# Lecture Notes

# Integration Testing and Deployment

## Integration Testing

In this session, you learnt about integration testing, its need, advantage over unit testing and different types of integration testing approaches. You also understood how integration testing is done in a developed UPSTAC application.

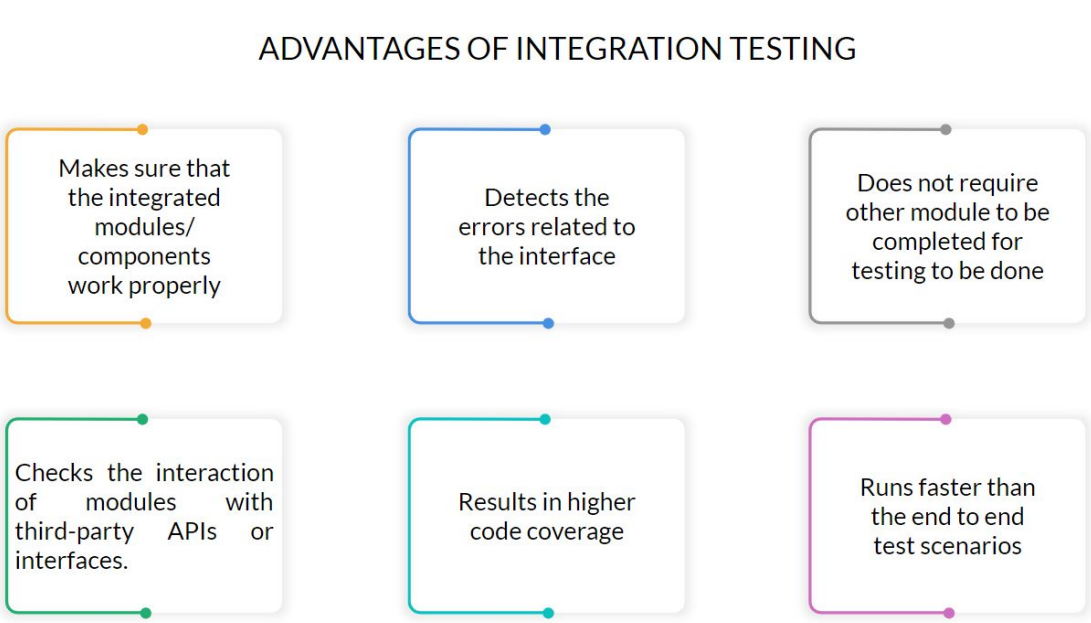
### Introduction to Integration Testing

**Integration testing** is a level of software testing where individual units/components are combined and tested as a group. It is usually performed after individual modules have undergone unit testing, i.e., when they are functioning well and independently.

Integration testing takes individual, unit-tested modules and combines them into larger groups. It then creates an integration test plan to test those larger groups of modules.

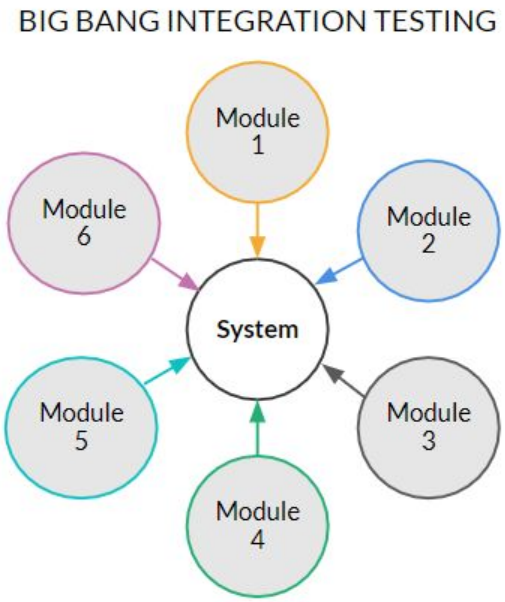
Once the results of integration testing are delivered, you can assess whether the combined units are working well as a whole or not. Integration testing tells us whether the final integrated system is functional and error-free.

The major advantages of integration testing are as follows:



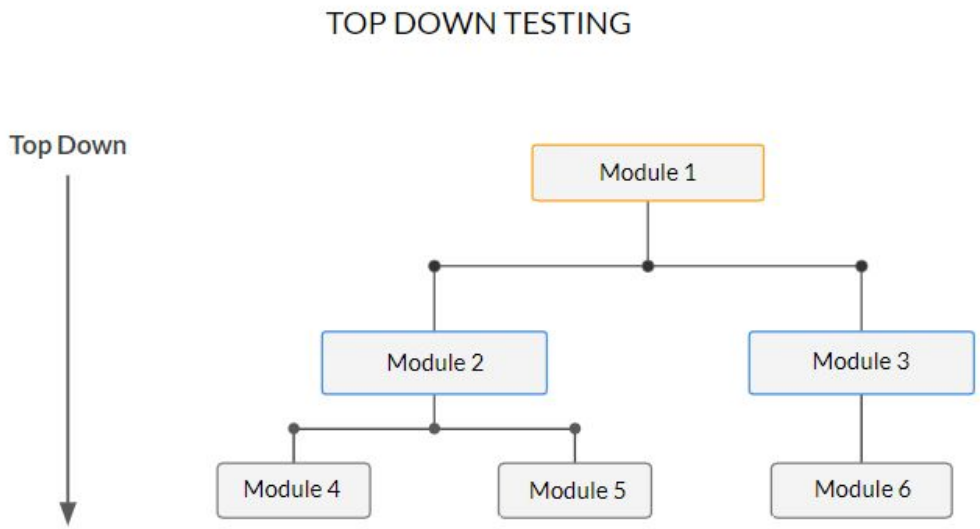

## Types of Integration Testing

**Big bang testing** is an approach to integration testing where all or most of the units are combined together and tested at one go. The following image visualises the big bang integration testing:



This approach is taken when the testing team receives the entire software in a bundle. If all the components in the unit are not complete, the integration process will not execute.
Big bang testing is convenient for small systems.
However, since the whole system is being tested at once, faults cannot be pinpointed. Also, since it can only be done after all the modules have been developed, the execution time for this testing is very less. Some modules that are more complex in nature and are more prone to defects cannot be isolated and tested.

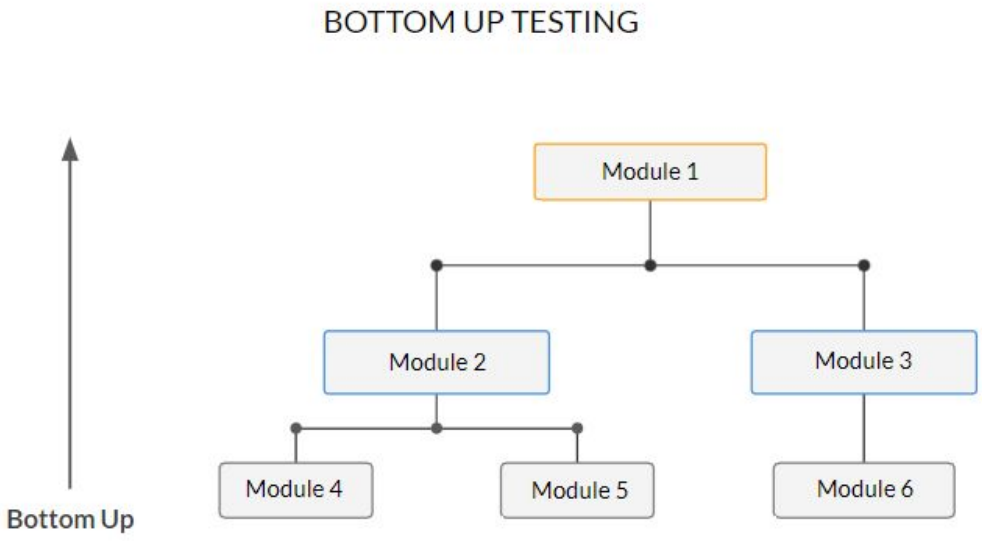
**Top-down testing** is an approach to integration testing where top-level units are tested first and lower-level units are tested one by one after that. So, this technique essentially starts from the topmost module and gradually progresses towards the lower ones. Only the top module is unit tested in isolation. After this, the lower modules are integrated one by one. The process is repeated until all the modules are integrated and tested.
The following image visualises top-down integration testing:




Here, **stubs** are the code snippets that accept the inputs/requests from the top module and return the results/response. Therefore, in spite of the lower modules not existing, we are able to test the top module. In top-down testing, fault localisation would be easier. The testing need not wait until all the modules are developed. The critical modules can be tested on priority.
However, since most of the modules are not developed while testing in a top-down manner, a lot of stubs are needed.

**Bottom-up integration testing** is a strategy in which the lower-level modules are tested first. These tested modules are then further used to facilitate the testing of higher-level modules. The process

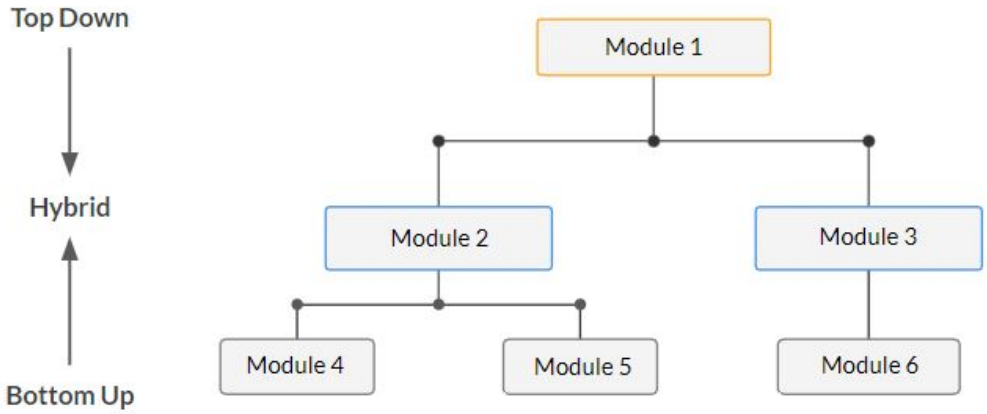
continues until all modules at the top level are tested. Once the lower-level modules are tested and integrated, then the next level of modules are formed.
The following image visualises bottom-up integration testing:




Here, **drivers** are the dummy programs that are used to call the functions of the lowest module in a case where the calling function does not exist. The bottom-up technique requires the module driver to feed the test case input into the interface of the module being tested.
The advantage of this approach is that if a major fault exists at the lowest unit of the program, it is easier to detect it, and corrective measures can be taken.
The disadvantage is that the main program does not exist until the last module is integrated and tested. As a result, the higher-level design flaws can be detected only at the end.

**Sandwich testing** is a strategy in which top-level modules are tested with lower-level modules at the same time; lower modules are integrated with top modules and tested as a system. It is a combination of the top-down and bottom-up approaches. Therefore, it is also known as Hybrid Integration Testing. It makes use of both stubs and drivers.
The following image visualises sandwich integration testing:

SANDWICH TESTING




When we test huge programs such as operating systems, we need to have some more techniques that are efficient and increase confidence of developers in the robustness of developer software. Sandwich testing plays a very important role here, where both the top-down and the bottom-up testing are started simultaneously.

Since both approaches start simultaneously, this technique is a bit complex and requires more people with specific skill sets, which, in turn, adds to the cost.

## Cloud Deployment

In this session, you learnt about cloud, cloud infrastructure and need in the modern software deployment process. You learnt about different cloud services that can be utilised. You also understood how cloud infrastructure can be leveraged to deploy the UPSTAC application.

# Introduction to Cloud

Cloud refers to servers that are accessed over the Internet and the software and databases that run on those servers.
Cloud computing is the on-demand delivery of IT resources over the Internet with pay-as-you-go pricing. Instead of buying, owning and maintaining physical data centres and servers, you can access technology services, such as computing power, storage and databases, on an as-needed basis from a cloud provider such Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP).

Advantages of cloud infrastructure:

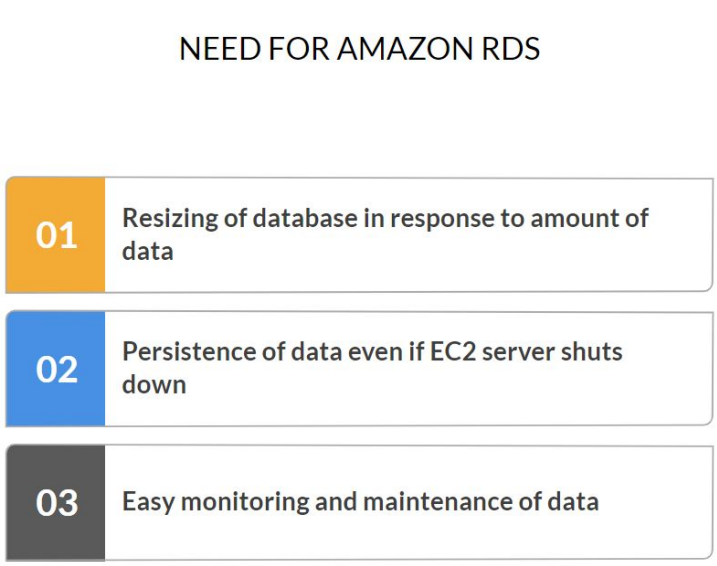
- Cloud gives you easy access to a broad range of technologies so that you can innovate faster and build nearly anything that you can imagine.
- With cloud computing, you do not have to overprovision resources up front to handle peak levels of business activity in the future. Instead, you provision the number of resources that you need. You can scale these resources up or down instantly to grow and shrink the capacity as your business demands.
- Cloud allows you to trade capital expenses (such as data centres and physical servers) for variable expenses and only pay for it as you consume it. Plus, the variable expenses are much lower than what you would pay to do it yourself because of economies of scale.
- With cloud, you can expand to new geographic regions and deploy the application globally in minutes. For example, AWS has infrastructure all over the world; so, you can deploy your application in multiple locations with just a few clicks. Putting applications in closer proximity to end users reduces latency and improves their experience.

Different AWS services:

**Amazon Elastic Compute Cloud (Amazon EC2)** is a web service that provides secure, resizable compute capacity in cloud. It is designed to make web-scale cloud computing easier for developers. Amazon EC2's simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment.

**Amazon Relational Database Service (Amazon RDS)** makes it easy to set up, operate and scale a relational database in cloud. It provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as hardware provisioning, database set-up, patching and backups. It frees up your time to allow you to focus on your applications so that you can give them the fast performance, high availability, security and compatibility that they need.

Advantages of using Amazon RDS for database management:

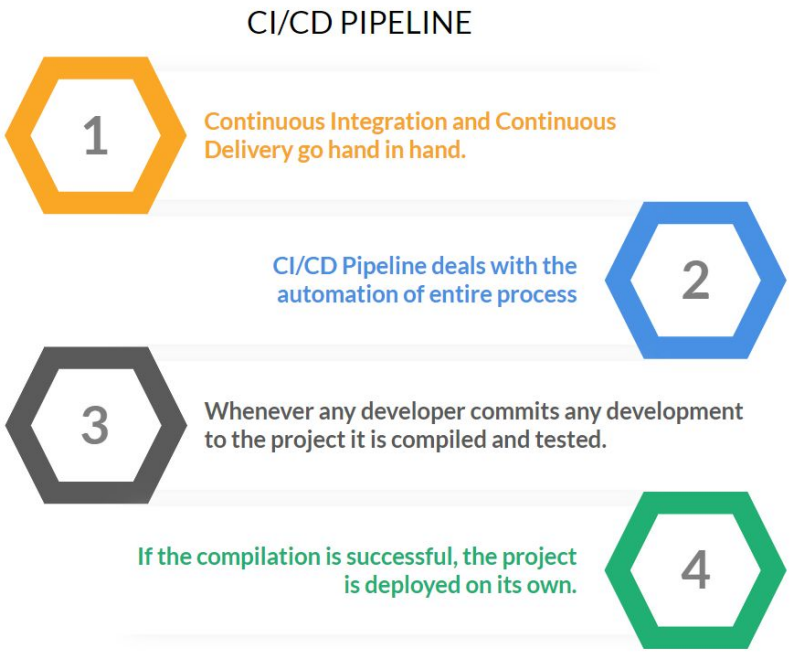



The modern approach of continuous integration and continuous delivery:

- **Continuous Integration (CI)** is an approach to establish a consistent and automated way to build, package and test applications. With consistency in the integration process in place, teams are more likely to commit code changes more frequently, which leads to better collaboration and software quality.
- **Continuous delivery (CD)** automates the delivery of applications to selected infrastructure environments. If the updated build is successful, it is deployed right away.

**Jenkins** is one of the most widely used automation tools; it helps the software development process in automating building, testing and deploying a project. It is deployed in a separate EC2 instance to automate the CI/CD process.

The automation of the continuous integration and continuous delivery processes is called developing a CI/CD pipeline:




# Software Documentation

In this session, you learnt what is software documentation, its importance in the software development lifecycle and the essential components of a good software documentation.

## Introduction to Software Documentation

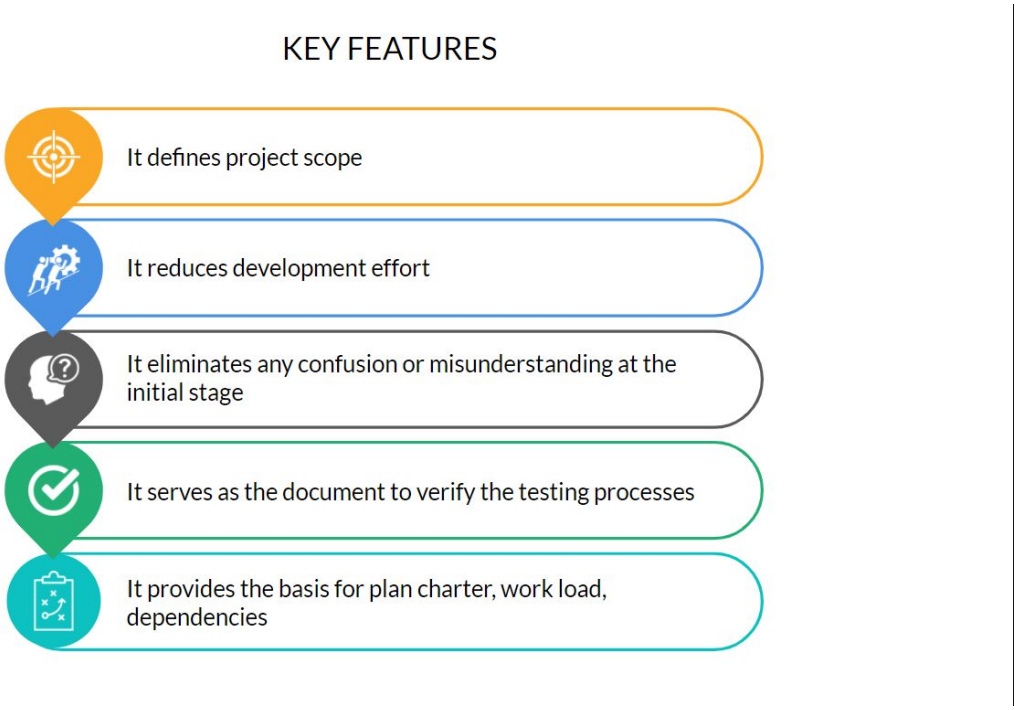
A **software requirements specification (SRS)** is a document that lays out the description of the software that is to be developed as well as the intention of the software under development. It contains details of what the software is supposed to do as well as how it is supposed to perform.

A standard software document is not all text, diagram or code; it is a combination of all of these. The major part of this document consists of descriptions about different components of the

software. Flow diagrams and algorithms may be specified to provide a better understanding of the software. However, the complete code used to build the software is not included in the SRS.

All this information about the requirements is used by different sections involved in the software process. An SRS is used by core developers, testers, software architects, project managers and even end-users.

The key features of an SRS are as follows:




## Components of Sofware Documentation

The key components of SRS include:

The **purpose** identifies the product whose software requirements are specified in this document, including the revision or a release number. It also describes the scope of the product that is covered by this SRS, particularly if this SRS describes only a part of the system or a single subsystem.

**Document conventions** explain the formats used in the document, such as what are bold letters used for, when is italic used and so on. For example, abbreviations that are used throughout the document would be mentioned here.

**Product scope** provides a short description of the software being specified along with its purpose, including relevant benefits, objectives and goals. It relates the software to corporate goals or business strategies. If a separate vision and scope document is available, it is referred to here rather than duplicating its contents.

**Overall description** has different segments, which are described below.

1. **Product perspective** describes the context within which the product (software) is being built. It also mentions whether the software is part of a product family, a replacement for an already existing member, or a completely new and unique product.

2. **Product function** is a list of the major functions that the product is supposed to perform or let users perform. This should be readable and easily understandable by all the intended readers of the SRS. Only a high-level summary of the product functions (such as a bullet list) is needed here. Sometimes, a top-level data-flow diagram may also be helpful.
3. **Operating environment** describes the environment in which the product will operate, such as the platform operating system version as well as any other component with which it must comply or be able to run alongside.
4. **Design and implementation constraints** lay out any constraints or issues that will limit options to developers. These might be hardware limitations, interfaces, particular technologies, parallel operations, language requirements, regulatory policies, corporate policies, security provisions, etc.
5. **User documentation** includes the user manuals, tutorials, etc. that are provided along with the software. These are primarily helpful for the end-users.
6. **Assumptions and dependencies** list any assumptions that could affect the requirements stated in the SRS. The software project might be affected if these assumptions are not provided or are incorrect, or change. Also listed here is any dependency that the product will have on external factors, such as third-party components that exist outside the current project.

**External interface requirements** describe the requirements of interfaces between the software and other components such as hardware or communication interfaces:

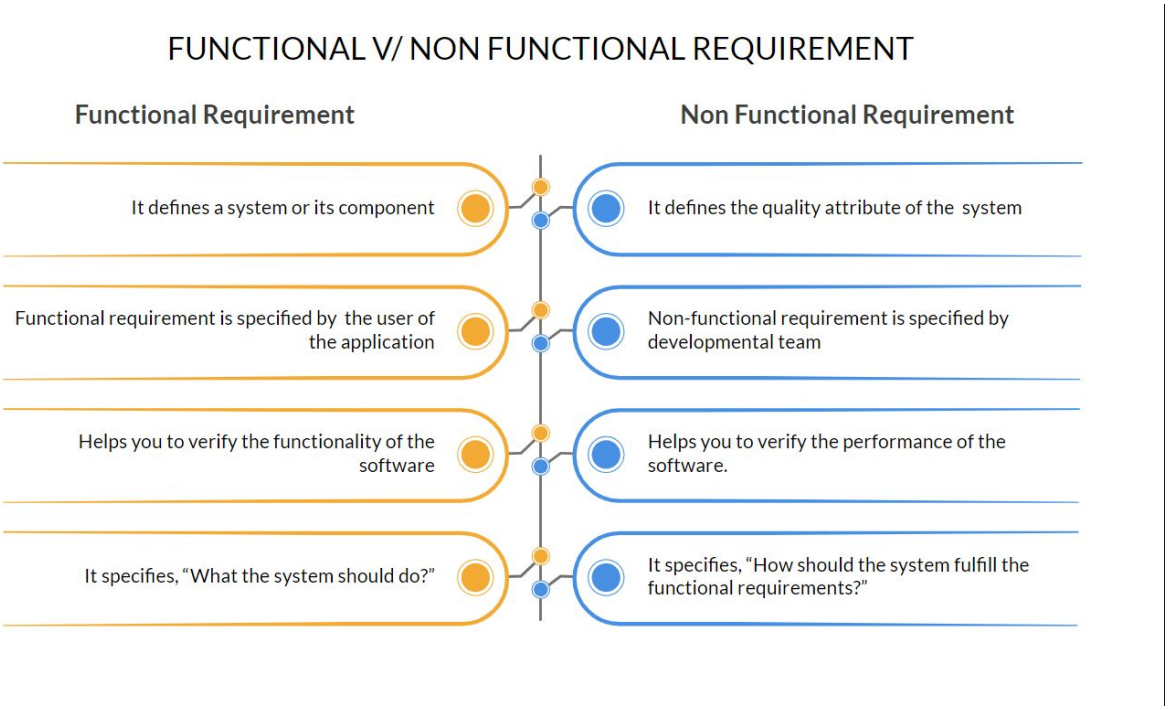
1. **User interface** describes the logical characteristics of an interface between the software and the user. This may include screen image samples, graphical user interface standards, screen layout constraints, buttons and functions, keyboard shortcuts, message displays, etc.
2. **Software interface** describes the connections between the product and any other specific software components, be it operating systems, databases libraries, etc.

3. **Communications interface** describes any communication functions required by the software, such as email, web browser, network server communications protocols and electronic forms. Any communication standards to be used are also identified. Communication security and/or encryption issues are specified.

**Functional requirements** define the basic system behaviour. Essentially, they are what the system does or must not do and can be thought of in terms of how the system responds to inputs. Functional requirements usually define if-then behaviours and include calculations, data input and business processes.

While functional requirements define what the system does or must not do, **non-functional** requirements specify how the system should do it. Non-functional requirements define system behaviour, features and general characteristics that affect the user experience. How well non-functional requirements are defined and executed determines how easy the system is to use and how it is performing. Non-functional requirements are product properties and focus on user expectations.

The differences between functional and non-functional requirements:



The differences between **User document** and **Functional document:**

1. A user document is intended to assist users in configuring and utilising the application, while a functional document is intended to help developers understand what features they need to build and how.
2. As the name suggests, a user document is more relevant to the user than a functional document. A user can quickly and easily know how the application should be configured, what can be done via this application and how it can be utilised.

**Disclaimer**: All content and material on the upGrad website is copyrighted material, either belonging to upGrad or its bonafide contributors and is purely for the dissemination of education. You are permitted to access, print and download extracts from this site purely for your own education only and on the following basis:

- You can download this document from the website for self-use only.
- Any copies of this document, in part or full, saved to disk or to any other storage medium may only be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.
- Any further dissemination, distribution, reproduction, copying of the content of the document herein or the uploading thereof on other websites or use of content for any other commercial/unauthorised purposes in any way which could infringe the intellectual property rights of upGrad or its contributors, is strictly prohibited.
- No graphics, images or photographs from any accompanying text in this document will be used separately for unauthorised purposes.
- No material in this document will be modified, adapted or altered in any way.
- No part of this document or upGrad content may be reproduced or stored in any other web site or included in any public or private electronic retrieval system or service without upGrad's prior written permission.
- Any rights not expressly granted in these terms are reserved.