* **Geo-Distribution:** Distributed content delivery is not only intuitive for internet users but also essential for global organizations. By distributing content across multiple geographic locations, distributed systems can reduce latency and improve the user experience for individuals accessing services from different parts of the world.

Microservices, or microservices architecture, is an approach to the design and implementation of enterprise applications in which a large application is built from modular components or services. Each module supports a specific task or business goal and uses a well-defined communications interface, such as an application programming interface (API), to communicate with other modules and services. Each task or service is created independently, and each one runs a unique process and usually manages its own database. A service can generate alerts, log data, support user interfaces (UIs), handle user identification or authentication, and perform various other computing and processing tasks.

A microservices architecture makes extensive use of virtual container and networking technologies, and is noted for its streamlined module development, deployment and scalability, characteristics that are particularly well-suited to application development for modern public clouds.

Typical characteristics of a microservices design and architecture include the following:

* **Unique components:** Services are designed and deployed as individual components working together to accomplish a specific function or address a specific requirement.
* **Decentralized:** Unique microservices components have few if any dependencies, although loose coupling requires frequent and extensive communication between components.
* **Resilient**: Services are designed for maximum fault tolerance. A single service failure shouldn't disable an entire application. This often requires excellent software design and site reliability engineering (SRE) practices, as well as redundant deployment and failover and high scalability techniques.
* **API-based:** A microservices architecture relies on APIs and API gateways to facilitate communication between components and other applications.
* **Data separation:** Each service accesses its own database or storage volume.
* **Automation:** Microservices application components can be many and can be cumbersome to deploy manually. Microservices rely on automation and orchestration technologies for component deployment and scaling.

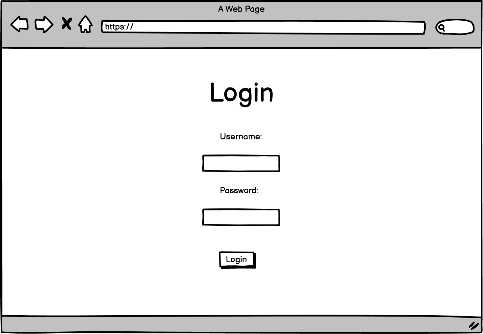
Docker is a popular platform that facilitates the development, deployment, and management of applications using containerization technology. Containers are lightweight, portable, and isolated environments that

encapsulate an application and its dependencies, ensuring consistency across different environments. Docker enables developers to package their applications into containers, along with all the necessary libraries and dependencies, making it easy to deploy them on any infrastructure that supports Docker. This approach streamlines the development process, improves deployment efficiency, and enhances scalability and reliability.

## User Stories

##### User stories: “As a user (superadmin or security administrator) I want to perform the login to the system so that I can use the software.”

### Mockup:



**Microservices Involved:**

The authentication process is facilitated by the Authentication Services, that is a microservice that allows the generation, the signature and sharing of the most important information with the JWT. This information are shared after the authentication process is done. Initially, we implemented a basic filter mechanism to handle incoming requests. Later on, we enhanced this mechanism by introducing a filter chain within the framework. This filter chain is responsible for generating and signing the JWT token after the authentication process is completed successfully. Once the JWT token is generated and signed, it is sent to the resource server (like the user-service or the other services involved in the process) and subsequently to the end user. When a user needs to interact with the resource server, they must include the JWT token in the HTTP request header. This token is obtained from the authentication service during the authentication process. Both the authentication service and the resource service share a secret key (for simplicity a symmetric key, but in a real scenario is better to use a public/private key mechanism. Using HMAC (Hash Based Message Authentication Code) signature, they verify the authenticity and validity of the token. This cryptographic system ensures the authenticity and integrity of the JWT token. The JWT token contains all relevant information about the user, such as the list of authorities or permissions associated with the user. After receiving the token, the resource server authorizes (after verifying the JWT signature) the user to access specific resources based on the information contained within the token. This authorization is possible because the JWT token is signed, and the cryptographic system guarantees its authenticity.

### Technical Choices:

###### Spring Boot:

* + We opted to build our microservice using Spring Boot, a powerful and widely adopted framework for developing Java-based applications.
  + Spring Boot offers rapid application development, auto-configuration, and a convention-over- configuration approach, streamlining the development process and reducing boilerplate code.

###### Spring Security:

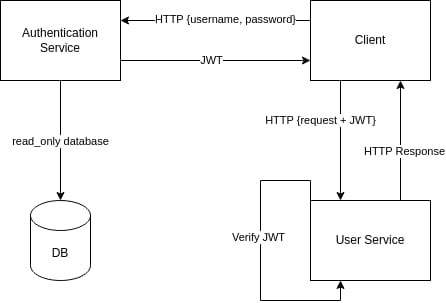
* + Spring Security serves as the cornerstone of our authentication and authorization framework.
  + Leveraging Spring Security, we can implement comprehensive security features such as authentication, authorization, and access control with ease.
  + Spring Security provides robust support for various authentication mechanisms, including JWT, OAuth, and form-based authentication, allowing us to choose the approach that best fits our application's requirements.

###### JWT (JSON Web Tokens):

* + We chose JSON Web Tokens (JWT) as the authentication mechanism for our microservice.
  + JWT offers a stateless and secure way to transmit authentication information between the client and the server.
  + By encoding user information into JWT tokens, we can achieve lightweight and scalable authentication without the need for server-side session management.

###### Role-Based Authentication System:

* + Our authentication system employs a role-based access control model to manage user permissions.
  + Roles encapsulate sets of permissions, allowing us to define granular access control policies based on user roles.
  + Users are assigned roles upon authentication, and their permissions determine the actions they can perform within the system.



### Acceptance criteria:

1. A user with a valid pair of username and password is able to perform the login only if the "disabled" field is set to false.
2. A user with a valid pair of username and password is not able to perform the login if the "disabled" field is set to true.
3. A valid user receives a valid JWT token with the list of all roles.
4. An invalid user (wrong username or password) is not able to obtain a valid JWT token.

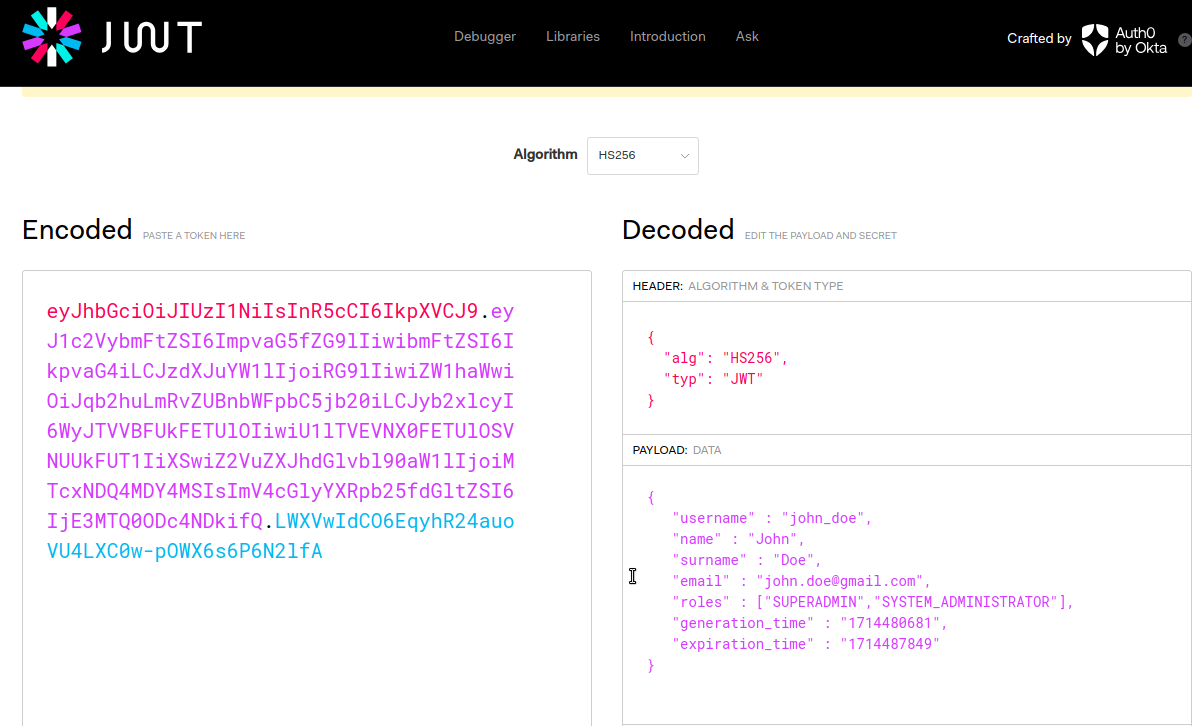
### Function Point:

Description:

The user, insert the username and password in any login form (UI Form, Application Android/IOS, etc) then produce an http request with username and password plain.

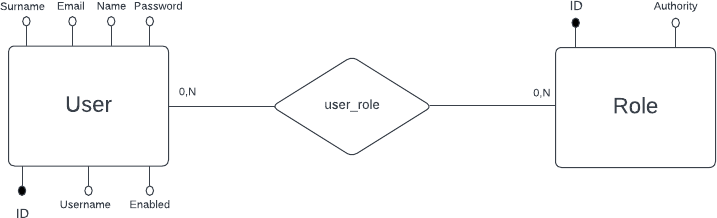
The request is sent to the Authentication Server, there, if the authentication has been successful, is returned a JWT token with HTTP Status Code 200; otherwise, an http 403 Error.

The JWT Token has the following structure:



The database is managed by an outside boundary service and this service does only reads.

###### ER Schema



Then the database translation is the following:

* User (ID, Email, Username, Password, Name, Surname, Enabled) PK: ID

UNQ [Email] UNQ [Username]

* Role (ID, Authority) PK: ID

UNQ: Authority

* User-Role (USED\_ID, ROLE\_ID)

PK: [User\_ID, Role\_ID]

Internal Logical Files (ILF): None, the database is managed externally. External Interface File (EIF): User, Roles, User\_Roles. Then we have 3 EIF.

External Input (EI): The http request with username and password field. Then we could represent this function has:

* EI\_1: Check username, password.
* EI\_2: Load roles of a user-id.

External Output (EO): Produce the JWT response. External Inquiry (EQ): None.

### ILF/EIF COMPLEXITY

ILF: None.

EIF\_ user: 7 DET (username, password, id, enabled, name, surname, email). EIF\_roles: 2 DET (ID, authority).

EIF\_ user-roles: 2 DET (User\_ID, Role\_ID). Then:

* RET= 3 (user, role and user-role)

- DET= 7+2+2=11

From the table:



So, EIF have Low Complexity with **5 FP.**

### EI Computation

**EI\_1:** Check username, password, enable => FTR= 1 (user) and DET=3 (username, enabled, password). From the table:



So, EI\_1 have Low Complexity with **3 FP**.

**EI\_2**: Load roles by user\_ID => FTR= 2 (user\_role, role) and DET= 3 (authority, user\_id, role\_id).

So, EI\_2 have Low Complexity with **3 FP**. Then EI= EI\_1 + EI\_2= 3+3= **6 FP**.

### EQ/EO calculation

EQ=None.

EO: Produce JWT => FTR (user, role, user-role) = 3; DET= 5 (username, name, email, role, surname). From the table:



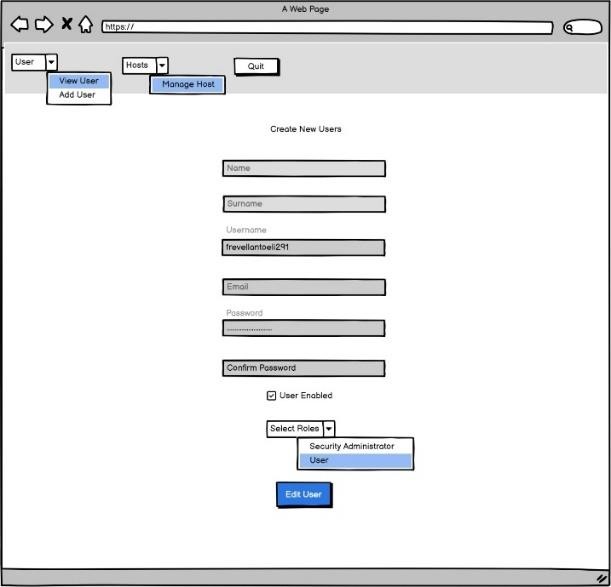
EO have Low Complexity with **4 FP**.

The total, UFP, is the sum of:

ILF + EI + EO + EQ +EIF = 0+6+4+0+5 =**15**.

##### User stories: “As a superadmin I want to insert a new user in the system so that I can authorize it to use the application with the right role.”

### Mockup:



**Microservices Involved:**

The user management part, when it comes to insertion, involves several steps. Initially, the client application, such as the user interface (UI), constructs an HTTP request. This request must include a JWT token in the header, which is necessary for authentication. Once the request is constructed, the client sends it to the user service.

The user service receives the request and begins the verification process. First, it extracts the JWT token from the request header and verifies its signature to ensure authenticity. Next, the user service checks the roles associated with the user contained in the JWT token. This check is performed to determine if the user has the required permission to perform the insertion operation.

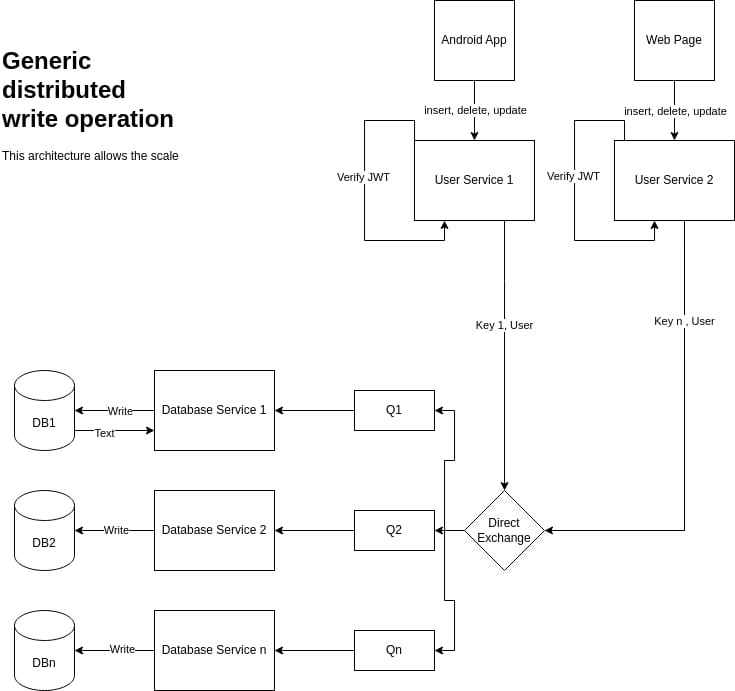
If the user has the required permissions, the user service proceeds by inserting a message into a queue. This queue is managed through a Direct Exchange policy, which is responsible for routing messages to the appropriate queues. In our case, the message is sent to a specific queue for user management.

A listener, which in our context is the database service, receives the message from the queue. The listener interprets the message and recognizes it as an insertion request. Using the information provided in the message, the listener verifies the data's consistency with the database constraints.

Once the data's consistency is verified, the listener proceeds with the actual insertion of the data into the database. This step is crucial as the database service is the only microservice authorized to directly access the database for data manipulation.

After completing the insertion into the database, the database service sends an ACK (acknowledgment) to the user service. This ACK contains a response indicating whether the insertion was successful or not.

In the end, the user service sends the ID of the new user as a response to the UI. This ID is a unique reference for the user inserted into the system.



### Technical Choices:

###### Spring Boot:

* + - Utilizing Spring Boot for developing the user service ensures rapid development and streamlined configuration of the microservice.
    - Spring Boot's convention-over-configuration approach reduces boilerplate code, enhancing productivity during development.

###### Spring Security:

* + - Employing Spring Security enables robust authentication and authorization mechanisms within the user service.
    - Spring Security provides comprehensive support for securing HTTP requests, including JWT token verification and role-based access control.

###### JWT (JSON Web Tokens):

* + - Adopting JWT as the authentication mechanism ensures secure transmission of user authentication data between the client application and the user service.
    - JWT tokens encapsulate user information and permissions, facilitating lightweight and stateless authentication processes.

###### Message Queue Implementation:

* + - Implementing a message queue system for user insertion operations allows for asynchronous communication between components and enhances system scalability.
    - Utilizing an Exchange and queues ensures reliable message routing and processing, contributing to overall system reliability.

###### Listener Design Pattern:

* + - Employing the listener design pattern within the database service enables efficient handling of insertion requests received from the message queue.
    - The listener interprets incoming messages and verifies data consistency with database constraints before proceeding with data insertion.

###### Database Service Authorization:

* + - Restricting direct database access to the database service enhances security and data integrity within the system.
    - Only authorized microservices, such as the database service, are permitted to directly access the database for data insertion operations.

###### ACK (Acknowledgment) Mechanism:

* + - Implementing an acknowledgment mechanism between the database service and the user service ensures reliable communication and error handling.
    - The ACK message contains information indicating the success or failure of the insertion operation, enabling proper feedback to the user interface.

###### Response Handling:

* + - Sending the ID of the newly inserted user as a response to the user interface facilitates seamless user interaction and provides a unique reference for the inserted user within the system.
    - Proper response handling enhances user experience and ensures smooth integration with the client application.

### Acceptance criteria:

1. A user with the right role (superadmin) is able to insert a new user into the system only if the inserted user is not already present in the system (e.g., email, username, etc.).
2. During the insertion step, it is possible to establish a relationship between the roles and the user.
3. During insertion, it is possible to set a new password.
4. Passwords in the database are hashed using the BCrypt algorithm, ensuring that the same input generates different output.
5. The user will receive an ACK (Acknowledgment) indicating the result of the operation (SUCCESS OR FAIL).

### Function Point:

External Inputs (EI):

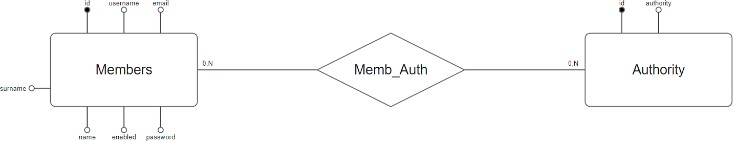
* + Http Request User Controller: HTTP Request sent from the UI.
  + Message Read from Queue: when the database service reads a message from the queue. External Outputs (EO):
  + ACK sent from Database Service.
  + Publish message to the queue: when user controller publishes the message to the queue.
  + Http response from the User Controller to the UI. External Inquiries (EQ)
  + JWT Verification: the verification process by user controller involves external inquiry to the authentication service.

Internal Logic File (ILF):

* + Data storage in database (user inserted) by database service. External Interface Files (EIF): None

**ILF/EIF COMPLEXITY**

ER Schema



* + - Members (id, username, email, password, name, surname, enable) PK [ id]

UK [ username] UK [ email]

* + - Authority (id, authority\_id) PK [ id]
    - Memb\_auth (member\_id, authority\_id) PK [ member\_id, authority\_id]

FK [ member\_id] ⊆ Member [ id] FK [authority\_id] ⊆ Authority [ id]

In this case we have 3 RET (Members, Authority and Memb\_Auth) and in total we have 11 DET.



So ILF is **LOW (7).**

EIF is **0**.

### EI Computation:

1. Http Request from UI.
2. Message read in the Queue. FTR= # of ILF/EIF involved in the EI.

DET= # of Fields involved in the process. In the first case:

* + There are involved 3 ILF (Members, Authority and Memb\_Auth).
  + For the http request the fields are username, email, password, name, surname, enable, authority.



So, from the table, in the first case we have 3 FTR and 7 DET => EI\_1: HIGH (6).

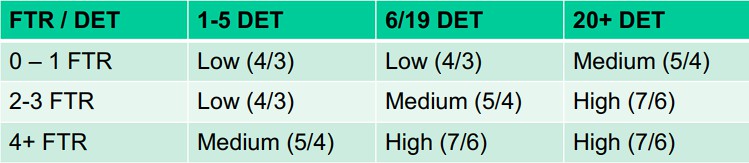
In the second case:

* + There is a Json object with the following fields: operationType, username, email, password, name, surname, enabled, authorities, so in total we have 8 DET.
  + The ILF involved are 3 (Members, Authority and Memb\_Auth). In the second case we have 3 FTR and 8 DET=> EI\_2: HIGH (6).

So, in total we have EI= EI\_1+EI\_2= 6+6=**12**.

### EO Complexity

1. ACK sent from Database Service.
2. Publish Message to the Queue (User Controller).
3. Http Response to the UI.
4. The ACK contains the following infos:
   * Message, Payload, Success / Failed so in total 3 DET.
   * There is no ILF involved in the process.



So for the first case we have EO\_1: LOW (4).

1. In the second case:
   * Message published by UserController contains: username, email, name password, surname, enable, authorities.
   * The ILF involved are 3 (Members, Authority and Memb\_Auth). So, for the second case we have EO\_2: MEDIUM (5).
2. The third case we have a http code so is gonna be EO\_3: LOW (3). So, in total we have EO=EO\_1+EO\_2+EO\_3= 4+5+3=**12**.

### EQ complexity

* + JWT verification include 1 ILF (key secret) and 3 DET (Header, Payload and Signature).

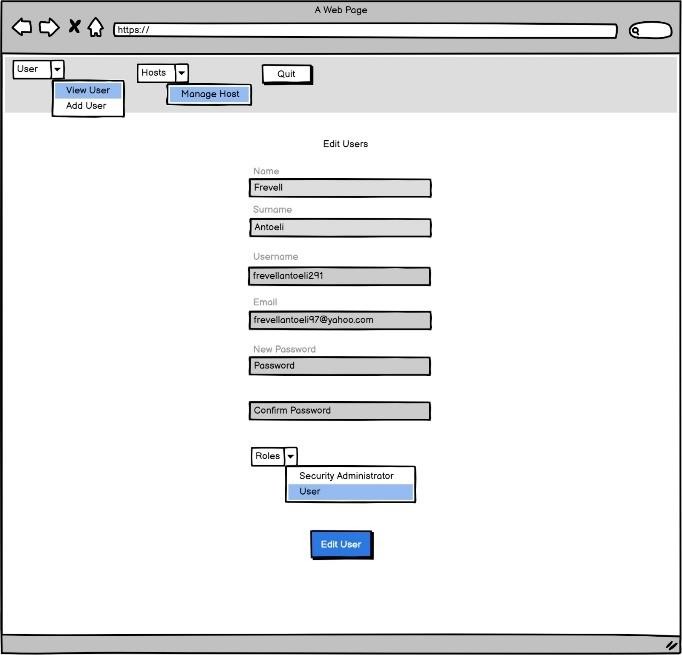
Then we have for EQ: **LOW (4).**

The total UFP is the sum of:

ILF + EI + EO + EQ +EIF = 7+12+12+4+0 =35.

##### User stories: “As a superadmin I want to edit a user in the system so that I can authorize a new security administrator to use the software.”

### Mockup:



**Microservices Involved:**

Similarly, the user service handles modification tasks in a comparable manner. The user service, responsible for validating tokens received from the UI, first verifies the token's correctness. Upon successful verification, it generates a message destined for a queue. However, instead of directly depositing the message into the queue, it routes it through an Exchange. This Exchange serves as an intermediary, facilitating the processing of the message before depositing it into the queue. Subsequently, the message is dispatched to the queue where a Listener, representing the database service, awaits. The database service interprets the message, identifying the modification operation it needs to execute. Once the database service determines the required operation, it promptly executes it, effectively writing the pertinent information within the database.

### Technical Choices:

###### Spring Boot:

* + Leveraging Spring Boot for the user service ensures rapid development and simplified configuration, enhancing productivity and maintainability.

###### Spring Security:

* + Utilizing Spring Security enables robust authentication and authorization mechanisms within the user service.
  + Spring Security facilitates token validation and role-based access control, ensuring secure user authentication and authorization processes.

###### Message Queue System:

* + Employing a message queue system for handling modification tasks enables asynchronous communication between components and enhances system scalability.
  + Messages for user modification are routed through an Exchange before being deposited into the queue, facilitating message processing and routing.

###### Listener Design Pattern:

* + Implementing the listener design pattern within the database service allows efficient handling of modification requests received from the message queue.
  + The listener interprets incoming messages, identifies the modification operation, and executes it promptly within the database.

###### Database Interaction:

* + Direct interaction with the database service ensures secure and controlled modification of user information within the database.
  + The database service executes modification operations effectively, updating relevant information within the database based on the received message.

###### Error Handling:

* + Implementing robust error handling mechanisms ensures proper handling of exceptional scenarios during user modification operations.
  + Error handling mechanisms within the user service and database service facilitate graceful recovery and error reporting.

###### Response Handling:

* + Proper response handling within the user service ensures timely feedback to the user interface regarding the success or failure of the modification operation.
  + Sending appropriate response messages to the user interface enhances user experience and facilitates seamless interaction with the application.

### Acceptance criteria:

1. The user with the right role can edit the users in the system.
2. The user with the right role (superadmin) will be able to update the roles and password for the target user.
3. The user with the superadmin role can deny the login of the target user by disabling the login action.
4. After the update request is issued, an ACK will be sent back with information about the success or failure.

### Function Point:

External Inputs (EI):

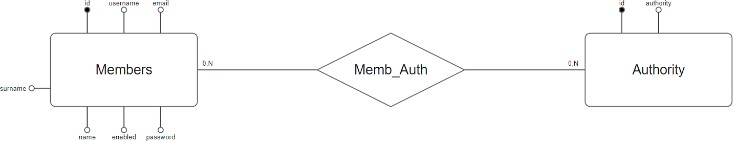
* + Http Request User Controller: HTTP Request sent from the UI.
  + Message Read from Queue: when the database service reads a message from the queue. External Outputs (EO):
  + ACK sent from Database Service.
  + Publish message to the queue: when user controller publishes the message to the queue.
  + Http response from the User Controller to the UI. External Inquiries (EQ)
  + JWT Verification: the verification process by user controller involves external inquiry to the authentication service.

Internal Logic File (ILF):

* + Data storage in database (user edited) by database service. External Interface Files (EIF): None.

**ILF/EIF COMPLEXITY**

ER Schema



* + - Members (id, username, email, password, name, surname, enable) PK [ id]

UK [ username] UK [ email]

* + - Authority (id, authority\_id) PK [ id]
    - Memb\_auth (member\_id, authority\_id) PK [ member\_id, authority\_id]

FK [ member\_id] ⊆ Member [ id] FK [authority\_id] ⊆ Authority [ id]

In this case we have 3 RET (Members, Authority and Memb\_Auth) and in total we have 10 DET.



So ILF is **LOW (7).**

EIF is **0**.

### EI Computation:

1. Http Request from UI.
2. Message read in the Queue. FTR= # of ILF/EIF involved in the EI. DET= # of Fields involved in the process. In the first case:
   * There are involved 3 ILF (Members, Authority and Memb\_Auth).
   * For the http request the fields are username, email, password, name, surname, enable, authority.



So, from the table, in the first case we have 3 FTR and 7 DET => EI\_1: HIGH (6).

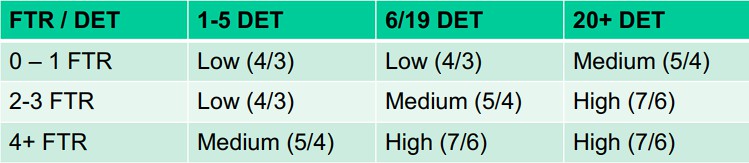
In the second case:

* + There is a Json object with the following fields: operationType, username, email, password, name, surname, enabled, authorities, so in total we have 8 DET.
  + The ILF involved are 3 (Members, Authority and Memb\_Auth). In the second case we have 3 FTR and 8 DET=> EI\_2: HIGH (6).

So, in total we have EI= EI\_1+EI\_2= 6+6=**12**.

### EO Complexity

1. ACK sent from Database Service.
2. Publish Message to the Queue (User Controller).
3. Http Response to the UI.
4. The ACK contains the following infos:
   * Message, Payload, Success / Failed so in total 3 DET.
   * There is no ILF involved in the process.



So for the first case we have EO\_1: LOW (4).

1. In the second case:
   * Message published by UserController contains: username, email, name password, surname, enable, authorities .
   * The ILF involved are 3 (Members, Authority and Memb\_Auth). So, for the second case we have EO\_2: MEDIUM (5).
2. The third case we have a http code so is gonna be EO\_3: LOW (3). So, in total we have EO=EO\_1+EO\_2+EO\_3= 4+5+3=**12**.

### EQ complexity

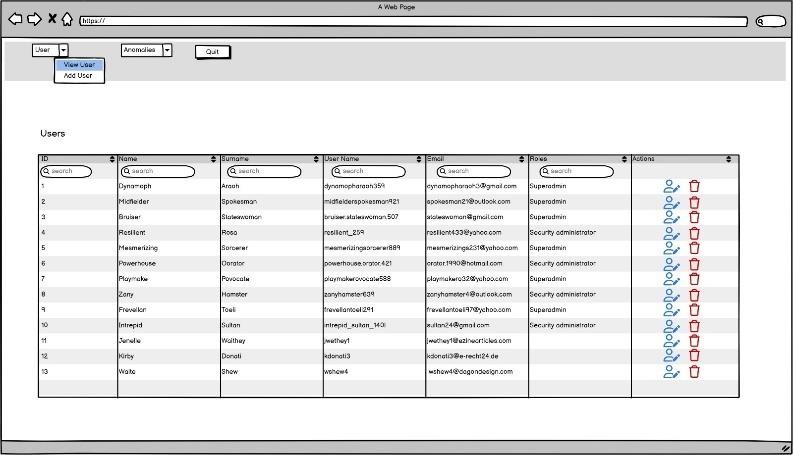
* + JWT verification include 1 ILF (key secret) and 3 DET (Header, Payload and Signature). Then we have for EQ: **LOW (4).**

The total UFP is the sum of:

ILF + EI + EO + EQ +EIF = 7+12+12+4+0 =35.

##### User stories: “As a superadmin I want to view and filter the user's information so that I can perform my research.”

### Mockup:



**Microservices Involved:**

Similarly, the user service handles visualization tasks in a comparable manner. Upon receiving a JWT token, it proceeds to validate the token's signature. Upon successful validation, it generates a message directed to the exchange. This message is then relayed to the queue. Here, the database service intercepts the message, discerns the operation to be performed, queries the database accordingly, and retrieves the necessary data. Upon completion, the database service sends an acknowledgment (ACK) to the exchange, along with the requested data. Subsequently, this data flows back to the user service, which in turn delivers it to the final consumer.

### Technical Choices:

###### Spring Boot:

* + The user service is built using Spring Boot, offering rapid development and simplified configuration, enhancing efficiency and maintainability.

###### Spring Security:

* + Spring Security is utilized for authentication and authorization within the user service, ensuring secure access to user information.
  + Token validation mechanisms provided by Spring Security validate the authenticity of JWT tokens, enhancing security during user information retrieval.

###### Message Queue System:

* + A message queue system is employed for handling visualization tasks, facilitating asynchronous communication between components and enhancing scalability.
  + Messages for user information retrieval are routed through an Exchange before being processed and relayed to the queue, streamlining message handling.

###### Database Querying:

* + The database service executes queries based on received messages, retrieving user information from the database.
  + Utilizing efficient database querying techniques ensures timely retrieval of user data, optimizing system performance.

###### Acknowledgment Mechanism:

* + An acknowledgment (ACK) mechanism is implemented to ensure reliable communication between the database service and the exchange.
  + Upon retrieving user information, the database service sends an ACK to the exchange, indicating the successful completion of the operation.

###### Response Handling:

* + Proper response handling within the user service ensures timely delivery of user information to the final consumer.
  + User information retrieved from the database is processed and delivered to the user interface, facilitating user research activities.

###### Error Handling:

* + Robust error handling mechanisms are implemented to manage exceptional scenarios during user information retrieval.
  + Error handling within the user service and database service ensures graceful recovery and error reporting, enhancing system reliability.

### Acceptance criteria:

1. The user with the role superadmin can see all the users saved in the system with their complete information.
2. The user with the role superadmin can filter the users to improve the search.

### Function Point:

External Inputs (EI):

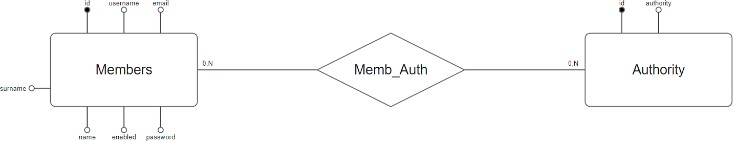
* + Http Request User Controller: HTTP Request sent from the UI.
  + Message Read from Queue: when the database service reads a message from the queue. External Outputs (EO):
  + ACK sent from Database Service.
  + Publish message to the queue: when user controller publishes the message to the queue.
  + Http response from the User Controller to the UI. External Inquiries (EQ)
  + JWT Verification: the verification process by user controller involves external inquiry to the authentication service.

Internal Logic File (ILF):

* + Data storage in database (user view) by database service. External Interface Files (EIF): None .

**ILF/EIF COMPLEXITY**

ER Schema



* + - Members (id, username, email, password, name, surname, enable) PK [ id]

UK [ username] UK [ email]

* + - Authority (id, authority\_id) PK [ id]
    - Memb\_auth (member\_id, authority\_id) PK [ member\_id, authority\_id ]

FK [ member\_id] ⊆ Member [ id ] FK [authority\_id] ⊆ Authority [ id ]

In this case we have 3 RET (Members, Authority and Memb\_Auth) and in total we have 11 DET.



So ILF is **LOW (7).**

EIF is 0.

### EI Computation:

1. Http Request from UI.
2. Message read in the Queue. FTR= # of ILF/EIF involved in the EI. DET= # of Fields involved in the process. In the first case:
   * There are involved 3 ILF (Members, Authority and Memb\_Auth).
   * For the http request the fields are: id, username, email, password, name, surname, enabled, authority.
   * 

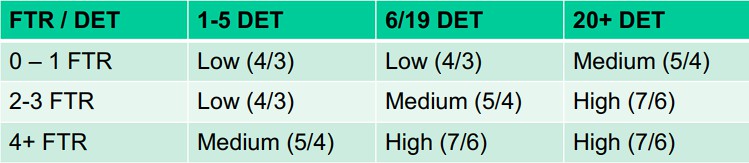
So, from the table, in the first case we have 3 FTR and 8 DET => EI\_1: HIGH (6). In the second case:

* + There is a Json object with the following fields: operationType, username, email, password, name, surname, enabled, authorities, so in total we have 8 DET.
  + The ILF involved are 3 (Members, Authority and Memb\_Auth). In the second case we have 3 FTR and 7 DET => EI\_2: HIGH (6).

So, in total we have EI= EI\_1+EI\_2= 6+6=**12**.

### EO Complexity

1. ACK sent from Database Service.
2. Publish Message to the Queue (User Controller).
3. http Response to the UI.
4. The ACK contains the following infos:
   * Message, Payload (The payload is not a field but a user object (id, username, etc.)), Success / Failed so in total 10 DET.
   * There is no ILF involved in the process.



So for the first case we have EO\_1: MEDIUM 5.

1. In the second case:
   * Message published by UserController contains: username, email, name password, surname, enable, authorities .
   * The ILF involved are 3 (Members, Authority and Memb\_Auth). So, for the second case we have EO\_2: MEDIUM (5).
2. The third case we have a http code so is gonna be EO\_3: LOW (3). So, in total we have EO=EO\_1+EO\_2+EO\_3= 5+5+3=**13**.

### EQ complexity

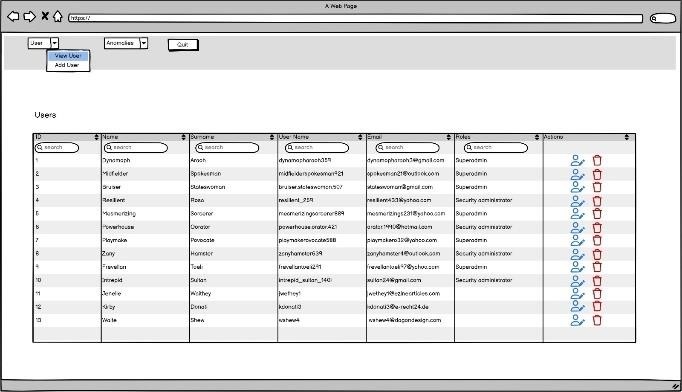
* + JWT verification include 1 ILF (key secret) and 3 DET (Header, Payload and Signature). Then we have for EQ: **LOW (4).**

The total UFP is the sum of:

ILF + EI + EO + EQ +EIF = 7+12+13+4+0 =**36**.

##### User stories: “As a superadmin I want to delete a user in the system so that I can remove the users that are not needed.”

### Mockup:



**Microservices Involved:**

The user service handles delete tasks similar to the other tasks.

When a delete request is initiated from the user interface (UI), it is sent to the user service. The user service, responsible for handling authentication and authorization, verifies the user's credentials and permissions. Upon successful verification, the user service generates a message intended for a queue.

Like the modification process, the user service doesn't directly deposit the message into the queue but routes it through an Exchange. This Exchange serves as an intermediary for processing the message before it reaches the queue. Once the message reaches the queue, a Listener, representing the database service, awaits its arrival.

The database service interprets the message, identifying it as a deletion operation. It then promptly executes the deletion operation within the database, removing the relevant information.

After successful deletion, the database service sends an acknowledgment back to the user service, confirming the deletion. The user service may then send a response to the UI, indicating the successful completion of the delete operation.

### Technical Choices:

###### Spring Boot:

* + - Utilizing Spring Boot for the user service ensures rapid development and streamlined configuration, enhancing productivity and maintainability.

###### Spring Security:

* + - Spring Security is employed for authentication and authorization within the user service, ensuring secure access to deletion functionalities.
    - Authentication mechanisms provided by Spring Security verify user credentials and permissions before initiating deletion operations, enhancing security.

###### Message Queue System:

* + - A message queue system is utilized for handling deletion tasks, enabling asynchronous communication between components and enhancing system scalability.
    - Messages for user deletion requests are routed through an Exchange before being processed and relayed to the queue, facilitating efficient message handling.

###### Listener Design Pattern:

* + - The database service employs the listener design pattern to efficiently handle deletion requests received from the message queue.
    - The listener interprets incoming messages, identifies deletion operations, and executes them promptly within the database, ensuring efficient deletion of user information.

###### Database Interaction:

* + - Direct interaction with the database service ensures secure and controlled deletion of user information within the database.
    - The database service executes deletion operations effectively, removing relevant user information from the database based on received messages.

###### Acknowledgment Mechanism:

* + - An acknowledgment (ACK) mechanism is implemented to ensure reliable communication between the database service and the user service.
    - Upon successful deletion of user information, the database service sends an ACK to the user service, confirming the deletion operation.

###### Response Handling:

* + - Proper response handling within the user service ensures timely communication of deletion status to the user interface.
    - Upon receiving the ACK from the database service, the user service sends a response to the user interface, indicating the successful completion of the deletion operation.

###### Error Handling:

* + - Robust error handling mechanisms are implemented to manage exceptional scenarios during user deletion operations.
    - Error handling within the user service and database service ensures graceful recovery and error reporting, enhancing system reliability.

### Acceptance criteria:

1. The user with the role superadmin is able to delete the user from the system.
2. After the delete operation, an ACK is sent back with the result of the operation (SUCCESS OR FAIL).

### Function Point:

External Inputs (EI):

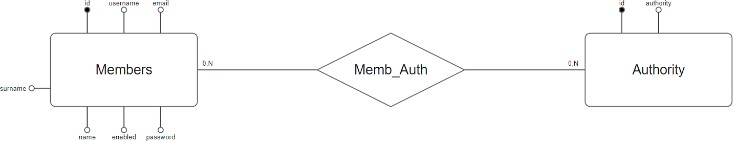
* + Http Request User Controller: HTTP Request sent from the UI.
  + Message Read from Queue: when the database service reads a message from the queue. External Outputs (EO):
  + ACK sent from Database Service.
  + Publish message to the queue: when user controller publishes the message to the queue.
  + Http response from the User Controller to the UI. External Inquiries (EQ)
  + JWT Verification: the verification process by user controller involves external inquiry to the authentication service.

Internal Logic File (ILF):

* + Data storage in database (user deleted) by database service. External Interface Files (EIF): None .

**ILF/EIF COMPLEXITY**

ER Schema



* + - Members(id, username, email, password, name, surname, enable) PK [ id]

UK [ username] UK [ email]

* + - Authority (id, authority\_id) PK [ id]
    - Memb\_auth (member\_id, authority\_id ) PK [ member\_id, authority\_id ]

FK [ member\_id] ⊆ Member [ id ] FK [authority\_id] ⊆ Authority [ id ]

In this case we have 3 RET (Members, Authority and Memb\_Auth) and in total we have 11 DET.



So ILF is **LOW (7).**

EIF is **0**.

### EI Computation:

1. Http Request from UI.
2. Message read in the Queue. FTR= # of ILF/EIF involved in the EI. DET= # of Fields involved in the process. In the first case:
   * There are involved 3 ILF (Members, Authority and Memb\_Auth).
   * For the http request the fields are id.



So, from the table, in the first case we have 3 FTR and 1 DET => EI\_1: MEDIUM (4).

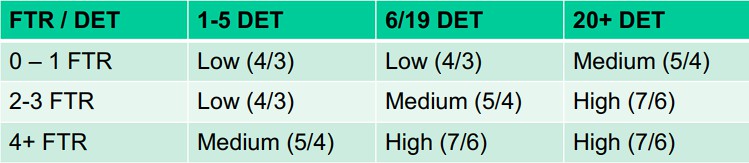
In the second case:

* + There is a Json object with the following fields: operationType, username, email, password, name, surname, enabled, authorities, so in total we have 8 DET.
  + The ILF involved are 3 (Members, Authority and Memb\_Auth). In the second case we have 3 FTR and 7 DET => EI\_2: HIGH (6).

So, in total we have EI= EI\_1+EI\_2= 6+4=**10**.

### EO Complexity

1. ACK sent from Database Service.
2. Publish Message to the Queue (User Controller).
3. http Response to the UI.
4. The ACK contains the following infos:
   * Message, Payload, Success / Failed so in total 3 DET.
   * There is no ILF involved in the process.



So for the first case we have EO\_1: LOW (4).

1. In the second case:
   * Message published by UserController contains: username, email, name password, surname, enable, authorities .
   * The ILF involved are 3 (Members, Authority and Memb\_Auth). So, for the second case we have EO\_2: MEDIUM (5).
2. The third case we have a http code so is gonna be EO\_3: LOW (3).

So, in total we have EO=EO\_1+EO\_2+EO\_3= 4+5+3=**12**.

### EQ complexity

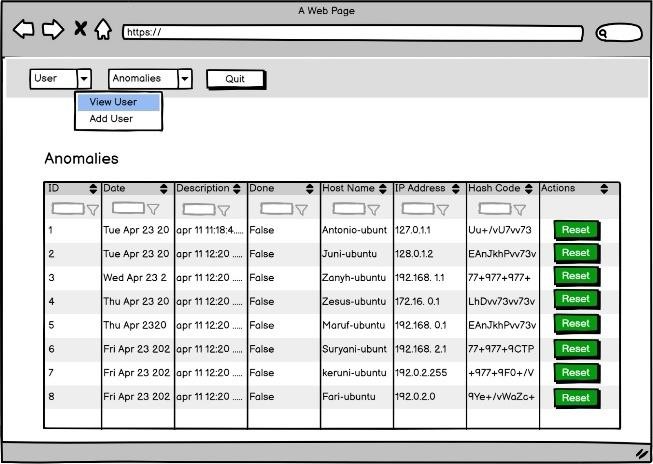
* + JWT verification include 1 ILF (key secret) and 3 DET (Header, Payload and Signature). Then we have for EQ: **LOW (4).**

The total UFP is the sum of:

ILF + EI + EO + EQ +EIF = 7+10+12+4+0 =**31.**

##### User stories: “As a System Administrator I want to see the list of the anomalies detected by hosts applications so that I can have the control of the entire architecture.”

### Mockup:



**Microservices Involved:**

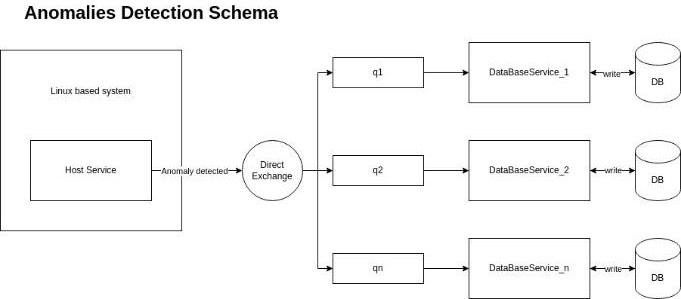
In the anomaly detection phase of our project, we focus on monitoring and detecting authentication failures within Linux-based systems. We employ a distributed architecture where a service, acting as an agent or background process, is deployed across multiple nodes, such as developer machines within a corporate network. This service, written in languages like Java or Bash, is responsible for performing various home

actions, with a primary task being the continuous monitoring of log files for authentication failure events. Unlike traditional approaches, our system operates without the need for explicit triggers, simplifying the deployment process.

Upon detecting an authentication failure, the service generates a message routed to a direct exchange, initiating a cascade of actions within our system. This message triggers a URL listener, which in turn interfaces with our database service, built upon the same database schema.

The database service consumes these messages, storing the detected anomalies for further analysis. Through a UI interface, security administrator can access and review these anomalies. In this way, an option is provided to flag resolved anomalies, allowing for efficient management and tracking of system integrity.

This implementation streamlines the anomaly detection process, offering a robust and scalable solution for monitoring authentication events within our distributed environment.



### Technical Choices:

###### Distributed Architecture:

* + - Employing a distributed architecture enables scalability and resilience in anomaly detection across multiple nodes within the system.
    - Distributing the anomaly detection service across various nodes facilitates efficient monitoring and detection of authentication failures in a large-scale environment.

###### Agent-Based Monitoring Service:

* + - Implementing a service deployed as an agent or background process on nodes allows continuous monitoring of log files for authentication failure events.
    - The agent-based approach simplifies deployment and operation, requiring minimal setup and configuration on each node.

###### Message Routing via Direct Exchange:

* + - Utilizing a direct exchange for message routing ensures efficient and targeted communication within the system.
    - Anomalies detected by the monitoring service trigger messages routed to a direct exchange, initiating subsequent actions in the system.

###### URL Listener and Database Service:

* + - Incorporating a URL listener facilitates interaction between the monitoring service and the database service for anomaly storage.
    - The database service consumes messages from the URL listener, storing detected anomalies in the database for further analysis and review.

###### Database Schema Design:

* + - Designing the database service based on a well-defined schema allows efficient storage and retrieval of anomaly data.
    - The database schema supports the storage of authentication failure events, enabling comprehensive analysis and reporting of detected anomalies.

###### User Interface (UI) for System Administrators:

* Providing a user interface tailored for system administrators allows easy access and review of detected anomalies.
* System administrators can view a list of anomalies detected by host applications, providing them with control over the system.

### Acceptance criteria:

1. Users with roles system administrator or superadmin can view the list of anomalies.
2. For each anomaly, it is possible to view information about the host that generated it.
3. Each authorized user with the appropriate role can filter the list of anomalies.
4. For each anomaly, the event that generated it is described.

### Function Point:

External Inputs (EI):

* + Http Request to Anomalies Controller: HTTP Request sent from the UI.
  + Message Read from Queue: when the database service reads a message from the queue. External Outputs (EO):
  + ACK sent from Database Service.
  + Publish message to the queue: when anomalies controller publishes the message to the queue.
  + Http response from the Anomalies Controller to the UI. External Inquiries (EQ)
  + JWT Verification: the verification process by user controller involves external inquiry to the authentication service.

Internal Logic File (ILF):

* + Anomalies data storage in database, sent by database service to anomalies controller and delivered to the UI.

External Interface Files (EIF): None.

**ILF/EIF COMPLEXITY**

#### The database translation is the following:

- Anomalies (id, date, description, done, hostname, ip address, hash code) PK [id]

In this case we have 1 RET (Anomalies) and in total we have 7 DET.



So ILF is **LOW (6).**

EIF is **0**.

### EI Computation:

1. Http Request from UI.
2. Message read in the Queue. FTR= # of ILF/EIF involved in the EI. DET= # of Fields involved in the process. In the first case:
   * There are involved 1 ILF (Anomalies).
   * For the http request the fields sent are id, date, description, done, hostname, ip address, hashcode, so in total we have 7 DET.



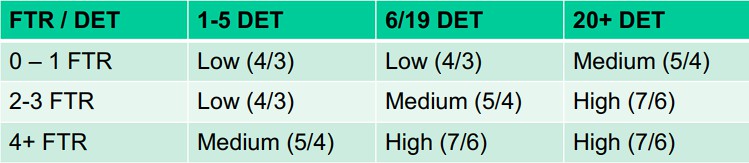
So, from the table, in the first case we have 1 FTR and 7 DET => EI\_1: Low (3). In the second case:

* + There is a Json object with the following fields: operationType, id, date, description, done, hostname, ip address, hashcode, so in total we have 8 DET. The ILF involved are 1 (Anomaly)

In the second case we have 1 FTR and 8 DET => EI\_2: LOW (3). So, in total we have EI= EI\_1+EI\_2= 3+3=**6**.

### EO Complexity

1. ACK sent from Database Service.
2. Publish Message to the Queue (Anomalies Controller).
3. Http Response to the UI.
4. The ACK contains the following infos:
   * Message, Payload, Success / Failed so in total 3 DET.
   * There is no ILF involved in the process.



So for the first case we have EO\_1: LOW (4).

1. In the second case:
   * Message published by Anomalies Controller contains anomaly information, so we have 8 DET.
   * There is 1 ILF (Anomaly) involved in the process. So, for the second case we have EO\_2: LOW (4).
2. The third case we have a http code so is gonna be EO\_3: LOW (3). So, in total we have EO=EO\_1+EO\_2+EO\_3= 4+4+3=**11**.

### EQ complexity

* + JWT verification include 1 ILF (key secret) and 3 DET (Header, Payload and Signature). Then we have for EQ: **LOW (4).**

The total UFP is the sum of:

ILF + EI + EO + EQ +EIF = 6+6+11+4+0 =**28.**

##### User stories: “As a System Administrator I want to run a background process on the host target to check a set of preestablished triggers, so that I can see if some anomalies occur.”

### Microservices Involved:

In our project's anomaly detection phase, we concentrate on monitoring and identifying authentication failures in Linux-based systems. We adopt a distributed architecture where a service, functioning as an agent or background process, is deployed across numerous nodes, including developer machines within a corporate network. This service, developed in languages such as Java or Bash, is tasked with executing various tasks, with its primary objective being the continuous monitoring of log files for authentication failure events. Unlike conventional methods, our system operates without requiring explicit triggers, simplifying the deployment process.

Upon detecting an authentication failure, the service generates a message routed to a direct exchange, initiating a sequence of actions within our system. This message triggers a URL listener, which subsequently interacts with our database service, built on the same database schema.

The database service consumes these messages, storing the identified anomalies for further analysis. Through a UI interface, security administrators can access and review these anomalies. Additionally, an option is provided to flag resolved anomalies, enabling efficient management, and tracking of system integrity.

### Technical Choices:

###### Distributed Architecture:

* + - Adopting a distributed architecture enables scalability and resilience in anomaly detection across multiple nodes within the system.
    - Distributing the anomaly detection service across various nodes facilitates efficient monitoring and identification of authentication failures in a large-scale environment.

###### Agent-Based Monitoring Service:

* + - Implementing a service deployed as an agent or background process on nodes allows continuous monitoring of log files for authentication failure events.
    - The agent-based approach simplifies deployment and operation, requiring minimal setup and configuration on each node.

###### Message Routing via Direct Exchange:

* + - Utilizing a direct exchange for message routing ensures efficient and targeted communication within the system.
    - Anomalies detected by the monitoring service trigger messages routed to a direct exchange, initiating subsequent actions in the system.

###### URL Listener and Database Service:

* + - Incorporating a URL listener facilitates interaction between the monitoring service and the database service for anomaly storage.
    - The database service consumes messages from the URL listener, storing identified anomalies in the database for further analysis and review.

###### Database Schema Design:

* + - Designing the database service based on a well-defined schema allows efficient storage and retrieval of anomaly data.
    - The database schema supports the storage of authentication failure events, enabling comprehensive analysis and reporting of detected anomalies.

###### User Interface (UI) for System Administrators:

* + - Providing a user interface tailored for system administrators allows easy access and review of detected anomalies.
    - System administrators can view a list of anomalies detected by host applications, providing them with control over system integrity.

### Acceptance criteria:

1. The watchdog application can execute custom commands defined in the main class by the end user (e.g., listening to journalctl).
2. The default operations allowed for each watchdog include each sudo operation and each remote SSH access to the target machine.
3. If an event is detected by the watchdog, it is able to send the event to the appropriate queue as an anomaly.
4. Each event will be considered an anomaly and will include information about the event (details), IP address, hostname, and hash code.
5. Each caught anomaly must be unique.

### Function Point:

External Inputs (EI):

* + Data coming from journalctl logs, including different event types such as errors and warnings. External Outputs (EO):
  + The data sent to the external queue, including event data (e.g. timestamp, event type, message) in a specific format.

External Inquiries (EQ): None. Internal Logic File (ILF):

* + Anomalies data storage in database, sent by database service to anomalies controller and delivered to the UI.

External Interface Files (EIF): None.

**ILF/EIF COMPLEXITY**

#### The database translation is the following:

- Anomalies (id, date, description, done, hostname, ip address, hash code) PK [id]

In this case we have 1 RET (Anomalies) and in total we have 7 DET.



So ILF is **LOW (6).**

EIF is **0**.

### EI Computation:

FTR= # of ILF/EIF involved in the EI. DET= # of Fields involved in the process .

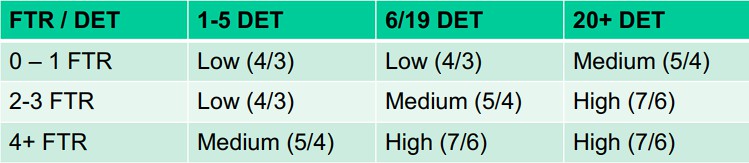
* There are involved 1 ILF (Anomalies).
* For journalctl the fields sent are date, description, done, hostname, ip address, so in total we have 5 DET.



So, from the table, in the first case we have 1 FTR and 8 DET => EI: **Low (3).**

### EO Complexity

1. ACK sent from Database Service.
2. Publish Message to the Queue (Anomalies Controller).
3. The ACK contains the following infos:
   * Message, Payload, Success / Failed so in total 3 DET.
   * There is no ILF involved in the process.



So for the first case we have EO\_1: LOW (4).

1. In the second case:
   * Message published by Anomalies Controller contains anomaly information and event data, so we have 10 DET.
   * There is 1 ILF (Anomaly) involved in the process. So, for the second case we have EO\_2: LOW (4).

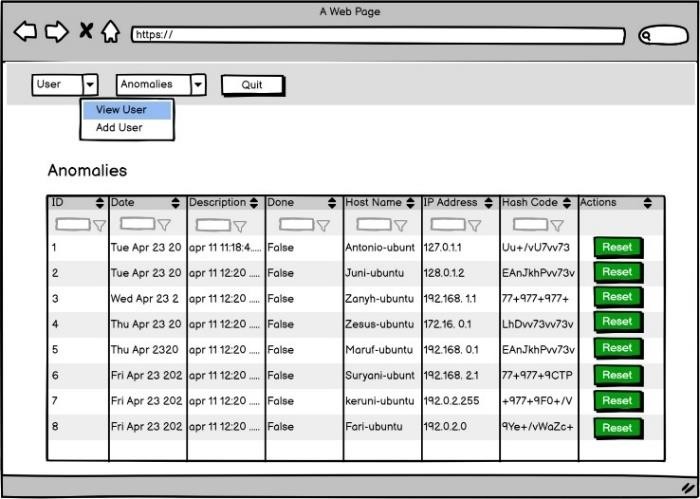
So, in total we have EO=EO\_1+EO\_2= 4+4=**8**.

The total UFP is the sum of:

ILF + EI + EO + EQ +EIF = 6+3+8+0+0 =**17**.

1. *User stories: “As A system Administrator I want to perform the action "reset to green" if some anomalies occur (after that a manually intervent is done), so that I see only the anomalies that aren't checked.”*

### Mockup:



**Microservices Involved:**

In our project's anomaly detection phase, we concentrate on monitoring and identifying authentication failures within Linux-based systems. We adopt a distributed architecture where a service, acting as an agent or background process, is deployed across multiple nodes, including developer machines within a corporate network. This service, developed in languages such as Java or Bash, is tasked with executing various tasks, with its primary objective being the continuous monitoring of log files for authentication failure events. Unlike traditional approaches, our system operates without the need for explicit triggers, simplifying the deployment process. Upon detecting an authentication failure, the service generates a message routed to a direct exchange, initiating a sequence of actions within our system. This message triggers a URL listener, which subsequently interfaces with our database service, built upon the same database schema. The database service consumes these messages, storing the detected anomalies for further analysis.

Through a UI interface, security administrators can access and review these anomalies. Additionally, a functionality is provided to perform the action "reset to green" manually after anomalies occur. This action resets the system to a normal state, allowing administrators to focus solely on unchecked anomalies.

This implementation streamlines the anomaly detection process, offering a robust and scalable solution for monitoring authentication events within our distributed environment.

### Technical Choices:

###### Distributed Architecture:

* + - Leveraging a distributed architecture enables scalability and resilience in anomaly detection across multiple nodes within the system.
    - Distributing the anomaly detection service across various nodes facilitates efficient monitoring and identification of authentication failures in a large-scale environment.

###### Agent-Based Monitoring Service:

* + - Implementing a service deployed as an agent or background process on nodes allows continuous monitoring of log files for authentication failure events.
    - The agent-based approach simplifies deployment and operation, requiring minimal setup and configuration on each node.

###### Message Routing via Direct Exchange:

* + - Utilizing a direct exchange for message routing ensures efficient and targeted communication within the system.
    - Anomalies detected by the monitoring service trigger messages routed to a direct exchange, initiating subsequent actions in the system.

###### URL Listener and Database Service:

* + - Incorporating a URL listener facilitates interaction between the monitoring service and the database service for anomaly storage.
    - The database service consumes messages from the URL listener, storing detected anomalies in the database for further analysis and review.

###### Database Schema Design:

* + - Designing the database service based on a well-defined schema allows efficient storage and retrieval of anomaly data.
    - The database schema supports the storage of authentication failure events, enabling comprehensive analysis and reporting of detected anomalies.

###### User Interface (UI) for System Administrators:

* + - Providing a user interface tailored for system administrators allows easy access and review of detected anomalies.
    - System administrators can view a list of anomalies detected by host applications, providing them with control over system integrity.

###### Manual Action "Reset to Green":

* + - Implementing functionality for system administrators to manually trigger the action "reset to green" allows for resetting the system to a normal state after anomalies occur.
    - This action enables administrators to focus solely on unchecked anomalies, streamlining the anomaly detection process and facilitating efficient management of system integrity.

### Acceptance criteria:

1. Users with roles superadmin or system administrator can perform the "Reset to green" action.
2. The "Reset to green" operation allows for the "deletion" (logical deletion) of the anomaly in the system.
3. If an anomaly is reset to green, it should not be present in the anomaly list.
4. After the "Reset to green" operation, an ACK is received indicating the result of the operation (SUCCESS or FAIL).

### Function Point:

External Inputs (EI):

* + Http Request from UI for “reset to green” action.
  + Message Read from Queue: when the database service get request from UI. External Outputs (EO):
  + ACK sent from Database Service after “reset to green” action.
  + Publish message to the queue using controller.
  + Http response from the Anomalies Controller to the UI External Inquiries (EQ)
  + JWT Verification: the verification process by user controller involves external inquiry to the authentication service.

Internal Logic File (ILF):

* + Anomalies data storage in database, sent by database service to anomalies controller and delivered to the UI.

External Interface Files (EIF): None.

**ILF/EIF COMPLEXITY**

#### The database translation is the following:

- Anomalies (id, date, description, done, hostname, ip address, hash code) PK [id]

In this case we have 1 RET (Anomalies) and in total we have 7 DET.



So ILF is **LOW (6).**

EIF is **0**.

### EI Computation:

1. Http Request from UI.
2. Message read in the Queue. FTR= # of ILF/EIF involved in the EI. DET= # of Fields involved in the process. In the first case:
   * There are involved 1 ILF (Anomalies).
   * For the http request the fields sent are id, date, description, done, hostname, ip address, hashcode so in total we have 7 DET.



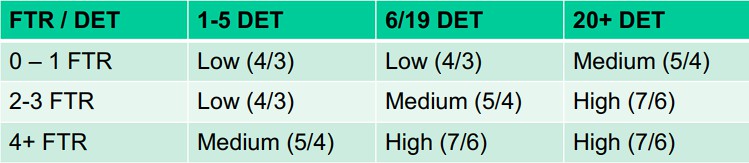
So, from the table, in the first case we have 1 FTR and 7 DET => Low (3). In the second case:

* + There is a Json object with the following fields: operationType, id, date, description, done, hostname, ip address, hashcode, so in total we have 8 DET.
  + The ILF involved are 1 (Anomaly).

In the second case we have 1 FTR and 8 DET => EI\_2: LOW (3). So, in total we have EI= EI\_1+EI\_2= 3+3=**6**.

### EO Complexity

1. ACK sent from Database Service.
2. Publish Message to the Queue (Anomalies Controller).
3. Http Response to the UI.
4. The ACK contains the following infos:
   * Message, Payload, Success / Failed so in total 3 DET.
   * There is no ILF involved in the process.



So for the first case we have EO\_1: LOW (4).

1. In the second case:
   * Message published by Anomalies Controller contains anomaly information, so we have 8 DET.
   * There is 1 ILF (Anomaly) involved in the process. So, for the second case we have EO\_2: LOW (4).
2. The third case we have a http code so is gonna be EO\_3: LOW (3).

So, in total we have EO=EO\_1+EO\_2+EO\_3= 4+4+3=**11**.

### EQ complexity

* + JWT verification include 1 ILF (key secret) and 3 DET (Header, Payload and Signature). Then we have for EQ: **LOW (4).**

The total UFP is the sum of:

ILF + EI + EO + EQ +EIF = 6+6+11+4+0 =**28**.

# Technologies

The technologies implemented in the project are the following:

* **JSON Web Token (JWT):** is an open standard that defines a compact and selfcontained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA It is used to authenticate and authorize users in web applications and APIs. In the case of authorization this is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.

A JSON token consists of three parts:

* 1. A Header containing information about the type of token and algorithms used to generate the signature.
  2. A Payload containing the “claims” (ID and authentication verifications) made by the user that can include a User ID, the user’s name, an email address, and metainformation about the operation of the token.
  3. A Signature, or cryptographic mechanism, is used to verify the token’s integrity.

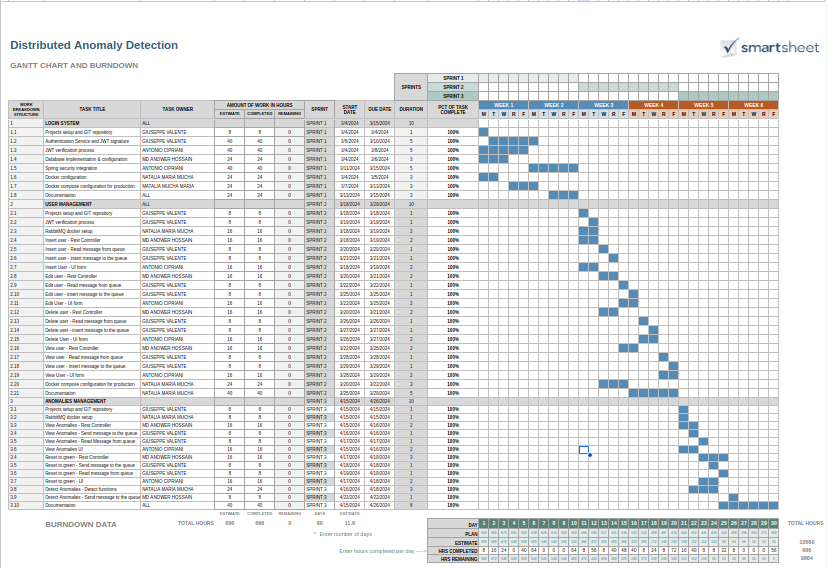
Together, the header, payload, and signature make up the JSON Web Token, typically passed between the client and the server in the HTTP Authorization header or in the body of an HTTP request or response. The server can then verify the signature to ensure that the token is valid and has not been modified and use the information in the payload to authenticate the user.

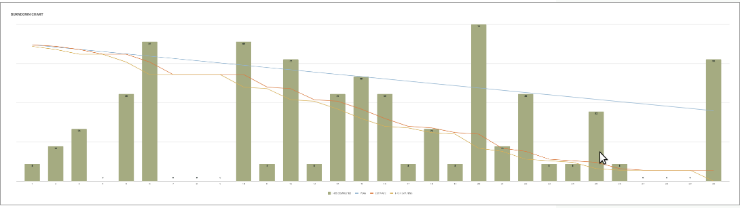
* **Spring Boot:** This is an open-source framework that simplifies the process of building and deploying Java-based applications. It provides a streamlined approach to setting up and configuring Spring applications, reducing the need for manual configuration. Spring Boot achieves its simplicity through convention-over-configuration principles and auto-configuration. It automatically configures many aspects of the application based on dependencies and settings, allowing developers to focus on writing business logic rather than boilerplate configuration code.
* **Spring Security:** This is a powerful and customizable authentication and access control framework for Java applications. It provides comprehensive security features for web applications, including authentication, authorization, session management, CSRF protection, and more. Spring Security works by intercepting requests to the application 14 and applying security measures based on pre- configured rules and settings. It integrates tightly with Spring applications, leveraging Spring’s inversion of control (IoC) and aspect-oriented programming (AOP) features. Spring Security allows developers to configure authentication mechanisms such as form-based authentication, HTTP basic authentication, OAuth, and JWT (JSON Web Tokens). It also provides support for role-based access control and method-level security.
* **JPA (Java Persistence API):** The Java Persistence API (JPA) is a programming interface that allows developers to work with databases using Java. It is used to store, retrieve, and update data in a database. JPA is based on Object-Relational Mapping (ORM) technology to map Java objects to database tables and vice versa. It can be used to perform CRUD (Create, Read, Update, Delete) operations on data and provides a set of APIs for querying data using the Java Persistence Query Language (JPQL). It also offers a way to manage transactions, caching, and concurrency control. JPA can be implemented by various ORM frameworks such as Hibernate, EclipseLink, TopLink, and can also be used in combination with Java technologies like Spring, Java EE, and others.
* **Hibernate:** Hibernate is a library ecosystem, with its main component typically referred to as Hibernate ORM, an open-source Object-Relational Mapping (ORM) framework. It is a Java framework for mapping object-oriented domain models to a relational database. Essentially, Hibernate is used to persist data from the Java environment to the database. Hibernate implements the Java Persistence API (JPA) specifications for data persistence. Other notable Hibernate libraries include Hibernate Search (full-text search), Hibernate Validator (constraint management), Hibernate OGM (Java Persistence support for NoSQL databases), and Hibernate Tools (collects command-line tools and plugins for working with Hibernate).

# Scrum

In this section, it is possible to analyze how the overall teamwork was divided among the three available periods of time, called sprint. We considered a sprint as a 10-day work period, with approximately eight hours of work per day. The tasks to be accomplished during the sprints were selected dynamically through team meetings.

Each sprint was dedicated to a part of the project, the first one for the login, the second one for user management, and the third one for anomalies.



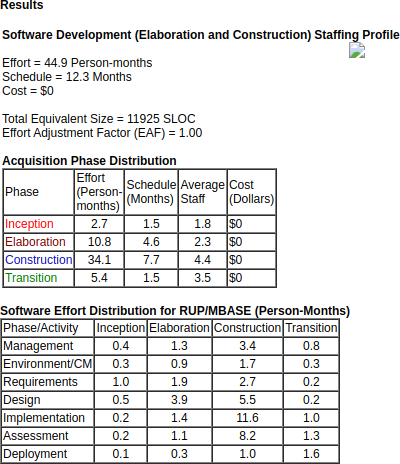


# Estimation (Function Points s COCOMO II)

#### For each user story we have provided the computation of the UFP then we can summarize the computation in the following table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **User Story** | **ILF** | **EIF** | **EI** | **EO** | **EQ** | **SUM** |
| **As a user (superadmin or security administrator) I want to**  **perform the login to the system so that I can use the software.** | 0 | 5 | 6 | 4 | 0 | 15 |
| **As a superadmin I want to insert a new user in the system so**  **that I can authorize it to use the application with the right role.** | 7 | 0 | 12 | 12 | 4 | 35 |
| **As a superadmin I want to edit a user in the system so that I can**  **authorize a new security administrator to use the software.** | 7 | 0 | 12 | 4 | 4 | 35 |
| **As a superadmin I want to view and filter the user's information**  **so that I can perform my research.** | 7 | 0 | 12 | 13 | 4 | 36 |
| **As a superadmin I want to delete a user in the system so that I**  **can remove the users that are not needed.** | 7 | 0 | 10 | 12 | 4 | 31 |
| **As a System Administrator I want to see the list of the anomalies detected by hosts applications so that I can have the control of**  **the entire architecture.** | 6 | 0 | 6 | 11 | 4 | 28 |
| **As a System Administrator I want to run a background process on the host target to check a set of preestablished triggers, so**  **that I can see if some anomalies occur.** | 6 | 0 | 3 | 8 | 0 | 17 |
| **As A system Administrator I want to perform the action "reset to green" if some anomalies occur (after that a manually intervent**  **is done), so that I see only the anomalies that aren't checked** | 6 | 0 | 6 | 11 | 4 | 28 |
| **TOTAL** | 46 | 5 | 67 | 75 | 24 | 225 |

Starting from FPA we can use one of the best tools to calculate the COCOMO II estimation (<http://softwarecost.org/tools/COCOMO/>), then:



# Code

This code can be downloaded from git-hub at the internet address: [https://github.com/giusvale-](https://github.com/giusvale-dev/distributed_anomalies_detection.git) [dev/distributed\_anomalies\_detection.git](https://github.com/giusvale-dev/distributed_anomalies_detection.git)