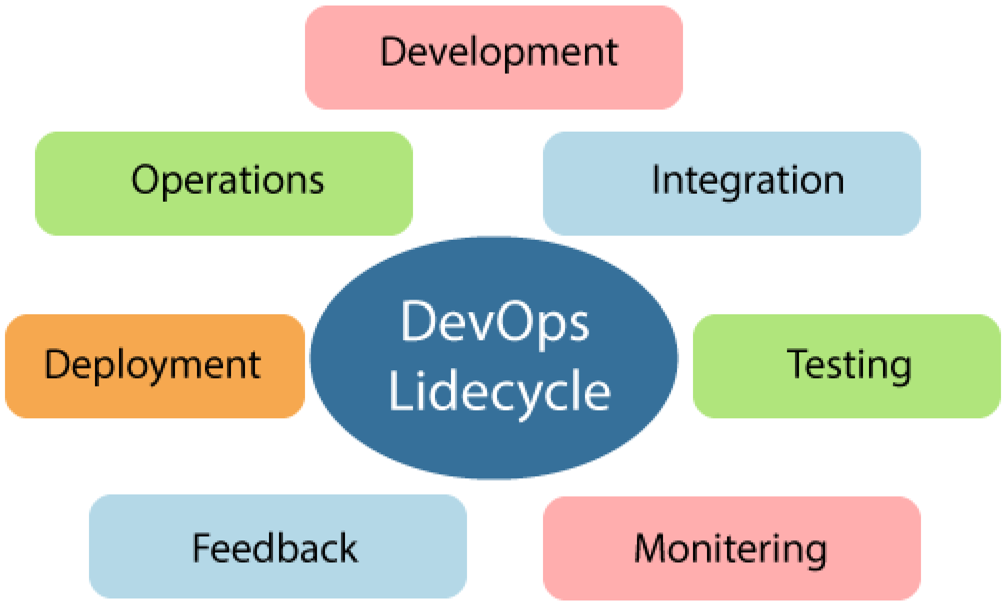
UNIT-2

Software development models and DevOps: DevOps Lifecycle for Business Agility, DevOps, and Continuous Testing. DevOps influence on Architecture: Introducing software architecture, The monolithic scenario, Architecture rules of thumb, The separation of concerns, Handling database migrations, Microservices, and the data tier, DevOps, architecture, and resilience.

DevOps Lifecycle for Business Agility:

DevOps defines an agile relationship between operations and Development. It is a process that is practiced by the development team and operational engineers together from beginning to the final stage of the product.

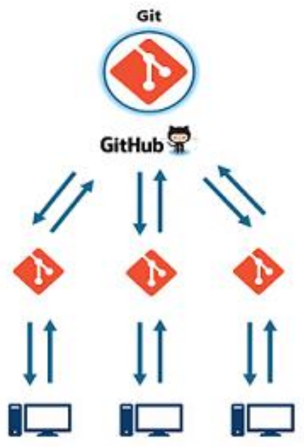


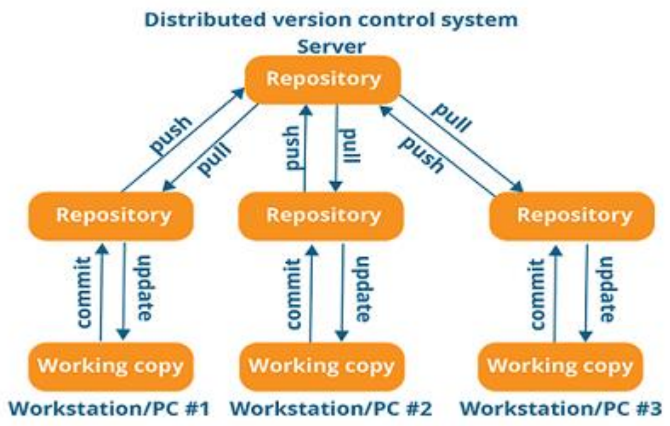
Learning DevOps is not complete without understanding the DevOps lifecycle phases. The DevOps lifecycle includes seven phases as given below:

1) Continuous Development

This phase involves the planning and coding of the software. The vision of the project is decided during the planning phase. And the developers begin developing the code for the application. There are no DevOps tools that are required for planning, but there are several tools for maintaining the code.

The code can be written in any language, but it is maintained by using Version Control tools. Maintaining the code is referred to as Source Code Management. The most popular tools used are Git, SVN, Mercurial, CVS, and JIRA. Also tools like Ant, Maven, Gradle can be used in this phase for building/ packaging the code into an executable file that can be forwarded to any of the next phases.





Now let us try to know a bit more about Git.

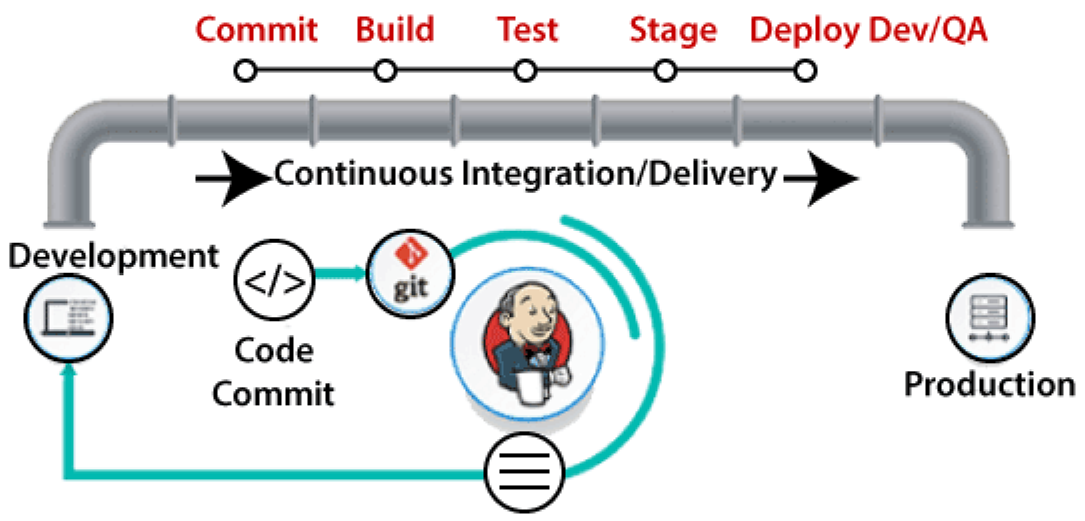
- Git is a distributed version control tool that supports distributed non-linear workflows by providing data assurance for developing quality software. Tools like Git enable communication between the development and the operations team.
- When you are developing a large project with a huge number of collaborators, it is very important to have communication between the collaborators while making changes in the project.
- Commit messages in Git play a very important role in communicating among the team. Apart from communication, the most important reason to use Git is that you always have a stable version of the code with you.
- Hence, Git plays a vital role in succeeding at DevOps.

2) Continuous Integration

This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. This may be on a daily or weekly basis. Then every commit is built, and this allows early detection of problems if they are present. Building code is not only involved compilation, but it also includes **unit testing, integration testing, code review**, and **packaging**.

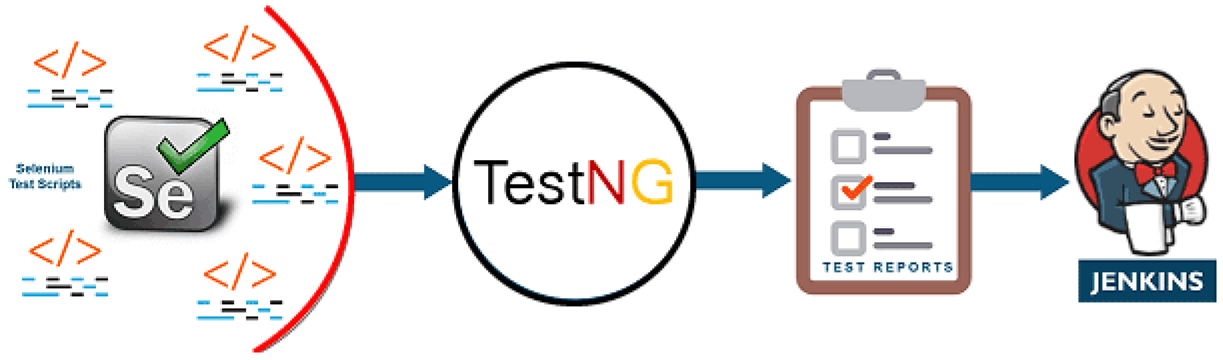
The code supporting new functionality is continuously integrated with the existing code. Therefore, there is continuous development of software. The updated code needs to be integrated continuously and smoothly with the systems to reflect changes to the end-users.

Jenkins is a popular tool used in this phase. Whenever there is a change in the Git repository, then Jenkins fetches the updated code and prepares a build of that code, which is an executable file in the form of war or jar. Then this build is forwarded to the test server or the production server.



3) Continuous Testing

This phase, where the developed software is continuously testing for bugs. For constant testing, automation testing tools such as **TestNG, JUnit, Selenium**, etc are used. These tools allow QAs to test multiple code-bases thoroughly in parallel to ensure that there is no flaw in the functionality. In this phase, **Docker** Containers can be used for simulating the test environment.



Selenium does the automation testing, and TestNG generates the reports. This entire testing phase can automate with the help of a Continuous Integration tool called **Jenkins**.

Automation testing saves a lot of time and effort for executing the tests instead of doing this manually. Apart from that, report generation is a big plus. The task of evaluating the test cases that failed in a test suite gets simpler. Also, we can schedule the execution of the test cases at predefined times. After testing, the code is continuously integrated with the existing code.

4) Continuous Monitoring

Monitoring is a phase that involves all the operational factors of the entire DevOps process, where important information about the use of the software is recorded and carefully processed to find out trends and identify problem areas. Usually, the monitoring is integrated within the operational capabilities of the software application.

It may occur in the form of documentation files or maybe produce large-scale data about the application parameters when it is in a continuous use position. The system errors such as server not reachable, low memory, etc are resolved in this phase. It maintains the security and availability of the service.

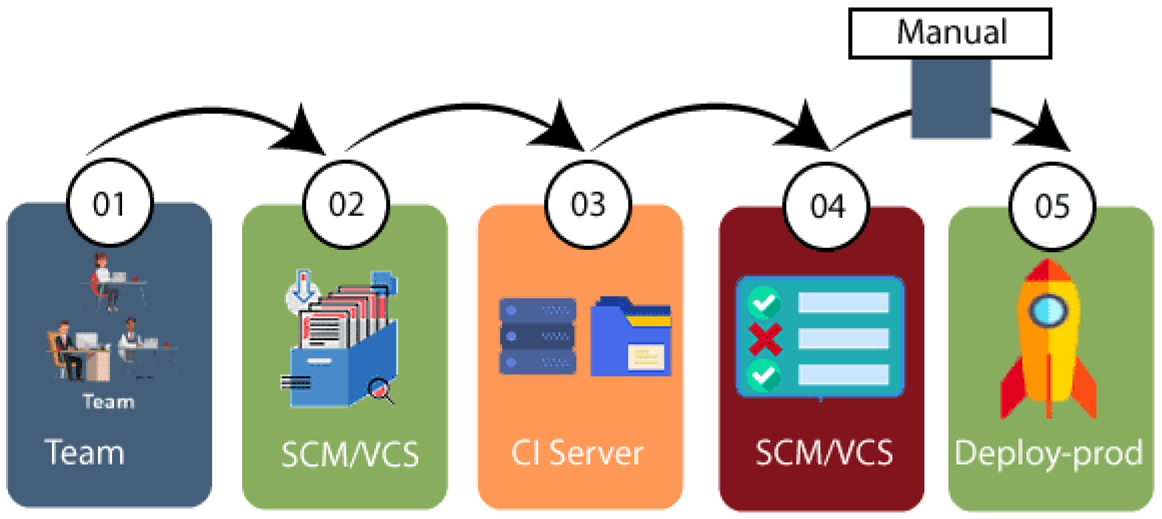
5) Continuous Feedback

The application development is consistently improved by analyzing the results from the operations of the software. This is carried out by placing the critical phase of constant feedback between the operations and the development of the next version of the current software application.

The continuity is the essential factor in the DevOps as it removes the unnecessary steps which are required to take a software application from development, using it to find out its issues and then producing a better version. It kills the efficiency that may be possible with the app and reduce the number of interested customers.

6) Continuous Deployment

In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers.



The new code is deployed continuously, and configuration management tools play an essential role in executing tasks frequently and quickly. Here are some popular tools which are used in this phase, such as **Chef, Puppet, Ansible**, and **SaltStack**.

Containerization tools are also playing an essential role in the deployment phase. **Vagrant** and **Docker** are popular tools that are used for this purpose. These tools help to produce consistency across development, staging, testing, and production environment. They also help in scaling up and scaling down instances softly.

Containerization tools help to maintain consistency across the environments where the application is tested, developed, and deployed. There is no chance of errors or failure in the production environment as they package and replicate the same dependencies and packages used in the testing, development, and staging environment. It makes the application easy to run on different computers.

7) Continuous Operations

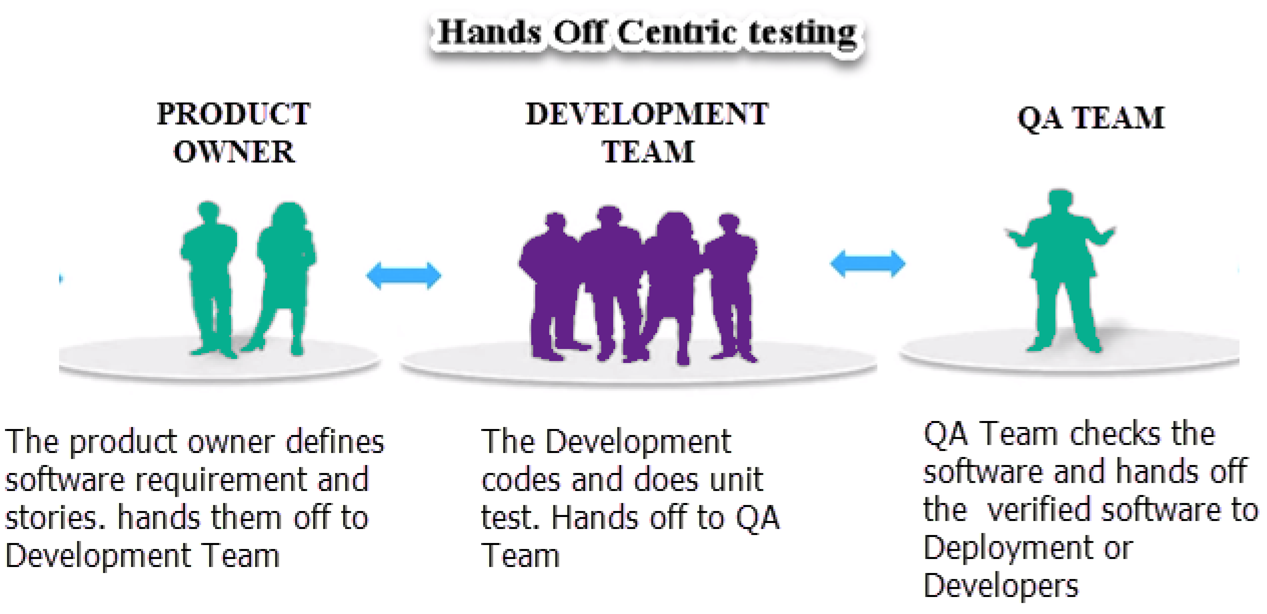
All DevOps operations are based on the continuity with complete automation of the release process and allow the organization to accelerate the overall time to market continuingly.

It is clear from the discussion that continuity is the critical factor in the DevOps in removing steps that often distract the development, take it longer to detect issues and produce a better version of the product after several months. With DevOps, we can make any software product more efficient and increase the overall count of interested customers in your product.

DevOps, and Continuous Testing

Continuous Testing in DevOps is a software testing type that involves testing the software at every stage of the software development life cycle. The goal of Continuous testing is evaluating the quality of software at every step of the Continuous Delivery Process by testing early and testing often.
The Continuous Testing process in DevOps involves stakeholders like Developer, DevOps, QA and Operational system.

How is Continuous Testing different?

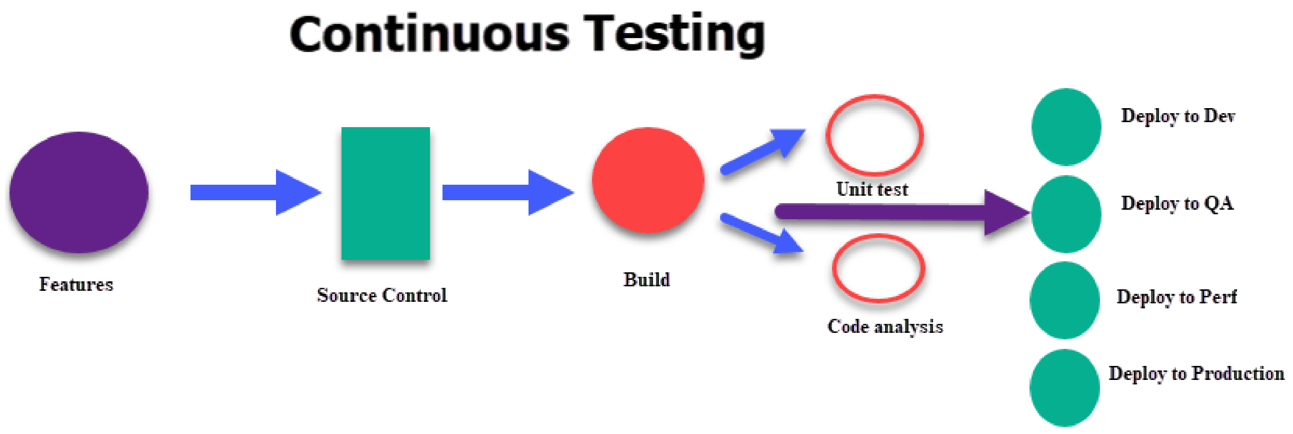

Hands Off Centric testing
PRODUCT OWNER
DEVELOPMENT TEAM
QA TEAM
The product owner defines software requirement and stories. hands them off to Development Team
The Development codes and does unit test. Hands off to QA Team
QA Team checks the software and hands off the verified software to Deployment or Developers

Continuous Testing

The old way of testing was hand off centric. The software was handed off from one team to another. A project would have definite Development and QA phases. QA teams always wanted more time to ensure quality. The goal was that the quality should prevail over project schedule.

However, business wants faster delivery of software to the end user. The newer is the software, the better it can be marketed and increase revenue potential of the company. Hence, a new way of testing was evolved.

Continuous means undisrupted testing done on a continuous basis. In a Continuous DevOps process, a software change (release candidate) is continuously moving from Development to Testing to Deployment.

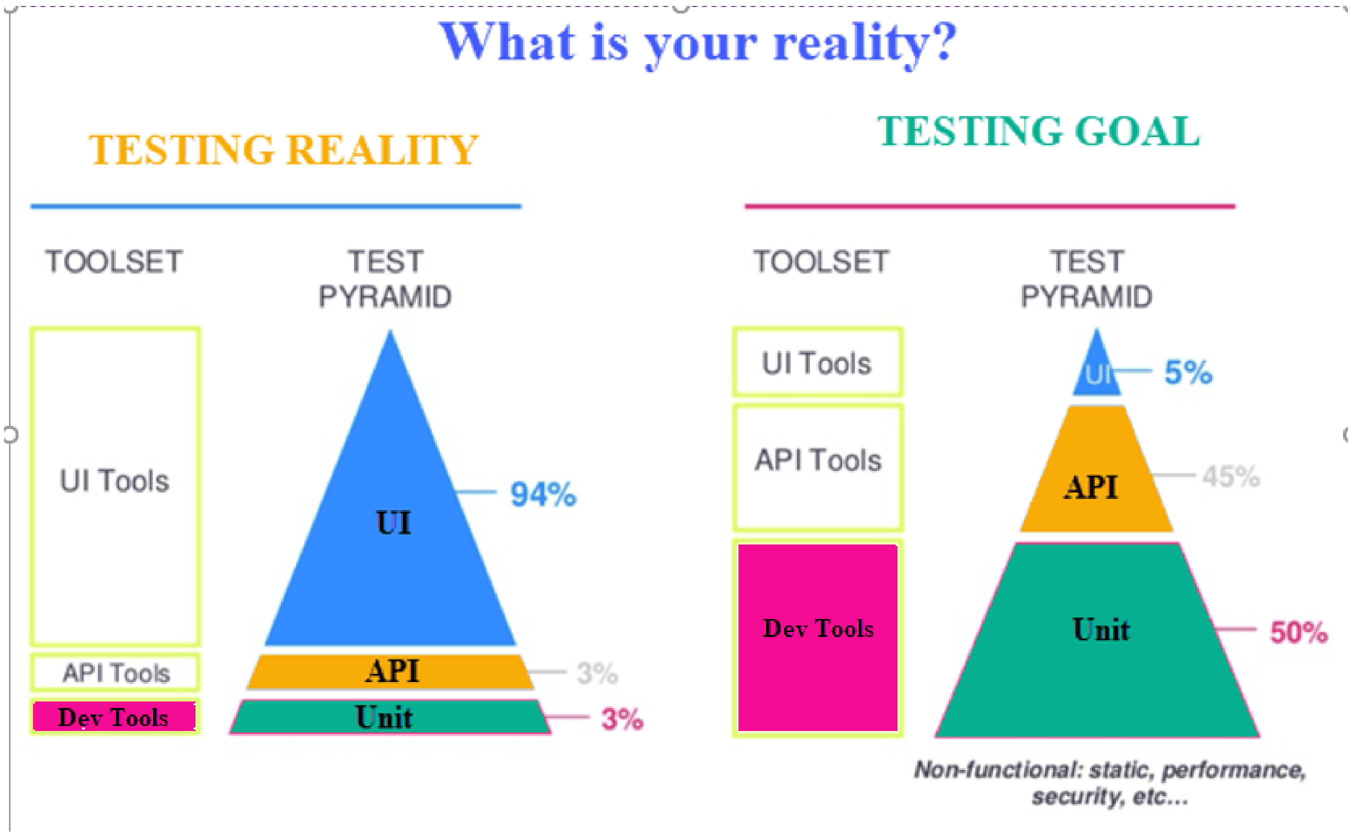


Continuous DevOps Process

The code is continuously developed, delivered, tested and deployed.

For Example, whenever a developer checks the code in the Source Code Server like Jenkins automated set of unit tests are executed in the continuous process. If the tests fail, the build is rejected, and the developer is notified. If the build passes the test, it is deployed to performance, QA servers for exhaustive functional and load tests. The tests are run in parallel. If the tests pass, the software is deployed in production.

Continuous Testing is a small cog in the Continuous Development, Integration and Deployment Cycle.



Current Testing Stack

Software development is not same as it is in the past we cut development from month to weeks. The current testing stack (see figure above) is titled towards UI testing. But the goal is to have more and more automated unit tests.

How Is Continuous Testing Different from Test Automation?

Test automation vs Continuous Testing

Parameter	**Test Automation**	**Continuous Testing**
Definition	Test automation is a process where tool or software is used for automating tasks.	It is a software testing methodology which focuses on achieving continuous quality & improvement.
Purpose	A set of similar or repetitive tasks, a machine can execute, faster, with a fewer mistake.	The continuous testing process helps to find the risk, address them and improve the quality of the product.
Prerequisite	Automation in testing possible without integrating continuous testing.	Continuous testing can not be implemented without test automation.

Parameter	Test Automation	Continuous Testing
Time	Software release can take a month to years.	Software release may be released weekly to hourly.
Feedback	Regular feedback after testing each release.	Feedback at each stage needs to be instant.
History	Automated testing has been done for decades to make the testing process faster.	Continuous testing is a relatively newer concept.

Continuous Testing Tools

Here is a curated list of best **Continuous Testing Tools** :

1) QuerySurge

QuerySurge is the smart data testing solution that is the first-of-its-kind full DevOps solution for continuous data testing. Key features include Robust API with 60+ calls, detailed data intelligence & data analytics, seamless integration into the DevOps pipeline for continuous testing, and verifies large amounts of data quickly.

2) Jenkins

Jenkins is a Continuous Integration tool which is written using Java language. This tool can be configured via GUI interface or console commands.

3) Travis

Travis is continuous testing tool hosted on the GitHub. It offers hosted and on-premises variants. It provides a variety of different languages and a good documentation.

4) Selenium

Selenium is open-source software testing tool. It supports all the leading browsers like Firefox, Chrome, IE, and Safari. Selenium WebDriver is used to automate web application testing.

Benefits of Continuous Testing

- **Find errors:** Ensure as many errors are found before being released to production
- **Test early and often:** Tested throughout the development, delivery, testing, and deployment cycles
- **Accelerate testing:** Run parallel performance tests to increase testing execution speed
- **Earn customer loyalty:** Accomplish continuous improvement and quality
- **Automation:** Automate your test cases to decrease time spent testing

- **Increase release rate:** Speed up delivery to production and release faster
- **Reduce business risks:** Assess potential problems before they become an actual problem
- **DevOps:** Incorporates into your DevOps processes smoothly
- **Communication transparency:** Eliminate silos between the development, testing, and operations teams
- **Available testing tools:** Available tools that support continuous testing to make the testing process easier, faster, and more reliable

While continuous testing has a myriad of key benefits, there are several challenges that software development teams must take into consideration:

- **Adjust to DevOps**: Professionals don't process the right tools and training for continuous testing within Agile and DevOps environments
- **Change in culture**: Cultural shifts among your development and testing teams may happen if traditional processes are maintained
- **Update testing strategy**: Maintaining only traditional testing methods and test data management that is not clearly defined keeps continuous testing from reaching its full potential
- **Code integration**: Developers who don't integration their code on a regular basis (recommended several times daily) create defect issues with duplicated coding efforts and non-compatible code
- **Test environments**: Make sure your test environments work within your code repository base for seamless testing of the newest available code
- **Production environments**: Also, make sure your production environments reflect the test environment to ensure every area was properly tested

DevOps influence on Architecture: Introducing software architecture

Introducing software architecture is a critical aspect of DevOps, as it lays the foundation for a resilient, scalable, and maintainable system. Software architecture refers to the high-level design of a system, including its structure, components, and interactions between them.

In DevOps, software architecture is typically developed collaboratively by multiple teams, including developers, operations personnel, and security professionals. This ensures that the architecture is designed to meet the needs of all stakeholders and is aligned with the organization's overall goals.

There are several key principles that should be considered when designing software architecture in a DevOps context, including:

Scalability: The architecture should be designed to accommodate growth and changing needs, including increased usage, new features, and changing requirements.

Resilience: The architecture should be designed to withstand failures and disruptions, including hardware failures, software bugs, and security breaches.

Maintainability: The architecture should be designed to be easy to maintain and update over time, including adding new features or fixing bugs.

Security: The architecture should be designed with security in mind, including authentication, access control, and data encryption.

Flexibility: The architecture should be designed to be flexible and adaptable, including the ability to integrate with third-party systems and technologies.

By following these principles and other best practices, DevOps teams can create software architectures that are robust, scalable, and maintainable, and that can support the needs of their organization over time.

The Monolithic Scenario:

IThe Monolithic Scenario is a traditional approach to software architecture where all components of an application are built as a single, integrated unit. In this scenario, the application typically consists of a single codebase and a single deployment artifact, and all components are tightly coupled together.

In recent years, there has been a shift away from the monolithic scenario towards more modular, service-oriented architectures that are better suited to the needs of modern, cloud-based systems. However, many organizations still use monolithic architectures for legacy applications or for simpler systems that do not require the scalability and flexibility of a microservices architecture.

In a DevOps context, the monolithic scenario can present several challenges, including:

Limited scalability: Monolithic architectures can be difficult to scale, as all components are tightly coupled together and cannot be easily scaled independently.

Limited flexibility: Monolithic architectures can be difficult to update or modify, as changes to one component may impact other components.

Limited resilience: Monolithic architectures can be more prone to failures or disruptions, as all components are tightly coupled together and a failure in one component can impact the entire system.

To address these challenges, DevOps teams can use techniques such as modular design, containerization, and automated testing and deployment to make monolithic architectures more scalable, flexible, and resilient. For example, by using containers to isolate components, teams can more easily scale individual components independently, and by using automated testing and deployment, teams can reduce the risk of errors or disruptions during updates or changes to the system.

Architecture Rules of Thumb

Architecture Rules of Thumb are guidelines or best practices that can help DevOps teams make important architectural decisions for their systems. These rules are based on experience and industry best practices, and can help teams avoid common pitfalls and ensure that their systems are robust, scalable, and maintainable.

Some examples of Architecture Rules of Thumb that are influenced by DevOps include:

Avoid vendor lock-in: Teams should avoid relying too heavily on proprietary technologies that may limit their ability to switch vendors or technologies in the future.

Use a modular architecture: A modular architecture can make it easier to add new features or functionality to a system, as well as to maintain and scale the system over time.

Use automation tools to manage the deployment process: Automation tools can help ensure that deployments are consistent and reliable, while also reducing the risk of human error.

Design for resilience and failure: Teams should design their systems to be resilient to failures and able to recover quickly in the event of an outage or disruption.

Monitor and measure everything: Teams should track and measure all aspects of their systems, from performance and availability to security and compliance, in order to identify and address issues proactively.

By following these rules of thumb and other best practices, DevOps teams can create systems that are more robust, scalable, and maintainable, while also reducing the risk of downtime, data loss, or security breaches.

Handling Database Migrations:

Handling database migrations is an important aspect of DevOps architecture, as it enables teams to make changes to the database schema in a safe and efficient manner. When making changes to the database schema, it's important to consider the impact that these changes may have on the overall functionality of the system. DevOps promotes practices that enable smooth database migrations, such as:

Versioning the database schema: Versioning the database schema involves keeping track of changes to the schema over time. This enables teams to deploy changes to the schema in a controlled manner, and to roll back changes if necessary.

Testing migrations in a non-production environment: Testing migrations in a non-production environment is a best practice, as it enables teams to identify and fix any issues before deploying the changes to the production environment.

Using automated tools to manage the migration process: Using automated tools to manage the migration process can help ensure that the migration is performed consistently and reliably.

In addition to these practices, it's important to communicate any database schema changes to all stakeholders, including developers, testers, and operations teams. This helps ensure that everyone is aware of the changes and can prepare accordingly. By using these practices, teams can deploy changes to the database more quickly and safely, and can reduce the risk of data loss or corruption.

Microservices and the Data Tier:

Microservices architecture is an alternative approach to software architecture that emphasizes modularity and service-oriented design. In a microservices architecture, the application is divided into smaller, independent services, each responsible for a specific functionality. These services can be developed, tested, and deployed independently, allowing for greater flexibility and scalability.

When it comes to the data tier in a microservices architecture, there are several key considerations that DevOps teams need to keep in mind:

Data storage: In a microservices architecture, each service may have its own database, which can lead to data redundancy and inconsistency. To address this, DevOps teams can use techniques such as data replication, event-driven architectures, and data aggregation to ensure that data is consistent and up-to-date across all services.

Data access: In a microservices architecture, different services may require access to the same data, which can lead to data silos and duplication. To address this, DevOps teams can use techniques such as API gateways, data virtualization, and service meshes to provide secure and efficient access to data across all services.

Data security: In a microservices architecture, each service may have its own security model, which can make it difficult to ensure consistent and effective data security across all services. To address this, DevOps teams can use techniques such as identity and access management, data encryption, and security testing to ensure that data is protected throughout the system.

By considering these and other factors when designing and implementing the data tier in a microservices architecture, DevOps teams can create systems that are more scalable, flexible, and resilient, and that can support the needs of their organization over time.

DevOps, architecture, and resilience

DevOps plays an important role in ensuring the resilience of software architecture. Resilience refers to the ability of a system to withstand and recover from disruptions, such as hardware failures, software bugs, or cyber attacks. By integrating DevOps practices into the software development and deployment process, teams can improve the resilience of their architecture in several ways:

Continuous testing and monitoring: DevOps teams can use tools such as automated testing, code reviews, and monitoring to detect and address potential issues early in the development process. This can help prevent problems from escalating into full-blown failures later on.

Continuous delivery and deployment: DevOps teams can use techniques such as continuous delivery and deployment to ensure that new features and updates are rolled out smoothly and with minimal disruption to users. This can help reduce the risk of downtime and user frustration.

Infrastructure as code: DevOps teams can use infrastructure as code (IaC) techniques to automate the provisioning and management of infrastructure resources, such as servers and databases. This can help ensure that the infrastructure is always up-to-date and configured correctly, reducing the risk of configuration errors or security vulnerabilities.

Collaboration and communication: DevOps teams can foster a culture of collaboration and communication between development and operations teams, which can help ensure that everyone is working towards the same goals and that issues are addressed quickly and efficiently.

By incorporating these and other DevOps practices into their architecture design and implementation, teams can create more resilient systems that can withstand disruptions and continue to deliver value to their users over time.

Introducing software architecture

Software architecture is the high-level structure and design of a software system. It involves making decisions about the system's components, their interactions, and the technologies used. DevOps has a significant influence on software architecture, as it promotes practices that enable agility, scalability, and resilience. By using DevOps practices, software teams can create software systems that are easier to maintain, scale, and update.

The monolithic scenario
The traditional approach to software architecture involves building monolithic applications, where all the code is in a single codebase. This approach can lead to problems with scaling, testing, and maintenance. As the application grows larger, it becomes more difficult to manage, and changes to the code can impact the entire application. DevOps promotes breaking down monolithic applications into smaller, independent components, such as microservices. This approach enables teams to work more independently on different parts of the system and to deploy changes more quickly and safely.

Architecture rules of thumb
DevOps emphasizes certain rules of thumb in software architecture, such as separation of concerns, loose coupling, and high cohesion. These principles help ensure that different parts of the system can be developed, tested, and deployed independently, without affecting the overall functionality of the system. Separation of concerns means that each component of the system should have a well-defined responsibility, and should not be overly dependent on other components. Loose coupling means that the components should be able to interact without being tightly coupled to each other. High cohesion means that the components should have a strong relationship to each other, based on a common purpose or goal.

The separation of concerns
The separation of concerns is a fundamental principle in software development, and is a key aspect of DevOps architecture. This approach involves breaking down complex systems into smaller and more manageable components, each with its own well-defined responsibilities. By separating concerns, teams can work more independently on different parts of the system without impacting the overall functionality of the system. This enables teams to deploy changes more quickly and safely, and makes it easier to maintain and scale the system.

Handling database migrations
Database migrations can be a challenging aspect of software development, as changes to the database schema can impact the entire application. DevOps promotes practices that enable smooth database migrations, such as versioning the database schema, testing migrations in a non-production environment, and using automated tools to manage the migration process. By using these practices, teams can deploy changes to the database more quickly and safely, and can reduce the risk of data loss or corruption.

Microservices and the data tier
Microservices architecture involves breaking down a large application into smaller, independent components, each with its own well-defined responsibilities. DevOps promotes the use of microservices architecture for scalability, flexibility, and resilience. When designing microservices, it's important to consider the data tier, as each microservice may have its own data store. DevOps promotes practices that enable data consistency and resilience, such as using event-driven architecture

and distributed transactions. By using these practices, teams can ensure that the data is consistent across all microservices, even in the event of failures or disruptions.

DevOps, architecture, and resilience

DevOps promotes practices that enable software systems to be resilient, or able to recover quickly from failures or disruptions. Resilience is an important aspect of software architecture, as it helps ensure that the system can handle unexpected events, such as hardware failures, network outages, or cyber attacks. DevOps promotes practices such as continuous monitoring, automated testing, and rapid deployment, which help ensure the resilience of the system. By using these practices, teams can ensure that their software systems are reliable and can recover quickly from any disruptions.