SOFTWARE ENGINEERING

UNIT – 1

TOPIC – 1

INTRODUCTION

Introduction to Software Engineering

1. What is Engineering?

- **Engineering** is the process of designing and building things to solve problems.
- It involves creating a plan and then making something useful, such as a bridge or an application.

2. What is Software?

- **Software** is a set of instructions that tells a computer what to do.
- It's similar to a recipe that guides a computer in performing tasks.

3. Definition of Software Engineering

- **Software Engineering** is a structured approach to developing, operating, and maintaining software.
- It involves the application of engineering principles to software development in a systematic method.

Why Software Engineering is important?

Software engineering provides a structured approach to designing and developing software, which is crucial for creating reliable and efficient applications.

Real-World Impact: As a graduate student, mastering software engineering equips you with the skills to build software that solves real-world problems, from simple apps to complex systems.

Real-World Examples of Software Engineering Implementation:

1. **Healthcare Systems:** Software engineering principles are used to develop electronic health records (EHR) systems that manage patient data, ensuring accuracy, security, and efficiency in healthcare delivery.
2. **E-commerce Platforms:** Online shopping sites like Amazon and Flipkart rely on robust software engineering practices to handle millions of transactions daily, ensuring a seamless user experience.
3. **Mobile Applications:** Apps like WhatsApp and Uber are built using software engineering techniques to ensure they are scalable, secure, and user-friendly.

For graduate students, understanding software engineering is essential because it not only helps in creating functional software but also ensures that the software is maintainable, scalable, and reliable. This knowledge is applicable in various industries, making it a valuable skill set for your career.

SOFTWARE ENGINEERING

UNIT – 1

TOPIC – 2 – PART – 1

NATURE OF SOFTWARE AND SOFTWARE APPLICATION DOMAINS

I. Definition

Software is program/ set of programs containing instructions which provide desired functionality. Also comprises of data structures that enable the program to manipulate information

Engineering is the process of designing and building something that serves a particular purpose

Software Engineering is a systematic approach to the development, operation and maintenance of desired software

II. Evolving Role of Software

Software serves as a:

- Product: It delivers the computing potential of a H/W i.e., enables the h/w to deliver the expected functionality. Acts as information transformer
- Vehicle to deliver the product: Helps in creation and control of other programs i.e., it helps other software to do functions and helps as platform. E.g., Operating System

Today, software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time—information. Software transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide

information networks (e.g., Internet) and provides the means for acquiring information in all of its forms. The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems. The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.

III. Why Software Engineering is important?

i. Imposes discipline to work that can become quite chaotic - Lot of steps are involved in the development of a s/w, so if a systematic approach is not taken, it becomes difficult/clumsy

ii. Ensures high quality of software - If a s/w could deliver the features and functionalities required then it is a high-quality software.

iii. Enables us to build complex systems in a timely manner - Whenever we have a huge/complex project, then we need to set proper deadlines and milestones of the time taken in each step, so that in time we can deliver the s/w to the customer.

IV. Difference between Software and Hardware

Software:

- S/W is logical unit
- No spare parts for s/w
- Problem statement may not be complete and clear initially
- Requirements may change with time
- Multiple copies (less cost)
- Idealized and actual graph

Hardware:

- H/W is physical unit
- Spare parts for h/w exist

- Problem statement is clearly mentioned at the beginning of development
- Requirements are fixed before manufacturing
- Multiple copies (more cost)
- Bath tub graph

Bath Tub Curve:

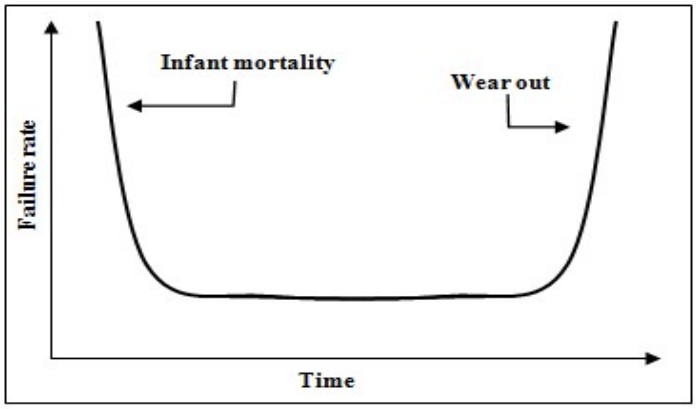


Figure 1: Bath tub Curve- Failure curve for Hardware

- Given the name bath tub because its shape is same as bath tub.
- It explains the reliability of the product i.e. until how many days / time the product works
- We have 3 stages in bath tub curve:
  - Decreasing failure rate or Infant mortality
  - Constant failure rate
  - Increasing failure rate or wear out (H/W effected by the environment factors like dust, temperature, pollution, etc). S/W does not have wear out.
- Decreasing failure rate:
  - As the product in this stage is new, there are very less chances of it to fail. The product still fails because of
    - Manufacturing defect
    - We haven't assembled it properly

- It is a weak part or product.

- Constant failure rate:
  - It is named as constant failure rate because the graph remains in straight line according to the time.
  - Most of the products fail in this stage
  - It is the service life of the product.
- Increasing failure rate:
  - This is the stage where we use the product more than its service period. The suddenly, the product may fail.
  - It is named increasing failure rate as the curve moves upwards as shown according to the time.

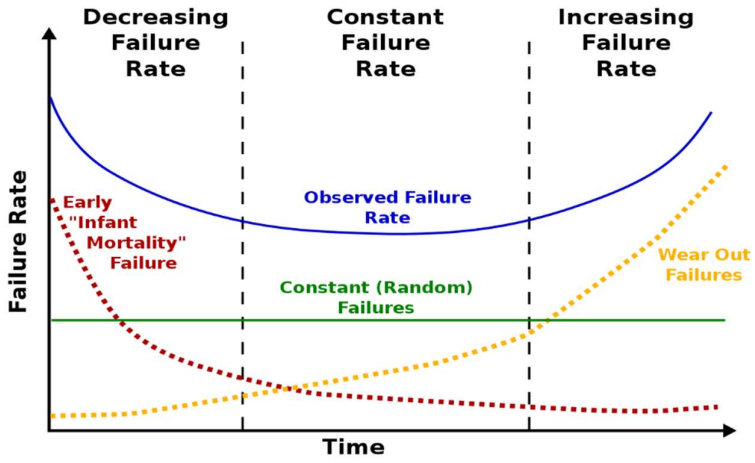


Figure 2: Failure rates in Bath tub Curve

Idealized Curve:

- Since there is no wear out in s/w the curve must undergo on 2 phases i.e., decreasing failure rate and constant failure rate which is called as Idealized curve.
- But in reality, idealized curve is not possible.
- Initial failure happens just like h/w due to undetected defects.

- After correcting the defects, the curve reaches a steady state, but if any change occurs in the s/w then there is spike in the graph.
- Most of the times the failure rate increases when a change effect is requested. The actual curve is higher than the idealized curve.

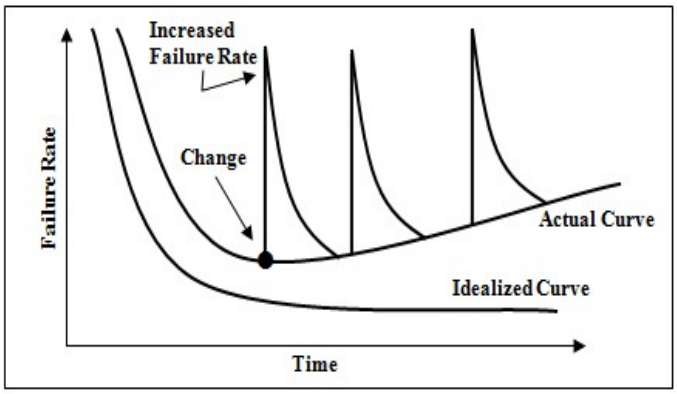


Figure 3: Idealized Curve- Failure curve for Software

SOFTWARE ENGINEERING

UNIT – 1

TOPIC – 2 – PART – 2

NATURE OF SOFTWARE AND SOFTWARE APPLICATION DOMAINS

I. Work Product:

Work product is the result or outcome of Software Engineering

The outcome is in 2 perspectives:

1. Software Engineer: The set of programs, the content (data) along with documentation that is a part of s/w.

2. User/Customer: The functionality delivered by the s/w that improves user experience

II. Software Engineering focuses on:

- Quality (While Software development)

– Functional: To what extent that we have delivered the correct s/w and is it reaching the expectations. Degree to which correct software is produced.

– Non functional: Also called structural attributes. Features other than functions of s/w like robustness, security, etc.

- Maintainability (After the software has been developed and delivered)
  - Should be easily enhanced and adapted to changing requirements whenever required.

III. Nature of Software:

Software is a key component of modern technology, driving everything from simple mobile apps to complex systems in industries like healthcare, finance, and transportation. Unlike

physical products, software is intangible—it can't be seen or touched. Instead, it exists as a collection of ideas and logic, embodied in code that tells computers what to do.

Key Characteristics of Software:

1. Intangible Nature: Software differs from physical products as it cannot be physically interacted with. Its existence is in the form of code and logical structures.

2. Complexity: Software systems can be incredibly complex, involving millions of lines of code. This complexity can make software development, maintenance, and management challenging.

3. Evolution Over Time: Software is not static. It needs to be continuously updated and improved to adapt to new user requirements and technological advancements.

4. No Wear and Tear: Unlike hardware, software does not physically degrade over time. However, it can become outdated or incompatible with newer technologies.

5. High Dependability: Modern society relies heavily on software, making its reliability essential. Software plays a critical role in everything from communication to transportation.

IV. Software Application Domains

Software is used in a variety of domains, each serving different purposes and industries. Here are some key areas where software plays a vital role:

1. Web Applications: These include websites and online platforms such as social media, e-commerce sites, and blogs. Web applications are accessible through browsers and are essential for modern communication and commerce.

2. Mobile Applications: Designed for smartphones and tablets, mobile apps include games, messaging apps, and fitness trackers. These apps are tailored to mobile devices' operating systems and hardware.

3. Enterprise Applications: Businesses use enterprise software to manage operations, such as accounting systems, customer relationship management (CRM) tools, and human resources (HR) software. These applications are crucial for efficient business operations.

4. Embedded Systems: Embedded software runs on devices like smartwatches, home appliances, and car systems. This type of software controls specific functions and is often embedded directly into the hardware it operates.

5. Scientific and Engineering Applications: These are specialized programs used in scientific research, simulations, or engineering tasks. Examples include weather prediction software and computer-aided design (CAD) tools.

6. Artificial Intelligence (AI) and Machine Learning (ML): AI and ML systems can learn from data and make decisions. Examples include chatbots, recommendation engines, and self-driving cars.

7. Gaming: Gaming software ranges from simple mobile games to complex video games. This domain focuses on entertainment and is a major industry in its own right.

These domains highlight the diverse applications of software and its importance across various sectors.

SOFTWARE ENGINEERING

UNIT – 1

TOPIC – 3

LEGACY SOFTWARE AND SOFTWARE MYTHS

I. Legacy Software:

What is Legacy Software?

- Legacy software is old software still used by companies even though it was made with outdated technology.
- It's hard to maintain and upgrade but is kept because it's important for daily operations.
- **Example:** Like an old classic car, it might be tough to repair, but it's valuable, so the owner keeps using it.

Types of Legacy Software:

1. **Old Operating Systems:** These are outdated systems no longer supported by developers.

   Example: Many hospitals still use Windows XP because upgrading could disrupt patient care.

2. **Mainframe Systems:** These are large, powerful computers used for critical tasks by big organizations.

   Example: Banks like the Royal Bank of Scotland still use these old systems to handle transactions.

3. **Custom-Built Applications:** Software specifically made for a company's needs many years ago, often without updates.

   Example: A manufacturing company might use an old system to manage production that doesn't work well with modern tools.

4. **Outdated Business Applications:** Old software used for tasks like accounting or inventory that may not work with new technology.

Example: Retail stores might use an old point-of-sale system that doesn't support mobile payments.

5. **Obsolete Databases:** Old databases that still store data but might not integrate with modern software.

   Example: Some government agencies still use databases from the 1980s, making data sharing difficult.

Who Uses Legacy Software?

1. **Government:** Many government offices use old software like COBOL.

   Example: The IRS faced delays in 2017 because its old system couldn't handle a high volume of tax returns.

2. **Unemployment Systems:** During COVID-19, some U.S. states had issues because their systems were over 40 years old.

   Example: The old systems couldn't handle the surge in unemployment claims.

3. **Banking:** Banks use old mainframe computers for critical operations.

   Example: The Royal Bank of Scotland had customer issues because its old system struggled with modern demands.

4. **Background Checks:** Outdated systems can lead to security risks.

   Example: In 2015, the U.S. National Background Investigations Bureau was hacked due to their old system.

5. **Retail:** Retailers keep old sales systems because updating them is expensive and disruptive.

   Example: Some large chains still use outdated cash registers that don't work well with new inventory systems.

Why Do Companies Still Use Legacy Software?

1. **Fear of Uncertainty:** Companies worry that new systems might not work well or cause new problems.

   Example: A company might hesitate to switch to a cloud system because they are unsure if it will be reliable.

2. **Budget Constraints:** Upgrading legacy software is expensive, and some companies can't afford it.

   Example: A small business might stick with old accounting software because buying and learning a new system is too costly.

3. **Operational Risks:** Updating systems can cause downtime, which some businesses can't afford.

   Example: Hospitals often keep old systems to avoid the risk of disrupting patient care.

4. **Custom Solutions:** Many legacy systems were custom-made for specific needs and still work well.

   Example: A logistics company might use an old tracking system that meets all their needs and hesitate to replace it.

5. **Training and Transition:** Switching to a new system means retraining employees, which takes time and reduces productivity.

   Example: A company might keep its old CRM system to avoid slowing down their sales team during a transition.

What Can Be Done with Legacy Software?

1. **Modernization:** Update the software to a newer version or move it to a modern platform.

   Example: Upgrading an old accounting system to a cloud-based version for better security.

2. **Integration:** Connect old software with new systems using APIs or middleware.

Example: A retailer might use middleware to link their old point-of-sale system with a new inventory system.

3. **Re-engineering:** Improve or rebuild parts of the software to work better.

   Example: A bank might enhance its old system to handle more transactions without replacing it entirely.

4. **Replacement:** Replace the old software with a new system.

   Example: A hospital might replace its old patient management system with a new, cloud-based solution.

5. **Maintenance and Support:** Keep using the legacy software but ensure it gets updates and support.

   Example: A government agency might pay for extended support to keep their old database system secure.

6. **Data Migration:** Move data from the old software to a new system.

   Example: Migrating data from an old customer database to a new CRM system.

7. **Training and Transition:** Help employees learn and adjust to new software.

   Example: Offering workshops to help staff get comfortable with a new project management tool.

II. Software Myths:

Software myths are common misconceptions or false beliefs about how software development works. These myths can lead to poor decision-making and problems in software projects. Understanding these myths helps avoid mistakes. There are three main types of myths: management myths, customer myths, and practitioner myths.

Management Myths:

These myths are often held by managers or decision-makers in a company:

Myth: "We can speed up the project by just adding more programmers."
Reality: Adding more programmers to a project doesn't always speed it up. In fact, it can slow things down because new team members need time to get up to speed, and coordinating more people can lead to more communication issues.
Example: If a project is already behind schedule, adding ten new developers might actually cause more delays as they need to be trained and integrated into the team.

Myth: "There's a book that has all the answers and procedures for software development."
Reality: While there are many helpful guidelines and best practices, each software project is unique and may require a different approach. There isn't a one-size-fits-all solution.
Example: A manager might rely on a popular project management book but find that the methods it suggests don't work well for their team's specific needs.

Myth: "We can use the same methods we used 10 years ago without any changes."
Reality: Technology and best practices evolve, so it's important to update methods and tools to stay effective. What worked a decade ago might not be the best approach today.
Example: A company might insist on using a waterfall development model because it worked for them in the past, even though agile methods could be more efficient for their current projects.

Customer Myths:

These myths are often held by the clients or customers who request the software:

Myth: "Just give a general idea; the details aren't important."
Reality: Clear and detailed requirements are essential for accurate development. Without specific details, developers might create something that doesn't meet the customer's needs, leading to wasted time and resources.
Example: A client might say, "I just need an app that tracks inventory," but without detailed requirements, the developers might not include critical features, like real-time stock updates or integration with accounting software.

Myth: "It's easy to make changes to the project at any time."

Reality: While some changes can be made easily, others can be complex and costly. It's important to consider the impact of changes before deciding to implement them.

Example: A customer might request a major feature change late in the project, not realizing that it could require reworking much of the existing code, delaying the project's completion.

Practitioner Myths:

These myths are commonly believed by software developers or engineers:

Myth: "Once the program works, the job is done."

Reality: The job isn't finished until the software is thoroughly tested and meets all requirements. Just because a program run doesn't mean it's ready for use.

Example: A developer might think their work is done when the software compiles and runs without errors, but without proper testing, there could still be bugs that affect its performance.

Myth: "I can't judge the quality of the code until it's running."

Reality: Code quality can be evaluated through code reviews, static analysis, and testing, even before the software is run. Good practices ensure that the code is clean, efficient, and maintainable.

Example: A programmer might skip code reviews thinking they'll spot any issues when they run the program, but this approach can lead to hard-to-find bugs and inefficient code.

Myth: "The only important deliverable is the working software."

Reality: While working software is crucial, other deliverables like documentation, code reviews, and user guides are also important for the long-term success and maintenance of the project.

Example: A team might deliver a fully functioning app but without any documentation. Later, when new developers join the team, they struggle to understand and maintain the codebase.

SOFTWARE ENGINEERING

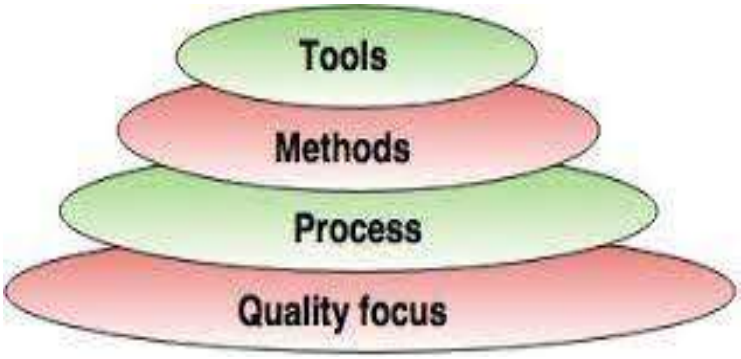
UNIT – 1

TOPIC – 4

SOFTWARE ENGINEERING – A LAYERED TECHNOLOGY, PROCESS FRAMEWORK, GENERIC PROCESS MODEL

I. SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY

Software engineering isn't just about writing code; it involves several layers that work together to produce reliable, high-quality software. These layers provide a structured approach to software development.



Software Engineering – A Layered Technology

Layers in Software Engineering

- **Process Layer:**
  - This layer defines the series of steps that need to be followed to develop software. It acts as a guide for what needs to be done and when.
  - **Example:** A company developing a mobile app may have a process that starts with gathering requirements, followed by design, coding, testing, and finally deployment.
- **Methods Layer:**
  - Methods are the techniques or procedures used during software development. They guide how specific tasks, such as designing the software architecture or testing the code, should be performed.

- **Example:** During the design phase, methods like UML diagrams (Unified Modelling Language) might be used to visually represent the system's components.

- **Tools Layer:**
  - Tools are software applications that help automate tasks or manage the process of software development. They make the work faster and more efficient.
  - **Example:** Version control tools like Git help developers keep track of changes in code, allowing multiple developers to work on the same project without conflicts.
- **Quality Focus (Bedrock):**
  - Quality is the foundation that supports all other layers. Every step, method, and tool used in the software engineering process must focus on ensuring the software meets the required quality standards.
  - **Example:** Before releasing a product, rigorous testing is conducted to ensure it performs well under various conditions, just like ensuring a cake is baked properly before serving.

II. THE SOFTWARE PROCESS FRAMEWORK

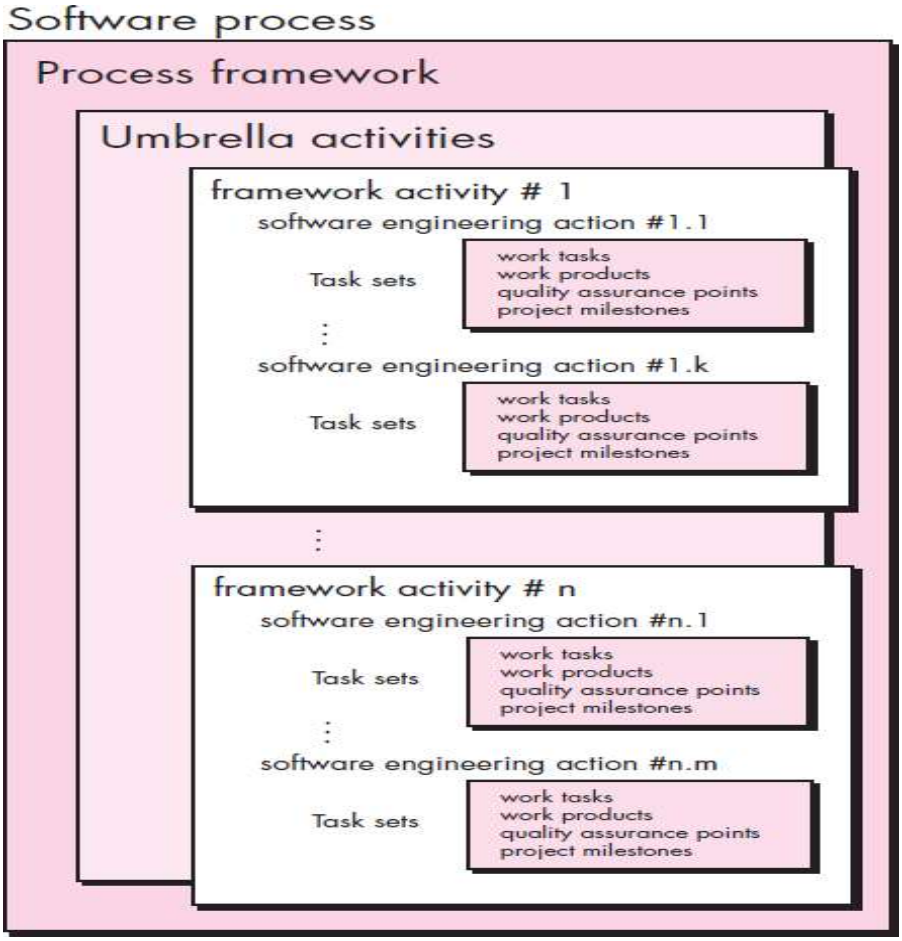
A software process framework is an organized approach that combines processes, methods, and tools to guide software development. It serves as the backbone of any software engineering project.

Definition and Structure:

- A software process framework outlines the essential steps and tasks that must be followed to develop software. These steps are general enough to apply to any software project.
- **Example Steps:**
  - **Gathering Requirements:** Understanding what the software needs to do.
  - **Communication:** Constant interaction with the team and stakeholders.
  - **Delivery:** Final product handover.

3. Detailed Breakdown of Software Process Framework:

The software process framework can be thought of as a structured plan or recipe, ensuring that all necessary activities are covered to produce high-quality software.



Step-by-Step Workflow:

1. **Umbrella Activities:** These are ongoing tasks that run throughout the software development process.
    - **Examples:** Risk management, quality assurance, and configuration management.
2. **Framework Activities:** Major phases that define the main tasks in a software project.
    - **Examples:** Communication, planning, modelling, construction, and deployment.
3. **Task Sets:** Each framework activity is further broken down into smaller, manageable tasks.

- **Example:** During the construction phase, tasks might include coding individual modules and running unit tests.

4. **Engineering Actions:** Specific tasks within the framework, focusing on creating work products, conducting quality checks, and achieving milestones.
    - **Example:** Writing code (engineering action) produces code files (work products), which are then reviewed for quality.
5. **Iteration of Framework Activities:** Each step is repeated as needed, refining the product until it meets the required standards.

III. A GENERIC PROCESS MODEL

The Generic Process Framework is a simplified, universal approach that applies to nearly every software development project. It is often referred to as a "universal recipe" that guides teams through the creation of software.

Key Activities in the Generic Process Framework:

1. **Communication:**
    - Purpose: To interact with customers and stakeholders to gather clear requirements and understand the project scope.
    - **Example:** A team developing an e-commerce website meets with the client to understand which features (like payment options or product filters) are needed.
2. **Planning:**
    - Purpose: To create a roadmap that outlines tasks, schedules, resources, and risks.
    - **Example:** The project manager creates a Gantt chart that details when each task (like coding the login system) should be completed and assigns it to team members.
3. **Modeling:**
    - Purpose: To create models or diagrams that represent how the software will function and be structured.
    - **Example:** Developers create flowcharts to map out user interactions within a mobile app.

4. **Construction:**
    - Purpose: To build the software by writing and testing code.
    - **Example:** The development team codes the main functionalities of the app, like user authentication, and runs unit tests to ensure each part works correctly.
5. **Deployment:**
    - Purpose: To deliver the finished software to the customer, who then uses it and provides feedback.
    - **Example:** After testing, the app is released on the App Store, and users can download and provide feedback on their experience.

5. Umbrella Activities in Detail

These activities support the main framework activities by addressing cross-cutting concerns that affect the entire development process.

Examples of Umbrella Activities:

1. **Risk Management:**
    - Involves identifying potential risks and creating strategies to mitigate them.
    - **Example:** A backup plan is created in case a critical team member is unavailable, or contingency resources are allocated if project delays occur.
2. **Software Quality Assurance (SQA):**
    - Ensures that the software meets predefined quality standards through planned reviews and testing.
    - **Example:** Before releasing a new app version, it undergoes thorough testing on different devices to ensure it functions smoothly across various platforms.
3. **Software Configuration Management (SCM):**
    - Manages changes in the software's configuration to keep track of all modifications and versions.
    - **Example:** Developers use tools like Git to manage code changes, enabling them to revert to previous versions if new changes cause issues.

4. **Measurement:**
    - Involves tracking project metrics like cost, time, and resources to ensure the project remains within budget and schedule.
    - **Example:** Regularly reviewing the time spent on coding and comparing it against the initial estimates to adjust schedules if needed.
5. **Formal Technical Review (FTR):**
    - Regular meetings that assess the project's progress and review technical aspects of the software.
    - **Example:** Weekly team reviews to check progress, identify challenges, and plan the next steps in development.

SOFTWARE ENGINEERING

UNIT – 1

TOPIC – 5

PROCESS MODELS – SDLC, WATERFALL MODEL, INCREMENTAL PROCESS MODEL

1. Introduction to Software Process Models

- **Software Process**:

  A software process refers to the set of activities, methods, practices, and tools used to develop and maintain software. It involves all the necessary steps to create a software product, including planning, designing, coding, testing, and maintenance.

  Example: When building a mobile banking app, the process begins with understanding user needs, designing the app, writing the code, testing the app, and finally launching it.

- **Software Process Model**:

  A software process model is a structured approach or plan that outlines how the software development process will be carried out. It helps organize and manage the steps needed to ensure that all important activities are completed in an orderly manner.

  Example: For the banking app, a process model helps decide whether to use a method like Waterfall or Agile, depending on the project's requirements.

2. Evolution of Software Development: From Frameworks to Process Models

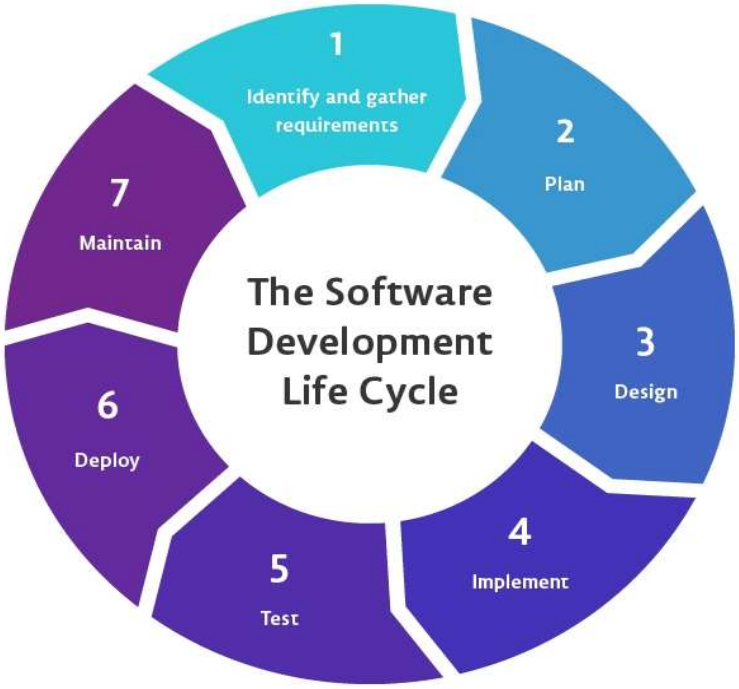
- **Basic Framework**:

  The starting point of any software project involves a basic framework that includes essential steps such as talking to users, planning, designing, building, testing, and releasing the software.

Example: In the banking app, the framework involves gathering user needs, designing the app, coding, testing, and finally launching the app.

- **Software Development Life Cycle (SDLC)**:

SDLC is an evolved form of the basic framework that introduces specific stages with a clear order for execution, making the software development process more organized.



Stages of SDLC:

- **Requirement Gathering**: Understanding what users need.
- **Design**: Planning how the software will function.
- **Implementation**: Writing the code for the software.
- **Testing**: Ensuring the software works correctly.
- **Deployment**: Releasing the software to users.
- **Maintenance**: Updating and fixing the software after release.

Example: The SDLC stages ensure that the banking app is developed in a structured and organized way, from gathering requirements to maintaining the software post-launch.

3. Overview of Process Models

- **Process Models**:

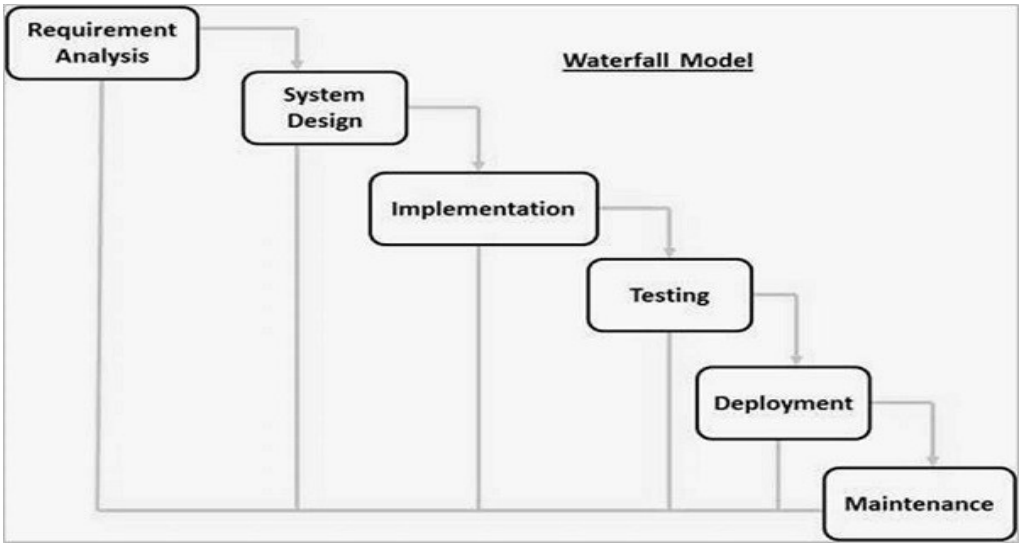
  Process models are like roadmaps that guide the software development process step-by-step. They evolved from the basic SDLC framework to address specific needs of software projects, making the development process easier, more organized, or flexible depending on the project requirements.

  Types of Process Models:

  - Waterfall Model
  - Incremental Process Model
  - Evolutionary Process Models (e.g., Iterative Model, Prototyping Model, Spiral Model, Concurrent Development Model)
  - Agile Model

4. Waterfall Model

- **Concept**:
  The Waterfall Model is a linear and sequential approach where each phase of the development process must be completed before moving on to the next. It is like following a recipe step-by-step where you cannot skip or go back to previous steps.



Steps in Waterfall Model:

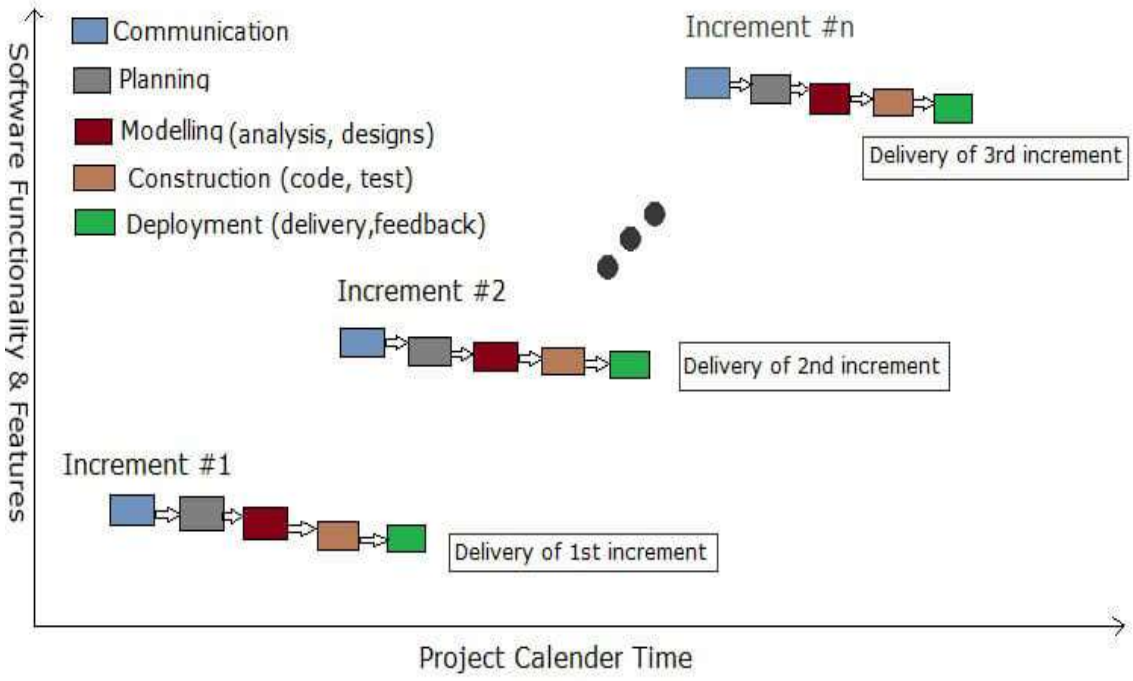
1. **Requirement & Analysis**: Understanding and gathering all requirements.
    - **Example**: Gathering features like account balance checking or money transfers for the banking app.
2. **System Design**: Planning how the software will work, including its structure.
    - **Example**: Designing the layout and data storage for the banking app.
3. **Implementation**: Writing the code to build the software.
    - **Example**: Coding features like login, money transfers, and balance checks in the banking app.
4. **Testing**: Checking the software to find and fix any issues.
    - **Example**: Testing if users can successfully log in and use all features of the banking app.
5. **Deployment**: Releasing the software to users.
    - **Example**: Launching the banking app on app stores.
6. **Maintenance**: Updating and fixing the software post-launch.
    - **Example**: Fixing bugs and releasing updates for the banking app.

- **Advantages**:
    - Simple to understand and easy to manage.
    - Each phase is well-documented and has a clear beginning and end.
    - Works well for smaller projects with clear requirements.
- **Disadvantages**:
    - Rigid and difficult to go back to previous phases.
    - Late changes can be costly.
    - Not flexible and might miss customer needs until the final product is complete.
    - Not suitable for large or complex projects.

5. Incremental Process Model

- **Concept**:

  The Incremental Process Model involves developing software by breaking the project into smaller, manageable parts called increments. Each increment adds a part of the overall system, allowing for step-by-step development and release.



How It Works:

1. **Communication**: Gather and discuss requirements for each increment.
2. **Planning**: Plan the development and integration of each increment.
3. **Modelling**: Design the system components for the current increment.
4. **Construction**: Code and test the increment.
5. **Deployment**: Release the increment to users and collect feedback.

Example: Developing a mobile shopping app:

- **Increment 1**: Release basic features like product browsing and cart functionality.
- **Increment 2**: Add a payment gateway and checkout process.
- **Increment n**: Introduce additional features like order tracking and customer reviews.

- **Advantages**:
  - Early delivery of working software.
  - Easier testing and issue resolution.
  - Flexibility to adjust based on user feedback.
  - Reduced risk as smaller increments minimize the chances of project failure.

- **Disadvantages**:
  - Integration of increments can be complex.
  - Requires detailed planning and coordination for each increment.
  - Initial versions may lack full functionality.
  - Repeated development and testing can increase overall costs.

6. Comparison: Waterfall Model vs. Incremental Process Model

- **Waterfall Model**:
  - Best for projects with a clear, unchanging plan.
  - Follows a strict sequential approach where each phase must be completed before moving to the next.
  - Not flexible and harder to adapt to changes.
- **Incremental Process Model**:
  - Ideal for projects where changes in requirements are expected.
  - Allows for gradual development and improvement of the product.
  - More flexible and better suited for projects that require frequent user feedback and updates.

SOFTWARE ENGINEERING

UNIT – 1

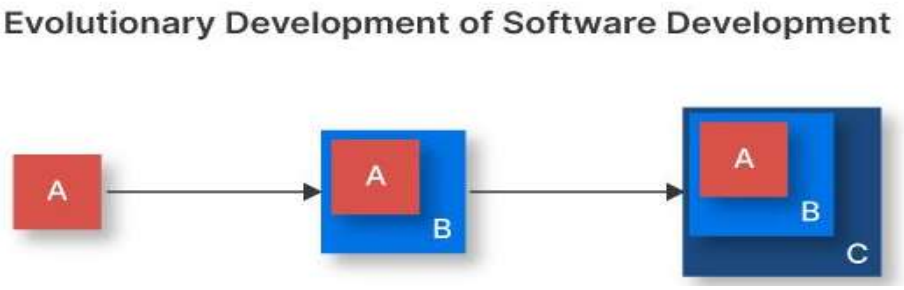
TOPIC – 6

PROCESS MODELS – EVOLUTIONARY PROCESS MODELS

1. Introduction to Evolutionary Process Models

- **What are Evolutionary Process Models?**

Evolutionary process models are flexible software development approaches that allow continuous improvement and adaptation of the software product based on user feedback. Unlike rigid models like Waterfall, evolutionary models embrace change, enabling developers to modify the software at any stage to meet evolving needs.



Example: Imagine you are developing a shopping app. Initially, you release a basic version of the app with essential features like browsing and adding items to a cart. As users provide feedback, you keep updating and improving the app, adding features like payment options and user profiles over time.

2. Iterative Model

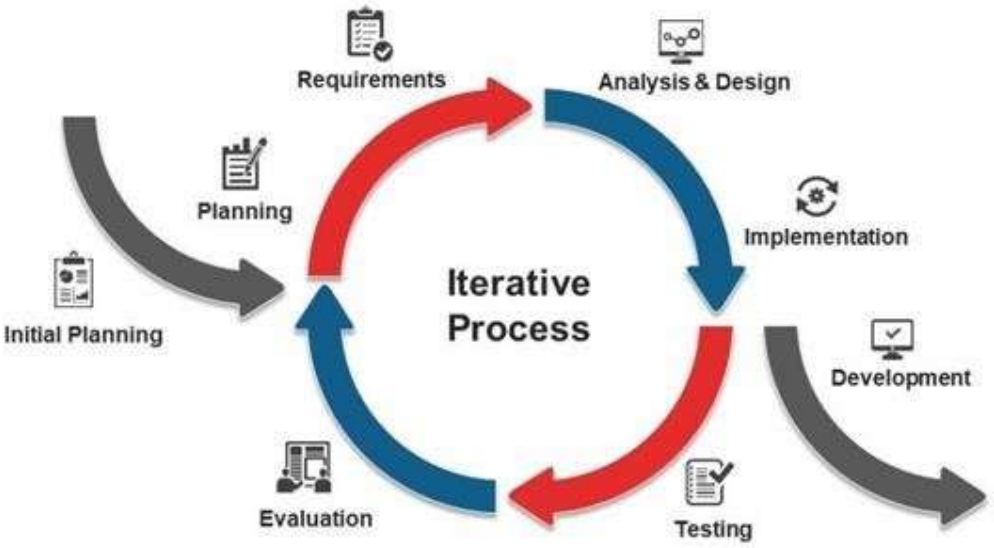
- **Concept**:

The Iterative Model is a development process where the system is built incrementally in small steps, with each iteration refining and improving the product based on user feedback.

Steps in the Iterative Model:

1. **Initial Planning**: Establish the overall goals, initial scope, and objectives for the project.
2. **Requirements**: Gather and define the detailed requirements for the specific iteration.
3. **Analysis & Design**: Create a design plan based on the requirements, focusing on how the system will be built.
4. **Implementation**: Write and integrate the code to develop the software for the current iteration.
5. **Development**: Ensure that the product is built according to the design and includes the necessary functionalities.
6. **Testing**: Validate the software through various tests, gather user feedback, and detect any issues.
7. **Evaluation**: Review the results of testing and user feedback to assess the quality of the iteration.
8. **Refinement (Planning for next iteration)**: Based on feedback, refine the design and plan for improvements in the next iteration.

The process repeats, continuously refining the system through each cycle of iteration until the software reaches its final form.



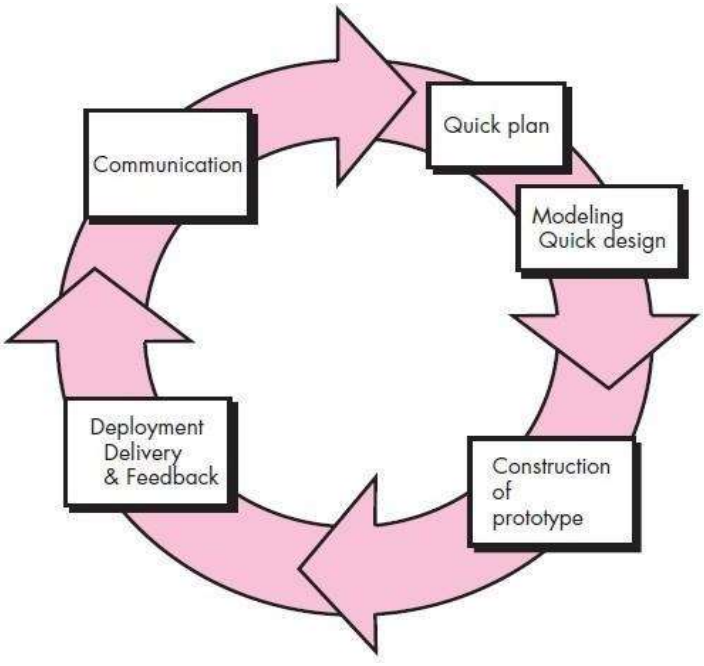
Example: Developing an e-learning platform by first launching basic courses. As time goes on, you add interactive features, more courses, and assessments, improving the platform with each iteration.

- **Advantages:**
  - **Early Problem Detection**: Issues are identified and addressed early in the development process.
  - **Continuous Feedback**: Regular feedback helps refine the product.
  - **Improved Quality**: The product improves with each iteration.
- **Disadvantages:**
  - **Time-Consuming**: Repeated cycles can take more time.
  - **Costly**: Multiple iterations and changes can increase development costs.
- **When to Use:** The Iterative Model is ideal when the project requires repeated refinements and when requirements are not fully understood from the start.

3. Prototype Model

- **Concept:**

  The Prototype Model involves building a simple, working version of the software (a prototype) to let users interact with it and provide feedback. The prototype helps clarify requirements and improve the final product before full-scale development.



Steps in the Prototype Model:

1. **Communication**: Engage with users to discuss and gather requirements for the system.
2. **Quick Plan**: Formulate a simple plan for building the prototype.
3. **Modelling/Quick Design**: Develop a basic model or quick design of the system, focusing on core functionalities.
4. **Construction of Prototype**: Build the prototype, creating an initial, simplified version of the software.
5. **Deployment, Delivery & Feedback**: Deploy the prototype to the users, delivering it for evaluation and collecting their feedback.
6. **Refinement through Communication**: Use the feedback gathered in the previous step to enhance and improve the prototype. This cycle is repeated until the system meets user expectations.

This iterative cycle continues until the final version of the system is developed.

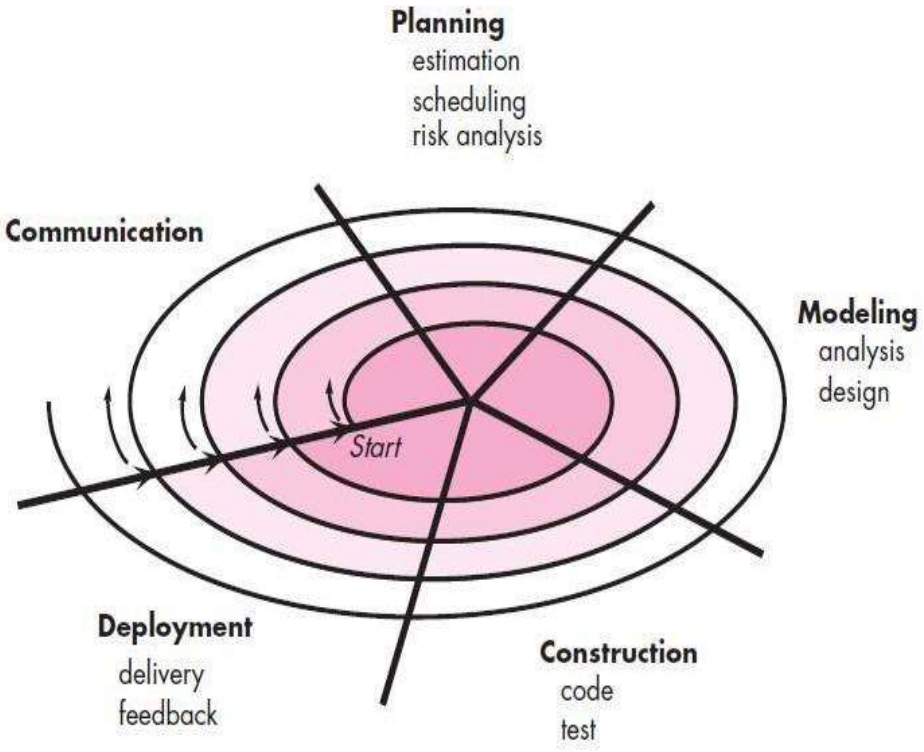
Example: Developing a mobile app for a new business. A simple version is quickly created and shown to users for feedback. Based on their input, the app is refined until it meets the users' needs.

- **Advantages:**
  - **Early Feedback**: Users can see and interact with the system early on.
  - **Flexible**: Easy to make changes during development.
  - **Risk Reduction**: Identifies potential problems or missing features early.
- **Disadvantages:**
  - **Time-Consuming**: Repeated revisions can take a lot of time.
  - **Incomplete System**: Users might confuse the prototype with the final product.
  - **Costly**: Multiple versions of the software can be expensive to develop.
- **When to Use:** The Prototype Model is best when users are unsure of their exact requirements, or when the project involves new or innovative features that need user validation.

4. Spiral Model

- **Concept:**

  The Spiral Model combines iterative development with systematic risk management. It is designed to manage risks by allowing the software to be developed in cycles, with each cycle addressing specific risks and adding new features.



Steps in the Spiral Model:

1. **Communication**: This initial step involves gathering and understanding requirements from users or stakeholders. It's crucial to ensure that everyone is on the same page before proceeding.
2. **Planning**: In this phase, you estimate the project cost, time, and resources. You also conduct risk analysis to identify potential challenges, such as technical difficulties, budget constraints, and market acceptance.
3. **Modelling**: This step focuses on the analysis and design of the system. You create models that represent the structure and behaviour of the system, which helps in visualizing the final product.

4. **Construction**: Here, the actual coding and testing take place. The system or its components are developed, tested, and refined based on the design created in the previous step.
5. **Deployment**: After construction, the system is delivered to the users. Feedback is gathered, which is essential for the next iteration of the spiral, ensuring continuous improvement of the system.

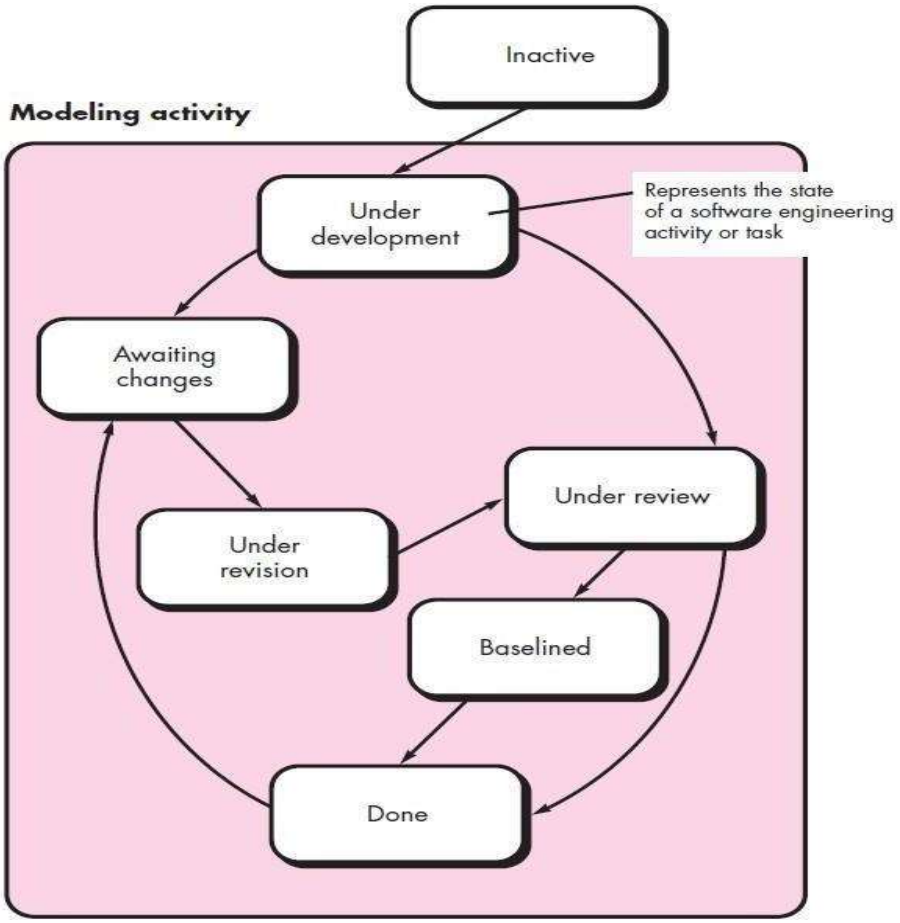
Each cycle of the spiral involves revisiting these steps with increasing detail and refinement, ultimately leading to the completed system.

Example: Developing a healthcare app. In the first cycle, you create basic features like patient registration. In the next cycle, you add appointment scheduling while assessing risks like data security and legal compliance.

- **Advantages:**
  - **Risk-Focused**: Identifies and mitigates risks early in the project.
  - **Flexible**: Allows for changes after each cycle based on feedback.
  - **Early Testing**: Parts of the system are tested after each phase.
- **Disadvantages:**
  - **Time-Consuming**: Multiple cycles can extend the development timeline.
  - **Expensive**: Revisiting phases and managing risks can increase costs.
  - **Complex**: Managing the project can be challenging, especially for large projects.
- **When to Use:** The Spiral Model is best for large, high-risk projects like banking software, healthcare systems, or mission-critical applications where managing risks such as security and timely delivery is crucial.

5. Concurrent Development Model

The Concurrent Development Model, also known as the Concurrent Engineering Model, allows multiple activities in software development to progress simultaneously rather than sequentially. This approach accommodates the natural overlaps in development tasks, enabling different components of the project to be in various stages at the same time.



Steps in the Concurrent Development Model:

- **Parallel Activities:**
    - In this model, tasks such as design, coding, and testing can occur in parallel. For example, while one team works on designing the user interface, another team could be coding the backend system, and yet another team might be testing modules that are already developed.
- **State Progression:**
    - Each task within the project can be in one of several states. The states in the diagram represent the lifecycle of a software engineering activity or task:
        - **Inactive**: The task has not yet started. It is planned, but no active work is being done.

- **Under Development**: The task is actively being worked on, whether it's coding, designing, or another activity.
- **Under Review**: The task is being evaluated for quality, completeness, and alignment with project goals. Reviews can involve code inspections, design walkthroughs, or test result assessments.
- **Awaiting Changes**: After review, the task may require modifications or corrections before it can proceed. It is on hold, waiting for the necessary changes to be made.
- **Under Revision**: The task is currently being revised based on feedback or new requirements. This involves making the necessary updates and corrections.
- **Baselined**: Once a task passes review and is approved, it is baselined. This means it is considered stable and set, serving as a reference point for further development or testing.
- **Done**: The task is completed, fulfilling all requirements and passing all necessary reviews. It is finalized and requires no further work.

Example:

Consider a project to develop a shopping app. The app's user interface (UI) is being designed while simultaneously, the backend payment system is being coded. At the same time, testing might already be underway for the basic functionalities. This parallel progression accelerates development by allowing multiple teams to work on different aspects of the project concurrently.

Advantages:

- **Faster Development**: By allowing multiple activities to occur simultaneously, the overall development process is faster, enabling quicker delivery of the product.
- **Flexibility**: The model's flexibility allows teams to adapt quickly to changes since various tasks are handled in parallel rather than sequentially.
- **Early Problem Detection**: Because different tasks progress simultaneously, issues in one area can be identified and addressed early, without waiting for a linear progression of tasks.

Disadvantages:

- **Complexity**: Managing multiple activities concurrently can be challenging, requiring careful coordination and oversight to ensure that tasks do not interfere with one another.
- **Coordination Required**: Effective communication and coordination between teams are essential to prevent confusion and ensure that all parallel tasks align with the overall project goals.
- **Risk of Overlap Issues**: If not managed properly, changes in one task may negatively impact other parallel tasks, leading to inconsistencies or integration problems.

When to Use:

The Concurrent Development Model is particularly useful for large, complex projects where different teams can work on various parts of the project simultaneously. It's ideal in scenarios where speed is critical, such as in the development of mobile apps, video games, or enterprise software. This model supports a dynamic development environment where tasks can evolve in parallel, adapting to new requirements as they arise.

SOFTWARE ENGINEERING

UNIT – 1

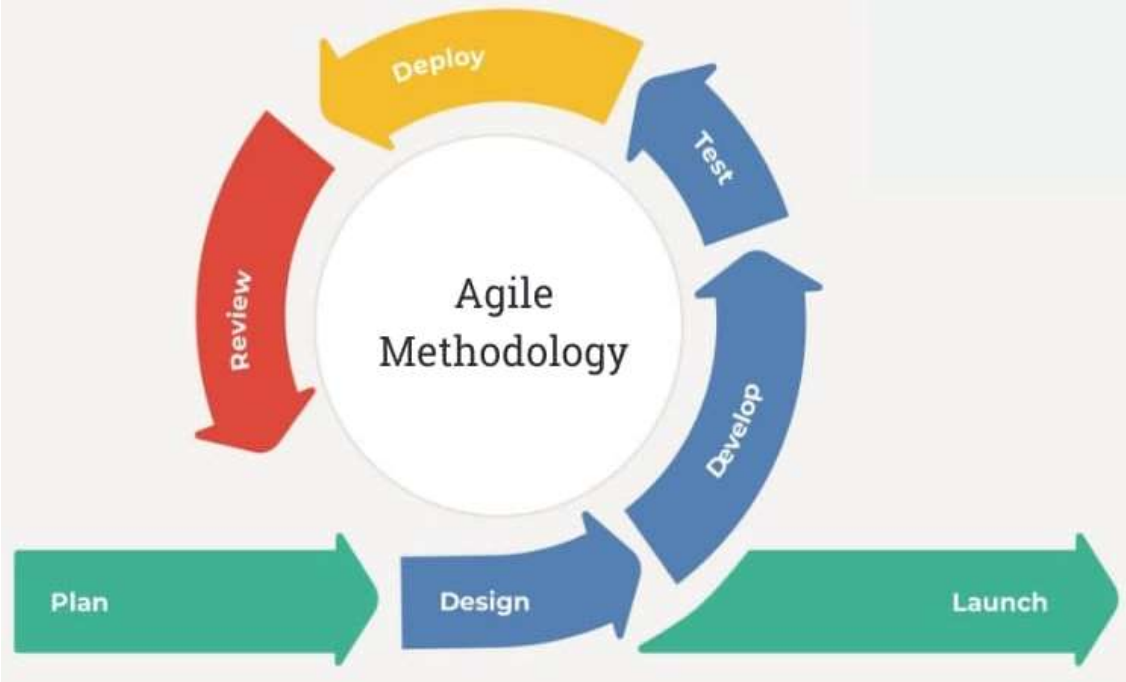
TOPIC – 7

INTRODUCTION TO AGILE

1. What is Agile?

Agile is a flexible approach to developing software. Unlike traditional methods where everything is planned in advance, Agile allows teams to adapt and make changes as they go.

Example: If you're building a mobile app, instead of planning everything from the start, you would release a basic version. Then, based on user feedback, you keep improving the app over time.



2. Agile Manifesto

The **Agile Manifesto** is a set of values and principles created in 2001 that guides how software should be developed. The Agile Manifesto consists of **64 words** that define the key principles of Agile software development.

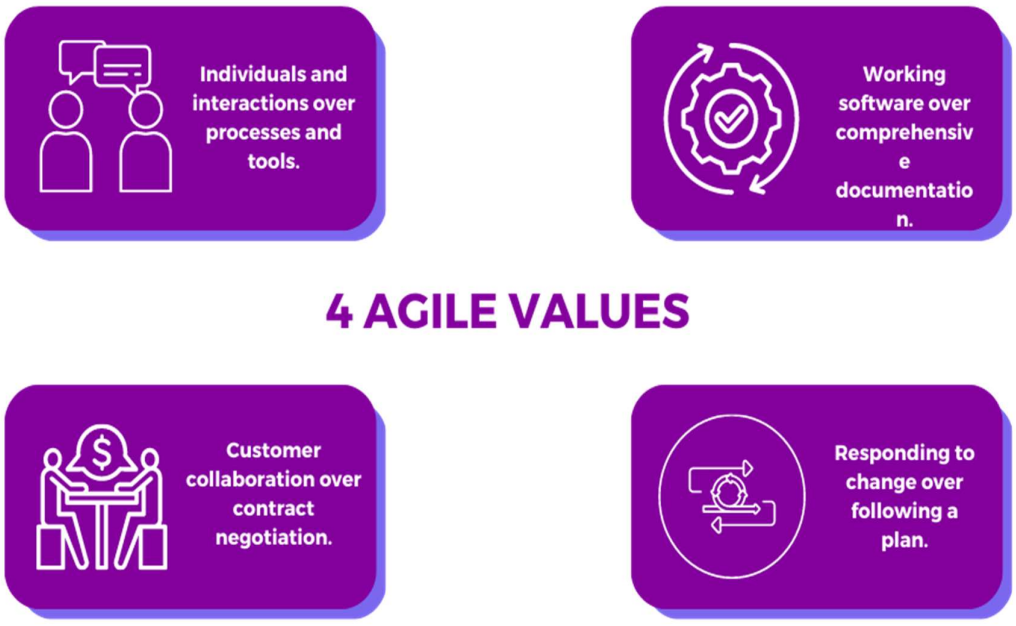
The **64 words** in the Agile Manifesto are the foundation of Agile methodology. Here they are:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions over processes and tools**
- **Working software over comprehensive documentation**
- **Customer collaboration over contract negotiation**
- **Responding to change over following a plan**

That is, while there is value in the items on the right, we value the items on the left more."

Four Key Values:



1. **Individuals and Interactions over Processes and Tools**
   - Agile prioritizes direct communication among team members rather than rigidly following processes.
   - **Example:** A developer can quickly ask another team member for clarification instead of waiting for formal documentation.

2. **Working Software over Comprehensive Documentation**
   - Focus on getting the software working rather than spending months writing detailed plans.
   - **Example:** Instead of writing a long report about how a feature should work, build the feature and let the users test it.
3. **Customer Collaboration over Contract Negotiation**
   - Work closely with the customer to understand their changing needs rather than sticking strictly to the initial contract.
   - **Example:** If the customer decides they need a new feature halfway through the project, you can adjust the work accordingly.
4. **Responding to Change over Following a Plan**
   - Be open to making changes, even if it means altering the original plan.
   - **Example:** If a new technology comes up that can improve the project, Agile allows you to adopt it, even if it wasn't planned from the start.

12 Agile Principles

These principles provide more specific guidance on how to implement the Agile values:

1. **Customer satisfaction** is the highest priority, achieved through early and continuous delivery of software.
   - **Example:** Release new features or updates every few weeks, so the customer always has something new to try.
2. **Welcome changing requirements**, even late in development.
   - **Example:** If the customer wants a change in the middle of development, it's welcomed rather than avoided.
3. **Deliver working software frequently**.
   - **Example:** Instead of waiting months, you release working software every two weeks (a cycle known as a "sprint").
4. **Daily collaboration** between business people and developers.
   - **Example:** The product owner (representing the customer) talks to the developers every day to make sure everyone is on the same page.
5. **Build projects around motivated individuals** and give them the freedom to make decisions.
   - **Example:** The development team is trusted to make technical decisions without waiting for approval from higher management.
6. **Face-to-face communication** is the best way to convey information.
   - **Example:** Instead of sending emails or creating documents, teams should hold short daily meetings to discuss progress and issues.
7. **Working software** is the primary measure of progress.
   - **Example:** The project is judged based on how much of the software is actually usable, rather than how much documentation or planning has been done.
8. **Sustainable development** means the team should be able to maintain a constant pace without burnout.
   - **Example:** Teams avoid working overtime constantly to ensure steady progress over time.
9. **Continuous attention to technical excellence** and good design enhances agility.
   - **Example:** Developers regularly refactor the code to keep it clean and maintainable, which makes future changes easier.
10. **Simplicity**, or the art of maximizing the amount of work not done, is essential.
    - **Example:** Focus only on building features that provide value, rather than adding unnecessary details.
11. **Self-organizing teams** produce the best designs and architectures.

- **Example:** Instead of following a strict hierarchy, the team decides the best way to achieve the project goals together.

12. **Regular reflection** on how to become more effective.
    - **Example:** After each sprint, the team holds a "retrospective" to discuss what went well and what could be improved.

3. Advantages of Agile

1. **Increased Customer Satisfaction**
    - Continuous feedback and updates keep the customer happy.
    - **Example:** The customer feels involved as they can see regular progress and suggest changes.
2. **Faster Delivery of Software**
    - Short development cycles mean features are delivered faster.
    - **Example:** Customers receive working software every few weeks, rather than waiting for months.
3. **Flexibility to Change**
    - Agile welcomes changes, even in later stages.
    - **Example:** If a new market trend or technology emerges, Agile allows the team to adjust their plans.
4. **Improved Product Quality**
    - Constant testing and feedback result in a more refined product.
    - **Example:** The product is tested at the end of each sprint, catching issues early.
5. **Enhanced Team Collaboration**
    - Agile promotes regular communication and teamwork.
    - **Example:** Daily meetings help everyone stay on the same page, improving overall team dynamics.
6. **Predictable Costs and Schedule**
    - Agile's regular sprints help teams manage time and budget more effectively.
    - **Example:** With fixed timeframes, it's easier to predict the costs and when a feature will be delivered.
7. **Early Risk Identification and Mitigation**
    - Problems are identified early due to frequent testing.

- **Example:** By releasing small updates, teams can spot potential issues early and adjust the project accordingly.

4. Disadvantages of Agile

1. **Not Suitable for Highly Complex Projects**

- Agile may struggle when strict planning and documentation are required.

2. **Requires Strong Leadership**

- A skilled leader is necessary to ensure the project stays on track.

3. **Tight Deadlines Can Be Challenging**

- Meeting frequent deadlines might result in rushed decisions or sacrifices in quality.

4. **Lack of Documentation**

- Since Agile relies on communication over documentation, it can be harder to onboard new team members or refer back to old decisions.

5. Agile Frameworks

Agile isn't a specific method but rather a collection of frameworks. Some popular ones include:

1. **Scrum**: A framework that uses iterative development with time-boxed iterations called sprints, typically lasting 2-4 weeks. It focuses on delivering small, incremental improvements.
2. **Kanban**: A visual management method that uses a board with columns to represent different stages of work. It emphasizes continuous delivery and improvement by managing work in progress.
3. **Extreme Programming (XP)**: A framework focused on technical excellence and customer satisfaction. It emphasizes practices like continuous integration, test-driven development, and pair programming.
4. **Lean Software Development**: Inspired by lean manufacturing principles, this framework focuses on eliminating waste, improving flow, and maximizing value delivered to the customer.

5. **Feature-Driven Development (FDD)**: A model-driven, short-iteration process that focuses on designing and building features. It emphasizes creating a feature list and then delivering those features incrementally.
6. **Crystal**: A family of methodologies tailored to different team sizes and project complexities. It emphasizes flexibility and the use of frequent delivery and reflective improvement.
7. **Dynamic Systems Development Method (DSDM)**: A framework that provides a structured approach to project management, focusing on delivering business value through iterative development and active user involvement.
8. **Agile Unified Process (AUP)**: A simplified version of the Rational Unified Process (RUP), AUP integrates Agile practices into the traditional Unified Process framework, focusing on iterative development and continuous improvement.
9. **Scaled Agile Framework (SAFe)**: A framework designed to scale Agile practices to larger organizations or projects. It includes principles and practices for aligning teams, programs, and portfolios.
10. **Large Scale Scrum (LeSS)**: An extension of Scrum for scaling Agile across multiple teams, maintaining the simplicity and principles of Scrum while addressing coordination and integration across teams.

    Each framework has its own unique approach and practices, so the choice often depends on the specific needs and context of the organization or project.

SOFTWARE ENGINEERING

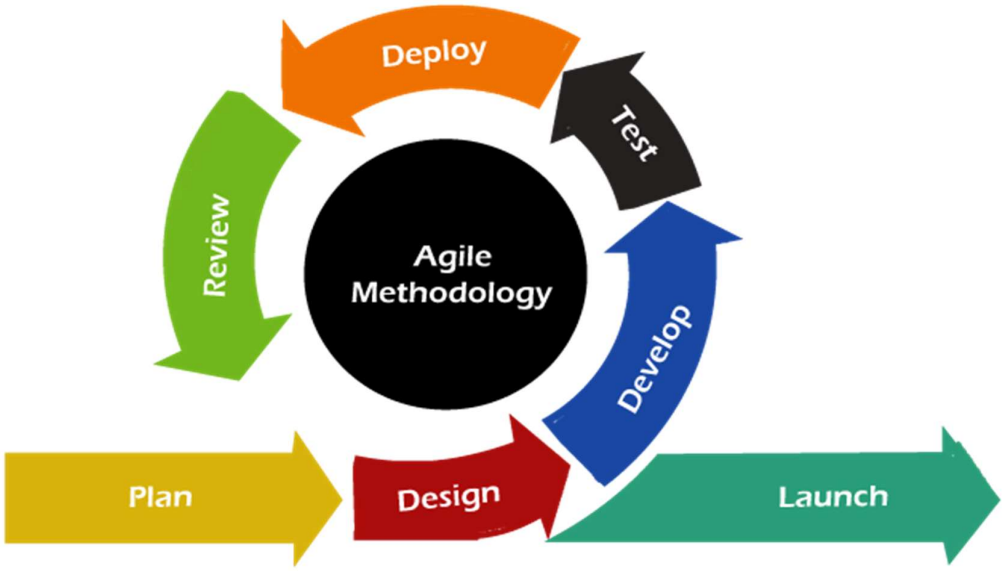
UNIT – 1

TOPIC – 8

SCRUM FRAMEWORK AND INTRODUCTION TO DEVOPS

I. Various Agile Frameworks

Agile is an approach to software development where work is divided into small, manageable parts called frameworks. These frameworks help teams deliver projects in a flexible and adaptive manner.



Some popular Agile frameworks are:

a) Rational Unified Process (RUP): Focuses on repeated cycles of development.
b) Adaptive Software Development (ASD): Emphasizes continuous learning and adaptation.
c) Feature Driven Development (FDD): Centers on building the project based on specific features.
d) Crystal Clear: Prioritizes simplicity and frequent delivery.
e) Dynamic Systems Development Method (DSDM): Offers a structured approach to rapid software development.
f) Extreme Programming (XP): Promotes continuous testing and customer feedback.
g) Scrum: A popular, team-based method that breaks work into short, manageable chunks.

II. SCRUM Framework

What is SCRUM?

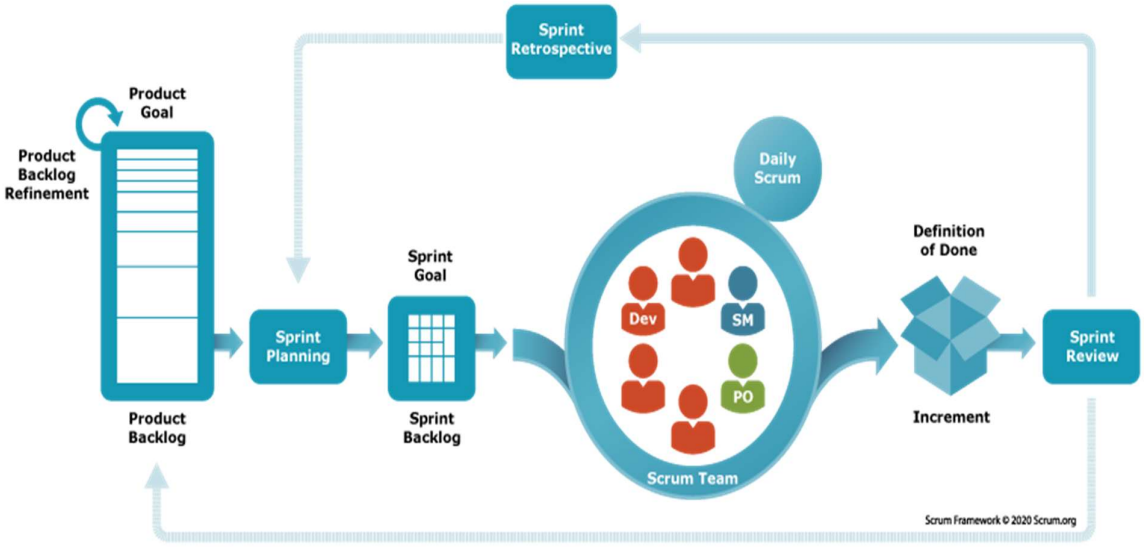
Scrum is a way to manage and organize the process of developing software. It's part of the Agile family, which means it helps teams stay flexible and open to changes. The main idea behind Scrum is to break big tasks into smaller, manageable pieces and complete them in Sprints—a set period of time (usually 2 weeks). Scrum encourages continuous improvement and frequent communication to make sure the team is always on track.

History of Scrum

Scrum was created in the early 1990s by two developers named Ken Schwaber and Jeff Sutherland. They wanted to help teams work together better and deliver software faster. They were inspired by how teams in Rugby work together in small groups to achieve their goals, so they used the word "Scrum" from Rugby.

In 1995, they officially introduced the Scrum Framework at a conference, and since then, it has become very popular. Today, Scrum is used not only in software development but also in other industries because it helps teams work efficiently and deliver results quickly.

Key Roles in Scrum:



Product Owner:

The Product Owner is like the manager of the product. They understand what the customer needs and create a list of tasks, called the Product Backlog. The Product Backlog contains

everything that needs to be done to create the final product, and it's the Product Owner's job to prioritize these tasks.

Scrum Master:

The Scrum Master acts as a coach for the team. They make sure the Scrum process is being followed and help the team stay organized. The Scrum Master removes any obstacles that might slow the team down, like solving problems and setting up meetings.

Scrum/Development Team:

This is the group of developers who do the actual work of building the product. They handle the technical tasks and work on the items from the Product Backlog.

Scrum Artifacts

Artifacts in Scrum are tools or documents that help keep the project organized.

Product Backlog:

This is a to-do list for the whole project. Created and managed by the Product Owner, it contains all the tasks, features, and improvements needed for the final product. The Product Backlog is constantly updated with new ideas, and tasks are prioritized based on what's most important.

Sprint Backlog:

A smaller, focused to-do list for each Sprint. The team picks tasks from the Product Backlog that they can finish during the Sprint (2 weeks or so). This list becomes the team's main focus for that Sprint.

Increment:

The Increment is the collection of all the work that has been completed in the current Sprint, plus everything from past Sprints. After each Sprint, the team should have a working version of the product that includes all the new tasks they finished. It's like reaching a small milestone.

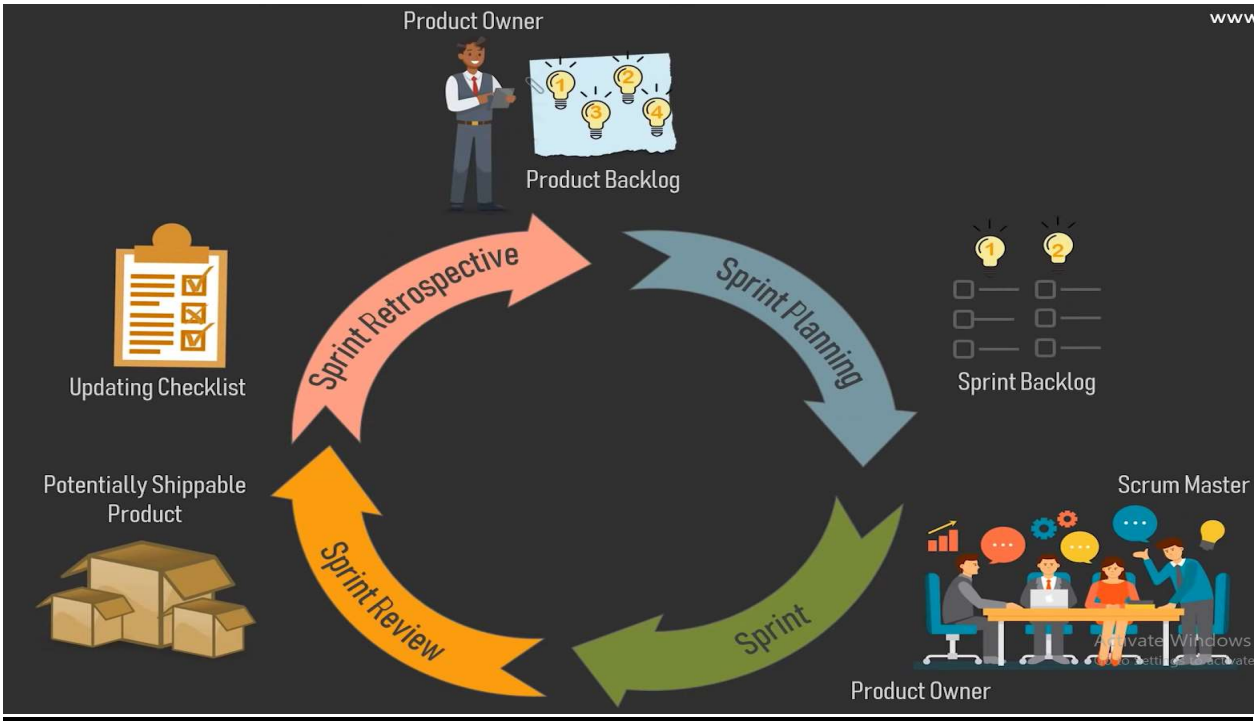
Burn-Down Chart:

This is a visual chart that tracks how much work is left compared to the time remaining in the Sprint. It helps the team see if they're on track to finish on time, and if they're falling behind, they can make changes to speed up. Ideally, the line on the chart should go downwards, showing that tasks are being completed.

Scrum Board (Task Board):

A visual tool that shows the status of tasks during the Sprint. Tasks are organized in columns like "To Do," "In Progress," and "Done." As tasks get completed, they are moved across the board. This helps everyone see what work has been done and what still needs to be finished.

Main Elements in Scrum:



- **Product Backlog:** A list of everything that needs to be done for the project, organized by the Product Owner in order of priority.
- **Sprint Backlog:** A smaller list of tasks chosen from the Product Backlog that the team will work on during the current Sprint.
- **Daily Scrum:** A short 15-minute meeting held every day. In this meeting, the team discusses:
  - What they worked on yesterday.
  - What they will work on today.
  - Any problems they are facing.
  - This keeps the team aligned and focused.

- **Sprint Review:** At the end of each Sprint, the team presents what they have completed to the Product Owner and other stakeholders. Feedback is gathered, and changes may be made for the next Sprint.
- **Sprint Retrospective:** After the Sprint Review, the team reflects on how the Sprint went and discusses what went well and what could be improved for the next Sprint.

Advantages of Scrum:

• **Quick Progress:** By breaking tasks into smaller parts, work is done faster, and parts of the product are ready quickly.

• **Flexibility:** Scrum allows for changes after every Sprint, so the team can make adjustments based on customer feedback.

• **Team Collaboration:** Everyone works together closely, which leads to better communication and problem-solving.

• **Continuous Improvement:** After each Sprint, the team reflects on what went well and what didn't. This helps them improve for the next Sprint.

Disadvantages of Scrum:

- **Confusing Roles:** Sometimes, team members might be unsure about their responsibilities, especially if they're new to Scrum.
- **Difficult to Manage Changes During a Sprint**: If a customer requests big changes in the middle of a Sprint, it can disrupt the team's workflow.
- **Requires Discipline:** Scrum needs the team to be very organized and disciplined in keeping up with daily meetings and task updates.
- **Not Ideal for Large Projects:** Scrum might not work well for extremely large projects, as it may not offer enough structure to handle everything.
- **Dependency on the Product Owner:** If the Product Owner is unavailable or slow to make decisions, the whole team can be delayed because they rely on the Product Owner to prioritize tasks.

Agile vs DevOps

- Agile is a software development approach that has been around for a few decades. It focuses on fast development cycles, ensuring that software is built and tested quickly.
- DevOps, introduced in the mid-2000s, also supports fast software development, but it goes beyond development. It includes continuous operations, meaning the software is constantly monitored to ensure it works smoothly with no downtime.

Differences Between Agile and DevOps

- Agile stops after development, testing, and deployment. There is no ongoing monitoring of the software.
- In DevOps, continuous monitoring ensures the software is always operational, with no issues for users.
- Agile typically has separate teams for development, testing, and operations, while DevOps merges all roles, making each individual responsible for all phases of the software life cycle.

Introduction to DevOps

The evolution of software development by comparing three approaches: Waterfall, Agile, and DevOps:

1. Waterfall Model (Traditional)

Best when all the requirements for the project are clear and won't change. The product is well-defined and stable from the start. Waterfall is a step-by-step method. You finish one step before moving to the next (like a waterfall flowing downward).

2. Agile Development (Modern Approach)

Best when the project's needs or requirements might change often. You need to develop the product quickly. Agile is flexible and works in short cycles, allowing for quick changes and improvements.

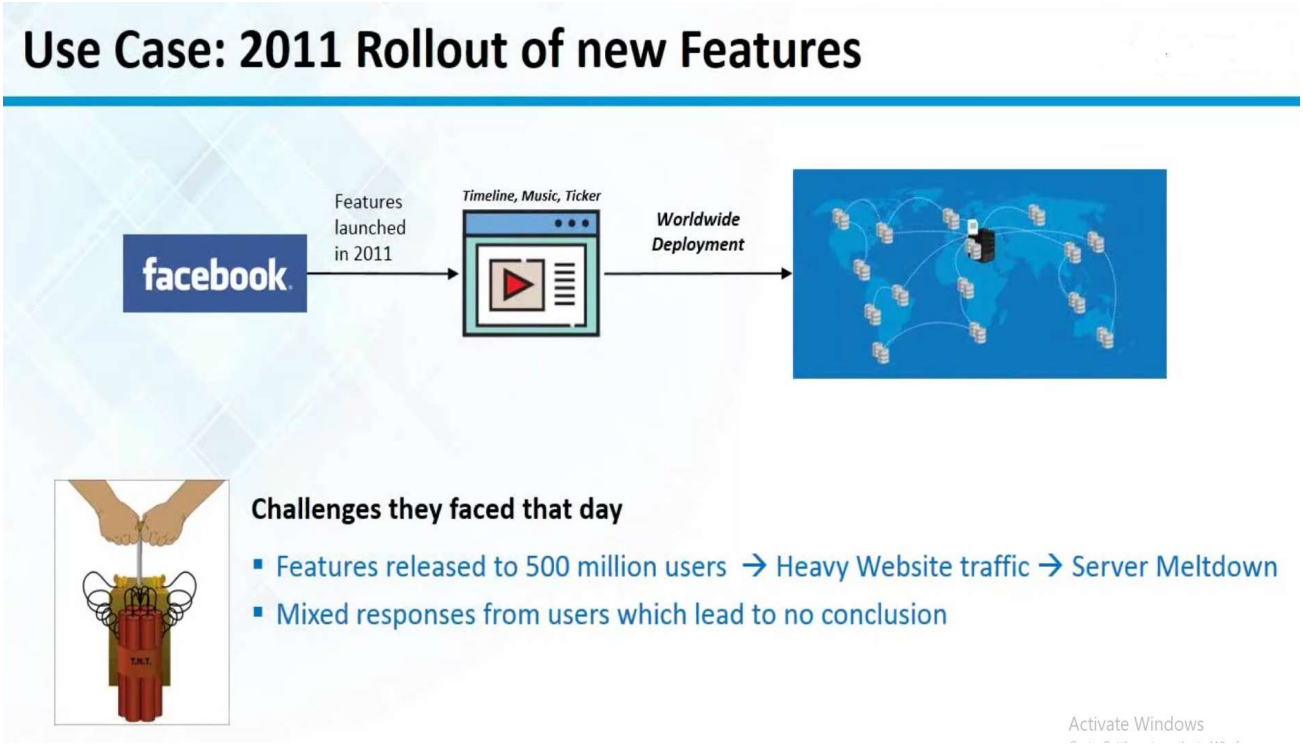
3. DevOps Approach (Latest Trend)

Best when requirements are still changing frequently, just like in Agile. Both the development and operations (maintenance and monitoring) need to happen quickly and smoothly. DevOps combines development and operations teams to ensure faster development and continuous monitoring of the product to keep it running without issues.

Use case

Facebook's 2011 Feature Rollout and How DevOps Helped

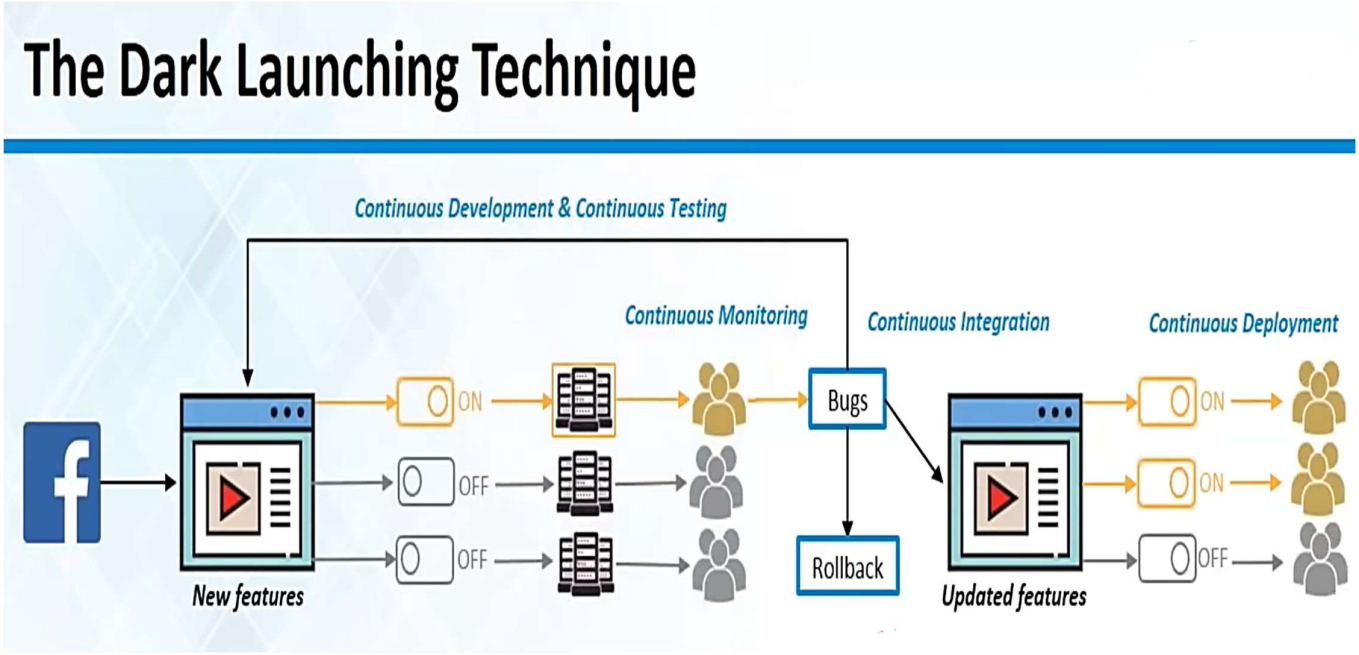
In 2011, Facebook introduced new features like **Timeline**, **Music**, and **Ticker**. They launched these features to **500 million users** all at once. However, the sudden increase in traffic was too much for Facebook's **servers**, causing a **server meltdown** where parts of the site stopped working. Users also gave **mixed feedback**, making it hard for Facebook to understand what to improve.



How DevOps Helped Fix These Issues

To solve these problems, Facebook used **DevOps**. DevOps brings together the teams that develop new features and the teams that manage the servers. This helped Facebook **fix the server problems quickly** by working together closely.

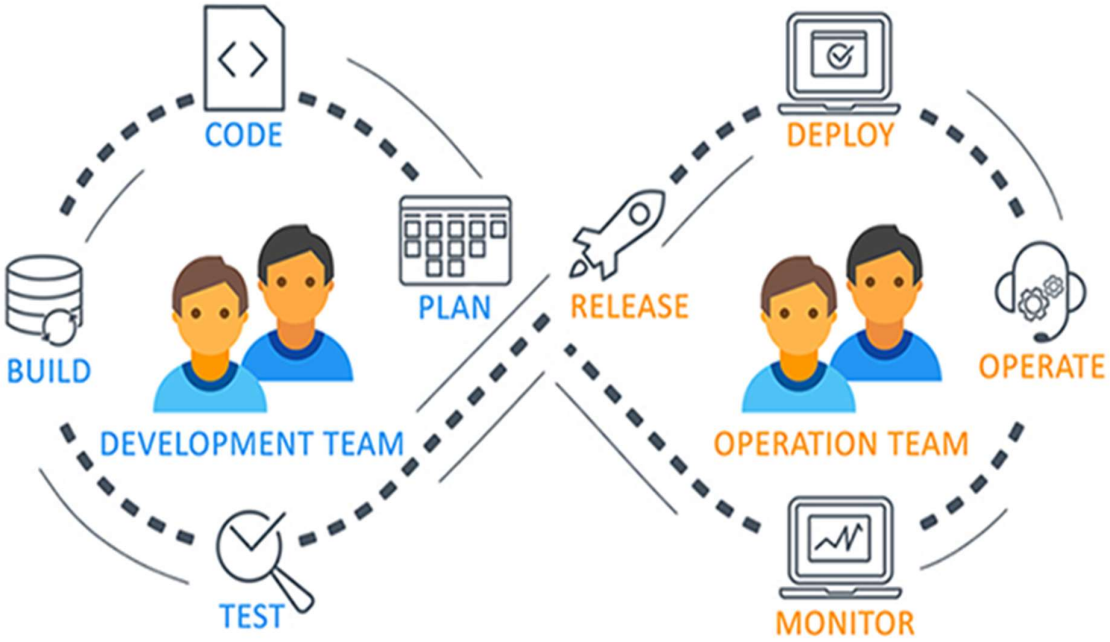
Facebook also used a technique called **dark launching**, which is part of the DevOps approach. Instead of giving the new features to everyone at once, they tested them with a **small group of users** first. This helped Facebook see how the servers handled the traffic and understand what users thought before releasing the features to everyone.



Using **DevOps** and **dark launching**, Facebook was able to avoid further big problems, gather better feedback, and make improvements without causing the website to crash again.

What is DevOps?

DevOps is a methodology that combines software development (Dev) and IT operations (Ops). Its main goal is to shorten the systems development lifecycle while delivering high-quality software. DevOps relies on practices like continuous development, integration, and monitoring to streamline the production process.



Key Features:

- **Continuous Development**: Ongoing and iterative updates to software code.
- **Continuous Testing**: Automated testing at every stage to catch bugs early.
- **Continuous Integration**: Regularly merging code changes to avoid conflicts.
- **Continuous Deployment**: Automating the release of software updates.
- **Continuous Monitoring**: Tracking performance and issues post-deployment.

DevOps Lifecycle

Description:

The DevOps lifecycle is a continuous, iterative process that links various stages of software development and operations. The goal is to create a seamless flow between coding, building, testing, deploying, and monitoring.

Key Phases:

1. **Plan**: Define the software objectives, features, and roadmap.
2. **Code**: Write the actual program according to the plan.
3. **Build**: Convert the code into executable software.
4. **Test**: Automate tests to ensure that the code works as expected.

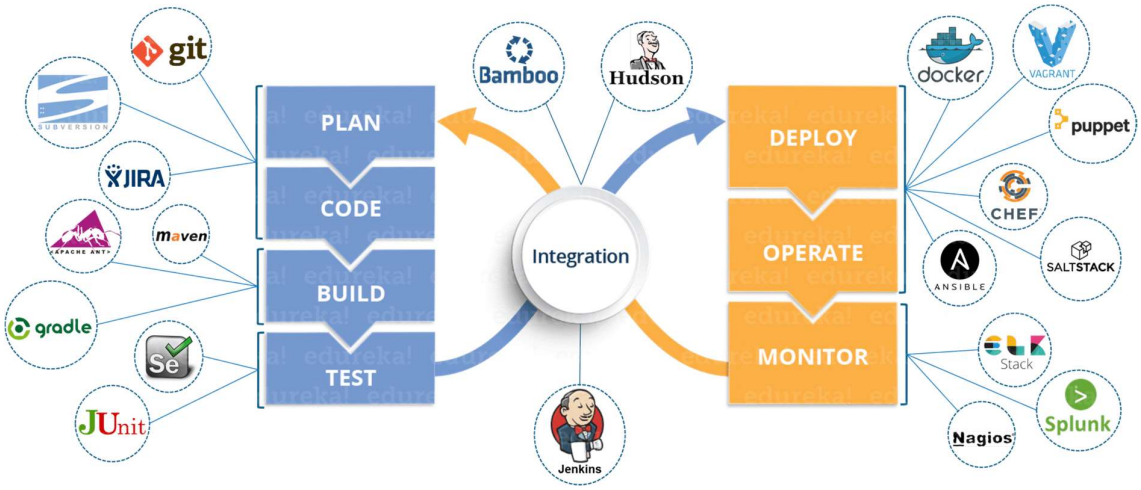
5. **Deploy**: Move the application to a production environment for users.
6. **Operate**: Ensure the software runs efficiently with minimal downtime.
7. **Monitor**: Continuously observe the system for performance and issues.

Integration: Throughout this lifecycle, integration ensures that each phase communicates smoothly, allowing for faster iterations and improvements.

DevOps Tools

Description:

DevOps tools automate processes across the lifecycle, ensuring that each step from coding to deployment is quick and efficient. These tools enhance collaboration between teams and ensure continuous workflows.



Key Tools:

- **Version Control Tools (Git, Subversion)**: Manage code versions and collaborate across teams.
- **Build Tools (Maven, Apache Ant)**: Automate the building and compilation of code.
- **Planning Tools (JIRA)**: Help manage tasks and track project progress.
- **Continuous Integration Tools (Jenkins, Bamboo, Hudson)**: Automate code integration and build processes.
- **Deployment Tools (Docker, Vagrant, Kubernetes)**: Manage containerized applications and deploy updates.
- **Monitoring Tools (Nagios, Splunk, ELK Stack)**: Track system performance and detect issues in real-time.

Each tool specializes in automating a specific phase of the DevOps lifecycle, ensuring smoother, faster, and more reliable workflows.

SOFTWARE ENGINEERING

UNIT – 1

TOPIC – 9

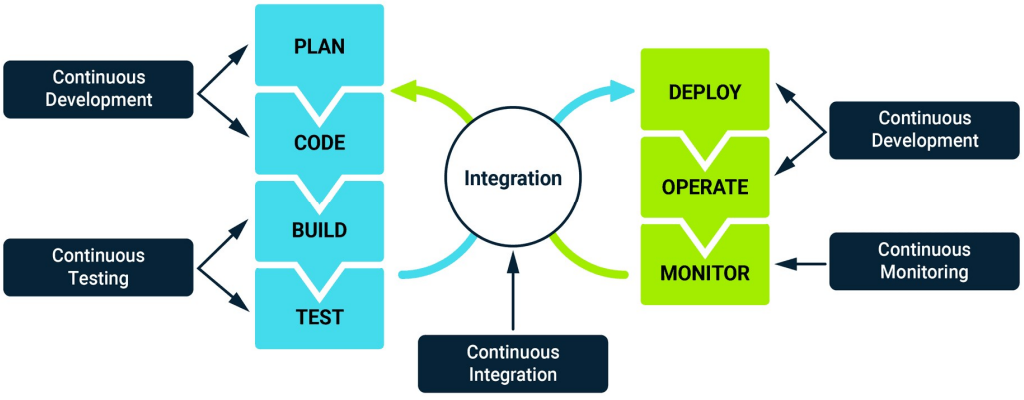
DEVOPS LIFECYCLE

I. Introduction to DevOps

DevOps is a combination of "Development" and "Operations". It is a set of practices that aims to automate and integrate the processes of software development (coding, testing) and IT operations (deployment, monitoring) to enhance collaboration and productivity between these teams.

The DevOps lifecycle consists of several key phases that are continuously repeated until the desired quality is achieved.

II. Phases of the DevOps Lifecycle:



Planning: In this phase, teams discuss what needs to be built and how it will meet the user's needs. Communication between team members is crucial here.

Key Point: There aren't specific tools used in planning, but clear communication is the main requirement.

Coding: Developers write the code based on the project's requirements. There are various tools to manage this code.

Tools Used: Version control tools: These helps keep track of code changes over time (**e.g., Git, SVN**).

Building: The code written by the developers is transformed into software that can be run. This is known as a build.

Tools Used: Build tools: Examples include **Ant, Maven, and Gradle**.

Integration: Developers continuously integrate their work, ensuring that the new code works well with what has already been built.

Tools Used: **Jenkins**: It automatically takes the latest code and compiles it into a build.

Testing: Automated tests check for issues in the software. If bugs are found, they are reported, fixed, and tested again.

Tools Used: Testing tools like **Selenium, TestNG, and JUnit**.

Deployment: The software is moved from a development or testing environment to production (where users can interact with it).

Tools Used: Configuration management tools: These ensure the software runs consistently across different servers (e.g., **Puppet, Chef, Ansible**). Containerization tools: Tools like **Docker** ensure the software behaves the same way across all environments.

Operation: Once deployed, the software is operational, and the team monitors it to ensure smooth functioning.

Monitoring: Monitoring tools check if the software is performing well and detect any issues. If problems arise, they are fed back to the development team for fixing.

Tools Used: Examples include **Splunk, Nagios, and New Relic**.

Continuous Loops in DevOps:

Continuous Development: The process of planning, coding, building, and testing keeps going in cycles. This ensures that new features are always being added and improved.

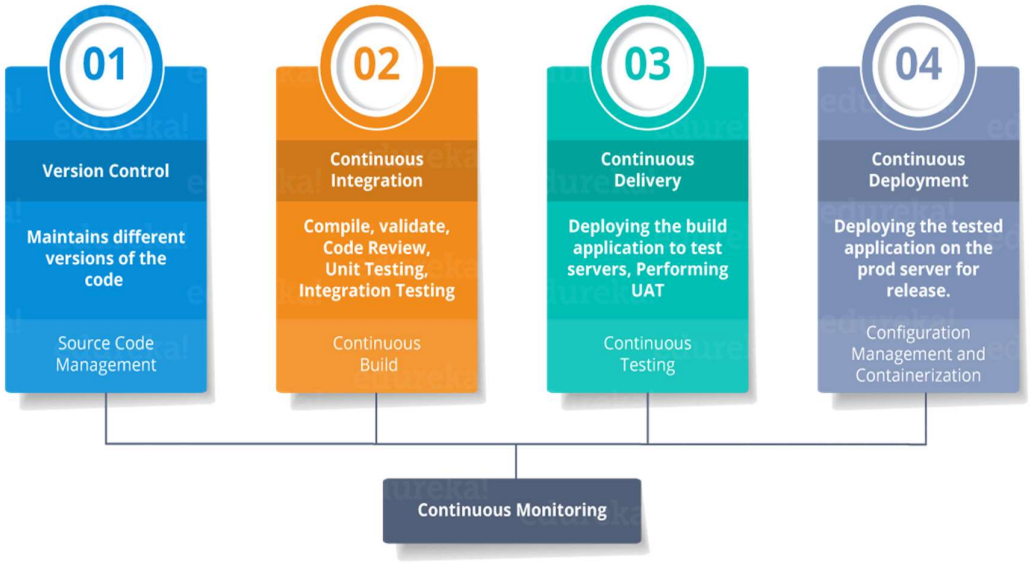
Continuous Testing: Regular testing ensures that bugs are caught early, before they reach the users.

Continuous Integration: This means merging code changes frequently to ensure everything works together smoothly.

Continuous Deployment: The software is continuously deployed to the live environment, so users always have the latest version.

Continuous Monitoring: After deployment, the software is constantly monitored to track its performance and identify any issues.

III. DevOps Stages



The stages of DevOps — Version Control, Continuous Integration, Continuous Delivery, and Continuous Deployment — ensure that software development is more collaborative, automated, and efficient. Each stage builds upon the previous one, leading to faster software releases with fewer errors. Continuous monitoring is an essential part of the process, providing real-time feedback that helps improve the software continuously. By integrating these stages, organizations can release high-quality software at a much faster pace, adapting quickly to user needs and market changes.

Stage 1: Version Control Version control is the backbone of DevOps practices. It helps teams track and manage changes to the codebase. Every time developers make changes, these are saved in different versions, so teams can easily go back to previous versions if needed. This

system is essential when multiple people work on the same project, as it allows them to collaborate without overwriting each other's work.

- Tools Used: Popular version control tools include **Git, SVN, and Mercurial**. These tools keep the source code organized, allowing teams to review previous changes, identify issues, and roll back to older versions if necessary.

Stage 2: Continuous Integration (CI) Continuous Integration (CI) involves frequently merging all the code changes from developers into the main codebase. It ensures that new code works seamlessly with the already existing code. This process involves several steps like compiling the code, running unit tests, and performing integration tests to validate the new changes. This phase ensures that bugs and integration issues are detected early. Continuous Integration automates this process, ensuring the software build is always up-to-date and stable.

- Tools Used: **Jenkins, Travis CI, and CircleCI** are examples of tools that help automate CI by constantly integrating, testing, and building the software.

Stage 3: Continuous Delivery (CD) Once the code passes through integration and testing, the next phase is Continuous Delivery. In this stage, the application is automatically delivered to a testing or pre-production environment, where it is further tested before being released to users. This ensures that every change made to the software can be automatically prepared for a release, which means the code is always ready for deployment at any time. The build application is deployed to test servers, and user acceptance testing (UAT) is performed to ensure that the product is ready for production.

- Tools Used: Tools like **Jenkins, GitLab, and Bamboo** help automate delivery processes, allowing smoother transitions between stages.

Stage 4: Continuous Deployment Continuous Deployment takes things a step further by automatically releasing the tested software to the production environment. This stage involves releasing the application into real-world environments where users can interact with it. In this phase, configuration management and containerization play a big role in ensuring the software works across various environments without manual intervention. This reduces human error and speeds up the time to market.

- Tools Used: **Docker, Kubernetes, Ansible, and Chef** are tools that handle configuration management and containerization, ensuring the application behaves the same across all environments.

Continuous Monitoring (Across All Stages) Continuous monitoring is an ongoing activity that runs across all DevOps stages. This phase ensures that once the software is deployed, its performance is constantly tracked. Monitoring tools help identify any potential issues such as server crashes, slow performance, or security vulnerabilities. If problems are detected, feedback is sent back to the development team, and the process starts again.

- Tools Used: Popular tools for continuous monitoring include **Nagios, Splunk, New Relic, and Prometheus.**

IV. DevOps Phases

DevOps is an ongoing process that integrates development and operations to ensure continuous delivery of high-quality software. By focusing on collaboration, automation, and feedback, the DevOps approach allows teams to build, test, and release applications faster and more reliably.

The key phases in the DevOps process:

1. Plan

Planning is the first and most critical step. It involves deciding what kind of application needs to be built, identifying its goals, and outlining the steps required to achieve those goals. You gather all the requirements from stakeholders and create a detailed plan or roadmap.

In this phase, you figure out what you're going to build, why it's needed, and how to go about building it.

2. Code

This is the phase where the actual development happens. Developers write the code for the application based on the plan. The code must align with user needs and business objectives, and collaboration among team members is essential.

This is where you write the actual instructions (code) that make the application work.

3. Build

Once the code is written, it needs to be compiled and combined to create the full application. During this phase, different modules or parts of the code are brought together to form a complete system. Automation tools are often used to speed up this process and avoid errors.

You bring together all the code pieces to create a functioning version of the application.

4. Test

Testing ensures that the application works as expected. Various tests (like unit tests, integration tests, and performance tests) are run to find bugs or issues. If something doesn't work, it's fixed and tested again until it's right.

You check to make sure the application works properly by looking for any errors or bugs.

5. Integrate

During integration, code from different developers is combined into a single, cohesive system. This phase helps ensure that everyone's work fits together and functions smoothly without conflicts. Continuous Integration (CI) tools help to automate this process, reducing manual effort.

You combine the work of different developers into one unified application.

6. Deploy

Deployment is about making the application available to users. The software is placed on servers (usually in the cloud) so that it can be accessed and used by people. This phase often involves automating the release process to make deployment faster and more reliable.

You make the application live, so people can start using it.

7. Operate

After the application is deployed, the operations team ensures it runs smoothly. This includes fixing any issues that come up, managing server resources, and handling routine maintenance tasks. The goal is to keep the application functioning well for users.

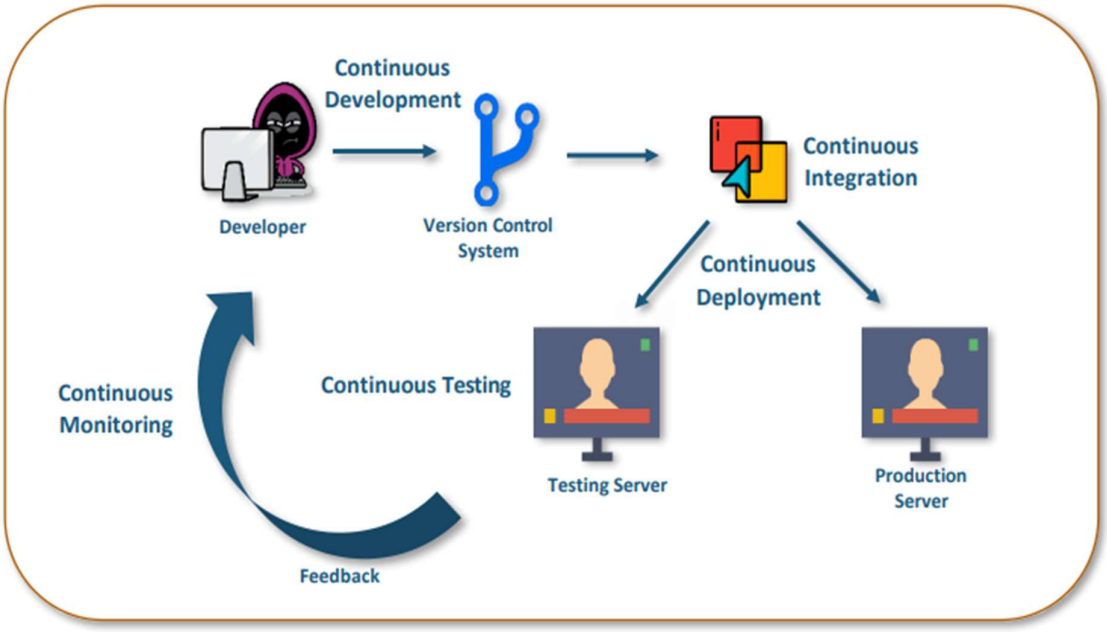
You take care of the application after it's live by fixing problems and keeping everything running smoothly.

8. Monitor

Continuous monitoring is key to ensuring the application performs well over time. Tools are used to track performance, uptime, and other key metrics. If issues arise, changes or fixes are made quickly to ensure users remain happy with the service.

You keep an eye on how well the application is working and make improvements when necessary to keep users satisfied.

Continuous DevOps Process



The DevOps process is a continuous flow, where software is constantly developed, tested, and improved in a cycle:

It all begins with the Developer writing new code or updating existing code. This code is then stored in a Version Control System, which helps keep track of all changes and versions of the software.

Once the code is ready, it moves to Continuous Integration, where it's automatically combined with the code from other developers. This step ensures that the new code works well with the rest of the project.

After integration, the process moves into Continuous Deployment. Here, the software is automatically sent to different servers. First, it goes to a Testing Server, where the code undergoes Continuous Testing to check if everything works as expected.

If the tests are successful, the code is then deployed to the Production Server, where users can access the new or updated software.

While the software is running on the Production Server, Continuous Monitoring takes place. This helps track performance and find any issues in real-time. Based on the results of this monitoring, feedback is collected and sent back to the development team.

This feedback initiates another cycle of development, ensuring the software is constantly improved and updated without any breaks in the process.

Thus, the process forms a loop that ensures continuous development, testing, deployment, and monitoring, making the software more efficient and reliable over time.