**What Is SDLC? Understand the Software Development Life Cycle**

The Software Development Life Cycle (SDLC) refers to a methodology with clearly defined processes for creating high-quality software. in detail, the SDLC methodology focuses on the following phases of software development:

* Requirement analysis
* Planning
* Software design such as architectural design
* Software development
* Testing
* Deployment

**What is the software development life cycle?**

SDLC or the Software Development Life Cycle is a process that produces software with the highest quality and lowest cost in the shortest time possible. SDLC provides a well-structured flow of phases that help an organization to quickly produce high-quality software which is well-tested and ready for production use.

The SDLC involves six phases as explained in the introduction. Popular SDLC models include the [waterfall model](https://economictimes.indiatimes.com/definition/waterfall-model), [spiral model](http://searchsoftwarequality.techtarget.com/definition/spiral-model), and [Agile model](http://istqbexamcertification.com/what-is-agile-model-advantages-disadvantages-and-when-to-use-it/).

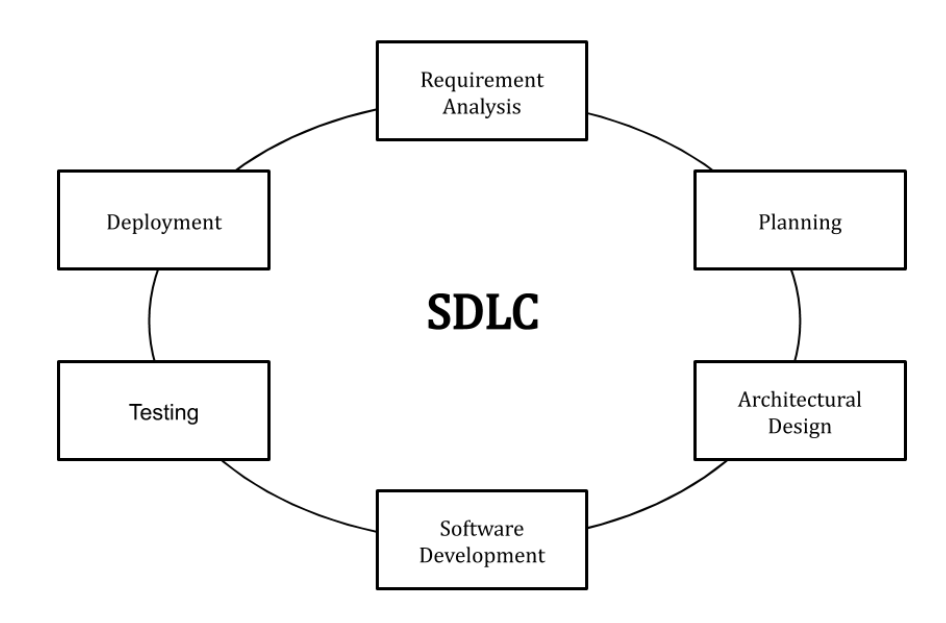
**How the SDLC Works**

SDLC works by lowering the cost of software development while simultaneously improving quality and shortening production time. SDLC achieves these apparently divergent goals by following a plan that removes the typical pitfalls of software development projects. That plan starts by evaluating existing systems for deficiencies.

Next, it defines the requirements of the new system. It then creates the software through the stages of analysis, planning, design, development, testing, and deployment. By anticipating costly mistakes like failing to ask the end-user or client for feedback, SLDC can eliminate redundant rework and after-the-fact fixes.

It’s also important to know that there is a strong focus on the testing phase. As the SDLC is a repetitive methodology, you have to ensure code quality at every cycle. Many organizations tend to spend few efforts on testing while a stronger focus on testing can save them a lot of rework, time, and money. Be smart and write the right types of tests.

Next, let’s explore the different stages of the Software Development Life Cycle.



## Stages and Best Practices

Following the best practices and/or stages of SDLC ensures the process works in a smooth, efficient, and productive way.

### **1. Identify the Current Problems**

“What are the [current](https://stackify.com/sdlc-phases-identify-problems/) problems?” This stage of the SDLC means getting input from all stakeholders, including customers, salespeople, industry experts, and programmers. Learn the strengths and weaknesses of the current system with improvement as the goal.

### **2. Plan**

“What do we want?” In this stage of the SDLC, the team determines the cost and resources required for implementing the analyzed requirements. It also details the risks involved and provides sub-plans for softening those risks.

In other words, the team should determine the feasibility of the project and how they can implement the project successfully with the lowest risk in mind.

### **3. Design**

“How will we get what we want?” This phase of the SDLC starts by turning the software specifications into a design plan called the Design Specification. All stakeholders then review this plan and offer feedback and suggestions. It’s crucial to have a plan for collecting and incorporating stakeholder input into this document. Failure at this stage will almost certainly result in cost overruns at best and the total collapse of the project at worst.

### **4. Build**

“Let’s create what we want.”

At this stage, the actual development starts. It’s important that every developer sticks to the agreed blueprint. Also, make sure you have proper guidelines in place about the code style and practices.

For example, define a nomenclature for files or define a variable naming style such as [camelCase](https://en.wikipedia.org/wiki/Camel_case" \t "_blank). This will help your team to produce organized and consistent code that is easier to understand but also to test during the next phase.

### **5. Code Test**

“Did we get what we want?” In this stage, we test for defects and deficiencies. We fix those issues until the product meets the original specifications.

In short, we want to verify if the code meets the defined requirements.

Try [Stackify’s free code profiler, Prefix](https://stackify.com/prefix" \t "_blank), to write better code on your workstation. Prefix works with .NET, Java, PHP, Node.js, Ruby, and Python.

### **6. Software Deployment**

“Let’s start using what we got.”

At this stage, the goal is to deploy the software to the production environment so users can start using the product. However, many organizations choose to move the product through different deployment environments such as a testing or staging environment.

This allows any stakeholders to safely play with the product before releasing it to the market. Besides, this allows any final mistakes to be caught before releasing the product.

### **Extra: Software Maintenance**

“Let’s get this closer to what we want.” The plan almost never turns out perfect when it meets reality. Further, as conditions in the real world change, we need to update and advance the software to match.

The [DevOps movement](https://stackify.com/what-is-devops/) has changed the SDLC in some ways. Developers are now responsible for more and more steps of the entire development process. We also see the value of shifting left. When development and Ops teams use the same toolset to track performance and pin down defects from inception to the retirement of an application, this provides a common language and faster handoffs between teams.

Application performance monitoring (APM) tools can be used in a development, QA, and production environment. This keeps everyone using the same toolset across the entire development lifecycle.

Read More: [3 Reasons Why APM Usage is Shifting Left to Development & QA](https://stackify.com/apm-shifting-left-development-qa/)

## Examples

The most common SDLC examples or SDLC models are listed below.

### **Waterfall Model**

This SDLC model is the oldest and most straightforward. With this methodology, we finish one phase and then start the next. Each phase has its own mini-plan and each phase “waterfalls” into the next. The biggest drawback of this model is that small details left incomplete can hold up the entire process.

### **Agile Model**

The Agile SDLC model separates the product into cycles and delivers a working product very quickly. This methodology produces a succession of releases. Testing of each release feeds back info that’s incorporated into the next version. [According to Robert Half](https://www.roberthalf.com/technology/blog/6-basic-sdlc-methodologies-the-pros-and-cons), the drawback of this model is that the heavy emphasis on customer interaction can lead the project in the wrong direction in some cases.

### **Iterative Model**

This SDLC model emphasizes repetition. Developers create a version very quickly and for relatively little cost, then test and improve it through rapid and successive versions. One big disadvantage here is that it can eat up resources fast if left unchecked.

### **V-Shaped Model**

An extension of the waterfall model, this SDLC methodology tests at each stage of development. As with waterfall, this process can run into roadblocks.

### **Big Bang Model**

This high-risk SDLC model throws most of its resources at development and works best for small projects. It lacks the thorough requirements definition stage of the other methods.

### **Spiral Model**

The most flexible of the SDLC models, the spiral model is similar to the iterative model in its emphasis on repetition. The spiral model goes through the planning, design, build and test [phases](https://stackify.com/sdlc-phases-identify-problems/) over and over, with gradual improvements at each pass.

## Benefits of the SDLC

SDLC done right can allow the highest level of management control and documentation. Developers understand what they should build and why. All parties agree on the goal upfront and see a clear plan for arriving at that goal. Everyone understands the costs and resources required.

Several pitfalls can turn an SDLC implementation into more of a roadblock to development than a tool that helps us. Failure to take into account the needs of customers and all users and stakeholders can result in a poor understanding of the system requirements at the outset. The benefits of SDLC only exist if the plan is followed faithfully.

Want to improve application quality and monitor application performance at every stage of the SDLC? [Try out Stackify’s Retrace tool](https://stackify.com/retrace/) for free and experience how it can help your organization at producing higher-quality software.

**Testing**

**What is Functional Testing?**

Functional testing is a type of testing which verifies that each **function** of the software application operates in conformance with the requirement specification. This testing mainly involves black box testing, and it is not concerned about the source code of the application.

Every functionality of the system is tested by providing appropriate input, verifying the output and comparing the actual results with the expected results. This testing involves checking of User Interface, APIs, Database, security, client/ server applications and functionality of the Application Under Test. The testing can be done either manually or using automation

**What is Non-Functional Testing?**

Non-functional testing is a type of testing to check non-functional aspects (performance, usability, reliability, etc.) of a software application. It is explicitly designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing.

A good example of non-functional test would be to check how many people can simultaneously login into a software.

Non-functional testing is equally important as functional testing and affects client satisfaction.

**KEY DIFFERENCE**

* Functional testing verifies each function/feature of the software whereas Non Functional testing verifies non-functional aspects like performance, usability, reliability, etc.
* Functional testing can be done manually whereas Non Functional testing is hard to perform manually.
* Functional testing is based on customer’s requirements whereas Non Functional testing is based on customer’s expectations.
* Functional testing has a goal to validate software actions whereas Non Functional testing has a goal to validate the performance of the software.
* A Functional Testing example is to check the login functionality whereas a Non Functional testing example is to check the dashboard should load in 2 seconds.
* Functional describes what the product does whereas Non Functional describes how the product works.
* Functional testing is performed before the non-functional testing.

**Functional Vs. Non-Functional Testing**

| **Parameters** | **Functional** | **Non-functional testing** |
| --- | --- | --- |
| **Execution** | It is performed before non-functional testing. | It is performed after the functional testing. |
| **Focus area** | It is based on customer's requirements. | It focusses on customer's expectation. |
| **Requirement** | It is easy to define functional requirements. | It is difficult to define the requirements for non-functional testing. |
| **Usage** | Helps to validate the behavior of the application. | Helps to validate the performance of the application. |
| **Objective** | Carried out to validate software actions. | It is done to validate the performance of the software. |
| **Requirements** | Functional testing is carried out using the functional specification. | This kind of testing is carried out by performance specifications |
| **Manual testing** | Functional testing is easy to execute by manual testing. | It's very hard to perform non-functional testing manually. |
| **Functionality** | It describes what the product does. | It describes how the product works. |
| **Example Test Case** | Check login functionality. | The dashboard should load in 2 seconds. |
| **Testing Types** | Examples of Functional Testing Types   * Unit testing * Smoke testing * User Acceptance * Integration Testing * Regression testing * Localization * Globalization * Interoperability | Examples of Non-functional Testing Types   * Performance Testing * Volume Testing * Scalability * Usability Testing * Load Testing * Stress Testing * Compliance Testing * Portability Testing * Disaster Recover Testing |

## What is a server?

 A server is a computer or system that provides resources, data, services, or programs to other computers, known as clients, over a network. In theory, whenever computers share resources with client machines they are considered servers. There are many types of servers, including web servers, mail servers, and virtual servers.

An individual system can provide resources and use them from another system at the same time. This means that a device could be both a server and a client at the same time.

Some of the first servers were mainframe computers or minicomputers. Minicomputers were much smaller than mainframe computers, hence the name. However, as technology progressed, they ended up becoming much larger than desktop computers, which made the term microcomputer somewhat farcical.  
  
Initially, such servers were connected to clients known as terminals that did not do any actual computing. These terminals, referred to as dumb terminals, existed simply to accept input via a keyboard or card reader and to return the results of any computations to a display screen or printer. The actual computing was done on the server.  
  
Later, servers were often single, powerful computers connected over a network to a set of less-powerful client computers. This network architecture is often referred to as the client-server model, in which both the client computer and the server possess computing power, but certain tasks are delegated to servers. In previous computing models, such as the mainframe-terminal model, the mainframe did act as a server even though it wasn’t referred to by that name.  
  
As technology has evolved, the definition of a server has evolved with it. These days, a server may be nothing more than software running on one or more physical computing devices. Such servers are often referred to as virtual servers. Originally, virtual servers were used to increase the number of server functions a single hardware server could do. Today, virtual servers are often run by a third-party on hardware across the Internet in an arrangement called cloud computing.  
  
A server may be designed to do a single task, such as a mail server, which accepts and stores email and then provides it to a requesting client. Servers may also perform several tasks, such as a file and print server, which both stores files and accepts print jobs from clients and then sends them on to a network-attached printer.

## What is containerization?

Containerization has become a major trend in software development as an alternative or companion to [virtualization](https://www.ibm.com/in-en/cloud/learn/virtualization-a-complete-guide). It involves encapsulating or packaging up software code and all its dependencies so that it can run uniformly and consistently on any infrastructure. The technology is quickly maturing, resulting in measurable benefits for developers and operations teams as well as overall software infrastructure.

Containerization allows developers to create and deploy applications faster and more securely. With traditional methods, code is developed in a specific computing environment which, when transferred to a new location, often results in bugs and errors. For example, when a developer transfers code from a desktop computer to a [virtual machine](https://www.ibm.com/in-en/cloud/learn/virtual-machines) (VM) or from a Linux to a Windows operating system. Containerization eliminates this problem by bundling the application code together with the related configuration files, libraries, and dependencies required for it to run. This single package of software or “container” is abstracted away from the host operating system, and hence, it stands alone and becomes portable—able to run across any platform or cloud, free of issues.

## Application of containerization

Containers encapsulate an application as a single executable package of software that bundles application code together with all of the related configuration files, libraries, and dependencies required for it to run. Containerized applications are “isolated” in that they do not bundle in a copy of the operating system. Instead, an open source runtime engine (such as the Docker runtime engine) is installed on the host’s operating system and becomes the conduit for containers to share an operating system with other containers on the same computing system.

Other container layers, like common bins and libraries, can also be shared among multiple containers. This eliminates the overhead of running an operating system within each application and makes containers smaller in capacity and faster to start up, driving higher server efficiencies. The isolation of applications as containers also reduces the chance that malicious code present in one container will impact other containers or invade the host system.

# **Kubernetes**

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

## What is root cause analysis?

Root cause analysis (RCA) is the process of discovering the root causes of problems in order to identify appropriate solutions. RCA assumes that it is much more effective to systematically prevent and solve for underlying issues rather than just treating ad hoc symptoms and putting out fires. Root cause analysis can be performed with a collection of principles, techniques, and methodologies that can all be leveraged to identify the root causes of an event or trend. Looking beyond superficial cause and effect, RCA can show where processes or systems failed or caused an issue in the first place.

**Internet Relay Chat (IRC)** is a text-based chat system. It enables discussions among any number of participants in so-called conversation channels, as well as discussions between only two partners — for example, in question-and-answer dialogues.[[1]](https://en.wikipedia.org/wiki/Internet_Relay_Chat#cite_note-RFC_1459-1) Any participant may open a new conversation channel, and a single computer user can also take part in several such simultaneous channels.

In order to establish or join a chat, a network program, called an IRC client, is required for accessing a channel by connecting to a server. The IRC client can be either an independent program on the local computer (e.g. mIRC, XChat) or take the form of a special user interface from within a larger program such as an Internet browser; the [Opera](https://en.wikipedia.org/wiki/Opera) browser, for example, includes an IRC client, and clients such as [Mibbit](https://en.wikipedia.org/wiki/Mibbit" \o "Mibbit) and [IRCCloud](https://en.wikipedia.org/wiki/IRCCloud" \o "IRCCloud) or the open source [KiwiIRC](https://en.wikipedia.org/wiki/KiwiIRC" \o "KiwiIRC) and [MIT](https://en.wikipedia.org/wiki/Massachusetts_Institute_of_Technology)'s The Lounge Chat can work in connection with many popular browsers.