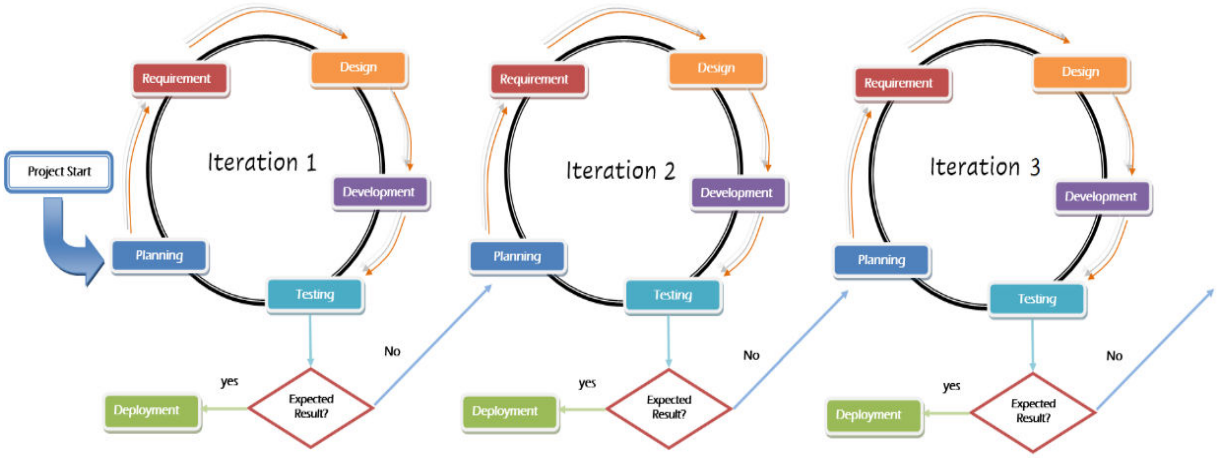
Agile development

An agile methodology is an iterative approach to software development. Each iteration of agile methodology takes a short time interval of 1 to 4 weeks. The agile development process is aligned to deliver the changing business requirement. It distributes the software with faster and fewer changes.

The single-phase software development takes 6 to 18 months. In single-phase development, all the requirement gathering and risks management factors are predicted initially.
The agile software development process frequently takes the feedback of workable product. The workable product is delivered within 1 to 4 weeks of iteration.

- The **Agile Model** is an **iterative and incremental approach** to software development where the project is **divided into small parts called "iterations" or "sprints"**, usually lasting **2 to 4 weeks**. Each sprint results in a working piece of software that is reviewed and improved in the next sprint.
- **Agile** is a **Project Management** and software development approach that aims to be more effective.
- It focuses on delivering smaller pieces of work regularly instead of one big launch.
- This allows teams to adapt to changes quickly and provide customer value faster.

Agile Model Process Phases



Types of Agile Methodologies

- Agile is a flexible framework with several approaches, each suited for different project needs. Here are some of the most common Agile methodologies:

1. Kanban

Is approach to evolutionary and incremental systems and process change for organizations. A work-in-progress limited pull system is the central mechanism to uncover system operation (or process) complications and encourage collaboration to continuously improve the system.

2. Scrum

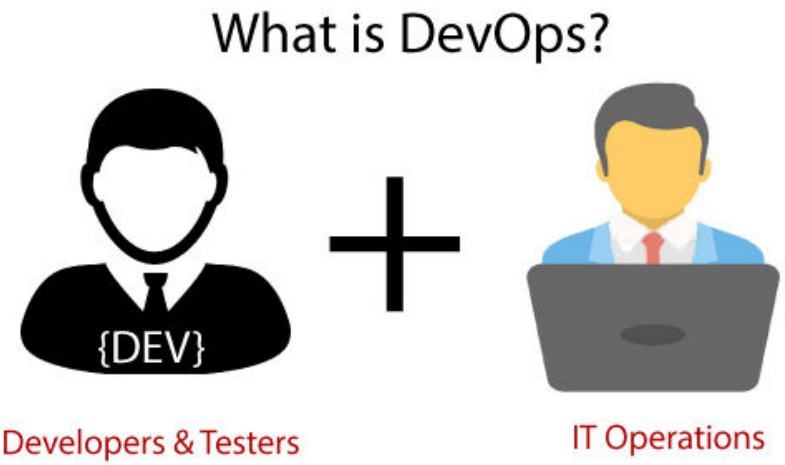
Scrum is great for small teams and works in **sprints**—short, focused work periods. A **Scrum Master** removes obstacles for the team. Scrum includes two key events:

Sprint Planning: Decides what the team will work on in the next sprint.

Sprint Retrospective: Reflects on the last sprint to improve the process for the next one.

Introduction of DevOps

The DevOps is the combination of two words, one is **Development** and other is **Operations**. It is a culture to promote the development and operation process collectively. The DevOps tutorial will help you to learn DevOps basics and provide depth knowledge of various DevOps tools such as **Git, Ansible, Docker, Puppet, Jenkins, Chef, Nagios**, and **Kubernetes**.



Why DevOps?

Before going further, we need to understand why we need the DevOps over the other methods.

- The operation and development team worked in complete isolation.
- After the design-build, the testing and deployment are performed respectively. That's why they consumed more time than actual build cycles.
- Without the use of DevOps, the team members are spending a large amount of time on designing, testing, and deploying instead of building the project.
- Manual code deployment leads to human errors in production.
- Coding and operation teams have their separate timelines and are not in synch, causing further delays.

DevOps History:

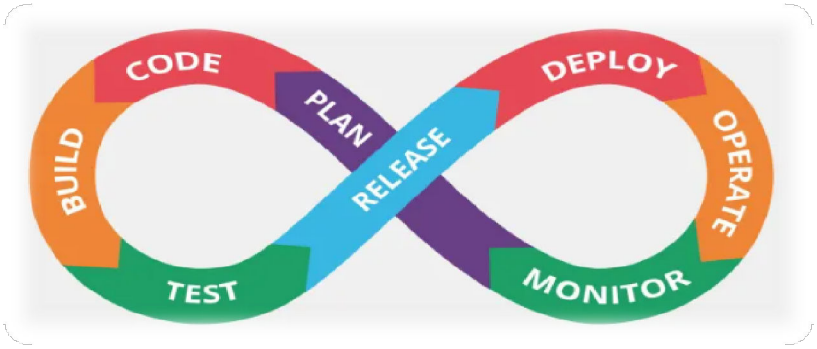
- In 2009, the first conference named **DevOpsdays** was held in Ghent Belgium. Belgian consultant and Patrick Debois founded the conference.
- In 2012, the state of DevOps report was launched and conceived by Alanna Brown at Puppet.
- In 2014, the annual State of DevOps report was published by Nicole Forsgren, Jez Humble,
  Gene Kim, and others. They found DevOps adoption was accelerating in 2014 also.
- In 2015, Nicole Forsgren, Gene Kim, and Jez Humble founded DORA (DevOps Research
  and Assignment).
- In 2017, Nicole Forsgren, Gene Kim, and Jez Humble published "Accelerate: Building and
  Scaling High Performing Technology Organizations".

Goals of DevOps:

- Faster time to market
- Continuous delivery
- Reliable and secure systems
- Automated workflows
- Collaboration across teams

The DevOps life cycle follows an **infinite loop**, representing continuous improvement and automation. The main phases are:

1. **Plan**
2. **Code**
3. **Build**
4. **Test**
5. **Release**
6. **Deploy**
7. **Operate**
8. **Monitor**



1. Plan

The planning phase is the foundation of the DevOps lifecycle, where the goals, features, and functionality of the application are defined. Teams collaborate to understand user needs, create user stories, estimate tasks, and prioritize work. Agile methodologies such as Scrum and Kanban are commonly used during this phase to ensure continuous planning, feedback, and improvement. Tools like Jira, Trello, and Confluence help in managing sprints, documenting decisions, and tracking progress efficiently.

Example: Imagine a team developing an e-commerce website. During the planning phase, they gather requirements like user login, product listing, and payment integration. These are then broken down into tasks in Jira and assigned to team members across multiple sprints. Clear communication during this stage ensures that developers, testers, and operations teams are aligned before coding begins.

2. Code

The development phase involves actual coding by developers using version control systems to collaborate and manage source code changes. The goal is to build modular, reusable, and testable code. Developers work on isolated branches using Git, and changes are regularly pushed to repositories like GitHub, GitLab, or Bitbucket. This phase supports continuous integration, where small and frequent commits are encouraged to reduce merge conflicts and bugs.

Example: Continuing with the e-commerce website, one developer works on the login module, another on the shopping cart. Both use feature branches in GitHub. Once features are completed, they raise pull requests for peer review. Automated triggers (like webhooks) can be set so that a commit automatically kicks off the build and test process using Jenkins or GitHub Actions.

3. Build

In the build phase, the source code is compiled, dependencies are resolved, and executable artifacts are generated. This process also includes static code analysis and error checking to ensure that the code is production-ready. Build tools such as Maven, Gradle, and npm automate this process. The generated builds are stored in repositories like Nexus or JFrog Artifactory for further use.

Example: Once the shopping cart code is merged, Jenkins automatically picks up the latest code, compiles it using Maven, runs unit tests, and generates a `.jar` file. If successful, the artifact is uploaded to Nexus. This ensures consistency and reliability in builds, and enables quick rollback in case of issues in future deployments.

4. Test

Testing ensures the quality and stability of the software. This phase includes multiple levels such as unit testing, integration testing, regression testing, and UI testing. The use of automation tools is encouraged for faster and repeatable testing. Tools like Selenium, JUnit, Postman, and SonarQube are commonly used to automate and measure code quality. Continuous testing ensures defects are caught early in the pipeline.

Example: After the build is generated, Selenium scripts are automatically run to simulate user actions like logging in or adding items to the cart. If any test fails (e.g., "Add to Cart" button is not clickable), the pipeline is stopped, and developers are notified to fix the issue before the code progresses to the next stage. This prevents bugs from reaching production.

5. Release

The release phase focuses on preparing the code for production deployment. It includes tagging the version, final review, documentation, and notifying stakeholders. This is often automated using CI/CD tools so that once code passes testing, it is packaged and ready to be deployed. Ensuring version control and traceability during releases is important to prevent regressions.

Example: After successful testing, Jenkins tags the latest commit as `v1.2.0`, generates a changelog, and sends notifications to the product team. The artifact is then stored in a release-ready state in the artifact repository. This version can now be deployed to staging or production, ensuring traceability of features and fixes in that release.

6. Deploy

Deployment involves moving code to live environments (such as staging, UAT, or production). Automation is used to ensure that deployments are consistent, repeatable, and can be rolled back if needed. Containerization with Docker and orchestration using Kubernetes or cloud services (like AWS ECS, Azure AKS) ensure scalability and isolation. Deployment strategies include Blue-Green, Rolling, and Canary deployments.

Example: For the e-commerce site, the app is containerized using Docker and deployed to AWS using Kubernetes. A Blue-Green deployment strategy is used: version `v1.1.0` remains active while `v1.2.0` is deployed in parallel. Once `v1.2.0` is validated, traffic is switched over. This reduces downtime and enables safe rollbacks if something goes wrong.

7. Operate

In this phase, the software is live and operational. The operations team is responsible for managing infrastructure, ensuring uptime, applying patches, and handling backups. Infrastructure as Code (IaC) tools like Ansible, Puppet, and Terraform automate infrastructure provisioning. DevOps ensures that operations are no longer reactive, but proactive and aligned with the development cycle.

Example: After deploying the app, the operations team uses Ansible playbooks to configure servers and manage updates. Kubernetes manages container health and auto-restarts failed pods. Infrastructure scaling policies (like autoscaling groups in AWS) handle traffic surges during sales or festive seasons without manual intervention.

8. Monitor

Monitoring provides visibility into system performance, application health, and user behavior. Metrics, logs, and alerts help detect issues early and allow teams to react quickly. Monitoring tools like Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), and Datadog provide dashboards, alerting, and log analysis for proactive issue resolution.

Example: Prometheus collects CPU, memory, and response time metrics from the live app. Grafana displays these in a dashboard. When latency crosses a threshold, an alert is sent via Slack or email. Logs from user activities (like login failures or payment errors) are visualized in Kibana, helping developers trace and fix issues quickly.

Phase	Purpose	Commonly Used Tools
1. Plan	Define requirements, track tasks & roadmap	**Jira**, Trello, Confluence, Asana, Notion, Azure Boards
2. Develop	Code writing, collaboration, versioning	**Git**, GitHub, GitLab, Bitbucket, VS Code, IntelliJ, Eclipse
3. Build	Compile code, manage dependencies	Maven, Gradle, Ant, Make, npm, **Jenkins**, Bazel
4. Test	Automated/manual testing, code quality	**Selenium**, JUnit, TestNG, Postman, Newman, SoapUI, SonarQube, Cypress, PyTest, Cucumber
5. Release	Prepare for deployment, versioning	**Jenkins, GitLab** CI/CD, Travis CI, Nexus, JFrog Artifactory, Helm, Git Tagging
6. Deploy	Send application to staging/production	**Docker**, Kubernetes, AWS CodeDeploy, OpenShift, Ansible, Chef, Terraform, Helm, Spinnaker
7. Operate	Manage infrastructure & runtime environment	Ansible, Puppet, Chef, Terraform, Docker Swarm, Kubernetes, **AWS** OpsWorks
8. Monitor	Monitor system/app performance & logs	Prometheus, **Grafana**, Nagios, ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, Datadog, New Relic

Compare and contrast Agile and DevOps models with examples

In today's IT industry, delivering software quickly, correctly, and with flexibility is most important. Two famous approaches used are **Agile** and **DevOps**. Although they seem similar and are often used together, they are **not the same**.

Many people get confused between Agile and DevOps. This document explains the **meaning, goals, key differences, similarities**, and **real-life examples** of both models, especially for Indian software engineers or students.

Introduction

Agile and DevOps are two popular methodologies used in the software industry to develop and deliver software faster and efficiently. Both aim to improve software quality, reduce time-to-market, and increase customer satisfaction. However, they follow different approaches and have different focus areas.

Definition of Agile

Agile is a **software development methodology** that focuses on **iterative and incremental development**. In Agile, the project is divided into **small parts called sprints**, usually 1 to 4 weeks. After every sprint, working software is delivered to the customer for feedback.

Key Features of Agile:

- Iterative development
- Regular customer feedback
- Continuous improvement
- Close collaboration within the team

Definition of DevOps

DevOps is a **combination of cultural practices, tools, and automation** that brings together **Development (Dev)** and **Operations (Ops)** teams. Its goal is to **automate the entire software delivery process** and ensure **continuous integration and continuous delivery (CI/CD)**.

Key Features of DevOps:

- Collaboration between development and operations
- Automation of build, test, and deployment
- Continuous monitoring
- Faster and reliable software delivery

Differences Between Agile and DevOps

Aspect	Agile	DevOps
Main Focus	Fast and flexible development	Fast delivery and reliable operations
Team Involved	Developers and testers	Developers, testers, and operations team
Process Type	Iterative and sprint-based	Continuous and automated
Customer Involvement	High – direct feedback from customer	Low – indirect feedback via monitoring
Release Cycle	After every sprint (1–4 weeks)	Can happen multiple times a day
Automation	Limited – mostly for testing	High – CI/CD, deployment, monitoring
Tools Used	Jira, Trello, Git, Selenium	Jenkins, Docker, Ansible, Kubernetes
Goal	Develop software in small parts with user feedback	Deliver and operate software quickly and reliably

---

Similarities Between Agile and DevOps

- Both aim for **faster delivery of software**.
- Focus on **collaboration and communication**.
- Support **continuous improvement**.
- Use of **automation tools**.
- Deliver **high-quality products**.

Real-Time Examples

✅Agile Example:

Scrum in a Web App Project

- A team building an e-commerce web app follows **2-week sprints**.
- Features like “add to cart”, “user login” are developed, tested, and demonstrated every sprint.
- Customers give feedback after each sprint review.
- Tools used: Jira (for sprint planning), Git (for version control).

✅DevOps Example:

CI/CD Pipeline for a Mobile App

- The same e-commerce app is pushed through a **Jenkins pipeline**.
- Code is automatically tested (Unit + Integration tests).
- Once tests pass, the app is built into a **Docker container**, deployed to **AWS** or **Kubernetes**.
- Operations team monitors app performance using **Prometheus + Grafana**.
- Rollbacks are automated if any deployment fails.

ITIL

ITIL is an abbreviation of **Information Technology Infrastructure Library**.

It is a framework which helps the IT professionals for delivering the best services of IT. This framework is a set of best practices to create and improve the process of ITSM (IT Service Management). It provides a framework within an organization, which helps in planning, measuring, and implementing the services of IT. The main motive of this framework is that the resources are used in such a way so that the customer get the better services and business get the profit. It is not a standard but a collection of best practices guidelines.

Service Lifecycle in ITIL

- ✓ **Service Strategy.**
- ✓ **Service Design.**
- ✓ **Service Transition.**
- ✓ **Service Operation.**
- ✓ **Continual Service Improvement.**

1. Service Strategy

Overview:
Service Strategy is the **foundation** of the ITIL lifecycle. It focuses on defining the **IT service provider's offerings**, target **customer base**, and **business value**. The aim is to ensure that IT services support business objectives by planning services strategically rather than just technically.

Processes in this stage include:

- **Service Portfolio Management**
- **Financial Management for IT Services**
- **Demand Management**
- **Business Relationship Management**

The main objective is to understand **what services to offer**, **to whom**, and **at what cost**. It's about making strategic decisions like whether to build new services, discontinue old ones, or outsource some functions. This stage ensures IT is seen as a **business partner** rather than just a support function.

In a real-world example, a telecom company might use Service Strategy to decide whether to develop a new cloud backup service based on customer demand, market trends, and cost-benefit analysis. This decision influences all the remaining ITIL stages.

1. Service Portfolio Management

Manages the complete lifecycle of all IT services—those being planned, in use, or retired. It ensures that the service provider offers the right mix of services to meet business needs and tracks their performance, cost, and value.

2. Financial Management for IT Services

Handles budgeting, accounting, and charging activities to manage IT service costs. It ensures financial transparency and helps IT align spending with business priorities.

3. Demand Management

Analyzes patterns of business activity to anticipate service demands. It ensures the right level of resources are available to meet user needs without over- or under-provisioning.

4. Business Relationship Management

Maintains a strong relationship between IT and the business. It identifies customer needs and ensures IT services deliver the desired outcomes through regular engagement.

2. Service Design

Overview:
Service Design ensures that new or changed services are designed effectively to meet business requirements. It transforms strategic objectives into **plans, policies, and specifications**. This includes everything from functionality to availability, security, and performance.

Key processes include:

- **Service Catalog Management**
- **Service Level Management**
- **Capacity Management**
- **Availability Management**
- **IT Service Continuity Management (ITSCM)**
- **Security Management**
- **Supplier Management**

This phase ensures that all components and processes required to deliver services are carefully planned and documented. For example, if the strategy decides to offer a 24/7 customer support service, Service Design would specify the infrastructure, staff, security, and backup needs to deliver it.

Well-designed services lead to fewer issues during deployment and operation. A failure in this phase might result in costly redesign or poor customer experience later. That's why Service Design is often seen as a blueprint phase—**design it right, avoid chaos later.**

1. Service Catalog Management
Maintains a centralized, up-to-date catalog of all IT services available to users, including details like service descriptions, SLAs, and availability.

2. Service Level Management (SLM)
Negotiates, documents, and manages Service Level Agreements (SLAs) between the IT provider and the business. It monitors service performance against agreed targets.

3. Capacity Management
Ensures that IT infrastructure and services have sufficient capacity to meet current and future demands in a cost-effective and timely manner.

4. Availability Management
Ensures that IT services are always available as needed by the business. It designs services for reliability and manages unplanned outages or planned downtime.

5. IT Service Continuity Management (ITSCM)
Prepares IT to recover quickly from disasters or major incidents. It ensures continuity of critical services through risk assessment and backup planning.

6. Information Security Management
Protects IT services and data from unauthorized access, breaches, and other security risks. It enforces confidentiality, integrity, and availability of information.

7. Supplier Management
Manages third-party suppliers to ensure they meet contractual obligations. It maintains good vendor relationships and tracks their service quality and performance.

3. Service Transition

Overview:
Service Transition is responsible for building and deploying IT services. It manages changes in a controlled way to minimize risk and ensures new or modified services are delivered into the live environment **smoothly and successfully**.

Major processes include:

- **Change Management**
- **Release and Deployment Management**
- **Service Asset and Configuration Management (SACM)**
- **Knowledge Management**
- **Transition Planning and Support**

This phase controls **how changes are introduced** into the environment. It's not just about coding and deployment—it includes testing, documentation, communication, and training. A good example is a bank updating its mobile app: Change Management ensures approvals, testing teams validate functionality, and Release Management handles rollout to users.

Service Transition reduces the risk of failed releases and service outages. By implementing structured change and release management, businesses can deliver innovations **without disrupting users or operations**.

1. Change Management

Controls the lifecycle of all changes in the IT environment. It ensures that changes are planned, tested, and implemented with minimal disruption to existing services.

2. Release and Deployment Management

Plans, schedules, and controls the movement of releases to testing and live environments. It ensures successful deployment and user readiness.

3. Service Asset and Configuration Management (SACM)

Maintains a detailed record of all IT assets and configurations, including their relationships and dependencies, to support effective change and incident management.

4. Knowledge Management

Ensures the right information is available to the right people at the right time. It reduces the need to rediscover knowledge and improves decision-making.

5. Transition Planning and Support

Coordinates all transition activities, resources, and schedules. It ensures new or changed services are delivered on time and within scope.

4. Service Operation

Overview:
This is the most **visible phase** to users and customers—it's where live services are monitored, maintained, and supported. The primary goal is to ensure IT services are delivered effectively and efficiently, meeting agreed service levels.

Important processes include:

- **Incident Management**
- **Problem Management**
- **Event Management**
- **Request Fulfillment**
- **Access Management**

This phase handles day-to-day IT operations. For example, if a user can't access their email, Incident Management ensures the issue is logged and resolved quickly. If multiple users face the same issue, Problem Management investigates the root cause to prevent recurrence.

Operational excellence in this phase ensures user satisfaction, business productivity, and minimum downtime. Tools like monitoring software, helpdesks, and automation play a major role here. A key principle is to **restore service quickly, not just fix everything permanently**—permanent solutions come later through Problem Management.

1. Incident Management

Restores normal service operation as quickly as possible after an unplanned interruption to minimize business impact and user inconvenience.

2. Problem Management

Finds the root causes of incidents and reduces the likelihood of recurrence. It involves both reactive problem-solving and proactive prevention.

3. Event Management

Monitors IT services and systems for significant events. It detects and interprets events to decide whether action needs to be taken.

4. Request Fulfillment

Handles standard user requests such as password resets, software installations, or information access quickly and efficiently.

5. Access Management

Grants authorized users the right to use a service and prevents unauthorized access. It enforces security policies by managing user permissions

5. Continual Service Improvement (CSI)

Overview:
CSI spans all stages and aims to improve IT services and processes **over time**. It uses performance metrics and feedback to identify what is working and what needs improvement.

Processes in CSI include:

- **Service Review**
- **Process Evaluation**
- **Definition of CSI Initiatives**
- **Monitoring of CSI Initiatives**

This stage uses the **PDCA (Plan-Do-Check-Act)** cycle for ongoing improvements. For instance, a company might measure how fast incidents are resolved and take steps to shorten resolution time. CSI analyzes trends, customer feedback, and KPI reports to find opportunities to improve service quality or reduce costs.

Continual Service Improvement ensures that the IT organization doesn't become stagnant. It promotes a culture of **learning and evolution**, keeping services aligned with changing business needs, user expectations, and technological advancements. Without CSI, even a well-running service can become obsolete or inefficient over time.

1. Service Review

Periodically assesses services and SLAs with customers to ensure their needs are being met and to identify areas for improvement.

2. Process Evaluation

Evaluates the performance of IT processes to detect inefficiencies, compliance issues, or improvement opportunities.

3. Definition of CSI Initiatives

Identifies, plans, and prioritizes improvement projects to enhance service quality and value. It sets clear goals and success measures.

4. Monitoring of CSI Initiatives

Tracks the progress and effectiveness of improvement activities. Ensures that defined goals are being met and benefits are realized.

Define Continuous Delivery. Describe Its Role in DevOps

Definition of Continuous Delivery (CD)

Continuous Delivery (CD) is a **DevOps practice** where **code changes** are automatically tested and prepared for **release to production** at any time.
In simple words:
Continuous Delivery means that the software is always **ready to be deployed** — even several times a day — with **minimum manual work**.
This allows developers to deliver updates **faster**, **safely**, and **more frequently**.

Key Features of Continuous Delivery

- Code is **automatically built, tested, and packaged**
- Code can be **released to production** at **any time**
- Focus is on **speed and stability**
- Teams can deploy **more often** with **less risk**
- Involves **automated pipelines**

Difference Between CI and CD

Term	Full Form	Meaning
CI	Continuous Integration	Developers regularly merge their code into a shared repository and test it.
CD	Continuous Delivery	The tested code is automatically made ready for deployment to production.

CI ensures code works.
CD ensures code can be deployed any time.

Steps in a Continuous Delivery Pipeline

1. **Code Commit**
   Developer writes and commits code to a version control system like Git.
2. **Build**
   Code is compiled, and all parts are combined.
3. **Automated Testing**
   Unit tests, integration tests, and UI tests are run automatically.
4. **Packaging**
   Code is packaged into a deployable format (e.g., Docker image or ZIP file).
5. **Ready for Deployment**
   The final product is stored in a place (like an artifact repository) and can be deployed to servers when needed.

Tools Used in Continuous Delivery

- **Git** – For version control
- **Jenkins**, **GitLab CI/CD**, **CircleCI** – For CI/CD automation
- **Docker** – For containerizing applications

- **Kubernetes** – For managing deployments
- **Ansible**, **Chef**, **Puppet** – For automation of environments

Role of Continuous Delivery in DevOps

Continuous Delivery is a **core part of the DevOps process**. Here's how it plays a key role:

1. Faster Releases
With CD, companies can release **new features, bug fixes, and updates quickly** — sometimes **multiple times per day**.

2. Less Risk
Every change is **tested automatically**, so there are **fewer bugs** in production. Also, if something breaks, it can be fixed and redeployed quickly.

3. Automation
CD automates the most **time-consuming tasks** like testing and deployment. This **reduces manual errors** and saves time.

4. Improves Developer Productivity
Developers don't have to wait for a long process to release their code. CD gives them **quick feedback**, so they can focus on writing better code.

5. Better Collaboration
CD encourages **developers, testers, and operations** to work together, just like DevOps promotes. Everyone is responsible for making sure the code is always deployable.

6. Continuous Improvement
With frequent releases, teams can **gather user feedback faster**, fix issues quickly, and improve the product regularly.

✓ Real-World Example

Let's say an **online shopping app** like Amazon wants to add a new feature – "Voice Search".

- Developers write and push the code.
- Automated tests run in the CD pipeline.
- The feature is built and tested automatically.
- It's ready to deploy with one click.
- If needed, it can be released to all users **on the same day**.

This is the power of Continuous Delivery in action.

Conclusion

Continuous Delivery is one of the most important parts of the **DevOps culture**. It allows teams to:

- Build better software,
- Deliver it faster,
- And keep customers happy.

By using Continuous Delivery, companies can stay **competitive**, **innovative**, and **customer-focused**.

Explain Scrum and Kanban. How Do They Differ in Managing Software Projects

ANSWER:

In software development, especially in **Agile methodology**, teams use different ways to **plan, track, and complete their work**. Two popular methods are:

- **Scrum**
- **Kanban**

Both help teams to work in an organized way and **deliver software faster**, but they have different rules and workflows.

What is Scrum?

Scrum is a type of **Agile framework** used to manage software development. It divides work into **fixed time periods** called **sprints**, usually 1 to 4 weeks long.
At the end of every sprint, the team delivers a **working software feature**.

Key Elements of Scrum

Element	Description
Sprint	A time-boxed period (e.g., 2 weeks) in which work is completed
Product Backlog	A list of all features and tasks needed in the product
Sprint Backlog	A smaller list taken from the product backlog for the current sprint
Scrum Master	A person who guides the team and removes obstacles
Product Owner	The person who represents the customer and decides what features are most important
Daily Scrum Meeting	A short 15-minute meeting held every day to check progress

Scrum Process (Step-by-step)

1. Product Owner creates a **Product Backlog**
2. Team selects tasks for the **Sprint Backlog**
3. Work begins for a sprint (e.g., 2 weeks)
4. Team meets daily in **Daily Scrum**
5. At the end, a **working feature** is delivered
6. Team holds a **Sprint Review** and **Sprint Retrospective**

What is Kanban?

Kanban is a **visual method** to manage work. It uses a **board** with columns to show the status of tasks.
Example:

- **To Do → In Progress → Testing → Done**

Work moves from one column to another as it progresses.

Key Elements of Kanban

Element	Description
Kanban Board	A board with columns showing work stages

Element	Description
Cards	Each task or feature is shown as a card
Work In Progress (WIP) Limit	Restricts the number of tasks in each stage to avoid overload
Continuous Flow	Work is pulled and delivered continuously, not in fixed time slots

Kanban Process (Step-by-step)

1. Create a **Kanban Board** with stages (To Do, In Progress, Done)
2. Add **cards** for each task or feature
3. Team members pull cards and move them across the board
4. Tasks are **finished continuously**, not after a fixed sprint

Comparison: Scrum vsKanban

Feature	Scrum	Kanban
Work Time	Fixed sprints (e.g., 2 weeks)	No fixed time; continuous flow
Planning	Planning happens at the start of every sprint	Planning is ongoing; as work comes
Roles	Scrum Master, Product Owner, Team	No fixed roles required
Meetings	Daily Scrum, Sprint Review, Retrospective	Meetings optional
Board	Sprint-based task board	Continuous Kanban board
WIP Limits	Not required	WIP limits are used to control work
Flexibility	Less flexible; fixed sprint time	More flexible; tasks can be added anytime

Example of Scrum in Use

A team building a **mobile banking app** uses Scrum:

- Sprint 1: Add login feature
- Sprint 2: Add money transfer feature
- Sprint 3: Add account statement page

After each sprint, they deliver and test the new feature.

Example of Kanban in Use

A team handling **software bug fixing** uses Kanban:

- Bugs are added to the **To Do** column
- Developers pull them to **In Progress**
- After testing, bugs move to **Done**
- This process happens **daily and continuously**

Some companies even combine both – using **Scrum with a Kanban board** (called **Scrumban**).

Conclusion

Both **Scrum and Kanban** are powerful tools to manage software projects. The main difference is:

- **Scrum** = Time-boxed + Roles + Sprints
- **Kanban** = Continuous Flow + Visual Board + WIP limits

Choosing the right one depends on the **team's needs, project type, and work style**.

What Are the Common Bottlenecks in a Software Delivery Pipeline?

Answer:

What is a Software Delivery Pipeline?

A **Software Delivery Pipeline** is a step-by-step process used to:

- **Develop**
- **Test**
- **Build**
- **Deploy**

  software from the developer's computer to the live production environment.

This pipeline includes stages like **coding → building → testing → deploying → monitoring**.

What Is a Bottleneck?

A **bottleneck** is a stage in the pipeline where the flow of work slows down or gets **stuck**. Just like a bottle has a narrow neck that limits how fast water flows out, in software delivery, a bottleneck **delays the whole process**.

Common Bottlenecks in Software Delivery Pipelines

1. Manual Code Integration

- Developers write code, but they wait too long to integrate it into the main project.
- This leads to **merge conflicts** and bugs.
- It delays testing and deployment.

Solution: Use **Continuous Integration (CI)** to automatically merge and test code regularly.

2. Slow or Manual Testing

- If testing is done manually, it takes a lot of time.
- Bugs may go unnoticed or be found too late.
- Delays the release of new features.

Solution: Use **automated testing tools** like Selenium, JUnit, etc.

3. Environment Issues

- Developers test on one environment, but production is different (e.g., different OS, settings, or databases).
- This causes unexpected bugs after deployment.

Solution: Use **Docker or virtual machines** to ensure environments are the same everywhere.

4. Long Approval or Review Time

- Code may be ready, but it waits for **manual approvals** from team leads or QA.
- This causes unnecessary delay.

Solution: Automate approvals where possible and use **code review tools** like GitHub pull requests.

5. Build Failures

- If the build system is not properly maintained, builds may often **fail or be slow**.
- Teams waste time fixing broken builds.

Solution: Use reliable build tools (like Maven, Gradle, Jenkins) and run builds in a clean, consistent environment.

6. Slow or Manual Deployment

- Some teams deploy software **manually** using copy-paste or FTP.
- This increases the chance of human error and takes more time.

Solution: Use **Continuous Deployment (CD)** tools like Jenkins, GitLab CI/CD, or Azure DevOps.

7. Poor Communication

- Developers, testers, and operations teams **do not coordinate** properly.
- This causes confusion and misalignment between code and production.

Solution: Follow **DevOps culture** to improve collaboration and use communication tools like Slack, Jira, or Trello.

8. Lack of Monitoring & Feedback

- After deployment, if the team is **not monitoring** the software, they may not notice bugs or performance issues.
- Problems go undetected for a long time.

Solution: Use **monitoring tools** like Prometheus, Grafana, New Relic to track performance and fix issues quickly.

9. Security Approvals or Checks

- Security reviews are sometimes done **at the very end** of development.
- If issues are found late, teams must go back and fix them, delaying the release.

Solution: Use **DevSecOps** – add security checks early in the pipeline (e.g., automated vulnerability scanners).

10. Too Many Parallel Tasks

- Developers or teams work on **too many features at the same time**.
- This divides attention and causes delays in completing any one feature.

Solution: Use **Work In Progress (WIP) limits** like in Kanban to keep focus.

Summary Table: Bottlenecks and Solutions

Bottleneck	Cause	Solution
Manual Integration	Late merging of code	Use Continuous Integration
Manual Testing	Slow and error-prone	Automate testing
Different Environments	Bugs in production	Use Docker or similar tools
Waiting for Approvals	Human delay	Automate and use review tools
Build Failures	Poor build setup	Reliable CI tools like Jenkins
Manual Deployment	Time-consuming	Continuous Deployment
Poor Team Communication	Silos between teams	Follow DevOps culture
No Monitoring	Delayed issue detection	Use monitoring tools
Late Security Checks	Last-minute risks	Implement DevSecOps
Too Much Work	Lack of focus	Use WIP limits in Kanban

Bottlenecks in the software delivery pipeline **slow down the entire software release process** and increase the risk of **bugs, delays, and unhappy customers**.
Using **DevOps tools, automation, and good team practices**, these bottlenecks can be **identified and removed** to make the pipeline **faster, smoother, and more reliable**.