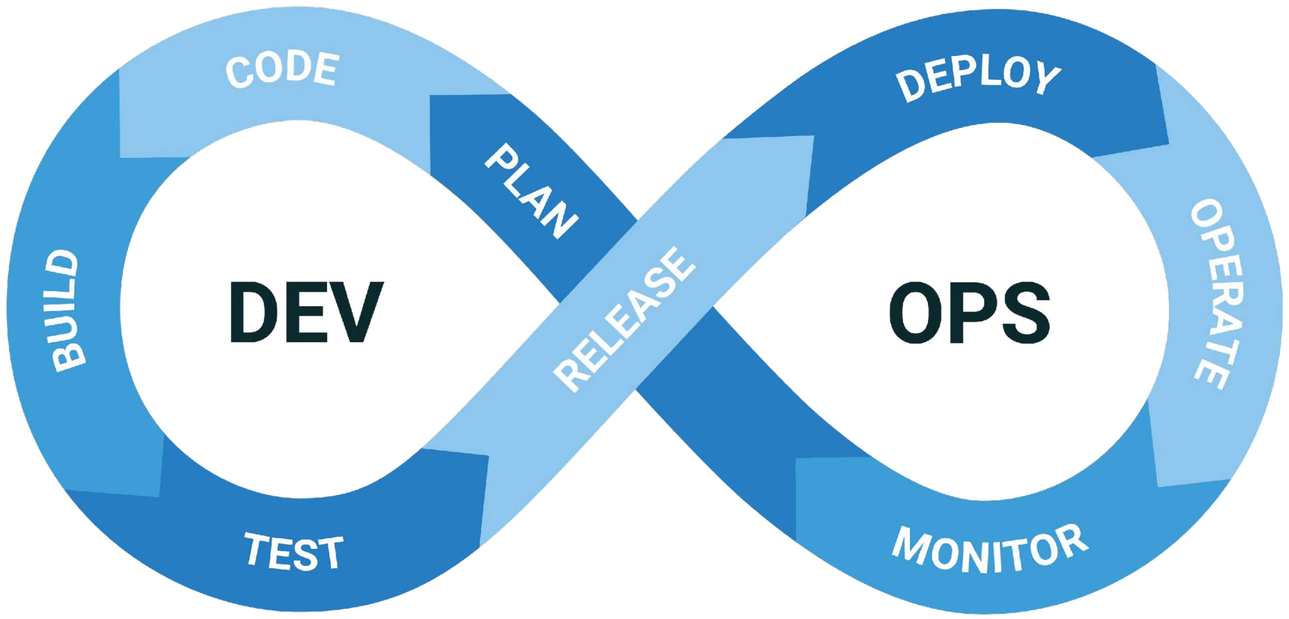
Introduction to DevOps

Understanding the DevOps Culture and Practices

By Jjtech inc.

Definition of DevOps

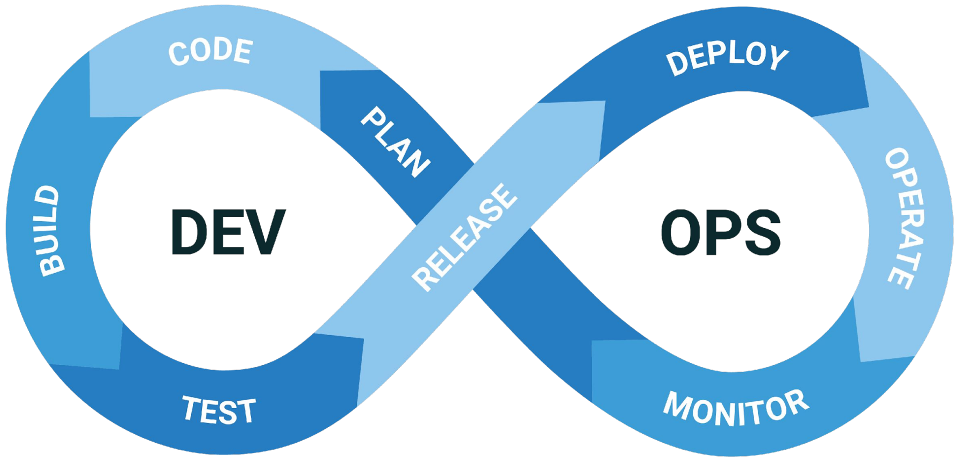
DevOps is a about a methodology, a set of practices, tools and a new approach businesses use to develop, deliver, and maintain software. The outcome being-fater and high quality of deployments.



Benefits of DevOps

Speed	Teams that practice DevOps release deliverables more frequently, with higher quality and stability
Improved Collaboratoon	The foundation of DevOps is a culture of collaboration between developers and operations teams, who share responsibilities and combine work.
Rapid Deployment	By increasing the frequency and velocity of releases, DevOps teams improve products rapidly. A competitive advantage can be gained by quickly releasing new features and repairing bugs.
Quality and Reliability	Practices like continuous integration and continuous delivery ensure changes are functional and safe, which improves the quality of a software product. Monitoring helps teams keep informed of performance in real-time.

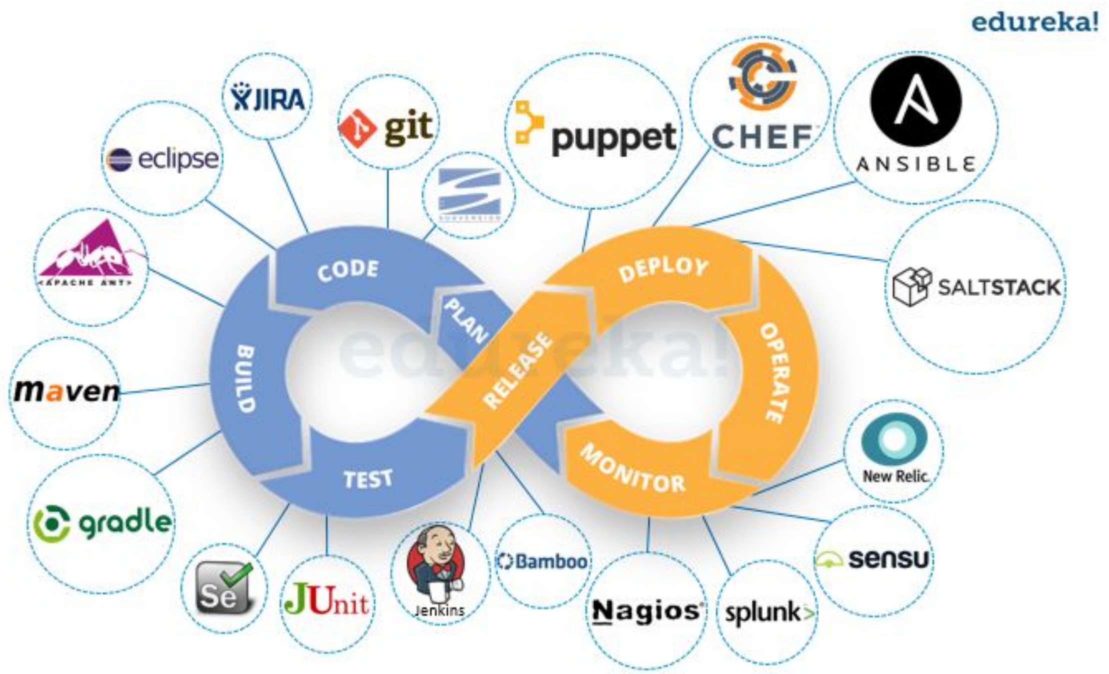
DevOps Lifecycle



Stages	Description
Plan	In this stage, teams identify the business requirement and collect end-user feedback. They create a project roadmap to maximize the business value and deliver the desired product during this stage.
Code	The **code development** takes place at this stage. The development teams use some tools and plugins like *Git* to streamline the development process, which helps them avoid security flaws and lousy coding practices.
Build	In this stage, once developers finish their task, they commit the code to the shared code repository using build tools like Maven and Gradle.
Test	Once the build is ready, it is deployed to the test environment first to perform several types of testing like user acceptance test, security test, integration testing, performance testing, etc., using tools like JUnit, Selenium, etc., to ensure software quality.
Release	The build is ready to deploy on the production environment at this phase. Once the build passes all tests, the operations team schedules the releases or deploys multiple releases to pre-prod and then production
Deploy	In this stage, Infrastructure-as-Code helps build the production environment and then releases the build with the help of different tools.
Operate	The release is live now to use by customers. The operations team at this stage takes care of server configuringand maintenance of the application.
Monitor	In this stage, the DevOps pipeline is monitored based on data collected from customer behavior, application performance, etc. Monitoring the entire environment helps teams find the bottlenecks impacting the development and operations teams' productivity.

DevOps Tools Overview

- Planning: Jira, Trello
- Coding: Git, GitHub, Bitbucket, VsCode
- Building: Jenkins, Maven, gradle, Ant
- Testing: Selenium, Junit, Sonarqube
- Releasing: Docker, Kubernetes, Ansible
- Deploying: AWS, Azure
- Operating: Nagios, Prometheus
- Monitoring: Grafana, ELK Stack



Continuous Integration and Continuous Delivery/Deployment

Continuous Integration (CI):

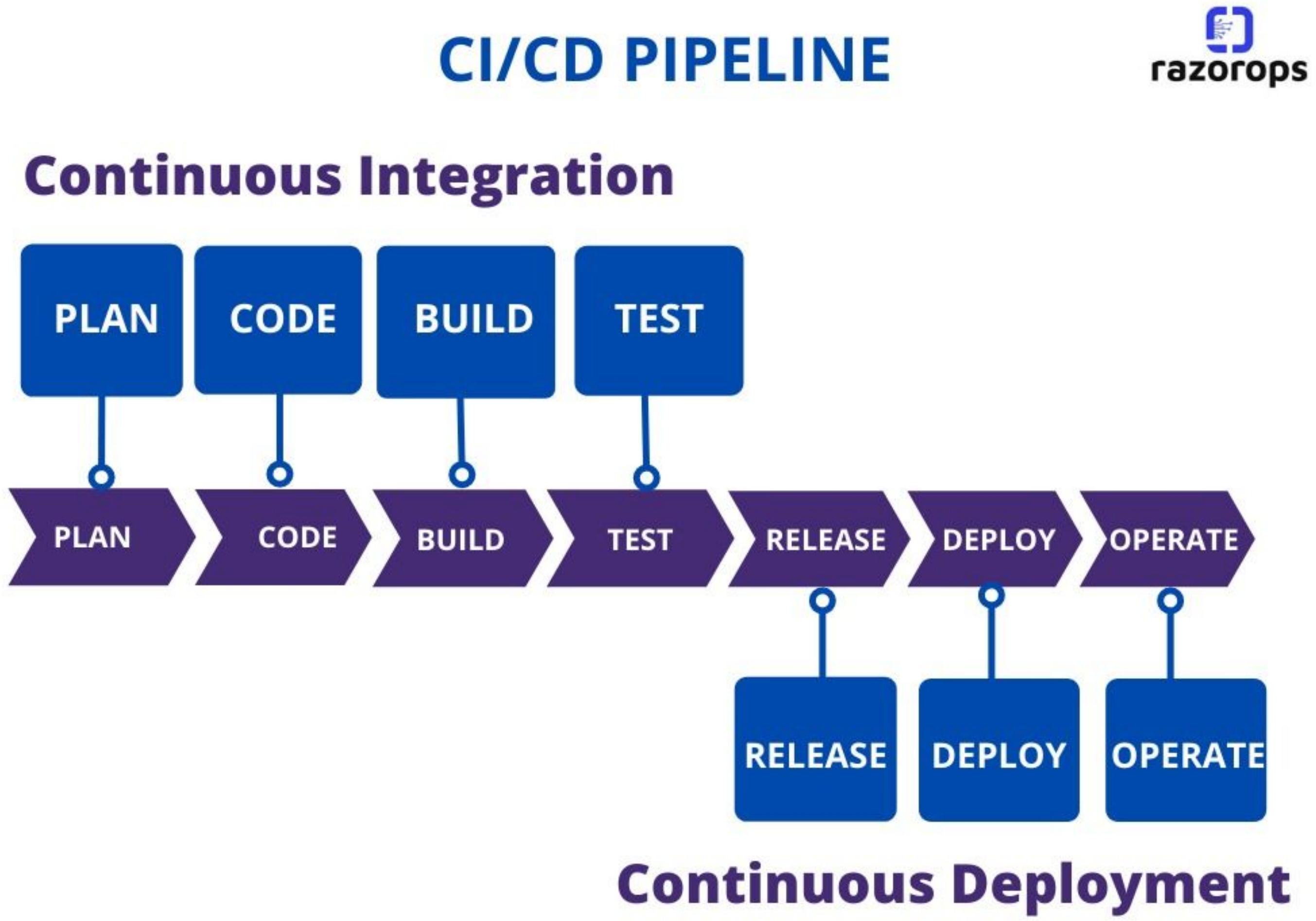
CI involves developers frequently integrating their code changes into a shared repository.

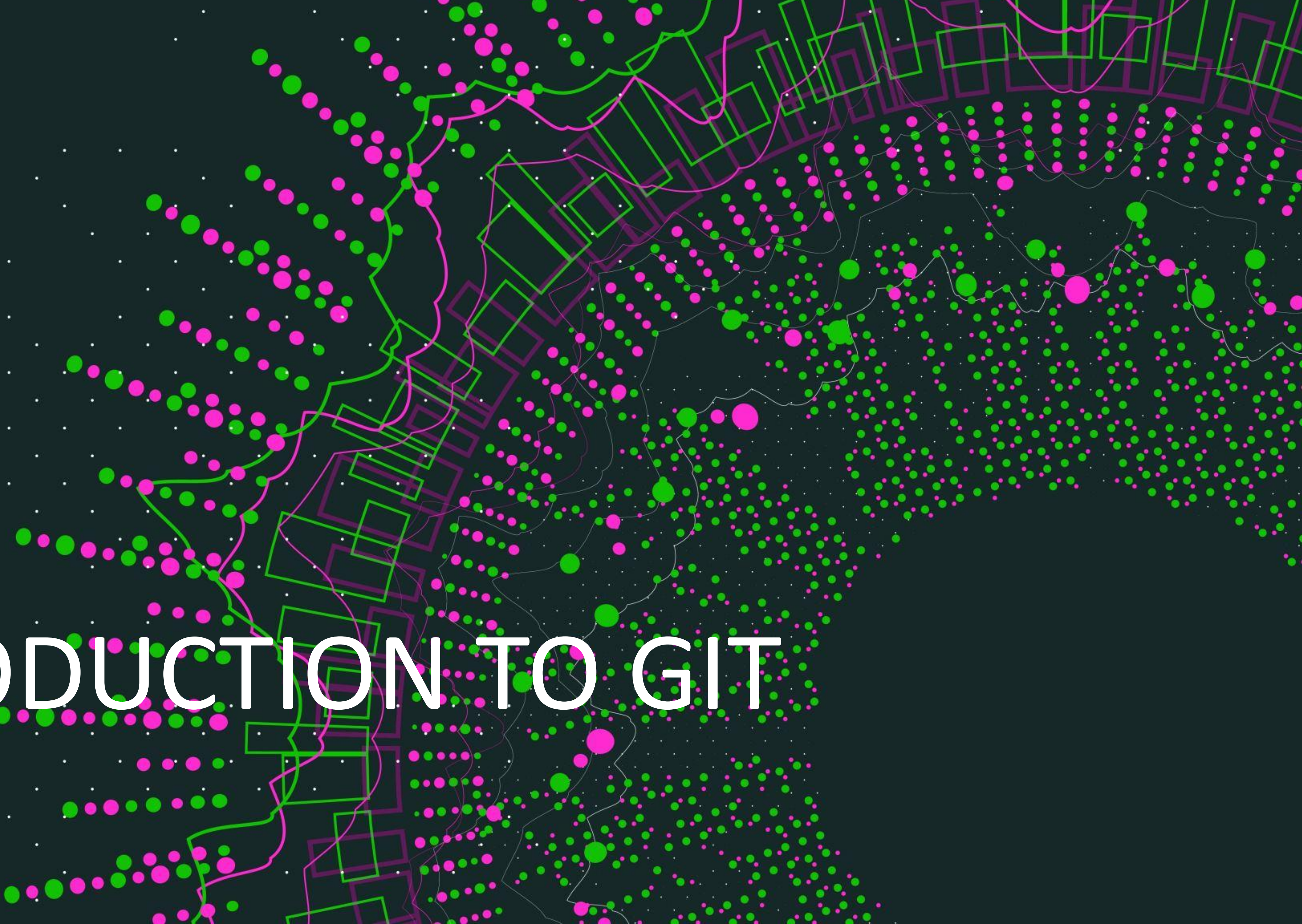
Each integration triggers an automated build and test process, where the code is compiled and tested automatically. The main objective of CI is to detect integration errors early in the development process, allowing teams to address issues quickly and maintain a stable codebase

Continuous Deployment (CD):

CD extends the principles of CI by automating the release process. With Continuous Deployment, every code change that passes through the CI pipeline is automatically deployed to production environments, assuming it meets all necessary quality criteria. CD pipelines encompass not only automated testing but also automated deployment to various environments, including development, testing, staging, and production. The aim of CD is to ensure that code changes are released to customers rapidly, reliably, and with minimal manual intervention

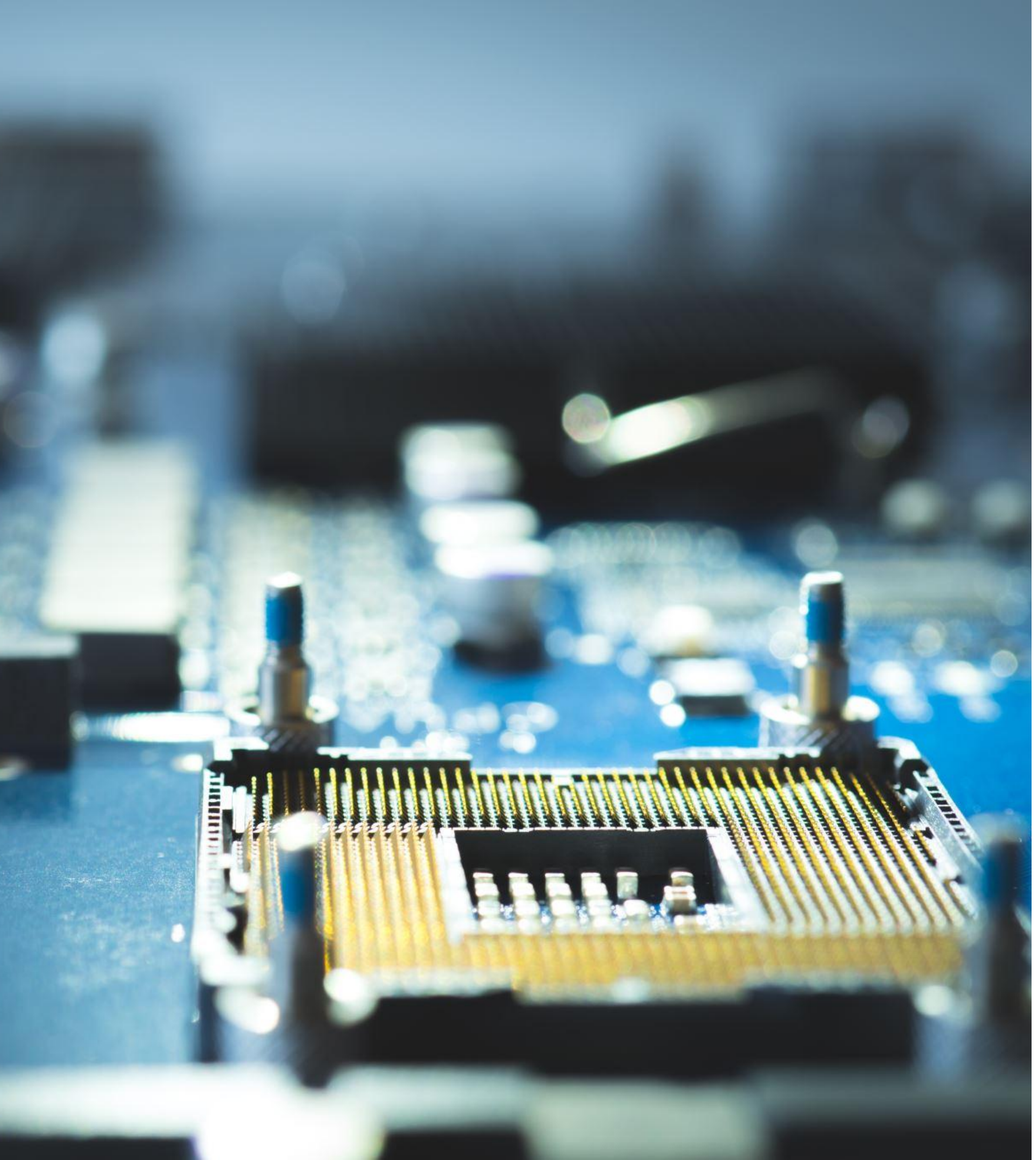
CI/CD Pipeline Diagram





INTRODUCTION TO GIT

BY JJTECH INC.

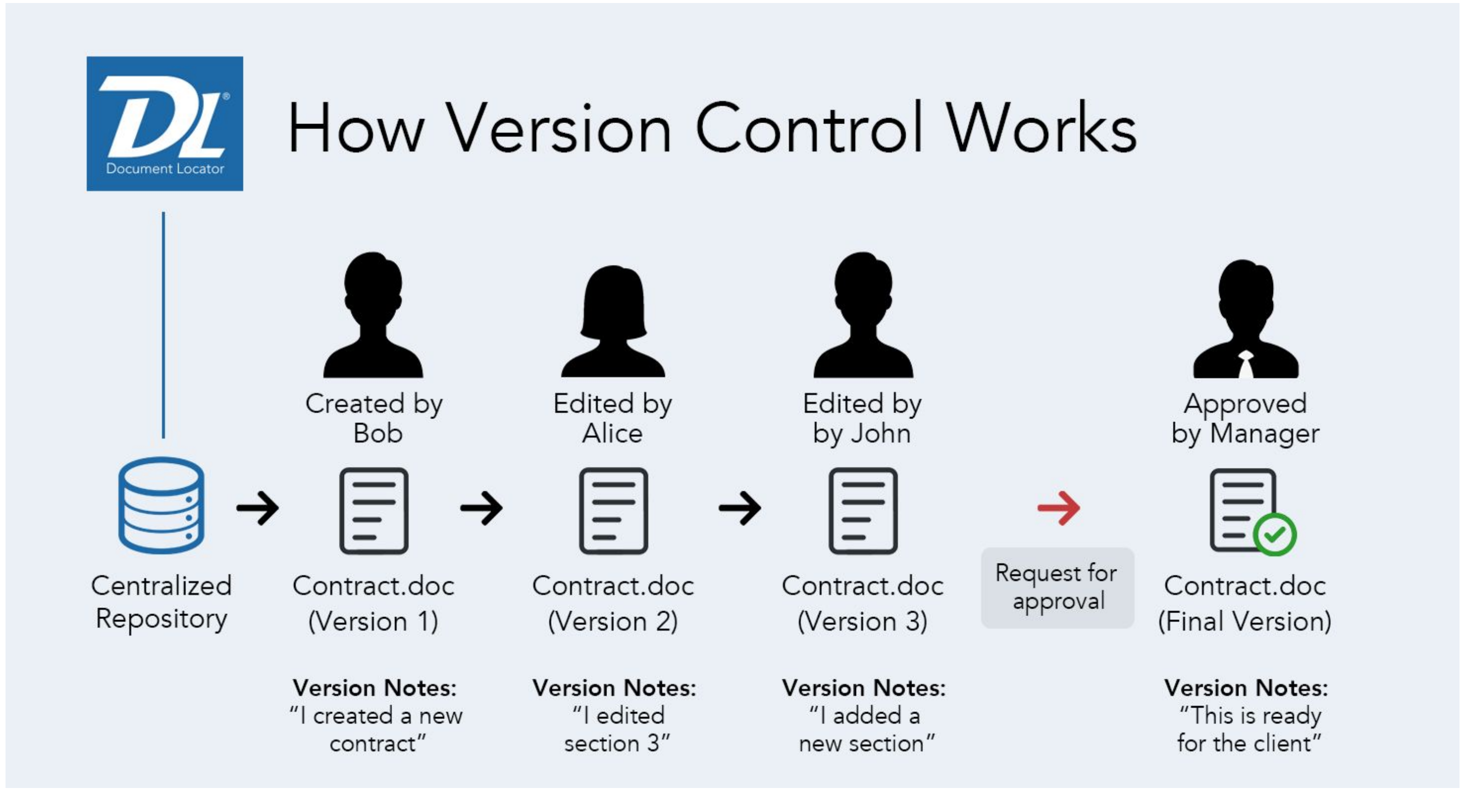


CONTENT

- Version control systems
- Git and Github
- Git Workflow
- Git Commands

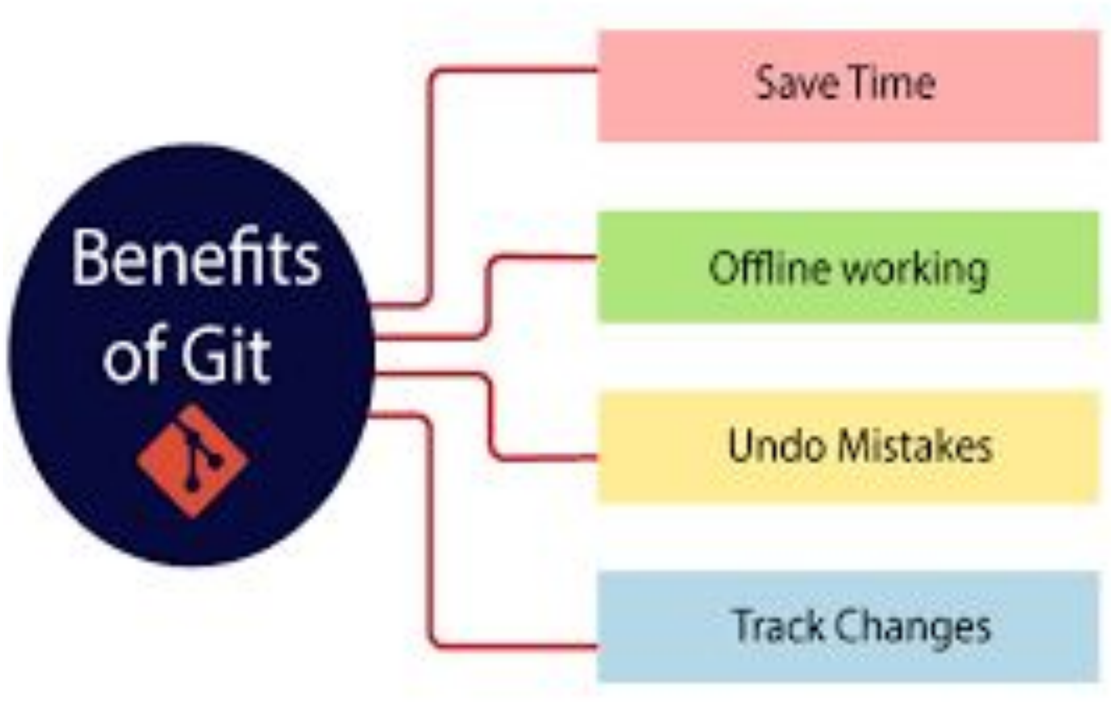
Version Control System

- VCS also known as **Source Control System** is a software tool or framework used to manage and track changes made to files and directories over time.
- It is commonly used in software development but can also be applied to other types of projects that involve multiple contributors working on shared files.


Document Locator
How Version Control Works
Created by Bob
Edited by Alice
Edited by by John
Approved by Manager
Centralized Repository
Contract.doc (Version 1)
Contract.doc (Version 2)
Contract.doc (Version 3)
Request for approval
Contract.doc (Final Version)
Version Notes: "I created a new contract"
Version Notes: "I edited section 3"
Version Notes: "I added a new section"
Version Notes: "This is ready for the client"

Benefits for VCS

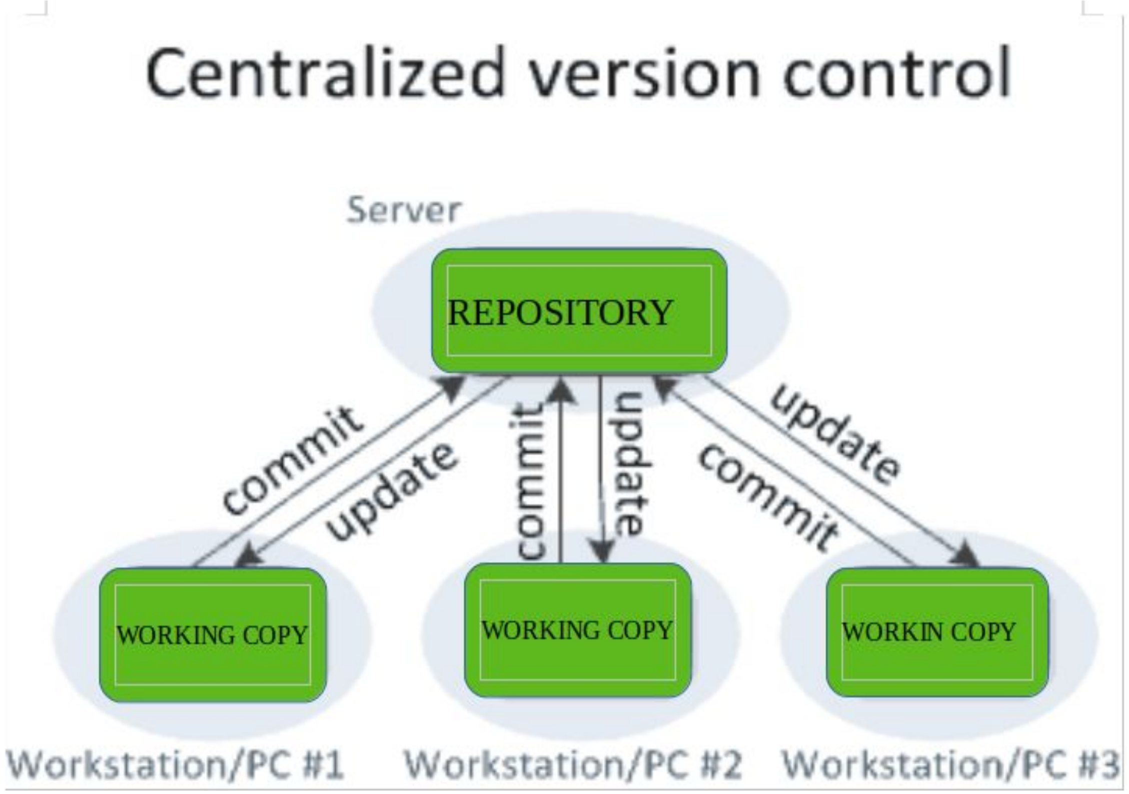
1. **Track changes:** Allows developers to keep track of every modification made to files, including additions, deletions, and modifications. This history can include details such as who made the changes, when they were made, and what specific changes were made.
2. **Version management**: Enables the creation and management of different versions or snapshots of a project. Each version represents a specific state of the codebase at a given point in time. This capability allows developers to access previous versions, compare differences, and revert to earlier versions if needed.
3. **Collaboration: VCS** systems facilitate collaboration among multiple developers working on the same project by providing mechanisms for merging changes made by different contributors, resolving conflicts, and coordinating the integration of code changes.
4. **Branching and merging:** VCS allows developers to create branches, which are separate lines of development that can diverge from the main codebase. Branches enable developers to work on new features, bug fixes, or experiments without affecting the main codebase.
5. **Conflict resolution:** VCS systems handle conflicts that arise when two or more developers modify the same file simultaneously. VCS provides mechanisms to merge conflicting changes or prompt developers to resolve conflicts manually

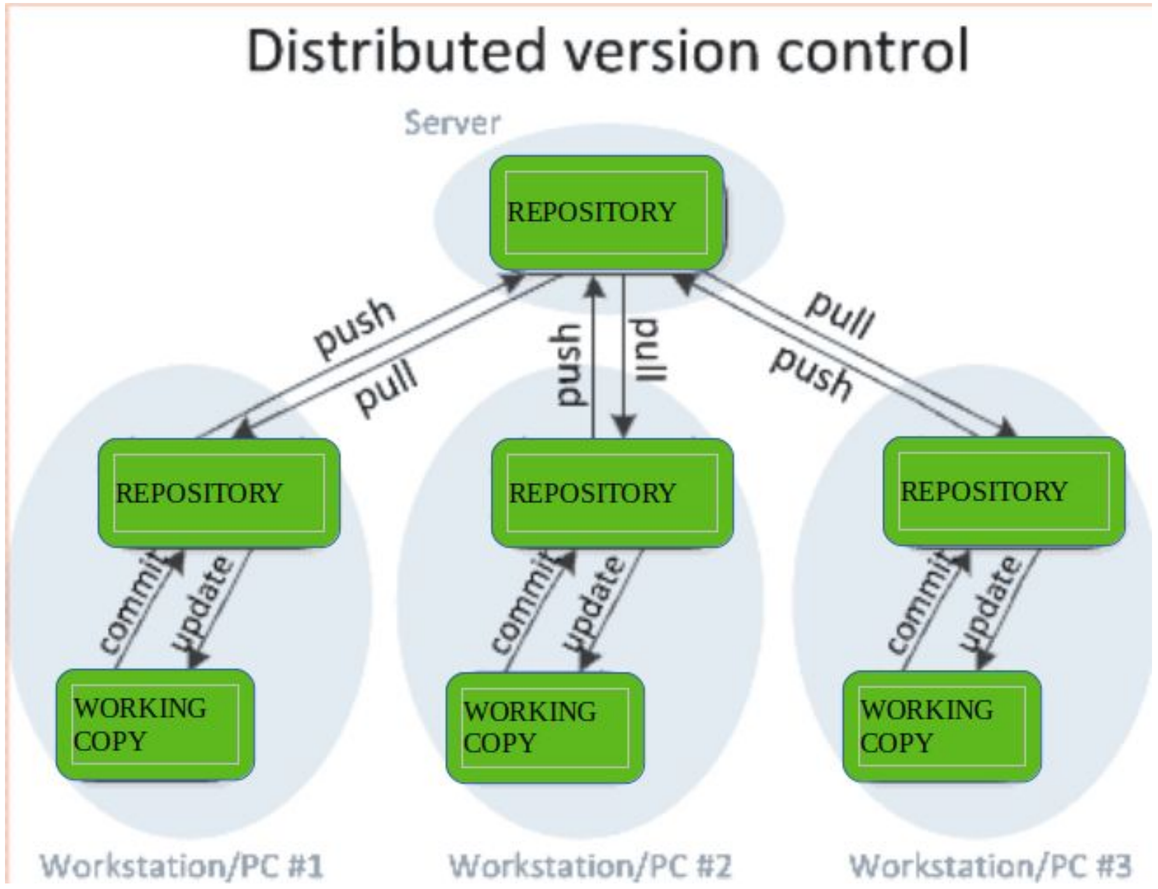


Types of VCS

Centralized VCS

- Central server that stores the entire codebase and its complete history.
- Developers typically work with a local working copy of the code and interact with the central server to commit changes, update their copy, or retrieve previous versions.
- Collaboration relies heavily on the central server, making it a potential single point of failure
- Branching and merging operations can be more challenging and involve coordination with the central server.
- Examples of CVCS include Subversion (SVN) and Perforce





Distributed Version Control Systems (DVCS)

- Each developer has a local repository that contains the entire project history, enabling them to work offline without relying on a central server.
- Developers can commit changes, create branches, merge code, and explore history locally, without network connectivity
- DVCS systems are generally more resilient to server failures because every developer has a complete copy of the repository I their local
- Example include Git and Mercurial

git

GitHub

Git

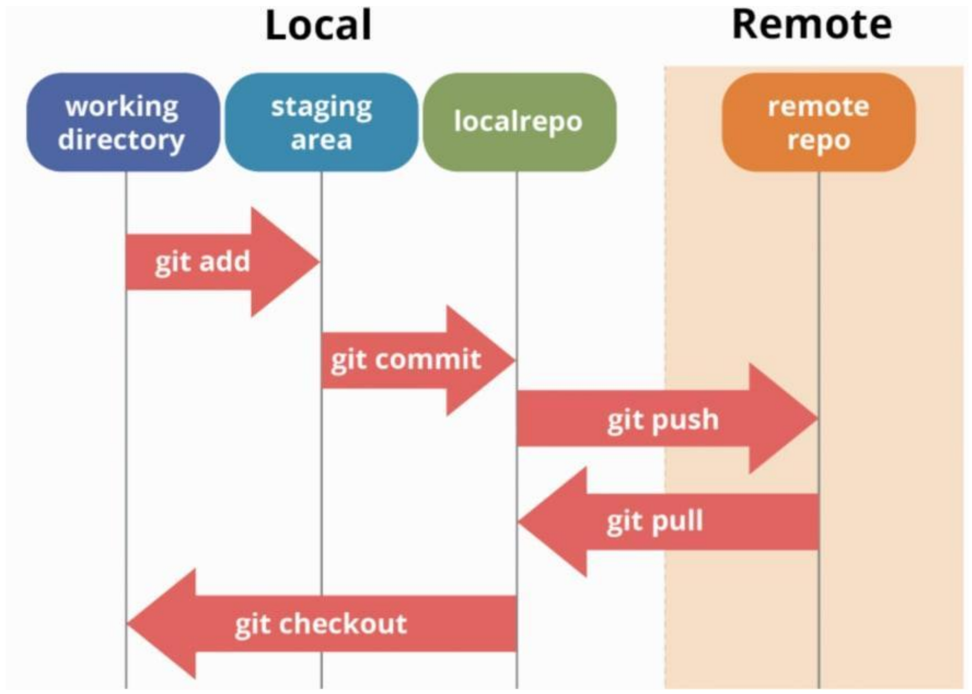
Git is a distributed version control system (DVCS) that was created by Linus Torvalds in 2005. It is a command-line tool that allows developers to track changes, manage versions, and collaborate on code efficiently. Features include:

- Distributed nature: Each developer has a complete local copy of the repository, allowing for offline work and decentralized collaboration.
- Branching and merging: Git makes branching and merging operations fast and easy, enabling parallel development and seamless integration of code changes.
- Speed and efficiency: Git is designed to be fast, even with large codebases and extensive histories.
- Flexibility: Git is not tied to any specific development workflow or centralized server.

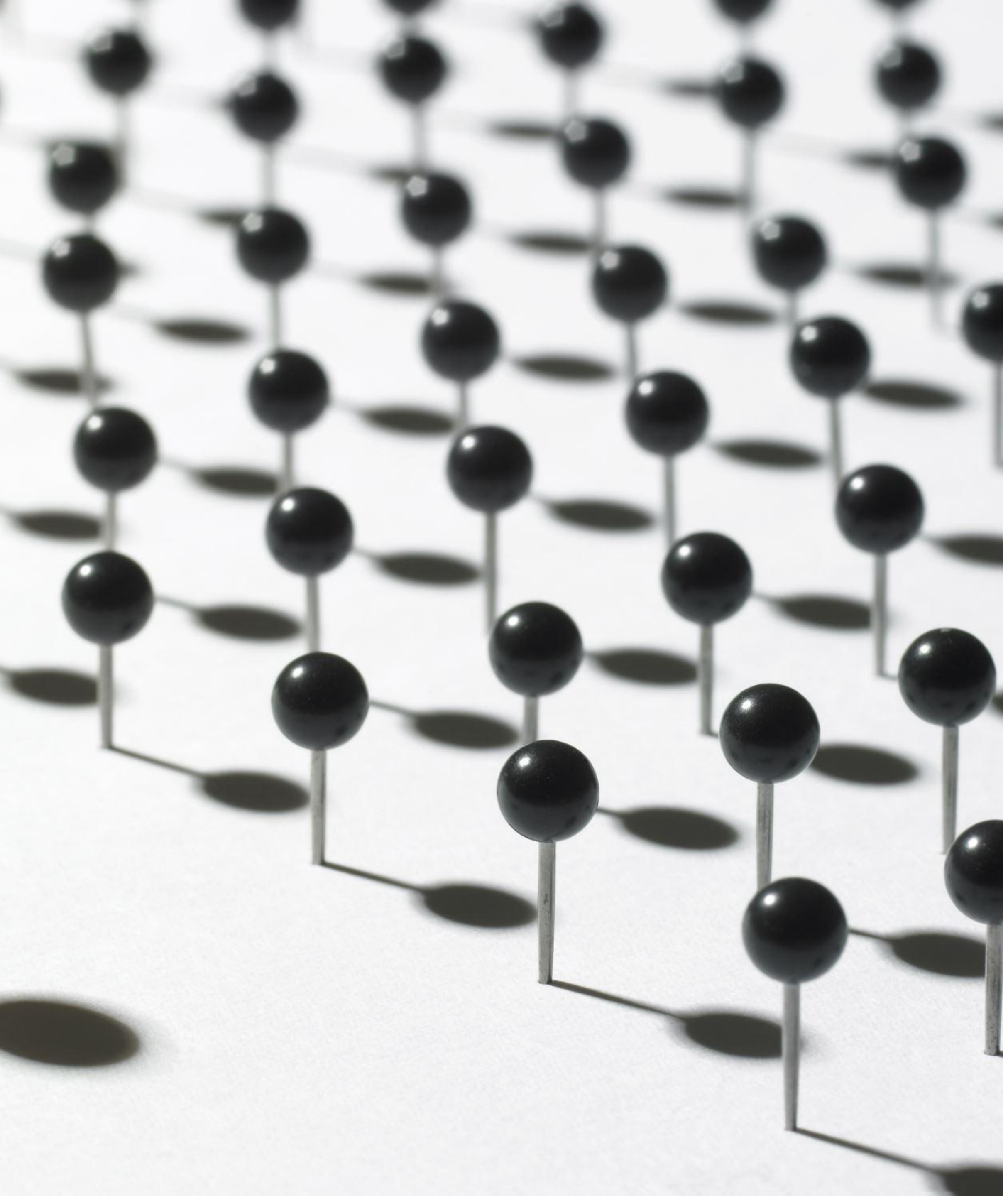
Github

- GitHub is a web-based hosting platform that provides a graphical user interface (GUI) and additional collaboration features.
- It was founded in 2008 and has become one of the most popular platforms for hosting and sharing Git repositories.
- GitHub allows developers to store their Git repositories in the cloud and provides a web interface for managing repositories.
- GitHub facilitates collaboration among developers by providing features such as pull requests, code reviews and issue tracking.
- GitHub offers access control mechanisms to manage user permissions and control who can access and contribute to repositories.
- GitHub integrates with various tools and services, including continuous integration (CI) systems, project management tools, and third-party integrations.

Git Workflow



1. Initialize the repository:
    - Create a new Git repository or clone an existing one to your local machine.
2. Create branches:
    - Develop new features or work on bug fixes in feature branches,
    - Use descriptive branch names that reflect the purpose of the work being done.
3. Develop features:
    - Make changes, commit them to your branch, and regularly push the branch to the remote repository to share your progress with others.
4. Review and integrate changes:
    - Once a feature is complete, create a pull request (PR) or merge request (MR) to request a code review from other team members.
    - Address any feedback or requested changes in subsequent commits.
    - Once approved, the changes are ready to be integrated into the main development branch.
5. Merge and release:
    - When a set of features is ready for release, create a release branch from the development branch.
    - Perform any necessary testing and bug fixes on the release branch.
    - Once the release is stable, merge the release branch into the main branch (often called "master" or "main").
    - Tag the release with a version number for future reference.
6. Hotfixes:
    - If critical bugs are discovered in the released version, create a hotfix branch from the main branch.
    - Fix the bug on the hotfix branch and merge it back into both the main branch and the development branch.



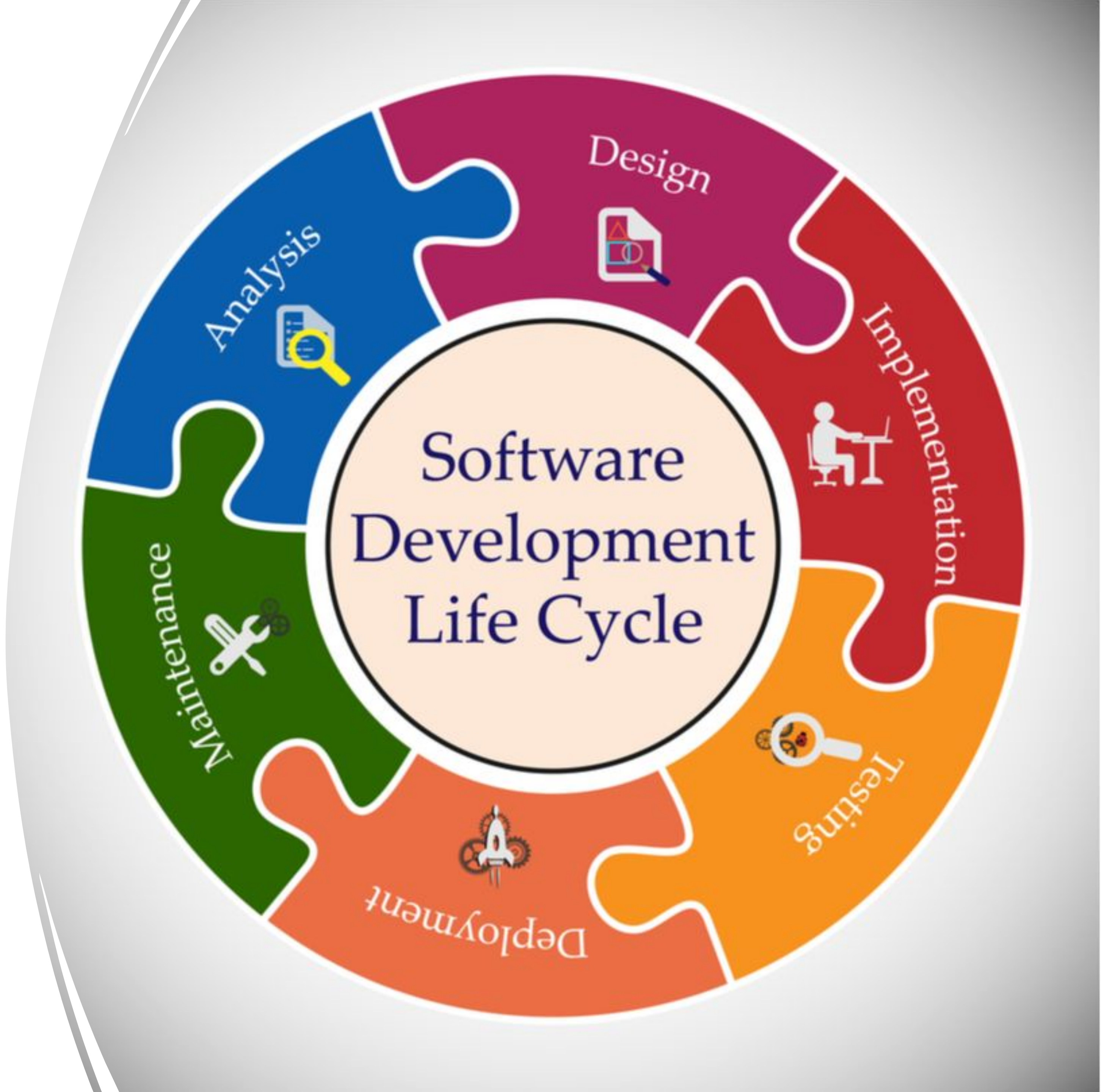
Software Development Lifecycle (SDLC)

Software Development Life Cycle (SDLC) describes the systematic and step by step approach to developing a software.

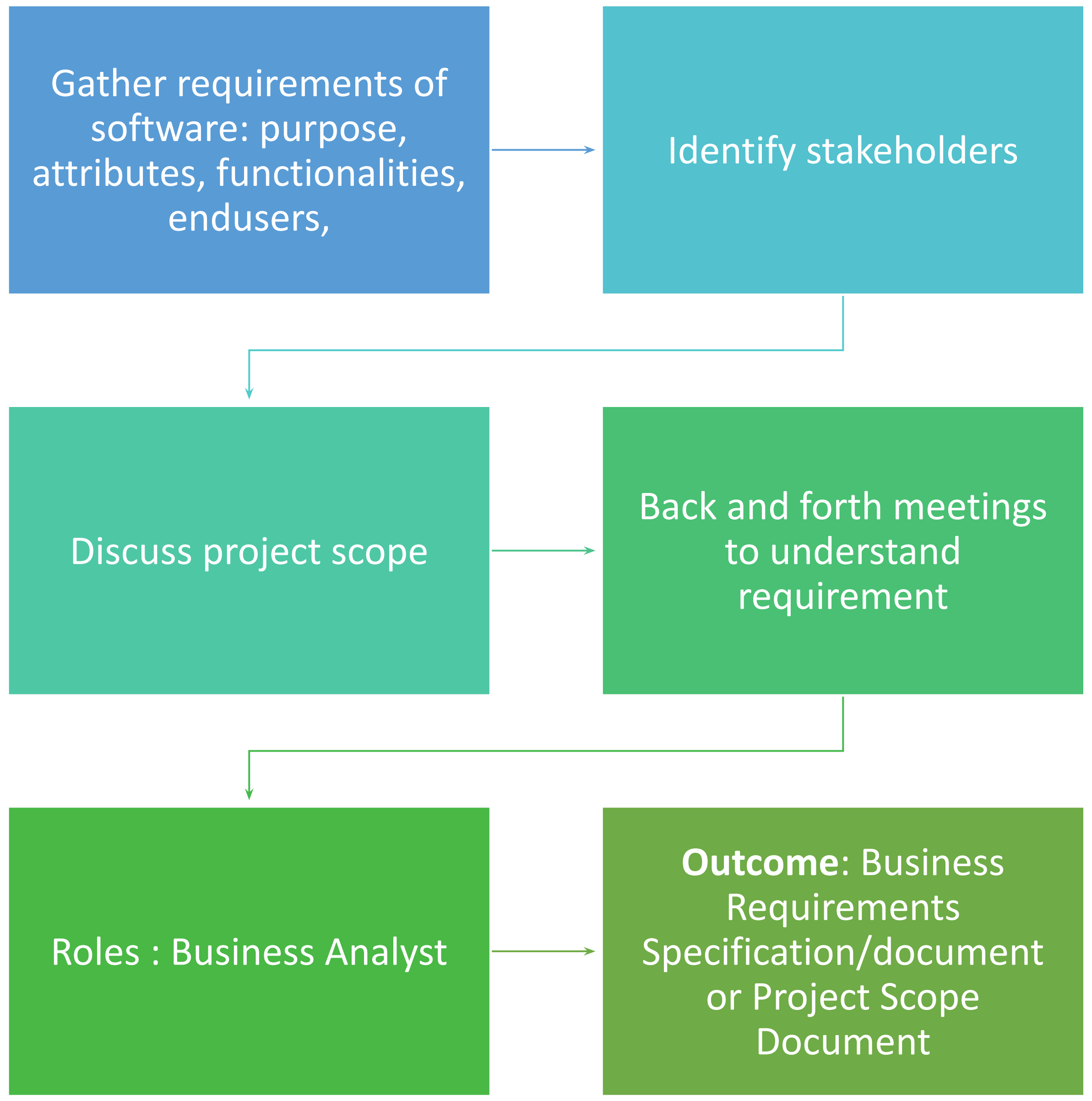
Describes different phases involved in development process

Software Development Lifecycle (SDLC)

- Planning & Requirements
- Requirement Analysis
- Design
- Implementation
- Testing
- Deployment
- Maintenance



Planning Stage/Phase


Gather requirements of software: purpose, attributes, functionalities, endusers,
Identify stakeholders
Discuss project scope
Back and forth meetings to understand requirement
Roles : Business Analyst
Outcome: Business Requirements Specification/document or Project Scope Document

Requirement Analysis Phase/stage

Analysis and feasibility of the requirements

The detailed software requirements are documented. This document is called Software Requirement Specification (SRS)

Role: Business Analyst, Project Manager, Team lead

Outcome: SRS document

Design

Develop architectural design of the software. This serves that blueprint for development phase

HLD- High level design

LLD- Low level design

Role: Architects, Team Lead, Senior Developers

Outcome: HLD and LLD documents

Implementation/Development Phase

Actual coding and development of application based on Requirement and Design documents

Role: Developers

Outcome: Application Source Code, Developed product

Testing Phase

Developed software is tested for bugs and other issues

Different types of testing: Unit, integration, system, functional and user acceptance test ()UAT

Role: Technical lead, Testing team

Outcome: Test Artifacts(test plan, results)

Types of Software Tests

- **Unit Test:** Focuses on testing individual units or components of a software application in isolation. It ensures that each unit functions correctly and independently. Developers typically write unit tests to catch bugs early and ensure the reliability of individual code units.
- **Integration Test:** Integration testing verifies the interactions between different units or components of a software system. It ensures that the integrated components work together as expected and that data flows correctly between them. Integration tests help identify issues that arise when multiple components are combined.
- **Functional Test:** Functional testing validates whether the software application's features and functionality align with the specified requirements. It aims to ensure that the software behaves as expected and meets the user's needs.
- **User Acceptance Test (UAT):** User Acceptance Testing is the final phase of testing before a software application is released to end-users. It involves testing the software from the end-users' perspective to ensure it meets their requirements and expectations.
- **Regression Test:** Regression testing is performed after making changes or updates to the software to ensure that the existing functionalities have not been adversely affected. It helps detect unintended side effects caused by code changes.
- **Load Test:** Load testing examines how the software performs under high user loads and determines its capacity to handle concurrent users and transactions.
- **Compatibility Test:** Compatibility testing ensures that the software functions correctly on different platforms, operating systems, browsers, and devices.
- **Performance Test:** Performance testing evaluates the application's responsiveness, scalability, and stability under varying workloads. It assesses the system's performance and identifies any bottlenecks or performance issues.

Deployment

App/software is finally released to different environments test, stage, Dev, prod

Role: Deployment engineers

Outcome: Ready-to-use application in prod env

Maintenance

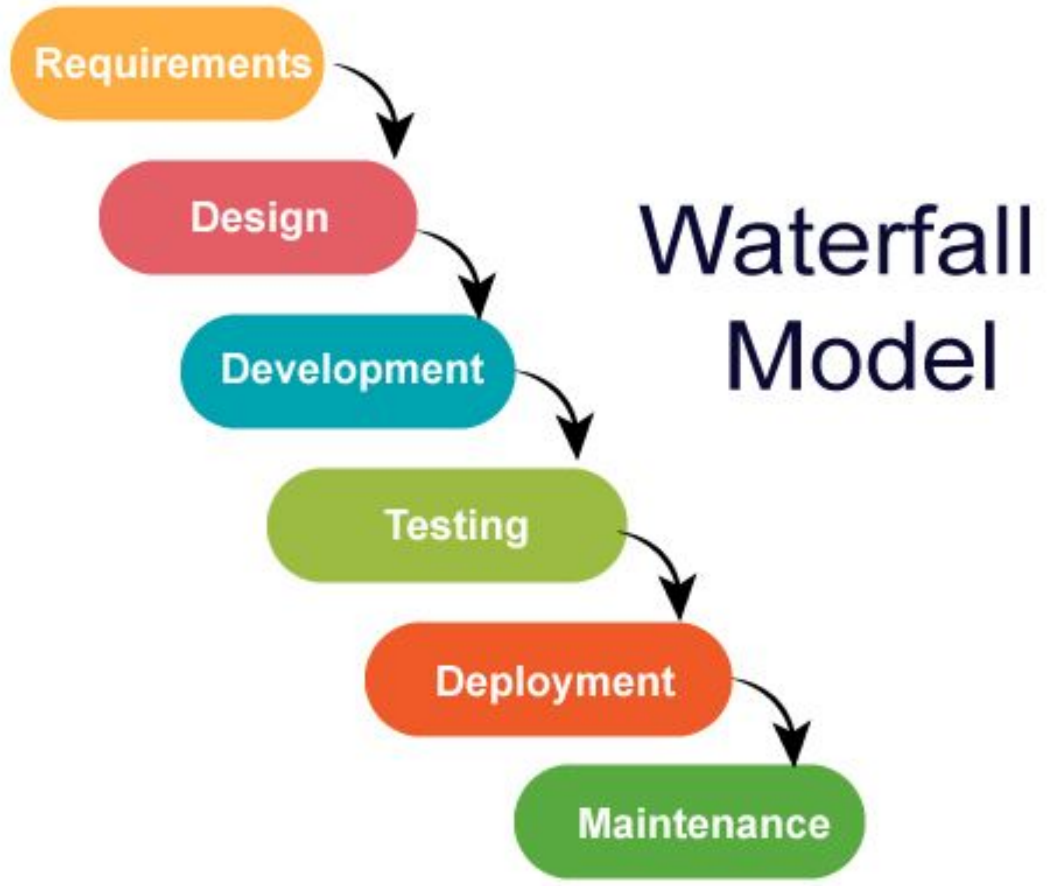
After deployment, developers and support teams address issues, fix bugs and make updates and enhancements as needed for software to be functional

Outcome: Service Level Agreement (SLAs)

Role: Developers, Testers, Tech lead, support engineers

Types of SDLC Models

- Waterfall /Sequential
- Spiral
- Agile



Waterfall SDLC Models

- Classical SDLC model, One of the earliest model and most traditional approach to SDLC
- Linear and Sequential: Each phase is completed before moving on to the next one, and there is no overlap between phases.
- Rigidity: Changes to requirements or design at later stages can be challenging and costly to implement.
- Emphasis on Documentation: Extensive documentation is created at each phase, serving as a record of the project's progress.
- Suitable for Well-Defined Projects: The Waterfall model works best when the requirements are stable and well understood from the beginning.

Advantages Waterfall SDLC Model

Clear and Well-Defined Phases: The Waterfall model's linear and sequential approach provides clear and distinct phases, making it easy to understand and manage the development process.

Documentation and Traceability: The emphasis on documentation in each phase ensures a comprehensive record of project progress, making it easier to review and validate the project at different stages.

Easier Project Management: The sequential nature of the Waterfall model simplifies project management, as each phase has specific deliverables and milestones that can be easily monitored.

Well-Suited for Stable Requirements: The Waterfall model works well for projects with well-defined and stable requirements, where changes are less likely to occur during the development process.

Suitable for Small Teams: Waterfall can be suitable for small teams or projects where the requirements and scope are relatively straightforward.

Disadvantages Waterfall SDLC Model

Rigidity to Changes: The Waterfall model is inflexible to changes in requirements or design once the development process moves to the next phase. Implementing changes at later stages can be challenging and costly.

Late User Feedback: User feedback is typically collected only during the later stages, such as during user acceptance testing (UAT), which may lead to significant changes or rework at that point.

High Risk of Project Failure: The Waterfall model's lack of early testing and validation increases the risk of discovering major issues late in the development cycle, leading to project delays or failures.

Long Development Cycles: As each phase is completed before moving to the next, the overall development cycle can be lengthy, potentially delaying the delivery of the final product.

Limited Collaboration: The Waterfall model may hinder collaboration and communication between different teams and stakeholders, as there is limited interaction until the end of each phase.

Agile SDLC Model

The Agile Software Development Life Cycle (SDLC) model is an iterative and incremental approach to software development that focuses on flexibility, collaboration, and customer feedback.

The Agile SDLC model is based on the **Agile Manifesto**, which emphasizes the following values:

1. Individuals and interactions over processes and tools.
2. Working software over comprehensive documentation.
3. Customer collaboration over contract negotiation.
4. Responding to change over following a plan.

Key Characteristics of the Agile SDLC Model

Iterative Development: Agile divides the development process into small iterations or time-boxed periods called sprints, where the team works on small increments of the software and delivers potentially shippable features at the end of each iteration.

Continuous Feedback and Improvement: Agile promotes frequent reviews and feedback from stakeholders and customers throughout the development process. This enables continuous improvement and allows for early identification and resolution of issues.

Flexibility to Change: Agile embraces changing requirements and priorities, enabling the team to adapt quickly and reprioritize work based on customer feedback and evolving needs.

Collaborative Approach: Agile emphasizes close collaboration among team members, including developers, testers, product owners, and customers, fostering open communication and a shared sense of ownership.

Customer-Centric: The focus of Agile is to deliver value to customers continuously. Customer needs and feedback drive the development process, ensuring that the product meets customer expectations.

Frequent Delivery: Agile aims to deliver working software in short iterations, allowing customers to see progress and provide early feedback on features and functionality.

Common Agile Frameworks

Scrum: A popular Agile framework that organizes development into fixed-length sprints, typically two to four weeks long. Scrum includes specific roles (Product Owner, Scrum Master, Development Team), events (Daily Standup, Sprint Review, Sprint Planning), and artifacts (Product Backlog, Sprint Backlog, Increment).

Kanban: An Agile approach that visualizes the flow of work on a Kanban board, enabling teams to manage and optimize their workflow continuously.

Advantages of the Agile SDLC Model

Customer Satisfaction: Agile prioritizes customer needs and feedback, leading to higher customer satisfaction with the delivered software.

Early and Frequent Delivery: Agile allows for the delivery of working software at the end of each iteration, providing value to customers early in the development process.

Adaptability: Agile's flexibility to change enables the team to adapt to evolving requirements, reducing the risk of delivering outdated or irrelevant features.

Improved Collaboration: Agile fosters a collaborative and empowered team environment, enhancing communication and alignment among team members.

Continuous Improvement: Agile's iterative nature allows for continuous improvement of the software and development process based on feedback

Disadvantages of the Agile SDLC Model

Requires Active Customer Involvement: Agile relies heavily on customer feedback, which may require significant involvement and availability from customers throughout the development process.

Complexity in Large Projects: Scaling Agile practices to large and complex projects can be challenging, requiring additional coordination and management.

Dependency on Team Collaboration: Agile's success depends on effective collaboration among team members, and any breakdown in communication can impact project progress.

Documentation Emphasis: Agile may place less emphasis on comprehensive documentation, which could be a concern for projects with strict regulatory requirements.

Other SDLC Models

Iterative Model: The Iterative model involves repeating the development process in cycles, with each iteration producing a working version of the software. It allows for feedback and refinement in each cycle.

Spiral Model: The Spiral model combines elements of both Waterfall and iterative models. It involves risk analysis and iterative development in a spiral manner, allowing for better risk management.

V-Model (Validation and Verification Model): The V-Model is an extension of the Waterfall model that emphasizes the validation and verification of requirements and corresponding test cases at each stage.

Big Bang Model: The Big Bang model is an informal and less structured approach where development starts with little planning or requirements gathering. It is suitable for small projects or experimental prototypes.

Prototype Model: The Prototype model involves building a partial or complete working prototype of the software to gather feedback and refine requirements before full development.

Incremental Model: The Incremental model breaks the software into small, manageable increments that are developed and delivered in stages. Each increment adds new functionality to the previous one.