


# 1. System-as-is

---

This section of the report is related to the project as can be found currently, the objective was to reverse engineer it in order to present diagrams and logic views that accurately represent the system.

## 1.1 Domain Model

 Domain Model

## 1.2 Logical View

### 1.2.1 Level 2 - Logical View

![Logical View](/Logical View As Is/Level 2/ARQSOFT\_LV\_LVL2\_AsIs.svg)

### 1.2.2 Level 3 - Logical View

![Logical View](/Logical View As Is/Level 3/ARQSOFT\_LV\_LVL3.svg)

## 1.3 Level 3 - Implementation View

 Implementation View

## 1.3 Physical View

 Physical View

## 1.4 Process View

### 1.4.1 Find Reviews by User

![Find Reviews by User](/Sequence Diagrams/ReviewController/findReviewsByUser-SD.svg)

### 1.4.2 Create Review

![Create Review](/Sequence Diagrams/ReviewController/createReview-SD.svg)

### 1.4.3 Create User

![Create User](/Sequence Diagrams/UserController/create-SD.svg)

### 1.4.4 Create Product

![Create Product](/Sequence Diagrams/ProductController/CreateProduct\_SD.svg)

# 2. Architecturally Significant Requirements

---

After the discovery of the system-as-is, the architecturally significant requirements were identified and the most relevant ones are listed below:

## 2.1 Use Case Model

![[Use Case Model]](/Use Case Model.svg)

Use Case	Description
UC1 - Get catalogue of Products	A user can get a list with all Products that were inserted into the system.
UC2 - Get Product from SKU	A user can insert the SKU of an specific product and that should return the product if it exists.
UC3 - Create a Product	A user can create a new Product to be sold.
UC4 - Get Reviews	It's possible to see a list of reviews.
UC5 - Add vote for a Review	In a review, the user should be able to send a vote, the vote can be either upvote or downvote.
UC6 - Post a Review	A user can post a review related to a product.
UC7 - Accept or reject Review	When a review is created, it is presented in a 'Pending' status, the user then can approve or deny the review to publish it.
UC8 - Create User	The admin can create new Users and add them into the system.
UC9 - Get User	The admin can get a specific user information by inserting their Id.

## 2.2 Quality Attributes

In addition to Use Cases, the group also identified the most relevant quality attributes for the system, and the most relevant ones are listed below:

ID	Quality Attribute	Scenario	Associated Use Cases
QA-1	Modifiability	A new SKU generation algorithm is implemented as part of a new client requirement. The system must successfully integrate the new algorithm without breaking the existing functionality and without changing any other component of the system.	UC-3
QA-2	Modifiability	A new Review listing algorithm is implemented as part of a new client requirement. The system must successfully integrate the new algorithm without breaking the existing functionality and without changing any other component of the system.	UC-4
QA-3	Security	A user creates a new review, it's possible to know who created the review 100% of the time.	UC-6
QA-4	Performance	During busy hours, the system can hit peak loads, when that happens, 100% of reviews posted must be successfully captured and stored.	UC-6

ID	Quality Attribute	Scenario	Associated Use Cases
QA-5	Availability	If an error occurs with the persistence mechanism, the system must be back to normal within less than a minute.	All
QA-6	Usability	User can display the list of pending reviews and select which one to update, the update should consist of both approve or reject the review.	UC-7

## 2.3 Constraints

Some Constraints were also identified, and are listed below:

- The system must support the integration of multiple persistence mechanisms, such as: MongoDB, Redis, Neo4j and MySQL.
- The system must support the integration of multiple SKU generation algorithms.
- The system must support the integration of multiple Review listing algorithms.
- The choice of which implementation to use regarding persistence, SKU generation and Review listing must be defined at the application properties.
- Injection of dependencies must be done through the use of Spring Framework.

## 2.4 Architectural Concerns

Considering the scope of the project, only a few concerns were raised, and they are listed below:

- The group's lack of experience with the Spring Framework.
- Supporting the use of multiple persistence mechanisms.
- Defining the initial structure of the project through the 4 + 1 architectural view model.

# 3. Architectural design alternatives and rational

This section will present the rational taken from the decisions the group did in relation to architectural choices, with alternatives being presented alongside positive and negative repercussions of each choice.

## 3.1 Architectural Patterns

Onion Architecture focuses on creating a clean and maintainable separation of concepts in an application. It follows the Dependency Inversion Principle, which says that high-level modules should not depend on low-level modules, but instead depend on abstractions. This approach tries to solve problems with coupling and separation of concerns. To achieve this, Onion Architecture works with the concept of layers, which represent the different levels of abstraction in the application. The inner layer contains the domain logic. above the domain, the application layer resides, it contains all logic present in the app and is used to communicate between the UI and the domain, finally, the outer layers consists of the UI layer, tests layer and infrastructure layer. Each layer can only depend on the layer directly below it, and not on any other layer.

Below the group listed some advantages and disadvantages of Onion Architecture:

### Advantages of Onion Architecture:

- **Clean Separation of Concerns:** Onion Architecture has a strict separation of concerns, making it easier to manage the complexity of large applications.
- **Testability:** Onion Architecture makes writing unit tests easier because the core business logic is isolated in the innermost circle, allowing developers to write tests without dependencies on external frameworks or infrastructure.
- **Flexibility and Maintainability:** The architecture is modular, which makes it easier to add new features, replace code, and maintain the application over time. Changes in one layer have minimal impact on other layers.
- **Independence from Frameworks and Libraries:** The inner domain layer doesn't depend on specific frameworks, making it easier to migrate to new technologies when needed.
- **Adaptability:** Onion Architecture is not tied to a specific delivery mechanism so it can be used with different types of applications.

**Disadvantages of Onion Architecture:**

- **Complexity:** Onion can be considered complex and over-engineered, especially for small or simple applications. It requires careful planning and design, which can be time-consuming.
- **Overhead:** While the modularity and separation of concerns are beneficial, they may introduce some overhead in terms of development time and effort.
- **Too abstract:** Heavily used interfaces.

Now, let's briefly compare Onion Architecture to other approaches like Microservices, Three-Tier Architecture, and Monolithic Architecture:

Architecture	Advantages over Onion	Disadvantages over Onion
Microservices	In some cases, especially for small microservices, a simpler architecture might be more appropriate. Onion Architecture's modularity can lead to overengineering in small, simple microservices.	Inter-service communication in microservices can introduce latency compared to in-process calls in a Onion architecture.
Monolithic	When compared to a monolithic approach, Onion Architecture requires more upfront design and planning. In situations where speed of development is a higher priority, a monolithic approach may be more suitable.	When compared to a traditional monolithic architecture, Onion Architecture offers better maintainability, testability, and flexibility, making it easier to transition from a monolithic system to a more modular architecture.
Three-Tier	Onion Architecture is more complex to set up and requires more initial effort compared to the simpler three-tier architecture. For very small or straightforward applications, the additional complexity may not be justified.	Three-tier architecture often leads to tighter coupling between layers. This makes it harder to change or replace components without affecting other parts of the application.

With these considerations in mind, the Onion architecture was chosen as the best approach for the application.

### 3.2 Database

The use of Non-relational (NoSQL) database was a big focus of this project, as the implementation of multiple NoSQL databases of different types was one of the main requirements. Because of that, the group found pertinent to demonstrate what advantages and disadvantages each type of NoSQL database has, and how they can be used in different scenarios. Beyond that, a comparison between NoSQL and Relational databases was also made, taking into consideration the already implemented SQL database.

The databases implemented in this project are:

- MongoDB (Document Database)
- Redis (Key-Value Database)
- Neo4j (Graph Database)
- SQL (Relational Database)

Database Type	Typical Projects
Document Databases	Content Management Systems, E-commerce, Log and Event data.
Relational Databases	Enterprise Applications, Customer Relationship Management, E-commerce.
Graph Databases	Social Networks, Knowledge graphs, Semantic search.
Key-Value Databases	User Profiles, Load balancing, Domain Name System.

As can be seen in the table above, each type of database is more suitable for a specific type of project, and that's because each type of database has its own advantages and disadvantages, the table below lists some of them for each type:

Database Type	Advantages	Disadvantages
Document Databases	Document databases allow you to store data without a predefined schema. Each document can have its own structure, which makes it easy to adapt to changing data requirements.	Document databases may not be the best choice for applications with structured, highly relational data that requires complex queries which is not well handled because of the not predefined schema.
Relational Databases	SQL databases enforce data integrity through features like primary keys, unique constraints, and foreign keys.	Performing joins on large tables can be resource-intensive and may require query optimization.
Graph Databases	Graph databases are optimized for traversing relationships, making them highly efficient for queries that involve navigating nodes and edges in a graph.	Graph databases are not very good for all types of applications.

Database Type	Advantages	Disadvantages
Key-Value Databases	Key-value stores are usually used for caching to make data retrieval faster.	Key-value databases might have problems with the size of values, which can be a problem for apps that deal with large documents.

### 3.3 SKU generation

For the generation of SKUs, two approaches could be taken, one where the use of external APIs to generate the SKUs was used, and one where a custom generator was implemented by the group. We have analysed the possible advantages and disadvantages of each approach:

**External APIs:** External APIs can be good when considering the implementation time and the fact that they are already tested and ready to use, however, some problems become obvious after first inspection, namely some security concerns in regard to the data, and also the lack of flexibility in the customization of the SKU generation, which would not be able to accommodate the 3 different generators that were created.

**Custom Generator:** The custom generator was the choice made by the team, because it allowed for more flexibility and is generally more performant, some problems that this approach gives however, can be found in the development time and the fact that it is not tested, needing custom testing to be implemented as well.

### 3.4 Result from Architectural analysis and choices

After analyzing the system-as-is and the architectural choices available to us such as the ones presented in the previous sections, the group decided to implement the following options:

- **Onion Architecture:** The group decided to implement the Onion Architecture, as we consider it the best option for the project due to its scalability, layer boundaries, code organization and familiarity. All positives that reflect well on the concerns and constraints raised in the ASR section.
- **Database:** The group implemented the following databases: MongoDB, Redis and Neo4j. The implementation makes use of one type of persistence at a time, depending on the configuration set in the application properties. Worth noting that the implementation was done in this way to follow the requests of the assignment, but in a real-world scenario, these databases could be used together to store the content they are best suited for, as an example: Neo4j could be used for the product review based on its relationship modeling focus, while MongoDB could be used for the product information, as it's a document database.
- **SKU generation:** 3 different types of SKU generation were implemented, they did not use any external APIs because of performance concerns and limited customization for our needs.

### 3.5 System To-be

**Logical View Level 2 - To-be** ![Use Case Model](/LogicalViewToBe/Level 2/ARQSOFT\_LV\_LVL2\_ToBe.svg)

**Logical View Level 3 - To-be** ![Use Case Model](/LogicalViewToBe/Level 3/ARQSOFT\_LV\_LVL3\_ToBe.svg)

These logical views show how the system was changed, the onion architecture is applied and the system has distinct layers, that can only communicate with the layer below and above, additionally, the system now follows an Open/Closed principle, it receives only one repository interface that can deal with distinct

databases, now any implementation beyond those implemented in the scope of this assignment would eliminate the need of changes to the system.

## 4. Principles and Patterns

---

### 4.1 Single Responsibility Principle

The Single Responsibility principle (SRP) states that a class or a module should encapsulate only one responsibility or functionality. This applies to the fact that having many responsibilities in a single class or module can make the code harder to modify, maintain and test, and also can lead to code duplication. Additionally, changes to one of its responsibilities may inadvertently affect the other responsibilities, leading to bugs and unexpected behavior. In this project, the SRP was applied in the following classes:

- In this project, we have Service classes, which each class have a specific, well-defined responsibility and do not mix unrelated operations or concerns. In that case, all methods and properties are directly related to its own Service's primary responsibility.
- The naming convention used for the classes and methods are also a good example of SRP, as they directly represent what the class or method is responsible for.

### 4.2 Open/Closed Principle

The Open/Closed principle (OCP) states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. It basically defends that instead of modifying existing code to accommodate new requirements, the system should be able to be extended by adding new code or components to it.

- The use of Interfaces in this project is an example of OCP, as it allows the system to be extended by adding new implementations of the interfaces, without the need to modify the existing code.
- The use of the Strategy Pattern is also an example of OCP, as this behavioral design pattern was used in the way the SKU and Review Recommendations algorithms have been implemented, it allows the addition of new algorithms without the need to modify the existing code that uses them.

### 4.3 Liskov Substitution Principle

The Liskov Substitution principle (LSP) states that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. This means that the new derived classes must extend the base classes without changing their behavior. Within this context, if class S is a subtype of class T, then S is expected to inherit the properties and behaviors of T and, ideally, extend or specialize them.

- The use of Interfaces in this project is an example of LSP, as it allows the system to be extended by adding new implementations of the interfaces, without the need to modify the existing code.
- The overriding methods in the derived class created in this project have the same method signatures as the methods in the base class.

### 4.4 Interface Segregation Principle

The Interface Segregation principle (ISP) states that no client should be forced to depend on methods it does not use. This means that a client should not be forced to implement an interface that it doesn't use. Instead,

the interface should be split into smaller, more specific interfaces so that the clients can implement only the functionality that they are interested in. It encourages breaking down large and monolithic interfaces into smaller and more specialized ones and limits that an interface should not contain more methods than what is necessary for the client classes to perform their tasks.

- The use of Interfaces for the SKU and Review Recommendations algorithms, and for the interfaces created for the Services, is an example of ISP, as they all contain only the methods that are necessary for the client classes to perform their tasks.

## 4.5 Dependency Inversion Principle

The Dependency Inversion principle (DIP) states that high-level modules should not depend on low-level modules, but instead depend on abstractions. This allows the system to be more flexible to changes, as the high-level modules can be easily replaced by other modules that implement the same interface. It's a concept that emphasizes the importance of designing software systems with flexibility and maintainability in mind, in order to achieve loosely coupled and highly cohesive software components.

- For this project, the use of Interfaces for the SKU and Review Recommendations algorithms, and for the interfaces created for the Services, is an example of DIP, as they all allow for different implementations to be added without affecting the core logic. The Interfaces are correctly defining the behaviors that the system needs, and the implementations are free to change as long as they follow the contract defined by the interfaces.