SOFTWARE ENGINEERING FUNDAMENTALS

Introduction

When we talk about learning DevOps, it is important to remember that DevOps is not a starting point, but a journey that sits on top of a strong foundation of software engineering. The very first step in this journey is to understand how software is planned, built, tested, deployed, and maintained over time. This structured process is known as the **Software Development Life Cycle (SDLC)**. Without understanding SDLC, DevOps would appear as just a collection of tools and practices, but its real power lies in how it improves and automates each stage of the life cycle.

Why is this so important? Imagine building a house without knowing the stages of construction. You may have workers, tools, and materials, but unless you know that you must first lay the foundation, then raise the walls, and finally build the roof, the process will be chaotic. Similarly, DevOps enhances software development only when we clearly understand the stages that exist before it.

In the real world, every project follows a journey. It begins with an **idea**, such as developing an e-commerce application or a mobile banking system. The idea then transforms into a set of **requirements**: what features must be included, who the users are, what problems the software should solve. Next comes the **design phase**, where system architecture, user interfaces, and data flow are planned. After this, the **development phase** takes place, where programmers translate the design into working code. Once the product is developed, it enters the **testing phase**, where it is validated for correctness, performance, and security. If the software meets expectations, it is **deployed** into a real-world environment for customers or users to access. Finally, the cycle continues into the **maintenance phase**, where updates, bug fixes, and new features are added over time.

To make this more concrete, consider the example of developing a **food delivery application**. The idea begins with a simple vision: "Allow users to order food from nearby restaurants." The requirements might include features such as user login, browsing restaurants, adding items to a cart, payment gateway integration, and real-time delivery tracking. In the design stage, engineers create app screen layouts, database structures for menus and orders, and architecture diagrams for servers and APIs. The development stage then converts these designs into working code. Testing ensures that users can successfully place an order, payments are processed correctly, and delivery updates are accurate. Once deployed, the app is released to the public, but the journey doesn't stop there—continuous maintenance is required to add new features like discount coupons or new payment methods and to fix any issues that arise.

This example illustrates that software development is not a one-time effort but a continuous journey. Every organization, whether building small mobile apps or large-scale enterprise systems, follows this life cycle. DevOps comes into the picture later, not to replace SDLC, but to make it **faster, more efficient, and more collaborative**. It automates testing, streamlines deployments, and creates a bridge between developers and operations teams, but the essence of what is being improved is still the SDLC itself.

In summary, the journey of DevOps begins with a deep appreciation of SDLC. It is the backbone of all software projects. Understanding SDLC gives clarity on what steps exist in the process of developing software, and then we can clearly see how DevOps enhances each of those steps.

Software Development Life Cycle (SDLC)

Definition

The **Software Development Life Cycle (SDLC)** is a structured process that defines the steps involved in developing, delivering, and maintaining software. It provides a **roadmap** that ensures the software product is delivered in a systematic, efficient, and high-quality manner.

In simple terms, SDLC answers the questions:

- *What needs to be built?*
- *How should it be built?*
- *Who will build it?*
- *How will we test it?*
- *When will it be delivered?*
- *How will it be maintained?*

Without SDLC, software development becomes chaotic, leading to missed deadlines, poor quality, and dissatisfied users.

Phases of SDLC

SDLC is divided into **sequential yet interconnected phases**. Each phase has a specific purpose and deliverable.

1. **Requirement Analysis**
    - Gathering and analyzing business needs.
    - Conducted through stakeholder meetings, surveys, or use-case discussions.
    - Output: A clear requirement document.
    - *Example:* For a banking app → requirement: “User should be able to transfer money securely.”

2. **System Design**
    - Creating the blueprint of the system.
    - Includes architecture design, database design, and user interface design.
    - Output: Design specifications, mockups, ER diagrams.
    - *Example:* Designing login screens, database schema for accounts, transaction flow.
3. **Development (Implementation)**
    - Actual coding of the system as per design.
    - Developers follow coding standards and use version control tools.
    - Output: Working software modules.
    - *Example:* Writing APIs for fund transfer and building the mobile app screens.
4. **Testing**
    - Validating the software to ensure it meets requirements.
    - Includes functional, performance, security, and user acceptance testing.
    - Output: Tested, quality-assured product.
    - *Example:* Checking whether transferring ₹100 from account A to B works correctly and securely.
5. **Deployment**
    - Releasing the software to the end-users or customers.
    - Can be a pilot launch, staged rollout, or full release.
    - Output: Live product in production environment.
    - *Example:* Publishing the banking app on Google Play Store / App Store.
6. **Maintenance**
    - Continuous updates, bug fixes, and enhancements after deployment.
    - Ensures product remains reliable and relevant.
    - Output: Long-term sustainability of software.
    - *Example:* Adding UPI payments or fixing login issues after user feedback.

Traditional SDLC Flow

- The earliest approach to SDLC was **linear and sequential**.
- Each phase had to be completed **before the next could begin**.
- This works well when:
    - Requirements are very clear.
    - Project scope is fixed.
    - Client expectations will not change.

For example: In **government projects** (like census systems or railway booking), requirements are frozen years in advance, so traditional SDLC works effectively.

Real-World Example: Banking Application

Let’s map the phases of SDLC to a banking app project:

- **Requirement Analysis:** Define features → login, balance check, fund transfer.
- **Design:** Create system flow diagrams, database schema for accounts, user interface wireframes.
- **Development:** Write code for login authentication, account management, and money transfer API.
- **Testing:** Validate transaction limits, check for security loopholes, ensure no double deductions.
- **Deployment:** Release app to production environment for customers.
- **Maintenance:** Fix issues like failed transactions and add new features like credit card integration.

Importance of SDLC

- Provides a **structured approach** to software building.
- Reduces project risks and delays.
- Ensures **quality and reliability** of software.
- Acts as the **foundation** for modern methodologies like **Agile and DevOps**.

Waterfall Model

Definition

The Waterfall Model is the **earliest and most traditional approach** to the Software Development Life Cycle (SDLC). It follows a **linear and sequential flow**, where each phase must be completed before moving on to the next. Just like water flows downwards step by step, the project progresses phase by phase without going back.

In this model, once a phase is finished, there is **little or no flexibility** to make changes.

Strengths of the Waterfall Model

1. **Simple and easy to use** – Since it is step-by-step, it is easy for teams to follow.
2. **Well-structured** – Each phase has clear goals and deliverables.
3. **Good documentation** – Because every phase produces detailed documents, knowledge transfer is easier.
4. **Useful for fixed requirements** – Works well when the customer knows exactly what they want and requirements will not change.

Weaknesses of the Waterfall Model

1. **Inflexible to changes** – If requirements change later, it is very hard and costly to modify.
2. **Late testing** – Testing happens only after development is complete, so issues are found very late.
3. **No customer feedback during development** – Users only see the product after it is built.
4. **High risk for long projects** – If assumptions made early are wrong, the whole project can fail.

Example

Consider a **government project** such as a passport application system. The requirements are usually well-defined and do not change frequently. For example: collecting personal data, verifying identity, and issuing passports. Since the process is stable and changes are rare, the Waterfall Model works here.

However, if the same model were used for a **mobile e-commerce application**, it would be risky. Customer needs change quickly (discount features, new payment methods), and Waterfall would not handle those changes well.

Importance of Understanding Waterfall

Even though modern methods like Agile and DevOps are widely used today, the Waterfall Model is still important to learn because:

- It shows the **evolution of software engineering practices**.
- Many legacy projects in industries like **banking, defense, and government** still follow this model.
- Interviewers may ask students to **compare Waterfall and Agile** to test their understanding of traditional vs modern methods.

Agile Methodology

What is Agile?

Agile is a **modern approach to software development** that focuses on **flexibility, collaboration, and delivering working software quickly**. Unlike the Waterfall Model, which is rigid and linear, Agile is **iterative and incremental**. This means the product is developed in **small parts (increments)**, and improvements are made through **continuous feedback**.

The main goal of Agile is to adapt to changing requirements and deliver value to the customer as early and as often as possible.

Principles of Agile (Agile Manifesto)

Agile is based on a set of values and principles defined in the **Agile Manifesto (2001)**. The four core values are:

1. **Individuals and interactions over processes and tools**
   – People and teamwork are more important than rigid tools or rules.
2. **Working software over comprehensive documentation**
   – Delivering a usable product is valued more than just writing documents.
3. **Customer collaboration over contract negotiation**
   – Customers are involved throughout the process, not just at the beginning or end.
4. **Responding to change over following a plan**
   – Flexibility is encouraged; if requirements change, the team adapts quickly.

Agile Frameworks

Agile is not a single method but a philosophy with multiple frameworks. The most common ones are:

1. **Scrum**
    - Work is divided into short cycles called **sprints** (usually 2–4 weeks).
    - A **product backlog** (list of features) is maintained.
    - Daily **stand-up meetings** ensure team coordination.
    - Roles include **Product Owner, Scrum Master, and Development Team**.
2. **Kanban**
    - Visual approach using **boards and cards** to track work.
    - Work is pulled as capacity allows, ensuring a **continuous flow**.
    - No fixed sprints; tasks move through stages like *To Do* → *In Progress* → *Done*.

Real-World Example

Suppose a company is building an **e-commerce application**.

- In **Sprint 1**, the team delivers the shopping cart feature.
- In **Sprint 2**, they add payment integration.
- In **Sprint 3**, they include discount coupons and offers.

The customer can use the product earlier, and the team can gather feedback at every stage to make improvements.

Strengths of Agile

1. Highly flexible and adaptable to change.
2. Delivers working software early and frequently.
3. Encourages customer involvement and satisfaction.
4. Reduces risk of failure by continuous testing and feedback.

Weaknesses of Agile

1. Requires close collaboration, which may be hard in distributed teams.
2. Less documentation, which may cause confusion later.

3. Needs experienced team members to work effectively.
4. Can be harder to manage for very large projects.

Why Agile Matters Today

- Agile is the **most widely used methodology in the industry today**.
- It supports modern practices like **DevOps and Continuous Delivery**.
- Most IT companies expect students and professionals to understand Agile and be comfortable working in **Scrum or Kanban environments**.

TASKS

Task 1: Timeline Activity

- On a sheet of paper, **draw a timeline** showing how software moves through the six SDLC phases.
- Under each phase, write **one risk** if that step is skipped.

Task 2: Role Identification

- Imagine you are part of a software project team. Assign yourself **one role** (e.g., analyst, designer, developer, tester, operations).
- Write **3 responsibilities** you would have in that role.
- Then write how **DevOps would support your role** (e.g., automation for testers, CI/CD for developers).

Task 3: Storytelling Exercise

- Write a **short story (5–6 lines)** about developing a **student attendance app**.
- The story should mention at least **three SDLC stages** (like requirement, design, testing).
- End with how **Agile + DevOps** would make it easier to improve the app quickly.