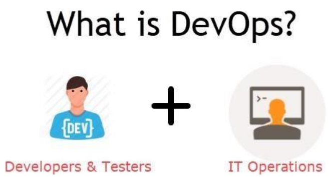
Unit-I

Fundamentals of DevOps: Architecture, Deployments, Orchestration, Need, Instance of Applications, DevOps delivery pipeline, DevOps eco system.
DevOps adoption in projects: Technology aspects, Agiling capabilities, Tool stack implementation, People aspect, processes

What is DevOps?

DevOps is a collaboration between Development and IT Operations to make software production and Deployment in an automated & repeatable way. DevOps helps increase the organization‟s speed to deliver software applications and services. The full form of „DevOps‟ is a combination of „Development‟ and „Operations.‟
It allows organizations to serve their customers better and compete more strongly in the market.
In simple words, DevOps can be defined as an alignment of development and IT operations with better communication and collaboration.



Why is DevOps Needed?

- Before DevOps, the development and operation team worked in complete isolation.
- Testing and Deployment were isolated activities done after design-build. Hence they consumed more time than actual build cycles.
- Without using DevOps, team members spend a large amount of their time testing, deploying, and designing instead of building the project.
- Manual code deployment leads to human errors in production.
- Coding & operation teams have separate timelines and are not synch, causing further delays.

There is a demand to increase the rate of software delivery by business stakeholders. As per Forrester Consulting Study, Only 17% of teams can use delivery software quickly, proving the pain point.

How is DevOps different from traditional IT

In this DevOps training, let‟s compare the traditional software waterfall model with DevOps to understand the changes DevOps brings.

We assume the application is scheduled to go live in 2 weeks, and coding is 80% done. We assume the application is a fresh launch, and the process of buying servers to ship the code has just begun-

Old Process	**DevOps**
After placing an order for new servers, the Development team works on testing. The Operations team works on extensive paperwork as required in enterprises to deploy the infrastructure.	After placing an order for new servers Development and Operations team work together on the paperwork to set up the new servers. This results in better visibility of infrastructure requirements.
Projections about failover, redundancy, data center locations, and storage requirements are skewed as no inputs are available from developers who have deep knowledge of the application.	Projections about failover, redundancy, disaster recovery, data center locations, and storage requirements are pretty accurate due to the inputs from the developers.
The operations team has no clue about the progress of the Development team. The operations team develops a monitoring plan as per their understanding.	In DevOps, the Operations team is completely aware of the developers‟ progress. Operations teams interact with developers and jointly develop a monitoring plan that caters to IT and business needs. They also use advanced Application Performance Monitoring (APM) Tools.
Before going go-live, the load testing crashes the application, and the release is delayed.	Before going go-live, the load testing makes the application a bit slow. The development team quickly fixes the bottlenecks, and the application is released on time.

Why is DevOps used?

DevOps allows Agile Development Teams to implement Continuous Integration and Continuous Delivery, which helps them launch products faster into the market.

Other Important reasons are:

1. Predictability: DevOps offers a significantly lower failure rate of new releases.

2. Reproducibility: Version everything so that earlier versions can be restored anytime.

3. Maintainability: Effortless recovery process in the event of a new release crashing or disabling the current system.

4. Time to market: DevOps reduces the time to market up to 50% through streamlined software delivery. It is particularly the case for digital and mobile applications.

5. Greater Quality: DevOps helps the team improve application development quality by incorporating infrastructure issues.

6. Reduced Risk: DevOps incorporates security aspects in the software delivery lifecycle, and it helps reduce defects across the lifecycle.

7. Resiliency: The Operational state of the software system is more stable, secure, and changes are auditable.

8. Cost Efficiency: DevOps offers cost efficiency in the software development process, which is always an aspiration of IT management.

9. Breaks larger code base into small pieces: DevOps is based on the agile programming method. Therefore, it allows breaking larger codebases into smaller and manageable chunks.

When to adopt DevOps?

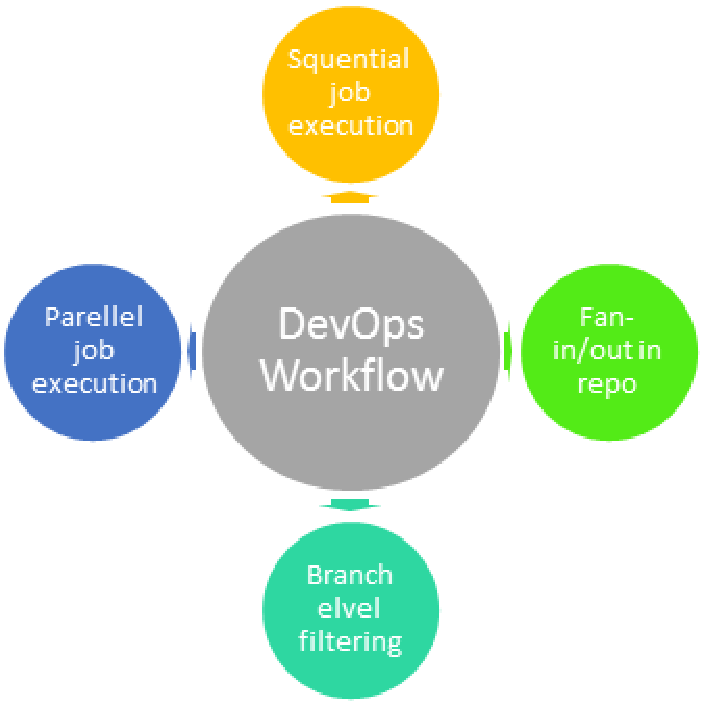
DevOps should be used for large distributed applications such as eCommerce sites or applications hosted on a cloud platform.

When not to adopt DevOps?

It should not be used in mission-critical applications like banks, power and other sensitive data sites. Such applications need strict access controls on the production environment, a detailed change management policy, and access control policy to the data centres.

DevOps Workflow

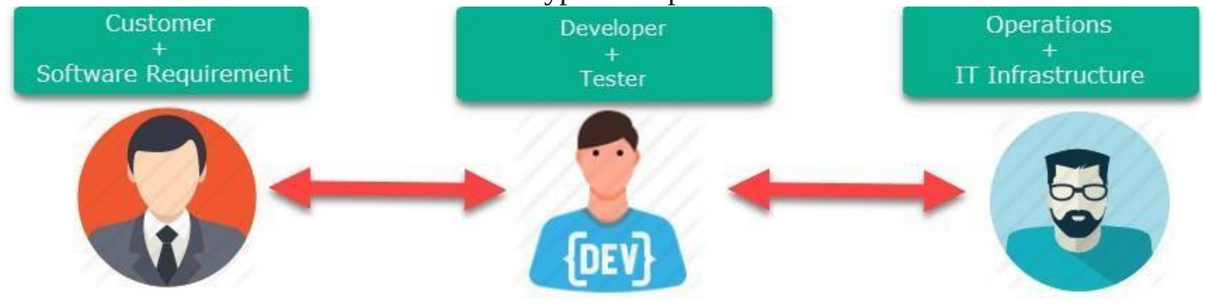
Workflows provide a visual overview of the sequence in which input is provided. It also tells about performed actions, and output is generated for an operations process.



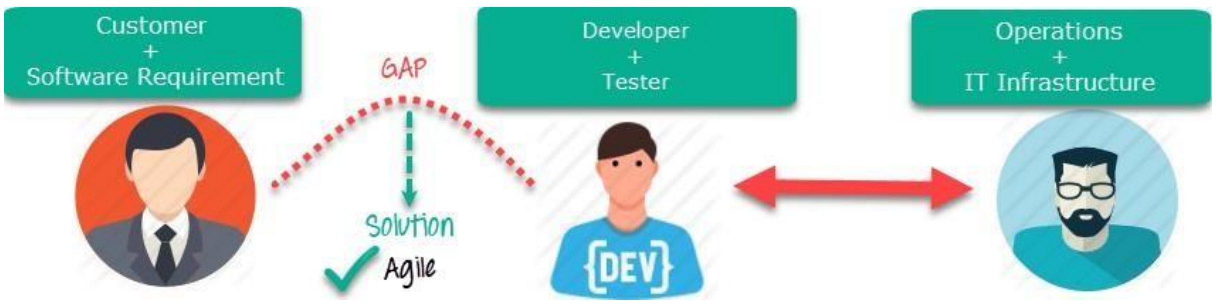
DevOps WorkFlow
Workflow allows the ability to separate and arrange jobs that the users top request. It also can mirror their ideal process in the configuration jobs.

How is DevOps different from Agile? DevOps Vs Agile

Stakeholders and communication chain a typical IT process.

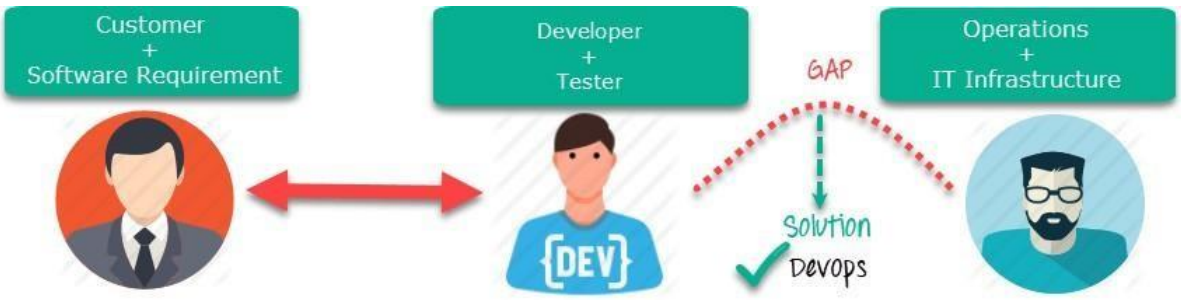


Agile addresses gaps in Customer and Developer communications



Agile Process

DevOps addresses gaps in Developer and IT Operations communications



DevOps Process

Difference between DevOps and Agile

Agile	**DevOps**
Emphasize breaking down barriers between developers and management.	DevOps is about software deployment and operation teams.

Addresses gaps between customer requirements and development teams.	Addresses the gap between the development and Operation team
Focuses more on functional and non-functional readiness	It focuses on operational and business readiness.
Agile development pertains mainly to the company‟s way development is thought out.	DevOps emphasises deploying software in the most reliable and safest ways that aren‟t always the fastest.
Agile development emphasises training all team members to have varieties of similar and equal skills. So that, when something goes wrong, any team member can get assistance from any member in the absence of the team leader.	DevOps likes to divide and conquer, spreading the skill set between the development and operation teams. It also maintains consistent communication.
Agile development manages on "sprints". It means that the timetable is much shorter (less than a month), and several features are to be produced and released in that period.	DevOps strives for consolidated deadlines and benchmarks with significant releases rather than smaller and more frequent ones.

DevOps Principles

Here are six principles that are essential when adopting DevOps:

1. Customer-Centric Action: The DevOps team must constantly take customer-centric action to invest in products and services.

2. End-To-End Responsibility: The DevOps team needs to provide performance support until they become end-of-life. This enhances the level of responsibility and the quality of the products engineered.

3. Continuous Improvement: DevOps culture focuses on continuous improvement to minimize waste, and it continuously speeds up the improvement of products or services offered.

4. Automate everything: Automation is a vital principle of the DevOps process, and this is not only for software development but also for the entire infrastructure landscape.

5. Work as one team: In the DevOps culture, the designer, developer, and tester are already defined, and all they need to do is work as one team with complete collaboration.

6. Monitor and test everything: Monitor and test everything: The DevOps team needs robust monitoring and testing procedures.

Who is a DevOps Engineer?

A DevOps Engineer is an IT professional who works with software developers, system operators, and other production IT staff to administer code releases. DevOps should have hard and soft skills to communicate and collaborate with development, testing, and operations teams.

The DevOps approach needs frequent, incremental changes to code versions, requiring frequent deployment and testing regimens. Although DevOps engineers need to code occasionally from scratch, they must have the basics of software development languages.

A DevOps engineer will work with development team staff to tackle the coding and scripting needed to connect code elements, like libraries or software development kits.

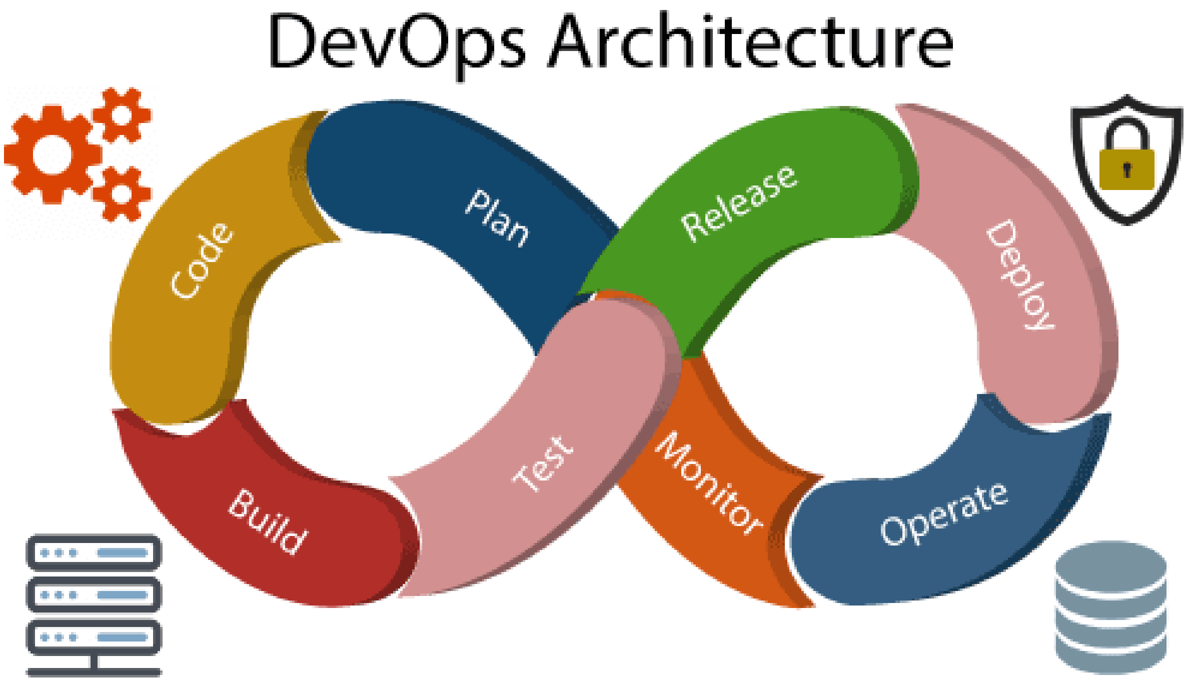
Roles, Responsibilities, and Skills of a DevOps Engineer

DevOps engineers work full-time, and they are responsible for the production and ongoing maintenance of a software application‟s platform.

Following are some expected Roles, Responsibilities, and Skills that are expected from DevOps engineers:

- Able to perform system troubleshooting and problem-solving across platform and application domains.
- Manage project effectively through open, standards-based platforms
- Increase project visibility thought traceability
- Improve quality and reduce development cost with collaboration
- Analyse, design and evaluate automation scripts & systems
- Ensuring critical resolution of system issues by using the best cloud security solutions services
- DevOps engineers should have the soft skill of problem-solver and quick-learner

DevOps Architecture

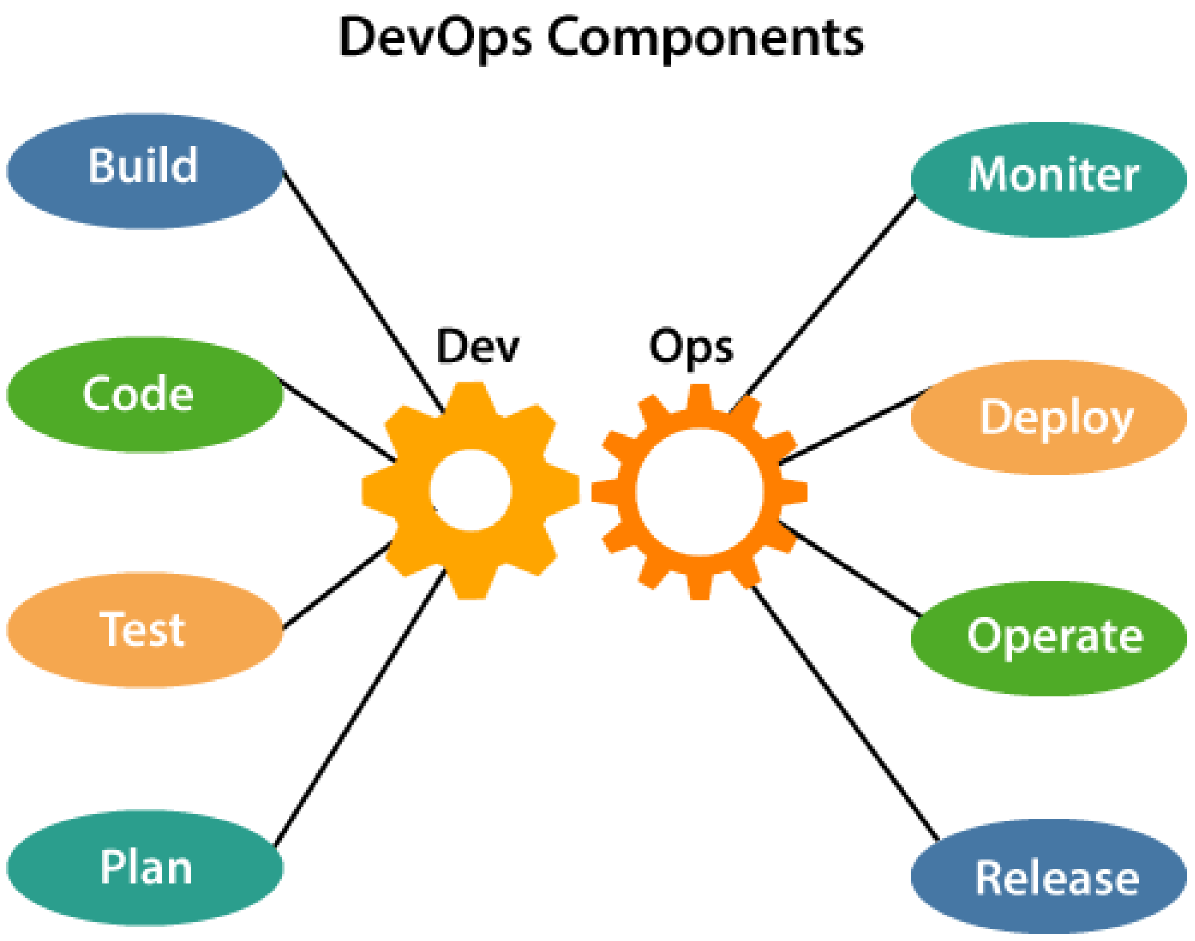


Development and operations both play essential roles in order to deliver applications. The deployment comprises analyzing the **requirements, designing, developing**, and **testing** of the software components or frameworks.

The operation consists of the administrative processes, services, and support for the software. When both the development and operations are combined with collaborating, then the DevOps architecture is the solution to fix the gap between deployment and operation terms; therefore, delivery can be faster.

DevOps architecture is used for the applications hosted on the cloud platform and large distributed applications. Agile Development is used in the DevOps architecture so that integration and delivery can be contiguous. When the development and operations team works separately from each other, then it is time-consuming to **design, test**, and **deploy**. And if the terms are not in sync with each other, then it may cause a delay in the delivery. So DevOps enables the teams to change their shortcomings and increases productivity.

Below are the various components that are used in the DevOps architecture:



1) Build
Without De Without DevOps, the cost of the consumption of the resources was evaluated based on the pre-defined individual usage with fixed hardware allocation. And with DevOps, the usage of cloud, sharing of resources comes into the picture, and the build is dependent upon the user's need, which is a mechanism to control the usage of resources or capacity.

2) Code

Many good practices such as Git enables the code to be used, which ensures writing the code for business, helps to track changes, getting notified about the reason behind the difference in the actual and the expected output, and if necessary reverting to the original code developed. The code can be appropriately arranged in **files, folders**, etc. And they can be reused.

3) Test

The application will be ready for production after testing. In the case of manual testing, it consumes more time in testing and moving the code to the output. The testing can be automated, which decreases the time for testing so that the time to deploy the code to production can be reduced as automating the running of the scripts will remove many manual steps.

4) Plan

DevOps use Agile methodology to plan the development. With the operations and development team in sync, it helps in organizing the work to plan accordingly to increase productivity.

5) Monitor

Continuous monitoring is used to identify any risk of failure. Also, it helps in tracking the system accurately so that the health of the application can be checked. The monitoring becomes more comfortable with services where the log data may get monitored through many third-party tools such as **Splunk**.

6) Deploy

Many systems can support the scheduler for automated deployment. The cloud management platform enables users to capture accurate insights and view the optimization scenario, analytics on trends by the deployment of dashboards.

7) Operate

DevOps changes the way traditional approach of developing and testing separately. The teams operate in a collaborative way where both the teams actively participate throughout the service lifecycle. The operation team interacts with developers, and they come up with a monitoring plan which serves the IT and business requirements.

8) Release

Deployment to an environment can be done by automation. But when the deployment is made to the production environment, it is done by manual triggering. Many processes involved in release management commonly used to do the deployment in the production environment manually to lessen the impact on the customers.

Features of DevOps Architecture

Below are the key features of DevOps Architecture.

1. Automation

Automation most effectively reduces the time consumption specifically during the testing and deployment phase. The productivity increases and releases are made quicker through automation with less issue as tests are executed more rigorously. This will lead to catching bugs sooner so that it can be fixed more easily. For continuous delivery, each code change is done through automated tests, through cloud-based services and builds. This promotes production using automated deploys.

2. Collaboration

The Development and Operations team collaborates together as DevOps team which improves the cultural model as the teams become more effective with their productivity which strengthens accountability and ownership. The teams share their responsibilities and work closely in sync which in turn makes the deployment to production faster.

3. Integration

Applications need to be integrated with other components in the environment. The integration phase is where the existing code is integrated with new functionality and then testing takes place. Continuous integration and testing enable continuous development. The frequency in the releases and micro-services lead to significant operational challenges. To overcome such challenges, continuous integration and delivery are implemented to deliver in a quicker, safer and reliable manner.

4. Configuration Management

This ensures that the application only interacts with the resources concerned with the environment in which it runs. The configuration files are created where the configuration external to the application is separated from the source code. The configuration file can be written while deployment or they can be loaded at the run time depending on the environment in which it is running.

DevOps Orchestration

DevOps orchestration is a logical and necessary step for any DevOps shop that is in the process of, or has completed, implementing automation. Organizations generally start with a local solution and then, after achieving success, orchestrate their best practices through a technology that unifies connectivity into one solid process. That‟s because automation can only go so far in maximizing efficiency, so orchestration in DevOps is needed if you want to take your releases to the next level.

To illustrate this concept, I‟m going to discuss DevOps orchestration in general and how practices like DevOps provisioning orchestration and DevOps release orchestration can apply both on-site and in the cloud.

DevOps Orchestration Basics

Define orchestration in DevOps:

DevOps automation is a process by which a single, repeatable task, such as launching an app or changing a database entry, is made capable of running without human intervention, both on PCs and in the cloud. In comparison, DevOps orchestration is the automation of numerous tasks that run at the same time in a way that minimizes production issues and time to market.

Automation applies to functions that are common to one area, such as launching a web server, or integrating a web app, or changing a database entry. But when all of these functions must work together, DevOps orchestration is required.

DevOps Orchestration – The Following Move after Automation

DevOps orchestration involves automating multiple processes to reduce issues leading to the production date and shorten time to market. On the other hand, automation is used to perform tasks or a series of actions that are repetitive.

DevOps orchestration is not simply putting separated tasks together, but it can do much more than that. DevOps orchestration streamlines the entire workflow by centralizing all tools used across teams, along with their data, to keep track of process and completion status throughout.

Besides, automation can be pretty complicated at scale, although normally it is focused on a specific operation to achieve a goal, such as a server deployment. When automation has reached its limitations, that‟s when orchestration plays to its strengths.

Why Invest in DevOps Orchestration?

SIX REASONS TO INVENST DEVOPS ORCHESTRATION

1. Speed up the automation process
2. Enhance cross-functional collaboration
3. Release products with higher quality
4. Lower costs for IT infrastructure and human resources
5. Build transparency across the SDLC
6. Improve the release velocity

Investing in DevOps Orchestration helps you:

1. Speed up the automation process
DevOps orchestration helps users deliver new builds into production quickly and seamlessly without putting much effort into these repetitive tasks. As a result, DevOps teams can focus on other critical projects and decision-making.

2. Enhance cross-functional collaboration
DevOps orchestration is a platform that constantly unifies and updates all activities to improve communication between operation and development teams, so every member is in sync in all steps.

3. Release products with higher quality
DevOps orchestration helps teams reduce software errors before reaching the end-user by controlling approvals, scheduling, security testing, automatic status reports.

4. Lower costs for IT infrastructure and human resources
Another benefit of DevOps orchestration is cutting infrastructure investment costs and the number of IT employees. In the long run, businesses can also expand their cloud service footprint and allocate costs flexibly.

5. Build transparency across the SDLC
Siloed tasks and information cause difficulty in creating clarity and openness throughout a project. DevOps orchestration helps businesses coordinate all tasks, centralize all operations‟ data, and allow key stakeholders to gain visibility into new updates and progress during the development lifecycle.

6. Improve the release velocity
DevOps orchestration also requires considerable automation and the automated progression of software through testing and releasing pipeline processes. As businesses can reduce the time of finishing manual tasks and deliver the program to the upcoming step in the process, software reaches the end-user more quickly, wait time is turned aside to the next project. With that reason, a higher level of automation helps release products faster, save money and increase profits.

DevOps Applications

Applications of DevOps

1. Application of DevOps in the Online Financial Trading Company
The methodology in the process of testing, building, and development was automated in the financial trading company. Using the DevOps, deployment was being done within 45 seconds. These deployments used to take long nights and weekends for the employees. The time of the overall process reduced and the interest of clients increased.

2. Use of DevOps in Network cycling
Deployment, testing and rapid designing became ten times faster. It became effortless for the telco service provider to add patches of security every day, which used to be done only every

three months. Through deployment and design, the new version of network cycling was being rolled out.

3. Application in Car Manufacturing Industries
Using DevOps, employees helped car manufacturers to catch the error while scaling the production, which was not possible before.

4. Benefits to Airlines Industries
With the benefit of DevOps, United Airlines saved $500,000 by changing to continuous testing standards. It also increased its coverage of code by 85%.

5. Application to GM Financial
Regression testing time was reduced by 93%, which in turn reduced the funding
period of load by five times.

6. Bug Reduction Benefit of DevOps
DevOps has reduced the bugs by up to 35% and in many cases of pre-production bugs up to 40%. By using DevOps, Rabobank was able to provide better quality applications for their clients within less time because it massively reduced the time taken for regression testing.

7. Less Time for Integration
Key Bank used DevOps to reduce the time taken for the integration of security and compliance into the process from 3 months to 1 week.

8. Decreased Computation Cost and Operation Time
By the use of DevOps, Computation time has been dramatically reduced. In many cases, it has reduced the computing time from up to 60%. When the time taken to complete a task is decreased, then the cost involved the process also decreases.

9. Faster Development of Software
The DevOps helps in the faster delivery of apps because it ensures speedier delivery.

10. Improvement in Team Collaboration
Transparency is required for better decision-making and works better efficiency of resources. By using DevOps, teams can be more transparent in their work of developing applications and software. There are many big tasks of a project which are broken down into many small tasks that are allotted to different teams or people in the organization.

11. Reliable Environments for Operations
DevOps provide a better environment that is more stable for the team to work together. The people in the group can rely on the environment for all kinds of operations and tasks.

12. Early Defects Detection
In the environment of DevOps, the error and the defects can be known at a very first stage. DevOps helps in the fast detection of defects.

13. Faster Correction
With the help of DevOps, All the defects are detected very early. Because of this, mistakes can be corrected very fast. There is a lot of time which gets saved in this kind of DevOps environment because the work is very fast, so the correction work is also completed very fast.

14. Continuous Operation of Monitoring, Testing, Deployment, and Release
There is a massive demand for software professionals to keep on delivering high-quality applications and software. DevOps Developer salary is ever increasing in India. A software

development team is expected to develop the software application within a shorter time period which can be launched into the market. The team of software is required to adopt a shorter release cycle.
All of the requirements of delivering high-quality software, developing software applications in less time frame. Also, a minimum period to launch the software applications and short release cycles can only be fulfilled by the use of DevOps.

15. Increased Focus on Operations
When less time is taken for the tasks and operation with the help of DevOps, It allows one person to focus more on the quality of activities and functions. There is more time for one to give his quality work because now there is DevOps minimize more time to focus as all the time for other services.

16. Automation Testing integration into DevOps
When the framework of automated testing is integrated into the DevOps, It helps to save a lot of money and time, which then increases the available time and quality of the work.

Top DevOps Tools for 2020
1) Jenkins
DevOps are divided into different stages. For integrating them, you‟d need to perform Continous Integration (CI). Jenkins is the tool that can help you in that regard. Jenkins enables companies to boost their software development processes. Developers use Jenkins to test their software projects and add changes seamlessly.

This tool uses Java with plugins, which help in enhancing Continuous Integration. Jenkins is widely popular with more 1 million users. So also get access to a thriving and helpful community of developers.

2) Git
It is a version control system, and it lets teams collaborate on a project at the same time. Developers can track changes in their files, as well as improve the product regularly. Git is widely popular among tech companies. Many companies consider it is a must-have for their tech professionals.

You can save different versions of your code on Git. You can use GitHub for holding repositories as well. GitHub allows you to connect Slack with your on-going projects so your team can easily communicate regarding the project.

3) Bamboo

Bamboo is similar to Jenkins as it helps you in automating your delivery pipeline. The difference is of their prices. Jenkins is free, but Bamboo is not. **Is bamboo worth paying for?** Well, Bamboo has many functionalities which are set up beforehand. With Jenkins, you would‟ve had to build those functionalities yourself. And that takes a lot of effort and time.
Bamboo doesn‟t require you to use many plugins too because it can do those tasks itself. It has a great UI, and it integrates with BitBucket and many other Atlassian products.

4) Kubernetes

Kubernetes deserves a place on this DevOps tools list for obvious reasons. First, it is a fantastic container orchestration platform. Second, it has taken the industry by storm.
When you have many containers to take care of, scaling the tasks becomes immensely challenging. Kubernetes helps you in solving that problem by automating the management of your containers.

It is an open-source platform. Kubernetes can let you scale your containers without increasing your team. As it is an open-source platform, you don‟t have to worry about access problems. You can use public cloud infrastructure or hybrid infrastructure to your advantage. The tool can also self-heal containers. This means it can restart failed containers, kills not-responding containers, and replaces containers.

5) Vagrant

You can build and manage virtual machine environments on Vagrant. It lets you do all that in a single workflow. You can use it on Mac, Windows, as well as, Linux.

It provides you with an ideal development environment for **better productivity and efficiency**. It can easily integrate with multiple kinds of IDEs and configuration management tools such as Salt, Chef, and Ansible.

Because it works on local systems, your team members won‟t have to give up their existing technologies or operating systems. Vagrant‟s enhanced development environments certainly make DevOps easier for your team. That‟s why we have kept it in our DevOps tools list.

6) Prometheus
Prometheus is an open-source service monitoring system, which you can use for free. It has multiple custom libraries you can implement quickly.
It identifies time series through metric names and key-value pairs. You can use its different modes for data visualization as well. Because of its functional sharding and federation, scaling the projects is quite easy.

It also enables multiple integrations from different platforms, such as Docker and StatsD. It supports more than ten languages. Overall, you can easily say it is among the top DevOps tools because of its utility.

7) Splunk
Splunk makes machine data more accessible and valuable. It enables your organization to use the available data in a better fashion. With its help, you can easily monitor and analyze the available data and act accordingly. Splunk also lets you get a unified look at all the IT data present in your enterprise.
You can deliver insights by using augmented reality and mobile devices, too, with the use of Splunk. From security to IT, Splunk finds uses in many areas. It is one of the best automation tools for DevOps because of the valuable insights it provides to the user. You can use Splunk in numerous ways according to your organization‟s requirements.

Some companies also use Splunk for business analytics and IoT analytics. The point is, you can use this tool for finding valuable data insights for all the sections of your organization and use them better.

8) Sumologic
Sumologic is a popular CI platform for DevSecOps. It enables organizations to develop and secure their applications on the cloud. It can detect Indicators of Compromise quickly, which lets you investigate and resolve the threat faster.

Its real-time analytics platform helps organizations in using data for predictive analysis. For monitoring and securing your cloud applications, you should choose Sumologic. Because of its power of the elastic cloud, you can scale it infinitely (in theory).

What is a DevOps Pipeline?

A DevOps pipeline is a set of practices that the development (Dev) and operations (Ops) teams implement to build, test, and deploy software faster and easier. One of the primary purposes of a pipeline is to keep the software development process organized and focused.

The term "pipeline" might be a bit misleading, though. An assembly line in a car factory might be a more appropriate analogy since software development is a continuous cycle.

Before the manufacturer releases the car to the public, it must pass through numerous assembly stages, tests, and quality checks. Workers have to build the chassis, add the motor, wheels, doors, electronics, and a finishing paint job to make it appealing to customers.

DevOps pipelines work similarly.
Before releasing an app or a new feature to users, you first have to write the code. Then, make sure that it does not lead to any fatal errors that might cause the app to crash. Avoiding such a scenario involves running various tests to fish out any bugs, typos, or mistakes. Finally, once everything is working as intended, you can release the code to users.

From this simplified explanation, you can conclude that a DevOps pipeline consists of the build, test, and deploy stages.

Components of a DevOps Pipeline

To ensure the code moves from one stage to the next seamlessly requires implementing several DevOps strategies and practices. The most important among them are **continuous integration** and **continuous delivery (CI/CD)**.

Continuous Integration

Continuous integration (CI) is a method of integrating small chunks of code from multiple developers into a shared code repository as often as possible. With a CI strategy, you can automatically test the code for errors without having to wait on other team members to contribute their code.

One of the key benefits of CI is that it helps large teams prevent what is known as *integration hell*.

In the early days of software development, developers had to wait for a long time to submit their code. That delay significantly increased the risk of code-integration conflicts and the deployment of bad code. As opposed to the old way of doing things, CI encourages developers to submit their code daily. As a result, they can catch errors faster and, ultimately, spend less time fixing them.

At the heart of CI is a **central source control system**. Its primary purpose is to help teams organize their code, track changes, and enable automated testing.

In a typical CI set-up, whenever a developer pushes new code to the shared code repository, automation kicks in to compile the new and existing code into a build. If the build process fails, developers get an alert which informs them which lines of code need to be reworked.

Making sure only quality code passes through the pipeline is of paramount importance. Therefore, the entire process is repeated every time someone submits new code to the shared repository.

Continuous Delivery

Continuous delivery (CD) is an extension of CI. It involves speeding up the release process by encouraging developers to release code to production in incremental chunks.

Having passed the CI stage, the code build moves to a holding area. At this point in the pipeline, it's up to you to decide whether to push the build to production or hold it for further evaluation.

In a typical DevOps scenario, developers first push their code into a production-like environment to assess how it behaves. However, the new build can also go live right away, and developers can deploy it at any time with a push of a button.

To take full advantage of continuous delivery, deploy code updates as often as possible. The release frequency depends on the workflow, but it's usually daily, weekly, or monthly. Releasing code in smaller chunks is much easier to troubleshoot compared to releasing all changes at once. As a result, you avoid bottlenecks and merge conflicts, thus maintaining a steady, continuous integration pipeline flow.

Continuous Deployment

Continuous delivery and continuous deployment are similar in many ways, but there are critical differences between the two.

While continuous delivery enables development teams to deploy software, features, and code updates manually, continuous deployment is all about automating the entire release cycle.

At the continuous deployment stage, code updates are released automatically to the end-user without any manual interventions. However, implementing an automated release strategy can be dangerous. If it fails to mitigate all errors detected along the way, bad code will get deployed to production. In the worst-case scenario, this may cause the application to break or users to experience downtime.

Automated deployments should only be used when releasing minor code updates. In case something goes wrong, you can roll back the changes without causing the app to malfunction.

To leverage the full potential of continuous deployment involves having robust testing frameworks that ensure the new code is truly error-free and ready to be immediately deployed to production.

Continuous Testing

Continuous testing is a practice of running tests as often as possible at every stage of the development process to detect issues before reaching the production environment. Implementing a continuous testing strategy allows quick evaluation of the business risks of specific release candidates in the delivery pipeline.

The scope of testing should cover both functional and non-functional tests. This includes running unit, system, integration, and tests that deal with security and performance aspects of an app and server infrastructure.

Continuous testing encompasses a broader sense of quality control that includes risk assessment and compliance with internal policies.

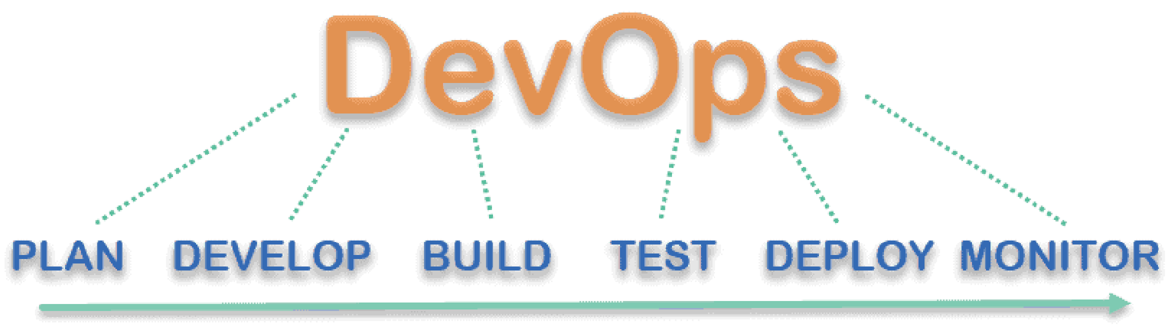
Continuous Operations

Having a comprehensive continuous operations strategy helps maintain maximum availability of apps and environments. The goal is for users to be unaware that of constantly releasing code updates, bug fixes, and patches. A continuous operations strategy can help prevent downtime and availability issues during code release.

To reap the benefits of continuous operations, you need to have a robust automation and orchestration architecture that can handle continuous performance monitoring of servers, databases, containers, networks, services, and applications.

Phases of DevOps Pipeline

There are no fixed rules as to how you should structure the pipeline. DevOps teams add and remove certain stages depending on their specific workflows. Still, four core stages make up almost every pipeline: **develop**, **build**, **test**, and **deploy**.



That set-up can be extended by adding two more stages - **plan** and **monitor** - since they are also quite common in professional DevOps environments.

Plan

The planning stage involves planning out the entire workflow before developers start coding. In this stage, product managers and project managers play an essential role. It's their job to create a development roadmap that will guide the whole team along the process.

After gathering feedback and relevant information from users and stakeholders, the work is broken down into a list of tasks. By segmenting the project into smaller, manageable chunks, teams can deliver results faster, resolve issues on the spot, and adapt to sudden changes easier.

In a DevOps environment, teams work in **sprints** - a shorter period of time (usually two weeks long) during which individual team members work on their assigned tasks.

Develop

In the Develop stage, developers start coding. Depending on the programming language, developers install on their local machines appropriate IDEs (Python IDEs, Java IDEs, etc), code editors, and other technologies for achieving maximum productivity.

In most cases, developers have to follow certain coding styles and standards to ensure a uniform coding pattern. This makes it easier for any team member to read and understand the code.

When developers are ready to submit their code, they make a pull request to the shared source code repository. Team members can then manually review the newly submitted code and merge it with the master branch by approving the initial pull request.

Build

The build phase of a DevOps pipeline is crucial because it allows developers to detect errors in the code before they make their way down the pipeline and cause a major disaster.

After the newly written code has been merged with the shared repository, developers run a series of automated tests. In a typical scenario, the pull request initiates an automated process that compiles the code into a build - a deployable package or an executable.

Keep in mind that some programming languages don't need to be compiled. For example, applications written in **Java** and **C** need to be compiled to run, while those written in **PHP** and **Python** do not.

If there is a problem with the code, the build fails, and the developer is notified of the issues. If that happens, the initial pull request also fails.

Developers repeat this process every time they submit to the shared repository to ensure only error-free code continues down the pipeline.

Test

If the build is successful, it moves to the testing phase. There, developers run manual and automated tests to validate the integrity of the code further.

In most cases, a **User Acceptance Test** is performed. People interact with the app as the end-user to determine if the code requires additional changes before sending it to production. At this stage, it's also common to perform security, performance, and load testing.

Deploy

When the build reaches the Deploy stage, the software is ready to be pushed to production. An automated deployment method is used if the code only needs minor changes. However, if the application has gone through a major overhaul, the build is first deployed to a production-like environment to monitor how the newly added code will behave.

Implementing a **blue-green deployment strategy** is also common when releasing significant updates.

A blue-green deployment means having two identical production environments where one environment hosts the current application while the other hosts the updated version. To release the changes to the end-user, developers can simply forward all requests to the appropriate servers. If there are problems, developers can simply revert to the previous production environment without causing service disruptions.

Monitor

At this final stage in the DevOps pipeline, operations teams are hard at work continuously monitoring the infrastructure, systems, and applications to make sure everything is running smoothly. They collect valuable data from logs, analytics, and monitoring systems as well as feedback from users to uncover any performance issues.

Feedback gathered at the Monitor stage is used to improve the overall efficiency of the DevOps pipeline. It's good practice to tweak the pipeline after each release cycle to eliminate potential bottlenecks or other issues that might hinder productivity.

How to Create a DevOps Pipeline

Now that you have a better understanding of what a DevOps pipeline is and how it works let's explore the steps required when creating a CI/CD pipeline.

Set Up a Source Control Environment

Before you and the team start building and deploying code, decide where to store the source code. **GitHub** is by far the most popular code-hosting website. GitLab and BitBucket are powerful alternatives.

To start using GitHub, open a free account, and create a shared repository. To push code to GitHub, first install Git on the local machine. Once you finish writing the code, push it to the shared source code repository. If multiple developers are working on the same project, other team members usually manually review the new code before merging it with the master branch.

Set Up a Build Server

Once the code is on GitHub, the next step is to test it. Running tests against the code helps prevent errors, bugs, or typos from being deployed to users.

Numerous tests can determine if the code is production-ready. Deciding which analyses to run depends on the scope of the project and the programming languages used to run the app.

Two of the most popular solutions for creating builds are **Jenkins** and **Travis-CI**. Jenkins is completely free and open-source, while Travis-CI is a hosted solution that is also free but only for open-source projects.

To start running tests, install Jenkins on a server and connect it to the GitHub repository. You can then configure Jenkins to run every time changes are made to the code in the shared repository. It compiles the code and creates a build. During the build process, Jenkins automatically alerts if it encounters any issues.

Run Automated Tests

There are numerous tests, but the most common are **unit tests**, **integration tests**, and **functional tests**.

Depending on the development environment, it's best to arrange automated tests to run one after the other. Usually, you want to run the shortest tests at the beginning of the testing process.

For example, you would run unit tests before functional tests since they usually take more time to complete. If the build passes the testing phase with flying colors, you can deploy the code to production or a production-like environment for further evaluation.

Deploy to Production

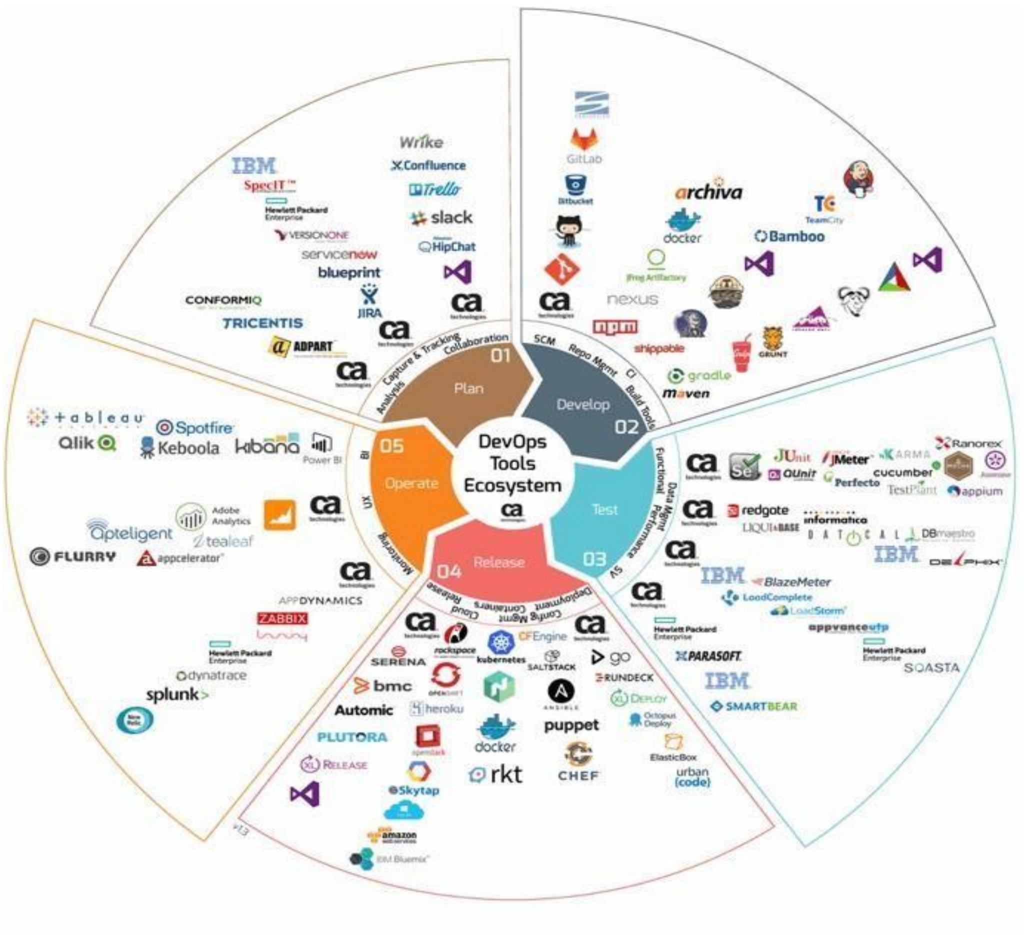
Before deploying the code to production, first set up the server infrastructure. For instance, for deploying a web app, you need to install a web server like Apache. Assuming the app will be running in the cloud, you'll most likely deploy it to a virtual machine.

For apps that require the full processing potential of the physical hardware, you can deploy to dedicated servers or bare metal cloud servers.

There are two ways to deploy an app - **manually** or **automatically**. At first, it is best to deploy code manually to get a feel for the deployment process. Later, automation can speed up the process, but only if you are confident, there are barriers that will stop bad code from ending up in production.

Releasing code to production is a straightforward process. The easiest way to deploy is by configuring the build server to execute a script that automatically releases the code to production.

DevOps Tools Ecosystem



Plan

Planning is the initial stage, and it covers the first steps of project management. The project and product ideas are presented and analyzed, in groups, alone, or on whiteboards. The developer, team, and organization decide what they want and how they want it and assign tasks to developers, QA engineers, product managers, etc. This stage requires lots of analysis of problems and solutions, collaboration between team members, and the ability to capture and track all that is being planned.

Develop

Developing is the stage where the ideas from planning are executed into code. The ideas come to life as a product. This stage requires software configuration management, repository management and build tools, and automated Continuous Integration tools for incorporating this stage with the following ones.

Test

A crucial part that examines the product and service and makes sure they work in real time and under different conditions (even extreme ones, sometimes). This stage requires many different kinds of tests, mainly functional tests, performance or load tests, and service virtualization tests. It‟s also important to test compatibility and integrations with third-party services. The data from the tests needs to be managed and analyzed in rich reports for improving the product according to test results.

Release

Once a stage that stood out on its own and caused many a night with no sleep for developers, now the release stage is becoming agile and integrating with the Continuous Delivery process. Therefore, the discussion of this part can‟t revolve only around tools, but rather needs to discuss methodologies as well. Regarding tools, this stage requires deployment tools.

Operate

We now have a working product, but how can we maximize the features we‟ve planned, developed, tested, and released? This is what this stage is for. Implementing the best UX is a big part of this, monitoring infrastructure, APMs, and aggregators, and analyzing Business Intelligence (BI). This stage ensures our users get the most out of the product and can use it error-free.

Obviously, this work cycle isn‟t one-directional. We might use tools from a certain stage, move on to the next, go back a stage, jump ahead two stages, and so on. Essentially, it all comes down to a feedback loop. You plan and develop. The test fails, so you develop again. The test passes, you release it, and you get information about customer satisfaction through measurement tools like google analytics or A/B testing. Then, you re-discuss the same feature to get better satisfaction out of the product, develop it again, etc. The most important part is that you cover all stages, as we will do in the upcoming weeks.

Assignment -II Questions

1. Analyse the DevOps with traditional process
2. Analyse DevOps tools echosystem
3. Analyse the phases of Devops Pipeline
4. Analyse phases of DevOps Architecture
5. Analyse the principles of Devops and roles and responsibilities of DevOps Engineer

UNIT – I(Extention)

DevOps adoption in projects: Technology aspects, Agiling capabilities, Tool stack implementation, People aspect, processes

Adopting DevOps offers a cultural change in the workforce by enabling engineers to cross the barrier between the development teams and operations teams.

Adopting a DevOps culture:

Progressive Collaboration
DevOps promises to bridge the gap between the two where both employ bottom-up and top-down feedback from each other. With DevOps, when development seeks operational help or when operations require immediate development, both remain ready for each other at any given time. In such a scenario, the software development culture brings in to focus combined development instead of individual goals; The development environment becomes more progressive as all the team members work in cohesion towards a common goal.

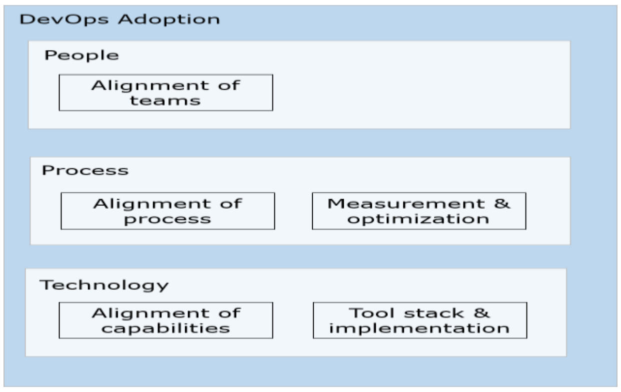
Processing Acceleration

With conjoined operational and developmental paradigms, the communication lag between the two is reduced to null. Organizations continuously strive for a better edge over their competing rivals, and if such acceleration is not achieved, the organization will have to succumb to competing forces— innovation will be slower, and the product market will decay.
Shorter Recovery Time
DevOps deployment functions on a more focused and exclusive approach which makes issues more accessible to spot; this helps error rectification faster and easier to implement. The resolution to problems is inherently quicker, as troubleshooting happens to take place at the current development level only, within a single team. Thus, the overall time for recovery and rectification is drastically reduced.
Lower Failure Rate

The abridged departments yield shorter development cycles which result in rapid production. The entire process becomes modular wherein issues related to configuration, application code, and infrastructure become more apparent and pre-accessible A decrease in error count also positively affects the success rates of development. Therefore, very few fixes will be required to attain a fully functional code for the desired output.
Higher Job Satisfaction

DevOps fosters equality by bringing different officials at the same level of interaction.
DevOps serves as a handy tool for achieving that feat; it enables the workforce to work in cohesion where chances for failure are minimal, and production is rapid. As a result, the processing becomes efficient and workspace more promising.

The DevOps adoption requires focus on the People, Process and Technology aspects.

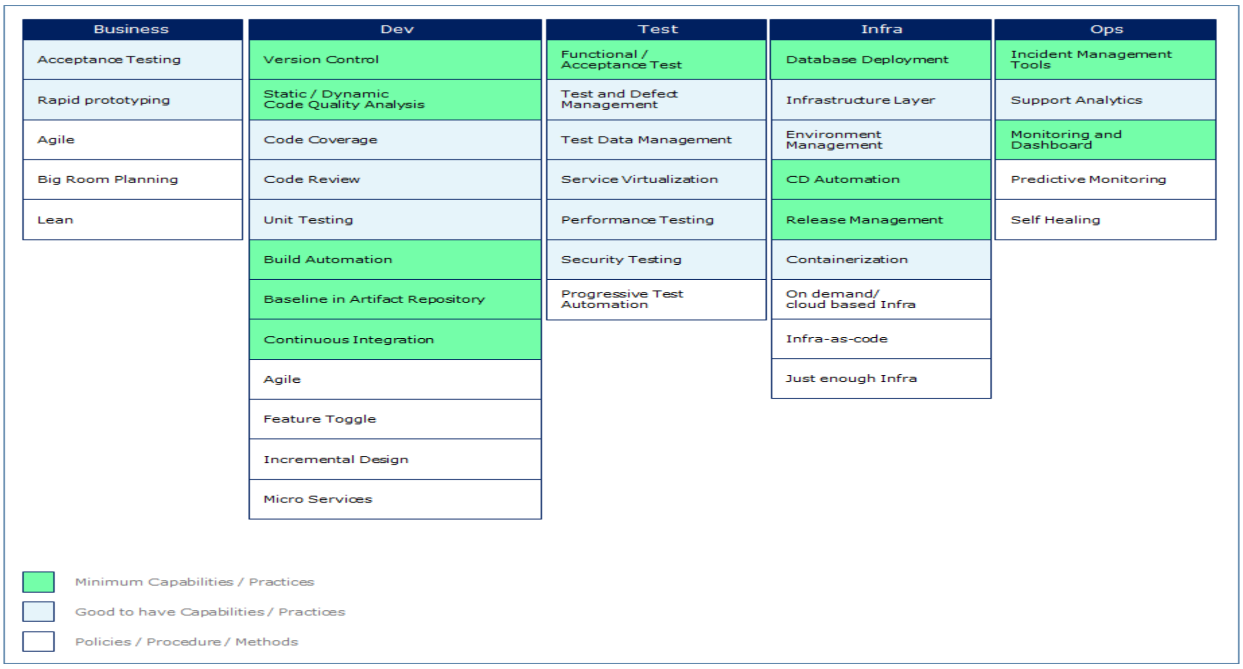

DevOps Adoption
People
Alignment of teams
Process
Alignment of process
Measurement & optimization
Technology
Alignment of capabilities
Tool stack & implementation

Technology challenges

- Lack of automation in the software development lifecycle and hence loss of quality due to error prone repeatability of steps (ex. Tests)
- Defects generated due to inconsistent environments for testing and deployment
- Delays in testing due to infrastructure unavailability
- Brittle point-to-point integration between modules

Aligning Capabilities :

The capabilities to be aligned are shown in the figure below.



Business	Dev	Test	Infra	Ops
🟦 Acceptance Testing	🟩 Version Control	🟩 Functional / Acceptance Test	🟩 Database Deployment	🟩 Incident Management Tools
🟦 Rapid prototyping	🟩 Static / Dynamic Code Quality Analysis	🟦 Test and Defect Management	🟦 Infrastructure Layer	🟦 Support Analytics
⬜ Agile	🟦 Code Coverage	🟦 Test Data Management	🟦 Environment Management	🟩 Monitoring and Dashboard
⬜ Big Room Planning	🟦 Code Review	🟦 Service Virtualization	🟩 CD Automation	⬜ Predictive Monitoring
⬜ Lean	🟦 Unit Testing	🟦 Performance Testing	🟩 Release Management	⬜ Self Healing
	🟩 Build Automation	🟦 Security Testing	🟦 Containerization	
	🟩 Baseline in Artifact Repository	⬜ Progressive Test Automation	⬜ On demand/ cloud based Infra	
	🟩 Continuous Integration		⬜ Infra-as-code	
	⬜ Agile		⬜ Just enough Infra	
	⬜ Feature Toggle			
	⬜ Incremental Design			
	⬜ Micro Services			

🟩 Minimum Capabilities / Practices
🟦 Good to have Capabilities / Practices
⬜ Policies / Procedure / Methods

Business perspectives:

Rapid prototyping

- For rapid and iterative design and delivery of software
- It involves using prototyping tools and collaborative review by stakeholders and refinement based on feedback
- **Benefits** - Provides a feel of the product early, room for customization, saves cost and time (if tools are used) and minimizes design flaws

Role and responsibilities of **Business teams should**

- be ready to adopt Agile approach and collaborate with the development teams for faster development and delivery of valuable software
- adopt lean practices and eliminate wasteful processes, documentation and the like
- be ready to allow iterations to happen and provide early feedback
- Ensure availability and clarity on requirements are on time

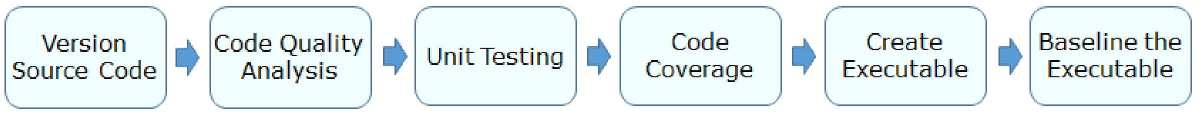
Alleviated issues

- Lack of automation in the software development lifecycle and hence loss of quality due to error prone repeatability of steps – in this case automation of Acceptance Tests
- Well defined and automated acceptance criteria and clarity on requirements

Development team perspective :

Continuous integration pipeline

- Continuous integration practice is mandatory to ensure that defects are captured early and a working version of the software is always available by automating the different development stage through integration of tools. This is a must-to-have to ensure continuous, frequent and automated delivery & deployment of software to customers
- Is a software development practice adopted as part of extreme programming (XP). The image below shows the activities in the development phase that need to be automated to form the Continuous Integration pipeline. It helps in automating the build process, enabling frequent integration, code quality checks and unit testing without any manual intervention by use of various open-source, custom-built and/or licensed tools. The sequence and whether an activity will be done will be decided by the orchestrator – i.e. the continuous integration tool based on the gating criteria set for the project



- **Benefits**
  - Reduced risks and fewer integration defects
  - Helps detect bugs and remove them faster
  - Less integration time due to automation
  - Avoids cumulative bugs due to frequent integration
  - Helps in frequent deployment

If Dev teams align with the capabilities mentioned, the following issues faced by “Project” would get alleviated.

Issues

- Lack of automation in the software development lifecycle and hence loss of quality due to error prone repeatability of steps – here the automation of activities in the build cycle activities
- Brittle point-to-point integration between modules – resolved due to continuous integration of modules

Test team perspective :

Continuous and automated testing

- In an automated development and delivery pipeline, integration will be done frequently as we saw in the previous section. Hence test cases need to be run frequently. The gate for developed software meeting the functionality are the test cases. This is possible only if the tests are automated
- Involves using automated testing tools which can execute tests, provide outcomes of the tests in the form of reports and can be run repeatedly. Test automation frameworks helps in this. Ex. JUnit

In the automated development and delivery pipeline, the following tests and their management are automated mandatorily. These tests are invoked in an automated fashion by the orchestrator – i.e. the continuous integration tool. The testing related automation is required.

Test and test data management automation

- Functional test
- Test and Test data management
- Performance test
- Security test

- **Benefits**
    - Increases the depth and scope of test coverage which helps in improving software quality
    - Ensures repeatability of running tests whenever required and helps in continuous integration of valuable software

Service virtualization

- In traditional software development, the testing starts post the integration of all the components that are needed – ex. Performance testing is delayed and testers may skip this for want of time. This leads to finding of defects late in the cycle and are costly to fix. It also impacts the delivery speed. Hence service virtualization is required to "shift-left" and detect defects early in the cycle (say during unit testing) by simulating the non-available components of the application
- Service virtualization helps in simulating application dependencies and begin testing earlier. Virtual components can be data, business rules, I/O configurations etc
    - **Features**

        - Are light-weight and hence testing is inexpensive. Example: If we have a legacy system on top of which business logic enhancements are done, setting up the latter every time for testing is cumbersome and costly
        - Creates a virtual asset which simulates the behaviour of the components which are impossible to access or unavailable
        - Components can be deployed in virtual environments
        - Created by recording of the live communication among the components that are used for testing
        - Provide logs representing the communication among the components
        - Analysis of service interface specifications like WSDL
        - Assets listen for requests and return an appropriate response. Example: Listen to an SQL statement and return the source rows from the database as per the query
- **Benefits**
    - Reduce cost of fixing defects
    - Decreases project risk
    - Improves the speed of delivery
    - Helps emulate the unavailable components/environments and represents a more realistic behaviour (stubs/mocks help in skipping unavailable components)

Infra team perspective:

CD automation

- Business requires frequent delivery of valuable software with efficiency. Continuous delivery helps create a repeatable, reliable and incrementally improving process for ensuring this
- CD allows constant flow of changes into production through an automated software production pipeline called "CD Pipeline". This involves Continuous Validation(CV) followed by Continuous Delivery. Quality is built-in to the pipeline. The pipeline provides feedback to the team and visibility into the flow of changes to everyone involved in delivering the new feature/s. We have seen these CI and CV earlier. CD therefore is a series of practices to ensure that quality code can be deployed fast and safe to production by delivering every change/new feature to a production-like environment. Since automation is used, the confidence level that this would work well in production environment is high. With the push of a button, the change can be deployed to the production environment. This is called continuous deployment. Continuous deployment may not be practical in all organizations due to regulatory and other processes, though it should be the goal of every organization. It follows continuous delivery
- **Benefits**
    - Customers can realize early ROI
    - Since it is based on automation, repeatability is ensured and quality software is delivered to pre-production environment which ensures that the same will work well in production environment also

Release management

- Traditionally, release management involves the complex process of planning, designing, building, testing and deploying new software and hardware in production environment. Integrity needs to be maintained while releasing the correct version. Traditionally, this process is very stressful and inefficient involving a lot of manual work and co-ordination. Also, since there is isolation of ops and dev teams, hence there are surprises, delays and errors in releases. There is lot of documentation that needs to be read prior to deployment every time. A path and workflow needs to be done through a system to allow for fast delivery of software

- The automated tools for release management allow the integration of the management and execution of releases. They help teams to plan, track and execute the releases through an integrated interface. They allow the approvals and notification to the concerned for various stages in the delivery pipeline. Release plans can therefore be run quickly
- **Benefits**
    - Errors in releases can be reduced
    - The workflow is automated and can be tracked through a system
    - Helps in bringing speed to releases

Database deploy

- In the continuous delivery pipeline, since database deployment has fundamental differences with application deployment and processes, the former is done in a manual fashion. Hence the benefits of continuous delivery pipelines are not optimized and may result in delays. To standardize database deployments to the CD delivery practices, automated database deployments are required
- The automated database deployment tools generate a single deployment script that contains the meta data and structure changes. It also contains the details of the changes in terms of configuration management
- **Benefits**
    - High levels of visibility into database deployments
    - Prevents errors due to manual scripts for database deployments
    - Provides an interface to package, verify, deploy, and promote database changes as done with application code and integration of database deployments to the CD pipeline

Infrastructure layer and environment management

- The various steps that we saw earlier (including environment provisioning, testing, deployments etc.) need an infrastructure to be in place for implementation. This layer is responsible for infrastructure management. The environment management tools help in this
- The infrastructure layer can be managed by tools like Chef which help spinning of virtual machines, syncing them, help make changes across multiple servers etc. The virtual machines mimic servers including the complete operating system, drivers, binaries etc. They run on top of a hypervisor system which in turn runs on top of another operating system
- **Benefits** - Provides the necessary hardware for automated deployments and the environment management tools help in managing and maintaining them

Containerization

- A virtual machine as seen in the earlier section has its own operating system. Hence precious operating system resources are wasted across virtual machines. In order to ensure that the virtual machines share the same resources, containerization is required
- It allows virtual machines to share a single host operating system and relevant binaries, drivers etc. This is called operating system level virtualization
- **Benefits**
    - Containers are smaller in size, easier to migrate and requires less memory
    - Allows a server to host multiple containers instead of virtual machines being spun

Ops team perspective:

Incident management

- With the large scale explosion of data centers and virtualization, the scale and fragmentation of IT alerts have increased dramatically. Hence the manual way of resolving alerts like constantly filtering through noisy alerts, connect them to get the bigger issue, prioritize and escalate to concerned, and manually managing the alerts should be avoided
- Centralized incident management solutions avoids redundant alerts. It combines all the monitoring systems and provides an easy tracking mechanism by which support teams can respond
- **Benefits**
    - Helps support teams to respond to alerts quick and easy
    - Since the automated pipelines may have several tools and layers, incident management tools help centralize the alerts and hence faster responses to them

Support analytics

- Faster release cycles demand automated deployment to get applications out faster and they demand discovering and diagnosing production issues gaining insight quickly and through actionable analytics. Focusing on business metrics is important in DevOps environments. To derive these metrics and the data to meet the key performance indicators becomes essential and hence the need for support analytics
- Tools for support analytics do a deep search for data, do centralized logging and parsing and display the data in a neat way
- **Benefits** - These tools help in collaboration across teams and provide exactly what is happening to the business from the data that is stored and logged

Monitoring dashboard

- In order to measure the success of DevOps adoption and also measure the health of the pipelines, monitoring dashboards are required
- A dashboard provides a complete view of the pipeline. The dashboard can be based on different perspectives. Some examples -
    - Business performance dashboard – May depict the revenue, speed of deployment, defect status etc. This can be for both technical and non-technical teams

    - End user dashboard – may provide code and API specific metrics like error rates, pipeline status etc.
- **Benefits** - Single point where teams get visibility of the DevOps implementation

Tool stack and its implementation

Now that the capabilities are understood, let us look at how to use tools to actionize the capabilities in the team

he aim of choosing the tool stack is to build an automated pipeline using tools for performing the various software development, testing, deployment and release activities. This helps in creating rapid, reliable and repeated releases of working software with low risk and minimal manual overhead. Here are the principles to be considered while choosing tools.

Principles to be considered

- Repeatability : The automated pipeline needs to be executed frequently and multiple times with consistency
- Reliability : The automated pipeline should ensure reliable software
- End to end automation : The activities from coding to release should be automated
- 100% source control : All the artifacts involved in the pipeline need to be version controlled (ex. Source code, automated test cases, reports, binaries etc.)
- Auto build quality : Pipeline should have quality auto-built by way of gating conditions
- Done is released : Pipeline should ensure that “done-ness” as per the definition of done is only released to production
- Continuous feedback : Tools provide continuous feedback by way of reports
- Customer appetite for tooling : The availability of budget from customer, existing tools and alliances, technology used in the project, feasibility of automation

The tools stacks are evolving and there are many vendors in this area.

Practical tips

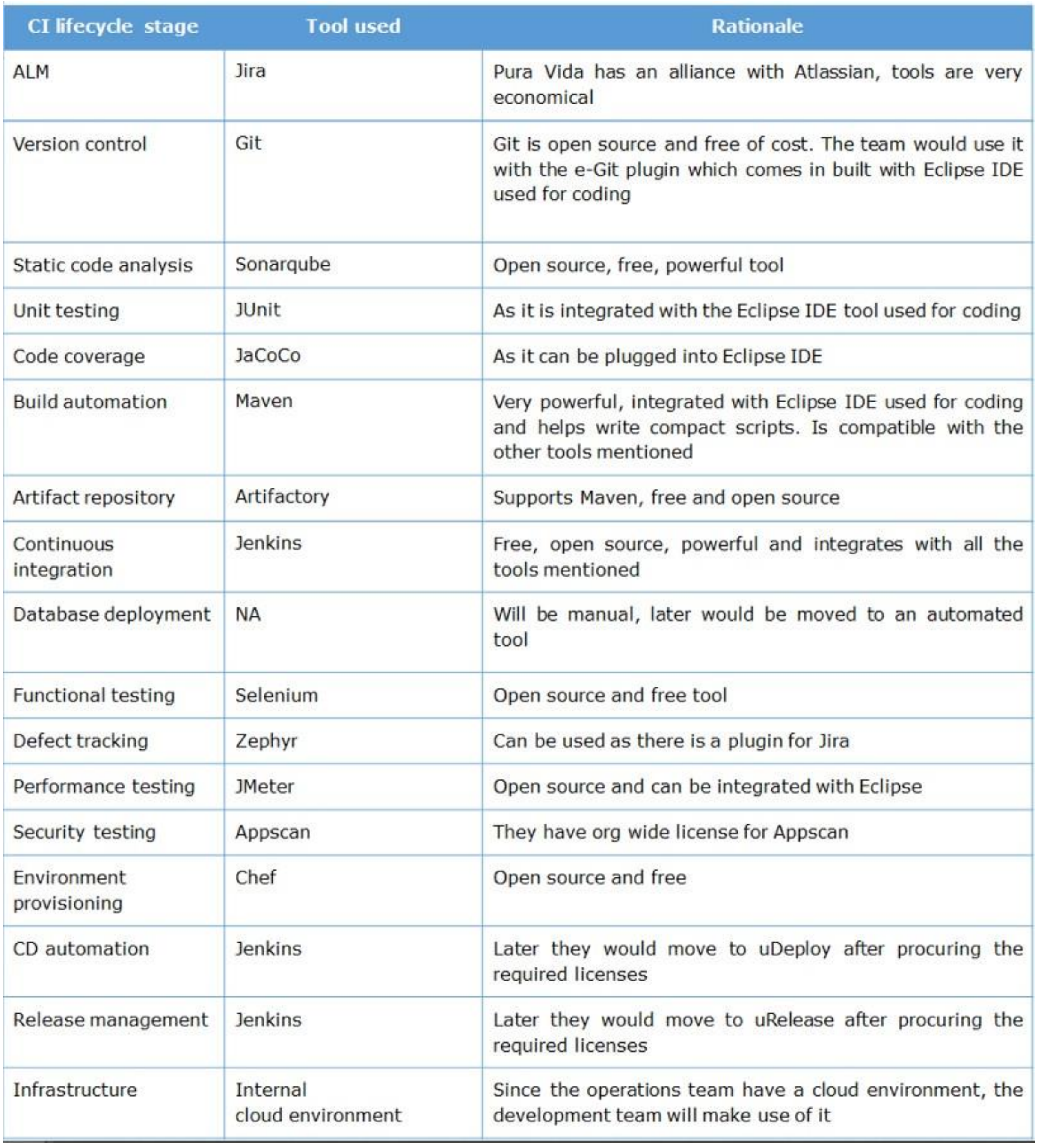
- Tooling is a consulting exercise.

The DevOps and Lean coach suggests using the Java and open source stack for the system development at "Project" for the reasons mentioned below.

Reasons

- The tools involved in this stack are primarily open source, free and powerful
- The project is an application development project using Java stack
- Quick availability of these tools and no overhead of maintenance of licenses

The team is advised to get the OSS (Open Source Software) compliance for the open source tools prior to the installation. OSS compliance refers to the compliance in terms of using approved and supported source code. There should be a policy and process to check the usage, purchase (as all open source software is not free), management and compliance(some can be used for training but payment is required if used commercially).Tools like Black Duck help in checking OSS compliance.



CI lifecycle stage	Tool used	Rationale
ALM	Jira	Pura Vida has an alliance with Atlassian, tools are very economical
Version control	Git	Git is open source and free of cost. The team would use it with the e-Git plugin which comes in built with Eclipse IDE used for coding
Static code analysis	Sonarqube	Open source, free, powerful tool
Unit testing	JUnit	As it is integrated with the Eclipse IDE tool used for coding
Code coverage	JaCoCo	As it can be plugged into Eclipse IDE
Build automation	Maven	Very powerful, integrated with Eclipse IDE used for coding and helps write compact scripts. Is compatible with the other tools mentioned
Artifact repository	Artifactory	Supports Maven, free and open source
Continuous integration	Jenkins	Free, open source, powerful and integrates with all the tools mentioned
Database deployment	NA	Will be manual, later would be moved to an automated tool
Functional testing	Selenium	Open source and free tool
Defect tracking	Zephyr	Can be used as there is a plugin for Jira
Performance testing	JMeter	Open source and can be integrated with Eclipse
Security testing	Appscan	They have org wide license for Appscan
Environment provisioning	Chef	Open source and free
CD automation	Jenkins	Later they would move to uDeploy after procuring the required licenses
Release management	Jenkins	Later they would move to uRelease after procuring the required licenses
Infrastructure	Internal cloud environment	Since the operations team have a cloud environment, the development team will make use of it

People Aspects

People challenges in "Project"

- The Dev, QA and Ops teams are separate
- Conflict in goals
  - Dev Team goals : Adapt to rapid changes and their implementation
  - QA Team goals: Write and execute test cases
  - Ops team goals: achieving stability & reliability

- Little collaboration between Dev, QA and Ops team and no shared responsibility
- Ops team not sensitized on the requirements and urgency of deployments
- Decision making is central and is not autonomous
- Geographically distributed teams

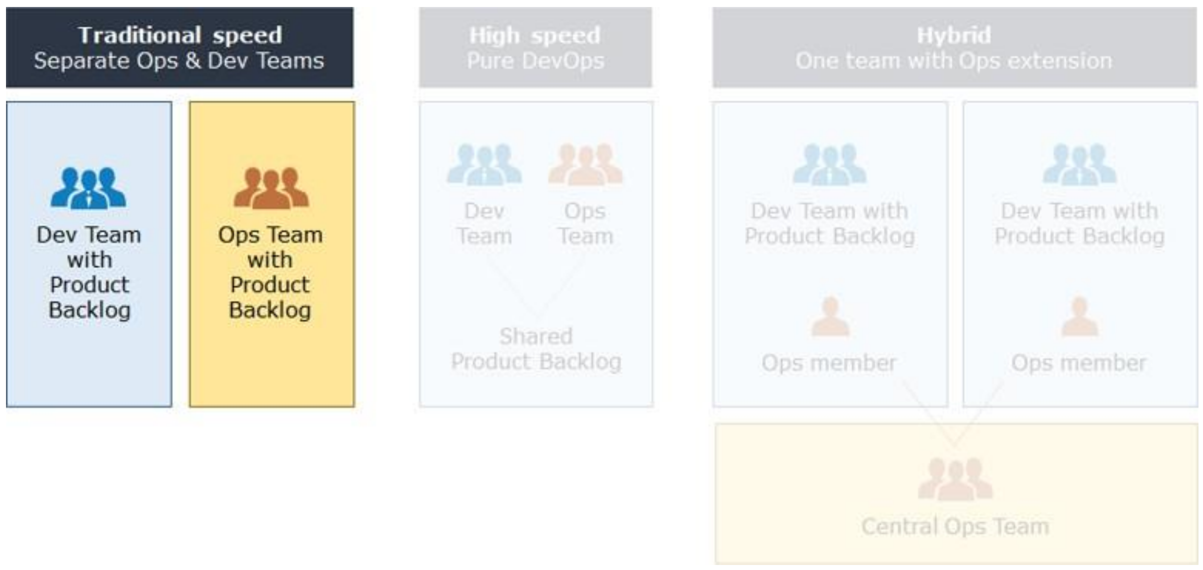
The DevOps & Lean coach now elaborate the people models available and what parameters are to be considered for choosing them.

Team Structure -Possible Organization:

Agile software development has broken down some of the isolation between requirements, analysis, development and testing teams. The objective of DevOps is to remove the silos between development (including testing) and operations teams and bring about collaboration between the teams.

However, since there are separate teams and also the fact the team may have niche skills rather than skills across the software development lifecycle, there could a phased approach to creating a pure DevOps team. Here are some of the possible team structures.

Team structure model 1: Separate dev and Ops teams


Traditional speed
Separate Ops & Dev Teams
Dev Team with Product Backlog
Ops Team with Product Backlog
High speed
Pure DevOps
Dev Team
Ops Team
Shared Product Backlog
Hybrid
One team with Ops extension
Dev Team with Product Backlog
Ops member
Dev Team with Product Backlog
Ops member
Central Ops Team

Key issue: Lack of collaboration between the teams as they are in silos.

However, it may not be possible to merge the teams, so the key is to improve the collaboration through common interventions.

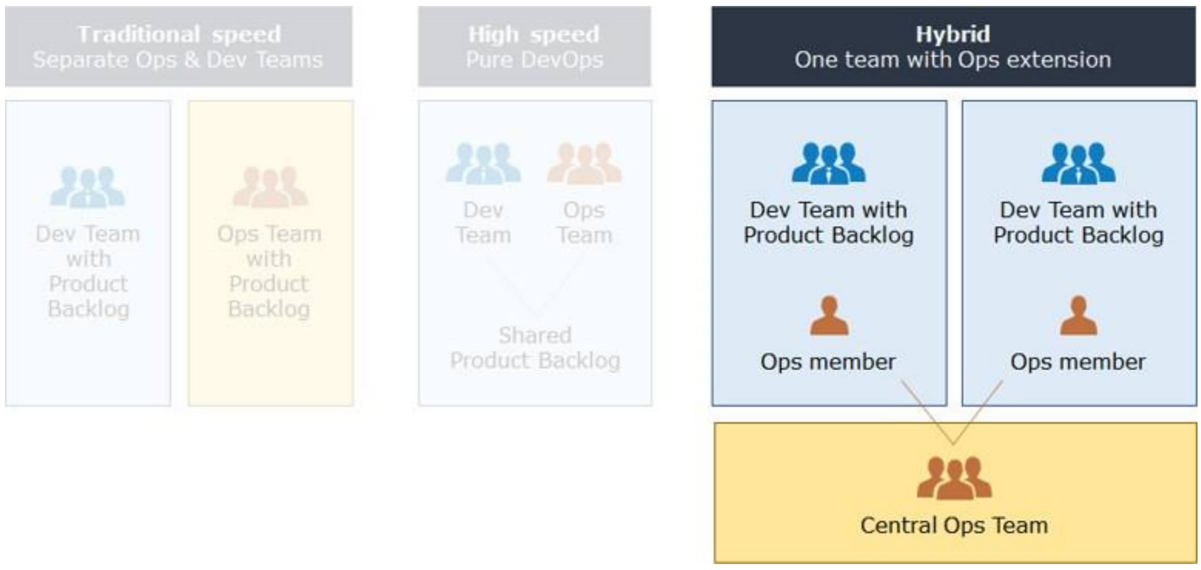
Salient Features

- Development and operations are separate (may apply to QA teams also)
- Interventions planned at regular intervals with no overhead processes

- Teams keep separate backlogs but take each other's stories in their backlogs

- Ops team gets knowledge about upcoming features, major design changes, possible impact on production
- Dev team understands what causes outages/ defects better, improves Dev processes to reduce impact (e.g. specific logging, perf testing for a cycle)
- Dev team improves dev processes over time by understanding Ops defects/outages better

Team structure model 2: One team with Ops extension


Traditional speed
Separate Ops & Dev Teams
Dev Team with Product Backlog
Ops Team with Product Backlog
High speed
Pure DevOps
Dev Team
Ops Team
Shared Product Backlog
Hybrid
One team with Ops extension
Dev Team with Product Backlog
Ops member
Dev Team with Product Backlog
Ops member
Central Ops Team

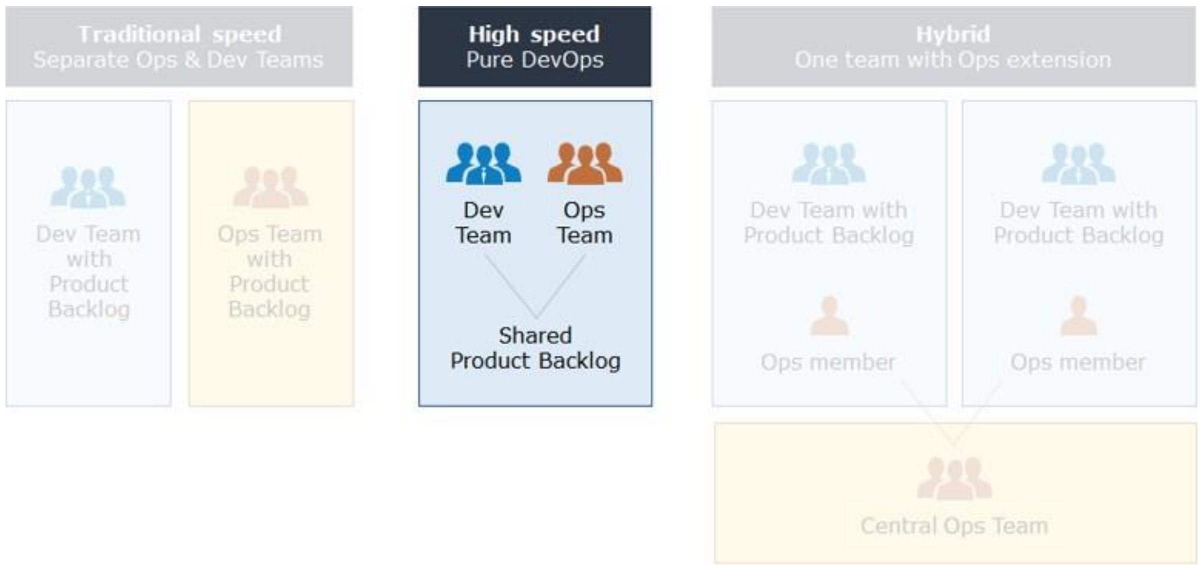
When a pure DevOps team cannot be constructed, a model closer to the pure devOps team can be constructed.

Salient Features

- A horizontal Ops team forms a backbone for all development teams
- It provides 24X7 support and performs the tasks which have larger impact on IT e.g. patch deployment
- Few Ops team members can become part of Dev team and perform tasks which are application specific.
- Ops team members can become part of Dev team
- Ops representative will focus on all the Ops activities which are this team/application specific while all centralized Ops activities will be taken by horizontal Ops team

When speed is increased, deployments are faster. Then teams realize that support service levels start dropping. That is when teams understand the importance of collaboration between development and ops teams.

Team structure model 3: Pure DevOps


Traditional speed
Separate Ops & Dev Teams
Dev Team with Product Backlog
Ops Team with Product Backlog
High speed
Pure DevOps
Dev Team
Ops Team
Shared Product Backlog
Hybrid
One team with Ops extension
Dev Team with Product Backlog
Ops member
Dev Team with Product Backlog
Ops member
Central Ops Team

The teams may be merged although DevOps skill set (end to end skills in a software lifecycle) may not be readily available.

Salient features

- Embedded team can be created by hiring people with blended skills or cross-training/on the job learning by Dev & Ops teams for each other's skills
- Team has single backlog with both Dev & Ops tasks
- Each team member is capable of selecting any item & work on it

Process Aspects:

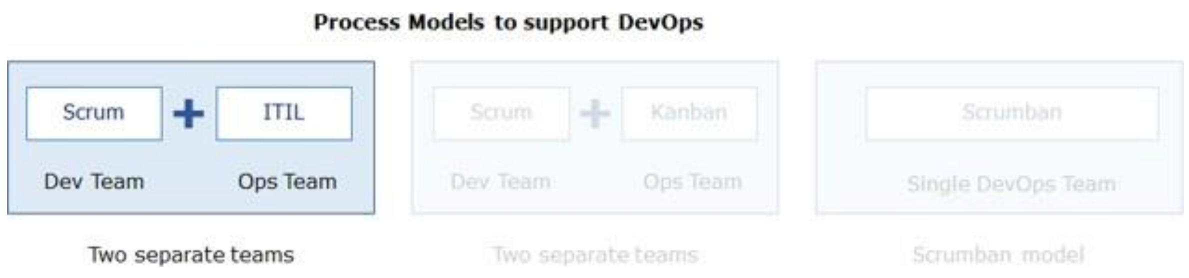
Process challenges in "Project"

- Delays due to formal knowledge transfer from Dev to Ops for every release
- Tedious Change Management process requiring lots of approvals
- Complex Release Management with manual checks impacts the operational efficiency

These challenges led to various issues. The coach suggests development and operations teams to follow a unified process.

When teams are merged, which process to follow gains more significance.

Model 1: Dev and Ops separate



The development team may be following an agile approach say Scrum as shown in the figure. The Ops team may be following another process, here ITIL (Information Technology Infrastructure Library. It is a framework for streamlining maintenance activities). Here is how the process can be fine-tuned for DevOps adoption.

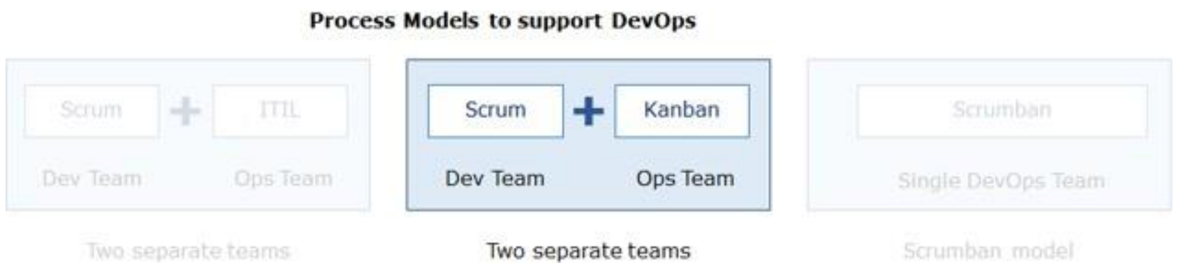
Process model

- Dev in Scrum and Ops in any iterative model
- Governance team (Program Manager) for conflict resolution
- Few team members cross-participate in daily standups

Limitations

- Frequent conflicts and less appreciation for each other's work
- Cross skilling of talents is not possible

Model 2: Dev and Ops separate but following similar process



The development and operations team may be following Agile approach say Scrum(by the development ream) and Kanban(by the operations team). Here is how the process can be fine-tuned for DevOps adoption.

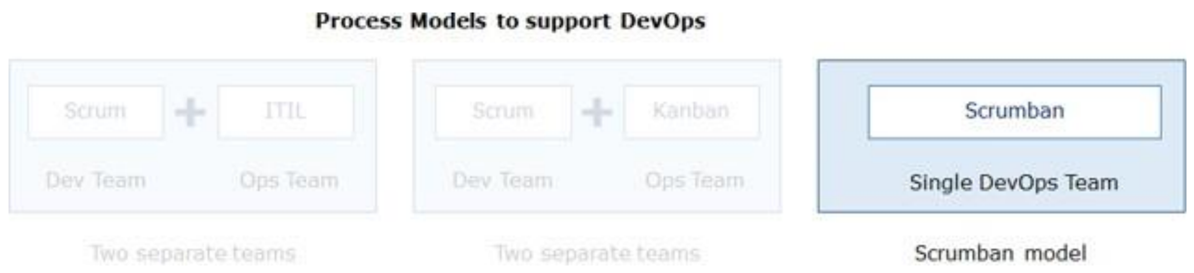
Process model

- One group does Scrum and the other Kanban as ONE team
- Two different product backlogs (PB), but single PO
- Dev team works on user stories and Ops works on high priority Kanban PB
- Any inter-dependent work items are prioritized by PO to resolve dependencies on time. Daily standup by both teams

Limitations

- Cross-skilling of talents is limited

Model 3: Unified process



When there is a unified DevOps team, they can follow a unified process like Scrumban as shown in the figure.

Process model

- Single PO with one PB
- Based on history, planned vs. unplanned effort considered across
- Team works on the prioritized user stories
- High priority is set for unplanned high severity incident,a team member having expertise in that takes up and resolves
- Cross skilling of talents is possible

Development team at "Project" follows Scrum and operations team adopts ITIL. The team starts with the first model and then move towards a unified process at "Project" for reasons stated below.

Reasons

- The development and operations teams have been separate at "Project". The team structure is optimized with this arrangement currently and hence it is difficult to have a unified process immediately.
- Over time, as the groups start to become cross-skilled, a unified process can be adopted.

UNIT- II

CI/CD: Introduction to Continuous Integration, Continuous Delivery and Deployment, Benefits of CI/CD, Metrics to track CICD practices

Introduction to CI/CD

CI/CD is a way of developing software in which you’re able to release updates at any time in a sustainable way. When changing code is routine, development cycles are more frequent, meaningful and faster.

“CI/CD” stands for the combined practices of Continuous Integration (CI) and Continuous Delivery (CD). Continuous Integration is a prerequisite for CI/CD, and requires:

- Developers to merge their changes to the main code branch many times per day.
- Each code merge to trigger an automated code build and test sequence. Developers ideally receive results in less than 10 minutes, so that they can stay focused on their work.

The job of Continuous Integration is to produce an artifact that can be deployed. The role of automated tests in CI is to verify that the artifact for the given version of code is safe to be deployed.

In the practice of **Continuous Delivery**, code changes are also continuously deployed, although the deployments are triggered manually. If the entire process of moving code from source repository to production is fully automated, the process is called Continuous Deployment.

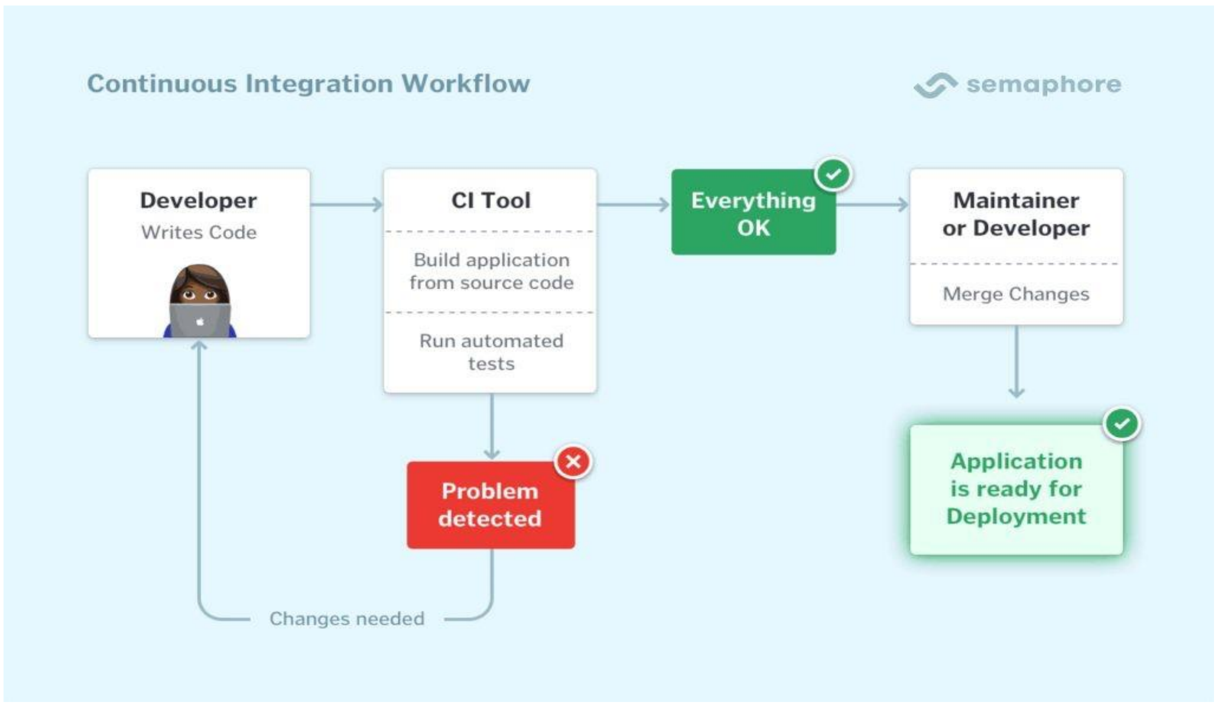
Continuous Integration (CI):

Continuous integration (CI) is a software development practice in which developers merge their changes to the main branch many times per day. Each merge triggers an automated code build and test sequence, which ideally runs in less than 10 minutes. A successful CI build may lead to further stages of continuous delivery.

If a build fails, the CI system blocks it from progressing to further stages. The team receives a report and repairs the build quickly, typically within minutes.

All competitive technology companies today practice continuous integration. By working in small iterations, the software development process becomes predictable and reliable. Developers can iteratively build new features. Product managers can bring the right products to market, faster. Developers can fix bugs quickly and usually discover them before they even reach users.

Continuous integration requires all developers who work on a project to commit to it. Results need to be transparently available to all team members and build status reported to developers when they are changing the code. In case the main code branch fails to build or pass tests, an alert usually goes out to the entire development team who should take immediate action to get it back to a "green" state.



Why do we need Continuous Integration?

In business, especially in new product development, we often don't have time or ability to figure everything upfront. Taking smaller steps helps us estimate more accurately and validate more frequently. A shorter feedback loop means having more iterations. And it's the number of iterations, not the number of hours invested, that drives learning.

For software development teams, working in long feedback loops is risky, as it increases the likelihood of errors and the amount of work needed to integrate changes into a working version software.

Small, controlled changes are safe to happen often. And by automating all integration steps, developers avoid repetitive work and human error. Instead of having people decide when and how to run tests, a CI tool monitors the central code repository and runs all automated tests on every commit. Based on the total result of tests, it either accepts or rejects the code commit.

Extension with Continuous Delivery

Once we automatically build and test our software, it gets easier to release it. Thus Continuous Integration is often extended with Continuous Delivery, a process in which code changes are also automatically prepared for a release (CI/CD).

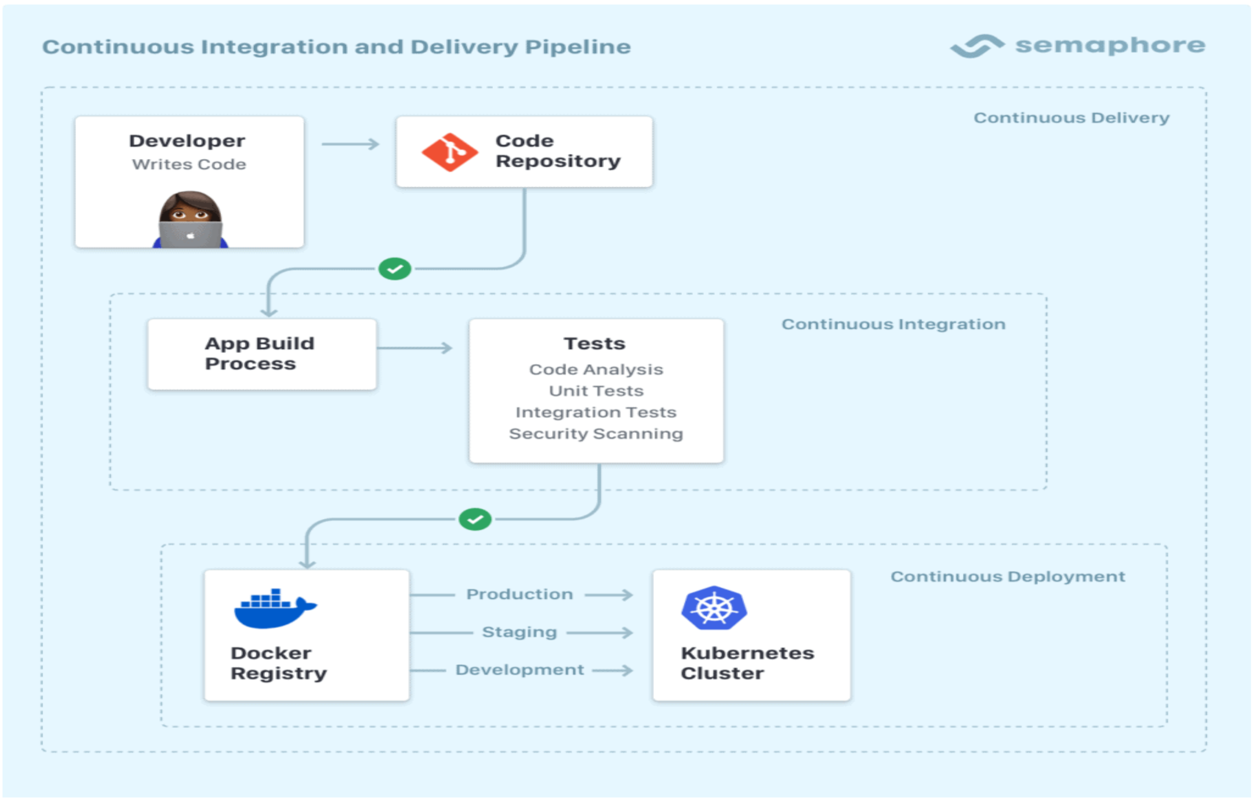
In a fine-tuned CI/CD process, all code changes are being deployed to a staging environment, a production environment, or both after the CI stage has been completed.

Continuous delivery can be a fully automated workflow. In that case, it’s usually referred to as Continuous Deployment. Or, it can be partially automated with manual steps at critical points. What’s common in both scenarios is that developers always have a release artifact from the CI stage that has gone through a standardized test process and is ready to be deployed.

CI and CD pipeline

CI and CD are often represented as a pipeline, where new code enters on one end, flows through a series of stages (build, test, staging, production), and published as a new production release to end users on the other end.

Each stage of the CI/CD pipeline is a logical unit in the delivery process. Developers usually divide each unit into a series of subunits that run sequentially or in parallel.



A Continuous Integration pipeline, extended with Continuous Delivery

For example, we can split testing into low-level unit tests, integration tests of system components working together, and high-level tests of the user interface.

Additionally, each stage in the pipeline acts as a gate that evaluates a certain aspect of the code. Problems detected in an early stage stop the code from progressing further through the pipeline. It doesn’t make sense to run the entire pipeline if we have fundamental bugs in code to fix first. Detailed results and logs about the failure are immediately sent to the team to fix.

Because CI/CD pipelines are so integral to the development process, high performance and high availability are paramount for developer productivity.

Benefits of Continuous Integration and Continuous Delivery

1. Smaller Code Changes

One technical advantage of continuous integration and continuous delivery is that it allows you to integrate small pieces of code at one time. These code changes are simpler and easier to handle than huge chunks of code and as such, have fewer issues that may need to be repaired at a later date.

Using continuous testing, these small pieces can be tested as soon as they are integrated into the code repository, allowing developers to recognize a problem before too much work is completed afterward. This works really well for large development teams who work remotely as well as those in-house as communication between team members can be challenging.

2. Fault Isolations

Fault isolation refers to the practice of designing systems such that when an error occurs, the negative outcomes are limited in scope. Limiting the scope of problems reduces the potential for damage and makes systems easier to maintain.

Designing your system with CI/CD ensures that fault isolations are faster to detect and easier to implement. Fault isolations combine monitoring the system, identifying when the fault occurred, and triggering its location. Thus, the consequences of bugs appearing in the application are limited in scope. Sudden breakdowns and other critical issues can be prevented from occurring with the ability to isolate the problem before it can cause damage to the entire system.

3. Faster Mean Time To Resolution (MTTR)

MTTR measures the maintainability of repairable features and sets the average time to repair a broken feature. Basically, it helps you track the amount of time spent to recover from a failure.

CI/CD reduces the MTTR because the code changes are smaller and fault isolations are easier to detect. One of the most important business risk assurances is to keep failures to a minimum and quickly recover from any failures that do happen. Application monitoring tools are a great way to find and fix failures while also logging the problems to notice trends faster.

4. More Test Reliability

Using CI/CD, test reliability improves due to the bite-size and specific changes introduced to the system, allowing for more accurate positive and negative tests to be conducted. Test reliability within CI/CD can also be considered *Continuous Reliability*. With the continuous merging and releasing of new products and features, knowing that quality was top of mind throughout the entire process assures stakeholders their investment is worthwhile.

5. Faster Release Rate

Failures are detected faster and as such, can be repaired faster, leading to increasing release rates. However, frequent releases are possible only if the code is developed in a continuously moving system.

CI/CD continuously merges codes and continuously deploys them to production after thorough testing, keeping the code in a release-ready state. It’s important to have as part of deployment a production environment set up that closely mimics that which end-users will ultimately be using. Containerization is a great method to test the code in a production environment to test only the area that will be affected by the release.

6. Smaller Backlog

Incorporating CI/CD into your organization’s development process reduces the number of non-critical defects in your backlog. These small defects are detected prior to production and fixed before being released to end-users.

The benefits of solving non-critical issues ahead-of-time are many. For example, your developers have more time to focus on larger problems or improving the system and your testers can focus less on small problems so they can find larger problems before being released. Another benefit (and perhaps the best one) is keeping your customers happy by preventing them from finding many errors in your product.

7. Customer Satisfaction

The advantages of CI/CD do not only fall into the technical aspect but also in an organization scope. The first few moments of a new customer trying out your product is a make-or-break-it moment.

Don’t waste first impressions as they are key to turning new customers into satisfied customers. Keep your customers happy with fast turnaround of new features and bug fixes. Utilizing a CI/CD approach also keeps your product up-to-date with the latest technology and allows you to gain new customers who will select you over the competition through word-of-mouth and positive reviews.

Your customers are the main users of your product. As such, what they have to say should be taken into high consideration. Whether the comments are positive or negative, customer feedback and involvement leads to usability improvements and overall customer satisfaction.

Your customers want to know they are being heard. Adding new features and changes into your CI/CD pipeline based on the way your customers use the product will help you retain current users and gain new ones.

8. Increase Team Transparency and Accountability

CI/CD is a great way to get continuous feedback not only from your customers but also from your own team. This increases the transparency of any problems in the team and encourages responsible accountability.

CI is mostly focused on the development team, so the feedback from this part of the pipeline affects build failures, merging problems, architectural setbacks, etc. CD focuses more on getting the product quickly to the end-users to get the much-needed customer feedback. Both

CI and CD provide rapid feedback, allowing you to steadily and continuously make your product even better.

9. Reduce Costs

Automation in the CI/CD pipeline reduces the number of errors that can take place in the many repetitive steps of CI and CD. Doing so also frees up developer time that could be spent on product development as there aren't as many code changes to fix down the road if the error is caught quickly. Another thing to keep in mind: increasing code quality with automation also increases your ROI.

10. Easy Maintenance and Updates

Maintenance and updates are a crucial part of making a great product. However, it's important to note within a CI/CD process to perform maintenance during downtime periods, also known as the non-critical hour. Don't take the system down during peak traffic times to update code changes.

Upsetting customers is one part of the problem, but trying to update changes during this time could also increase deployment issues. Make sure the pipeline runs smoothly by incorporating when to make changes and releases. A great way to ensure maintenance doesn't affect the entire system is to create microservices in your code architecture so that only one area of the system is taken down at one time.

Find bugs earlier, fix them faster

The automated testing process can include many different types of checks:

- Verify code correctness;
- Validate application behavior from a customer's perspective;
- Compare coding style with industry-standard conventions;
- Test code for common security holes;
- Detect security updates in third-party dependencies;
- Measure test coverage: how much of your application's features are covered by automated tests.

Building these tests into your CI pipeline, measuring and improving the score in each is a guaranteed way to maintain a high quality of your software.

A CI tool provides instant feedback to developers on whether the new code they wrote works, or introduces bugs or regression in quality. Mistakes caught early on are the easiest to fix.

Continuous integration tools

Continuous integration is, first and foremost, a process derived from your organization's culture. No tool can make developers collaborate in small iterations, maintain an automated build process and write tests. However, having a reliable CI tool to run the process is crucial for success.

Semaphore is designed to enable your organization to build a high-performing, highly available CI process with almost unlimited scale. Semaphore provides support for popular languages on Linux and iOS, with the ability to run any Docker container that you specify.

With the advantages of tight integration with GitHub and ability to model custom pipelines that can communicate with any cloud endpoint, Semaphore is highly flexible. Embracing the serverless computing model, your CI process scales automatically with no time spent on queues. Your organization has nothing to maintain and pays based on time spent on execution.

Whatever CI tool you choose, we recommend that you pick the one which maximizes your organization's productivity. Feature-wise, your CI provider should be a few steps ahead of your current needs, so that you're certain that it can support you as you grow.

Build and testing tools

You can consider all tools used within your build and test steps as your CI value chain. This includes tools like code style and complexity analyzer, build and task automation tool, unit and acceptance testing frameworks, browser testing engine, security and performance testing tools, etc.

Choose tools which are widely adopted, are well documented and are actively maintained.

Wide adoption is reflected by a large number of package downloads (often in millions) and open source contributors. Googling common use cases returns a solid number of examples from credible sources.

Well documented tools are easy to get started with. A searchable documentation website provides information on accomplishing more complex tasks. You will often find books covering their use in depth.

Actively maintained tools have had their latest release recently, at least within the last 6 months. The last commit in source code happened less than a month ago. The best tools evolve with the ecosystem and provide timely bug fixes and security updates.

Continuous Integration best practices

The following is a brief summary of CI best practices.

Treat master build as if you're going to make a release at any time. Which implies some team-wide don'ts:

- Don't comment out failing tests. File an issue and fix them instead.
- Don't check-in on a broken build and never go home on a broken build.

Keep the build fast: up to 10 minutes. Going slower is good but doesn't enable a fast-enough feedback loop.

Parallelize tests. Start by splitting by type (eg. unit and integration), then adopt tools that can parallelize each.

Have all developers commit code to master at least 10 times per day. Avoid long-running feature branches which result in large merges. Build new features iteratively and use feature flags to hide work-in-progress from end users.

Wait for tests to pass before opening a pull request. Keep in mind that a pull request is by definition a call for another developer to review your code. Be mindful of their time.

Test in a clone of the production environment. You can define your CI environment with a Docker image, and make the CI environment fully match production. An alternative is to customize the CI environment so that bugs due to difference with production almost never happen.

Use CI to maintain your code. For example, run scheduled workflows to detect newer versions of your libraries and upgrade them.

Keep track of key metrics: total CI build time (including queue time, which your CI tool should maintain at zero) and how often your master is red.

Continuous integration metrics and KPIs
Key performance indicators (KPIs) are used in practically every industry to show data around goals. In enterprises, KPIs are often applied to business functions as well as individuals to measure performance and progress towards certain initiatives. Like any business investment, teams need to be able to show how their continuous integration solution is meeting their needs.

Continuous integration metrics can range significantly, depending on a team’s priorities or even the industry. Successful CI strategies can look different from team to team, but there are metrics that can highlight potential problems or areas of opportunity for any team.

1. Cycle time

Cycle time is the speed at which a DevOps team can deliver a functional application, from the moment work begins to when it is providing value to an end user. In GitLab, we call this value stream analytics and it measures how long it takes your team to work in each stage of the developer workflow. By answering the question “How long does it take us to create something?” teams create a baseline that can then be revisited and improved upon.

2. Time to value

Once code is written, how long before it’s released? While cycle time measures the process, as a whole, time to value (TTV) focuses on the release process. This delay from when code is written to running in production is a bottleneck for many organizations. Having robust continuous delivery can help to overcome this barrier to quick deployments.

3. Uptime, error rate, infrastructure costs

Uptime is one of the biggest priorities for the ops team. Uptime is simply a measure of stability and reliability – how often is everything working as it should? With the right CI/CD strategy that automates the development lifecycle, ops leaders can focus more of their time on system stability and less time on workflow issues. If uptime and error rates seem high, it can illustrate a common CI/CD challenge between dev and ops teams. Operations goals are a key indicator of process success.

4. Team retention rate

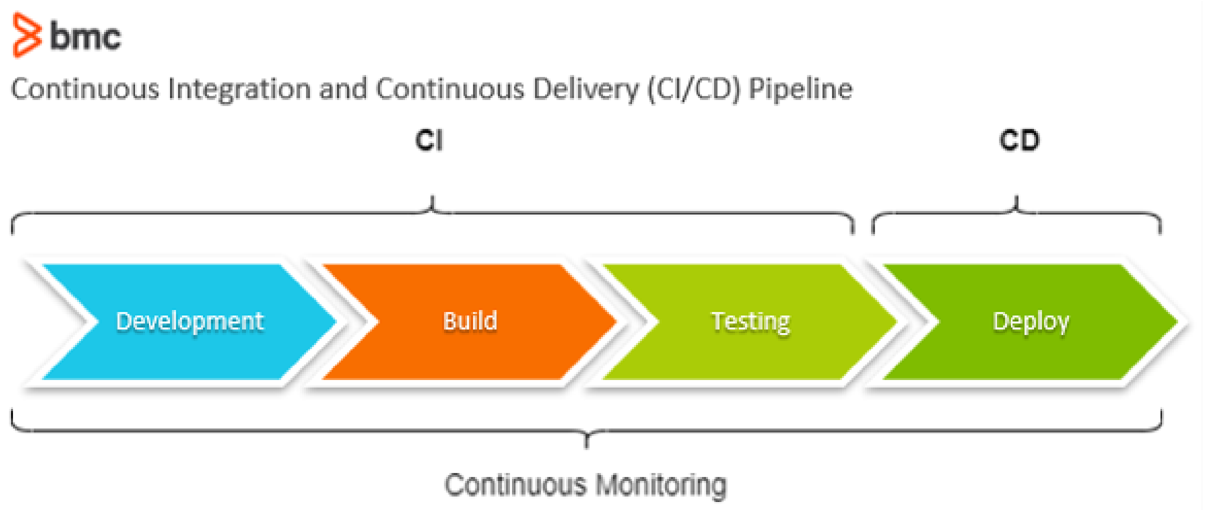
While happiness is a metric that's nearly impossible to measure, happy developers do tend to stick around. Retention rates can't measure happiness, but it can shed some light on how well processes and applications are working for the team. It might be tough for developers to speak up if they don't like how things are going, but looking at retention rates can be one step in identifying potential problems.

DevOps organizations monitor their CI/CD pipeline across three groups of metrics:

- Automation performance
- Speed
- Quality

With continuous delivery of high-quality software releases, organizations are able to respond to changing market needs faster than their competition and maintain improved end-user experiences. How can you achieve this goal?

Let's discuss some of the critical aspects of a healthy CI/CD pipeline and highlight the key metrics that must be monitored and improved to optimize CI/CD performance.


bmc
Continuous Integration and Continuous Delivery (CI/CD) Pipeline
CI
CD
Development
Build
Testing
Deploy
Continuous Monitoring

Metrics for optimizing the DevOps CI/CD pipeline

Now, let’s turn to actual metrics that can help you determine how mature your DevOps pipeline is. We’ll look at three areas.

Agile CI/CD Pipeline

In regard to delivering high quality software, infusing performance and security into the code from the ground up, developers should be able to write code that is QA-ready.

DevOps organizations should introduce test procedures early during the SDLC lifecycle—a practice known as shifting left—and developers should respond with quality improvements well before the build reaches production environments.
DevOps organizations can measure and optimize the performance of their CI/CD pipeline by using the following key metrics:

- **Test pass rate.** The ratio between passed test cases with the total number of test cases.
- **Number of bugs.** The number of issues that cause performance issues at a later stage.
- **Defect escape rate.** The number of issues identified in the production stage compared to the number of issues identified in pre-production.
- **Number of code branches.** Number of feature components introduced into the development project.

Automation of CI/CD & QA

Automation is the heart of DevOps and a critical component of a healthy CI/CD pipeline. However, DevOps is not solely about automation. In fact, DevOps thrives on automation adopted strategically—to replace repetitive and predictable tasks by automation solutions and scripts.
Considering the lack of skilled workforce and the scale of development tasks in a CI/CD pipeline, DevOps organizations should maximize the scope of their automation capabilities while also closely evaluating automation performance. They can do so by monitoring the following automation metrics:

- **Deployment frequency.** Measure the throughput of your DevOps pipeline. How frequently can your organization deploy by automating the QA and CI/CD processes?
- **Deployment size.** Does automation help improve your code deployment capacity?
- **Deployment success.** Do frequent deployments cause downtime and outages, or other performance and security issues?

Infrastructure Dependability

DevOps organizations are expected to improve performance *without*disrupting the business. Considering the increased dependence on automation technologies and a cultural change focused on rapid and continuous delivery cycles, DevOps organizations need consistency of performance across the SDLC pipeline.
Dependability of infrastructure underlying high performance CI/CD pipeline responsible for hundreds (at times, thousands) of delivery cycles on a daily basis is therefore critical to the success of DevOps. How do you measure the dependability of your IT infrastructure?

Here are a few metrics to get you started:

- **MTTF, MTTR, MTTD: Mean Time to Failure/Repair/Diagnose.**These metrics quantify the risk associated with potential failures and the time it takes to recover to optimal performance. Learn more about reliability calculations and metrics for infrastructure or service performance.
- **Time to value.** Another key metric is the speed of Continuous Delivery cycle release performance. It refers to the time taken before a complete written software build is released into production. The delaying duration may be caused by a number of factors, including infrastructure resources and automation capabilities available to test and process the build, as well as the governance process necessary for final release.
- **Infrastructure utilization.** Evaluate the performance of every service node, server, hardware, and virtualized IT components. This information not only describes the computational performance available for CI/CD teams but also creates vast volumes of data that can be studied for security and performance issues facing the network infrastructure.

Continuous Delivery metrics you should be tracking before you start:

1. Lead time to production

The clock starts ticking on this one at the point where enough is known about the work to begin and ends when the work is live in production. On this metric, Bowler points out that if lead time is “excessively long then we might want to track just cycle time.” This is because, “When teams are first starting their journey to continuous delivery, lead times to production are often measured in months and it can be hard to get sufficient feedback with cycles that long.” Measuring cycle time in the interim, or the time between when work starts on an item and that work meets the team’s definition of ‘done,’ “can be a good intermediate measurement while we work on reducing lead time to production.”

2. Number of bugs

“Shipping buggy code is bad and this should be obvious,” writes Bowler, “Continuously delivering buggy code is worse.” If there are quality issues with the code then continuous delivery is only going to get more bugs out into the world faster, causing more headaches later on.

3. Defect resolution time

Again, quality code is a prerequisite to successful Continuous Delivery. One way to measure quality, along with the team's commitment to quality, is to track the life of bugs. "I've seen teams that had bug lists that went on for pages and where the oldest was measured in years," Bowler observes, adding that, "Really successful teams fix bugs as fast as they appear."

4. Regression test duration

Tracking the time it takes to complete a full regression test "is important because we would like to do a full regression test prior to any production deploy." For teams with manual testing practices, this metric will be measured in "weeks or months," and "minutes or hours" for teams who primarily use automated tests. The argument that Bowler is putting forth here is that regression testing helps to reduce the risk of a failed deployment, and shortening this cycle down as much as possible will help increase the probability that continuous delivery will be successful.

5. Broken build time

"We all make mistakes," writes Bowler, but "the question is how important is it to the team to get that build fixed?" This is, again, a measure of the team's commitment to quality as a prerequisite to Continuous Delivery. If a team lets a build stay broken "for days at a time," continuous delivery won't be an achievable goal.

6. Number of code branches

Continuous Delivery advocates for main trunk development so that everything is maintained in the same pipeline. Tracking the number of branches you have to the code "now and tracking that over time will give you some indication of where you stand" in your preparedness to begin continuous delivery. Bowler adds that "if your code isn't in version control at all then stop taking measurements and just fix that one right now." No advanced practice like continuous integration or continuous delivery is possible without managing and tracking code in version control.

7. Production downtime during deployment

"Some applications such as batch processes may never require zero-downtime deploys," writes Bowler, stipulating that, "Interactive applications like web apps absolutely do." There is a cost to downtime for customer-facing or revenue-producing applications, and for those, the goal should be zero downtime during deployments. Bowler writes that "If you achieve zero-downtime deploys then stop measuring this one."

UNIT- III

Devops Maturity Model: Key factors of DevOps maturity model, stages of Devops maturity Model, DevOps maturity Assessment.

Understanding DevOps Maturity

By definition, DevOps Maturity is described as a model that determines an organization's standing in **principles of DevOps** journey along with deciding what more to be accomplished to achieve the desired results.

Understanding DevOps adoption 'as a continuous journey, not a destination' stands crucial to achieving DevOps maturity.

The majority of DevOps monitoring solutions are compatible with on-premise, cloud, and containerized infrastructure, which ensures a smooth DevOps transformation process.

The DevOps maturity model determines growth through continuous learning from both teams and organizational perspectives. More the capabilities and skills, more will be the ability to handle issues of scale and complexities.

Organizational DevOps maturity can be gauged by their abilities in the following four areas:

1) Culture and Strategy

DevOps has to be understood as a culture-driven approach that brings together different teams, driving them towards a common objective. Transition to **DevOps operating model** means a transformation in the organization's operating culture backed by a set of policies and process frameworks. So, that needs proper planning and perfect strategy.

2) Automation

Automation is key to continuous delivery and continuous deployment **tools** in the DevOps process. By automating repetitive tasks, the automation process eases development, testing and production in a DevOps cycle, thus saving time and enhancing resource efficiency.

3) Structure and Process

Modern-day IT functioning is process-oriented and involves processes across all stages of the Software Development Life Cycle (SDLC). This has advanced in a DevOps environment, where every stage is a set of procedures in line with corporate policies and business objectives.

4) Collaboration and Sharing

This is the most critical aspect of **principles of DevOps culture**. Collaboration and sharing are key to DevOps and teams (on the same location or a different location) will need to align tools and resources towards achieving common goals and objectives.

Organizations commonly find themselves in one of the following stages as part of their DevOps operational model journey:

- **Unconscious incompetence:** Organizations fail to understand **DevOps challenges** and its advantages
- **Conscious incompetence:** Organizations still see siloed processes even after 12-18 months of DevOps journey with some automation
- **Conscious competence:** After four years of DevOps implementation journey and successful automation, organizations focus on collaboration across teams and streamline sharing mechanism
- **Unconscious competence:** Here, organizations are all set with structured frameworks, in-depth collaboration, the concrete process for effective sharing

DevOps Maturity Model

A perfect DevOps maturity model determines DevOps maturity in three ways:

- Assessment of the current state of capabilities
- Identifying areas of improvement
- Outlining steps to achieve desired DevOps goals

In line with these three steps, the DevOps maturity block verifies maturity in building, deploying and testing stages across application, data and infrastructure levels:

1) DevOps Maturity for Application – Determines DevOps maturity by the ease in code movement from Development to Production phase. Achieving this requires having builds, tests, code coverage, security scans and monitoring as automated components of the deployment pipeline.

2) DevOps Maturity by Data – Determines DevOps maturity by ability to clear path to automate changes to data and validate functionality regularly, through DataOps.

3) DevOps Maturity by Infrastructure – Determines DevOps maturity by ability to ease infrastructure using capabilities around automation, streamlining and enabling self-service to provision environments, among other tasks.

Key Factors of DevOps Maturity Model

1) Automation

Automation is the buzzword of DevOps culture. DevOps places a high reliance on automation. Automation involves the deployment of several technologies to facilitate faster execution of the various functionalities that are a part of the software development cycle.

The deployment of automation also accelerates the transmission of updates and feedback among the multi-functional teams. Automation thus emphasizes minimizing the need for human beings to manually intervene in the operations associated with software development.

In fact, major DevOps processes like continuous integration, continuous delivery, and log analytics can be accomplished manually. But it will be at the expense of additional overheads in terms of communication, coordination, and time consumption. Automation facilitates faster execution of several operations as well as thereby guaranteeing better efficiency and time savings.

2) Collaboration

Collaboration is what enables the transformation of business ideas that are in their nascent stage into reality. DevOps collaboration is all about enhancing the quality and efficiency of the software development life cycle (SDLC) by ensuring that all the multi-functional teams and their members are efficiently kept in the loop as far as the transfer of information is concerned.

Information handoffs enable an organization to quickly act on any critical issues like bugs, errors, and other glitches.

3) Culture and Organization

DevOps is viewed as a cultural shift. To ensure the success of DevOps it is important to get the entire organization, from the top-level management leaders to the employees to be on the same page.

Culture and organizational change is the biggest roadblock to DevOps transformation as per a 2022 report by Gartner. Once everyone in an organization is convinced of the need to adopt a DevOps culture, the transformation to it will be more smooth and more successful.

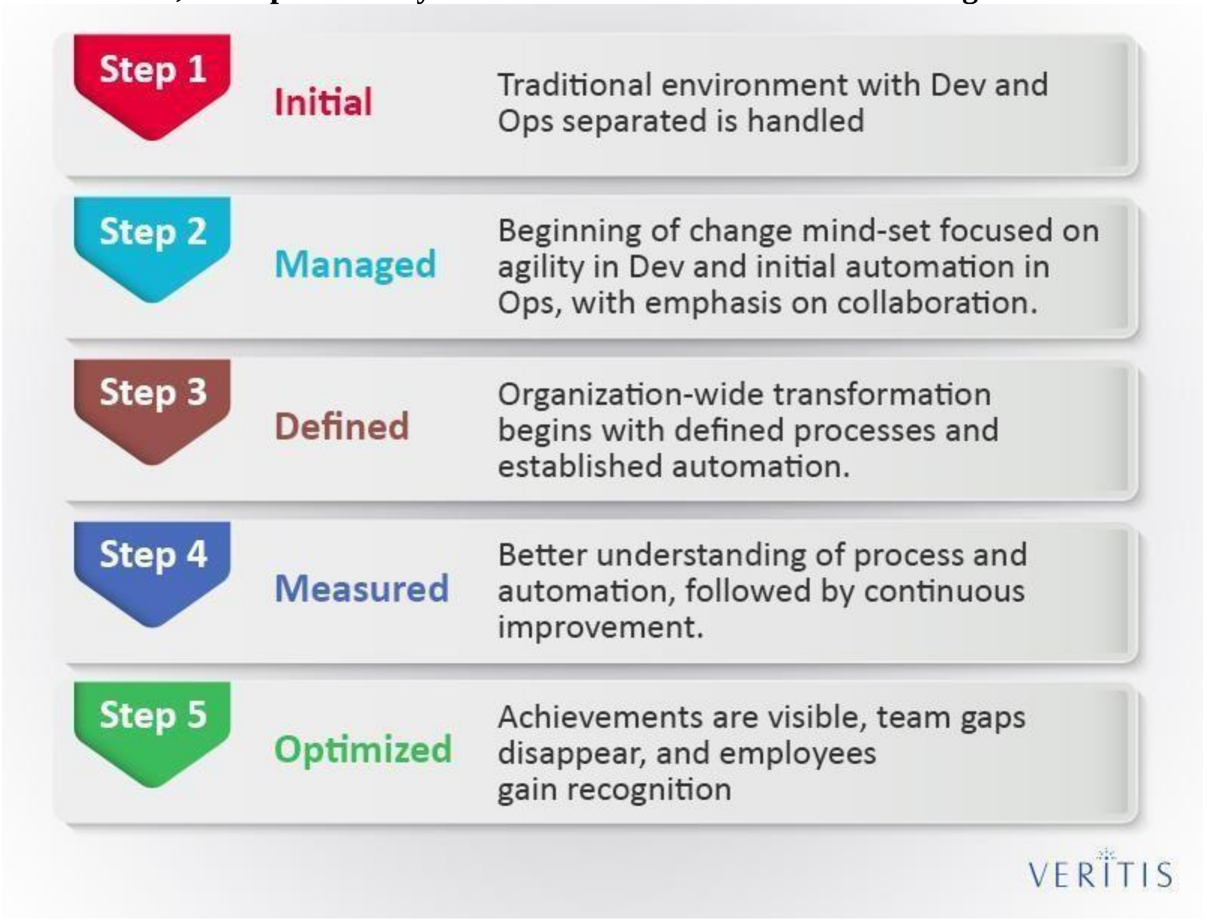
4) CI/CD

Continuous integration and continuous delivery/ deployment emphasize delivering higher quality software and facilitating bugs to be identified in the early stages of the development cycle by testing them earlier on. CI/CD is identified as a linchpin component of the DevOps strategy. It is good to consider deploying these practices that help in faster identification of issues as compared to deploying manual techniques and resulting in wastage of time. A scalable pipeline enables new requirements and features to be added easily.

5)Processes

Most organizations have now become largely process-oriented with functionalities being available for accomplishing A to Z of the multitude of processes in an organization. Unlike in the past when many processes were often required to rely on some specific sets of tools, today we have dedicated tools for accomplishing multiple processes and operations. Hence it is important to identify an apt tool depending on the nature of a process to ensure the success of the DevOps strategy.

Summarized, DevOps Maturity model involves five transformation stages:



Step 1	Initial	Traditional environment with Dev and Ops separated is handled
Step 2	Managed	Beginning of change mind-set focused on agility in Dev and initial automation in Ops, with emphasis on collaboration.
Step 3	Defined	Organization-wide transformation begins with defined processes and established automation.
Step 4	Measured	Better understanding of process and automation, followed by continuous improvement.
Step 5	Optimized	Achievements are visible, team gaps disappear, and employees gain recognition

1) Stage-1: Initial

Traditional environment with Dev and Ops separated is handled.

2) Stage-2: Managed

The beginning of change mindset focused on agility in Dev and initial automation in Ops, with emphasis on collaboration.

3) Stage-3: Defined

Organization-wide transformation begins with defined processes and established automation.

4) Stage-4: Measured

A better understanding of process and automation, followed by continuous improvement.

5) Stage-5: Optimized

Achievements are visible, team gaps disappear, and employees gain recognition.

While these 5 stages make a complete DevOps maturity model, it’s imperative for enterprises to keep checking their maturity at every step, and eventually identify focus areas and ways to evolve in the overall journey.

Measure in a DevOps Maturity Model

There are a set of parameters to be measured at every stage of the DevOps Maturity Model to confirm an organization’s level of DevOps maturity. These measures ideally define the direction the organization is advancing in its DevOps implementation journey. They are:

- Number of completed projects and the release frequency should ideally high resulting in ROI
- Percentage of successful deployments should maintain an edge over unsuccessful ones
- Mean Time To Recovery (MTTR) from an unexpected incident/failure from the time of occurrence, should be nil or as low as possible
- Lead time, from development of code to deployment in production, should be satisfactory
- Deployment frequency to determine the frequency of new code deployments

The stage-wise process and the above parameters define an organization’s DevOps maturity success.

DevOps Maturity Linked to Security

DevOps maturity is directly linked to DevOps security. As organizations progress in DevOps journey, the competitive edge becomes a pressing demand calling for faster release cycles and digital innovation demands a strong pitch.

Development, operations, and security are the three words that make up the DevOps security concept. The objective is to eliminate any barriers that might exist between IT operations and software development.

This is where the challenge of security starts becoming more serious and which is why DevOps culture calls for reconsidering security practices.

Eventually, organizations will have to make security an integral part of their DevOps process and take it closer to all application development stages.

DevOps experts work with security personnel for early security integration at the maturity level across all parts of the Software Development Lifecycle.

This can happen through effective DevSecOps implementation. solutions like Containerization can also help to some extent in addressing issues continuously by limiting the vulnerable resources.
Moreover, Security and DevOps teams can collaborate in applying security policies and frameworks to all the DevOps tools and resources.

Business Benefits of DevOps Maturity

Giving a complete picture of an organization's DevOps team standing, the DevOps maturity model presents a wide range of business benefits:

- Faster adaptability to change
- Ability to tap opportunities
- Identifying areas of fulfillment
- Improved scalability
- Operational efficiency
- Increased delivery speeds
- Enhanced quality

More such benefits of DevOps are part of the Maturity model that gives you the ability to witness the full DevOps potential.

DevOps Best Practices

Making the switch to DevOps offers numerous advantages. Despite having innovative approaches to improve the processes, DevOps challenges won't be entirely new. The following are some of the DevOps best practices

1) Collaborating teams

2) Continuous Integration (CI)

3) Continuous Development (CD)

4) Continuous Testing

5) Customer Satisfaction

6) Effective Leadership

7) Effective Tools

StagesofDevOpsMaturity

1)Nascent Stage

The initial stage is the nascent phase of DevOps transformation. As far as this stage is concerned it marks the transformation of an organization from its traditional development approach to DevOps. The limited understanding of DevOps makes an organization write off DevOps. This can also be due to fear of change from what is prevalent as is a normal situation. So this lack of understanding about DevOps often acts as a roadblock as far as an organization's transit to DevOps is concerned.

2) Managed Stage
In this phase, the need to implement the change is manifested among the various levels within the organization. So in the endeavor to accomplish this change, an organization adopts associated DevOps practices like automation, CI/CD practices, continuous testing, and continuous monitoring and emphasizes processes like continuous feedback, collaboration, and communication transfer among the various cross-functional teams.

3) Measured Stage
In this stage, the success of the adopted strategies is measured and it is improvised further by adopting continuous improvement methods. A number of parameters are deployed that can help an organization evaluate the success of its DevOps maturity model. As far as this stage is concerned, the DevOps teams create dashboards to view and share their insights as well as track how these changes affect an application, its performance, and health.

DevOps Maturity Assessment Factors

1) Deployment Frequency

Deployment frequency is one of the most often used ways to estimate the success of the DevOps maturity model by evaluating agility and efficiency. It is a measure of the frequency with which teams deploy the code.

2) Mean Time to Recovery

The mean time to recovery or repair is the time required to recover from a production failure. As far as teams with high DevOps maturity are concerned, MTTR will be considerably less and it is bound to become a minimum on acquiring a greater scale of DevOps maturity.

3) Lead time

Lead time is identified as the period that an organization takes when committing a code change to its very deployment. A successful DevOps approach enables teams to more frequently release their products and thus ensure that the lead time is kept minimized.

4) Percentage of Successful Deployments

The deployment success rate can be easily estimated by taking the ratio of the total number of successful deployments by the total number of deployments as far as a particular period of time is concerned.

5) Change Failure Rate

It is the percentage of code changes to be incorporated after the production and is a measure of the percentage of deployments that causes a failure in production. Analyzing the change failure rate helps an organization minimize the overall lead time and accelerate the velocity of software delivery.

The Merits of the DevOps Maturity Model

1) Agility

Business agility is one of the key benefits of the DevOps maturity model. An organization that has transcended higher levels of DevOps maturity can easily respond to the changing requirements of a business on the fly and thus ship their products at a faster pace and also incorporate new features by enabling easier product upgrades. Also, the deployment of automation solutions ensures that an organization can effectively get around the complexities associated with manual processes and operations.

2) Enhanced Scalability

Scalability is an uncompromising requirement in today’s business scenario. Organizations that have acquired a higher level of DevOps maturity are easily able to scale their operations based on business needs. Also, it enables an organization to make multiple deployments in a day and also accelerates the release of applications and products. It is not only the operations that become scalable by acquiring DevOps maturity but the teams also acquire better scalability to easily scale up or down based on the business needs.

3) Identify Opportunities

DevOps teams that have acquired a sufficient level of maturity will be in a position to better identify the threats and opportunities and can effectively make the best use of the latest set of technologies and toolings. DevOps tools are compatible with most of the latest technologies that are available in the market. A significant level of DevOps maturity enables an enterprise to effectively adapt to the changes and get reviews in a fast-paced manner.

Conclusion

DevOps maturity is all about acquiring an efficient DevOps strategy that enables an organization to acquire better business agility and scalability thus enabling them to effectively address the business requirements. It is important to evaluate the levels of DevOps maturity by monitoring the parameters that we have seen in this article and thus ensure that they are being refined to pave the way to better success.

IV Year B. Tech – I Semester
(MR20-1CS0118) Devops
UNIT-4

Deployment in DevOps - steps of DevOps deployment methodology

Deployment in DevOps is a process that enables you to retrieve important codes from version control so that they can be made readily available to the public and they can use the application in a ready-to-use and automated fashion.

In DevOps, deployment refers to the process of delivering software updates and applications to production environments. This typically involves building, testing, and releasing code changes, as well as configuring and maintaining the infrastructure required to run the software. Deployment is an important part of the development process, as it enables organizations to quickly and reliably bring new features and improvements to their users. To facilitate deployment, DevOps teams often use automation tools and practices such as continuous integration and delivery (CI/CD) to streamline the process and ensure that updates can be deployed quickly and consistently. Deployment can be done manually or automated through the use of tools and scripts. The goal of deployment in DevOps is to make the process of updating a production environment as efficient, reliable, and fast as possible and to make code changes available to end users as quickly and reliably as possible, while minimizing downtime and disruption.

While DevOps is not a technology, DevOps environments generally apply common methodologies. These include the following:

- continuous integration and continuous delivery or continuous deployment (CI/CD) tools, with an emphasis on task automation;
- systems and tools that support DevOps adoption, including real-time monitoring, incident management, configuration management and collaboration platforms; and
- cloud computing, microservices and containers implemented concurrently with DevOps methodologies.

The steps of DevOps are listed below -

Plan: task management, schedules

Code: code development and code review, source code management tools, code merging

Build: continuous integration tools, version control tools, build status

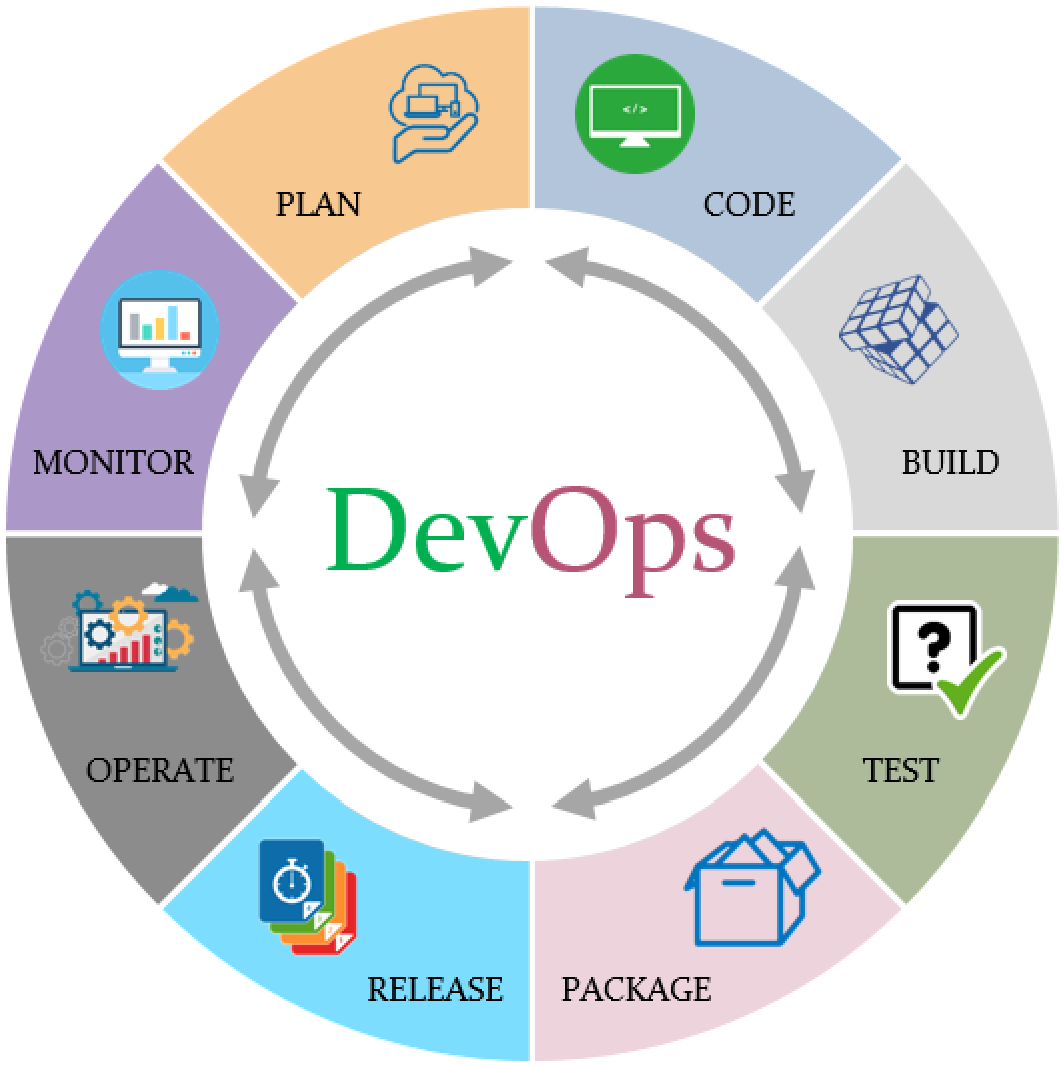
Test: continuous testing tools that provide feedback on business risks, determine performance

Package: artifact repository, application pre-deployment staging

Release: change management, release approvals, release automation

Operate: infrastructure installation, infrastructure changes (scalability), infrastructure configuration and management, infrastructure as code tools, capacity planning, capacity & resource management, security check, service deployment, high availability (HA), data recovery, log/backup management, database management

Monitor: service performance monitoring, log monitoring, end user experience, incident management



Seven phases of DevOps Lifecyle

The DevOps lifecycle is divided into seven different phases of continuous development that guide the software development process from beginning to end. To understand DevOps, it’s important to know each phase of the life cycle, and the processes and requirements of each phase.

1. Continuous development and delivery

Software development kicks off with planning and coding. In DevOps, this is done through the process of regular delivery with the goal of constant improvement.

Building off of core Agile values, DevOps encourages regular, frequent software releases. The standard way to achieve this is by automating the code’s integration and deployment, a process called continuous integration/continuous deployment (CI/CD).

Throughout development, whether in pre-or post-production, teams leverage feedback to identify problems and hypothesize solutions in planning.

Following the planning phase of the DevOps lifecycle, source code and asset creation begin with the goal of keeping production moving forward. Regardless of which coding language is used, maintaining the codebase using source code management tools is a priority.

2. Continuous integration

Continuous integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each checkin or branch is then verified by an automated build, allowing teams to detect problems early on, ensuring the main code branch is always viable and production-ready.

CI is designed to support many small, iterative changes rather than fewer, large changes. It helps teams scale through automated workflows for code builds, testing, merging, and checking into shared repositories.

The ultimate goal of continuous integration is to deliver better code, faster. Through smaller frequent changes coupled with automation, teams can find and address bugs more quickly and reduce time spent on validating and releasing new updates.

3. Continuous testing

Continuous testing goes hand-in-hand with continuous integration. CI/CD pipelines are dependent on automated tests rather than manual code validation. This is to ensure that what’s being deployed is of quality and won’t result in game-breaking errors before release.

DevOps relies on eliminating as many manual processes as possible. The more manual, tedious processes that are in place, the more time is wasted and the greater the chance of errors. The goal of continuous testing DevOps tools is not just to discover bugs, but to find them as quickly as possible so they don’t need to be addressed in the production phase via a patch or hotfix, which becomes much more complicated and time-consuming.

Automated tests are set up before release to the build and also prior to production. Teams may insert manual review as a final step before production and after automated testing has been completed.

4. Continuous monitoring

Continuous monitoring ensures that the DevOps lifecycle is well maintained, with the end goal of providing a great user experience. Software updates and usage are tracked closely, with insights gathered being used to ensure that the software runs correctly.

During the continuous monitoring phase, teams seek to detect and resolve system errors as quickly as possible. Automated error tracking is essential here. Automations can also provide visibility into other areas like overall software performance, user behavior, development infrastructure stability, and more.

In addition to overseeing automations, your DevOps team is responsible for ensuring all aspects of the pipeline are compliant with security standards. Manual processing of release management also occurs during this phase.

5. Continuous feedback

Continuous feedback requires implementing a feedback loop to gather insights on software performance from your internal team and your users. The feedback is then shared with the DevOps team to help guide product iteration. Sources can include surveys, questionnaires, focus groups, social media, forums, and more.

This process isn’t just about determining that your software functions correctly – it’s also about gauging overall customer satisfaction to guide business strategy and ensure the best possible outcomes. Continuous feedback should be used to steer your product roadmap and help you meet your audience’s wants, needs, and expectations.

6. Continuous deployment

Continuous deployment works in tandem with continuous integration by completing the automation cycle and minimizing or removing human intervention in the deployment process. Automated DevOps tools monitor source code updates and automatically deploy them into the production environment once they’ve passed the testing phase, saving time and improving user satisfaction.

Continuous deployment accelerates feedback loops with users through automation. Methods can also be deployed to separate deployment for a release, either hiding them from users (dark releases) or turning them on for specific users to test new features and solicit feedback (feature toggles or switches).

Since the code is released in small batches, it minimizes the risk you'd have with large code changes – all with minimal effort due to automation.

7. Continuous operations

Continuous operations aim to minimize downtime and prevent frustrating service disruptions for users. This phase of the DevOps lifecycle focuses on the optimization of applications and environments for stability and performance. It also completes the loop of the DevOps lifecycle by feeding the planning phase of continuous development with bug reports and user feedback for enhancements.

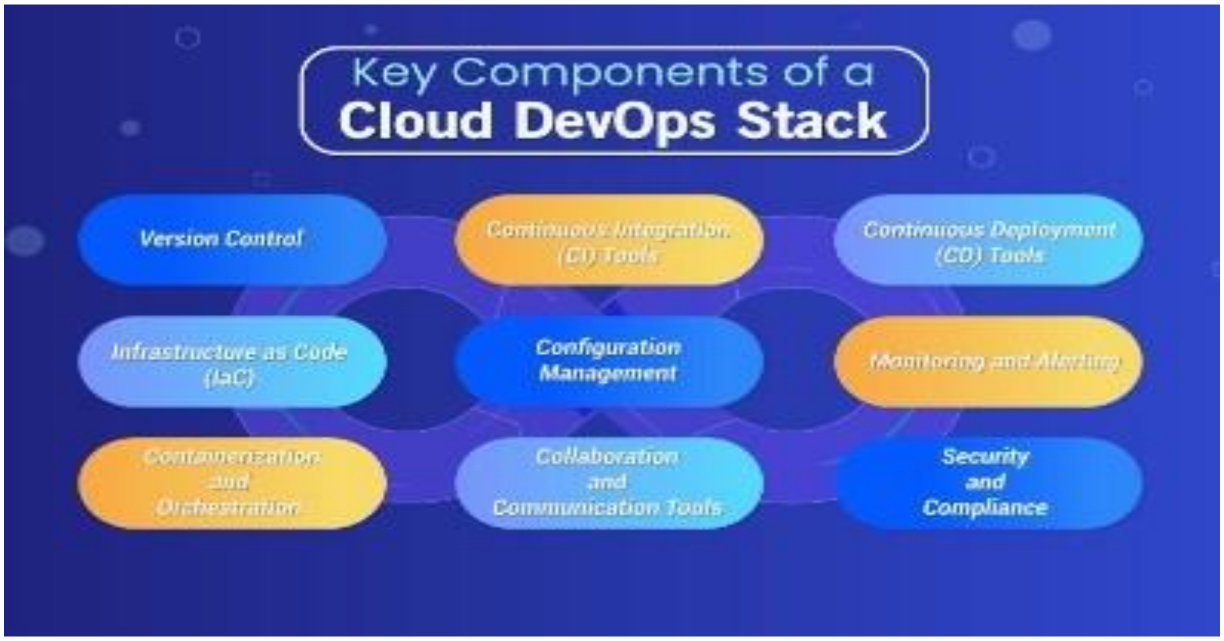
Through continuous collaboration between teams and with users, bugs, feedback, and security concerns can be continually passed on, assessed, and iterated on through the DevOps pipeline.

Best Practices for Implementing DevOps in a Cloud Environment

DevOps, short for Development and Operations, is a transformative approach to software development and IT operations that emphasizes collaboration, automation, and continuous improvement. It aims to break down silos between development and operations teams, fostering a culture of collaboration to deliver high-quality software faster. In a 2020 survey, Puppet reported that organizations practicing DevOps had 200 times more frequent deployments, 24 times faster recovery from failures, and a 3 times lower change failure rate.

The integration of DevOps with cloud environments is pivotal in today's software landscape. Cloud computing offers scalable infrastructure, enabling organizations to provision resources on demand. In a 2021 Flexera report, 92% of enterprises used a multi-cloud strategy. DevOps leverages the cloud's elasticity to automate provisioning, deployment, and scaling, which improves agility and resource utilization. The ability to model infrastructure as code (IaC) further streamlines operations, enhancing reliability and reducing human errors.

The synergy between DevOps and the cloud also empowers organizations to implement continuous integration and continuous delivery (CI/CD) pipelines seamlessly. This, as revealed in a 2022 State of DevOps report by the DevOps Research and Assessment (DORA) team, results in 7 times higher deployment frequency, 2,604 times faster lead time from commit to deploy, and a 24 times lower change failure rate.


Key Components of a
Cloud DevOps Stack
Version Control
Continuous Integration (CI) Tools
Continuous Deployment (CD) Tools
Infrastructure as Code (IaC)
Configuration Management
Monitoring and Alerting
Containerization and Orchestration
Collaboration and Communication Tools
Security and Compliance

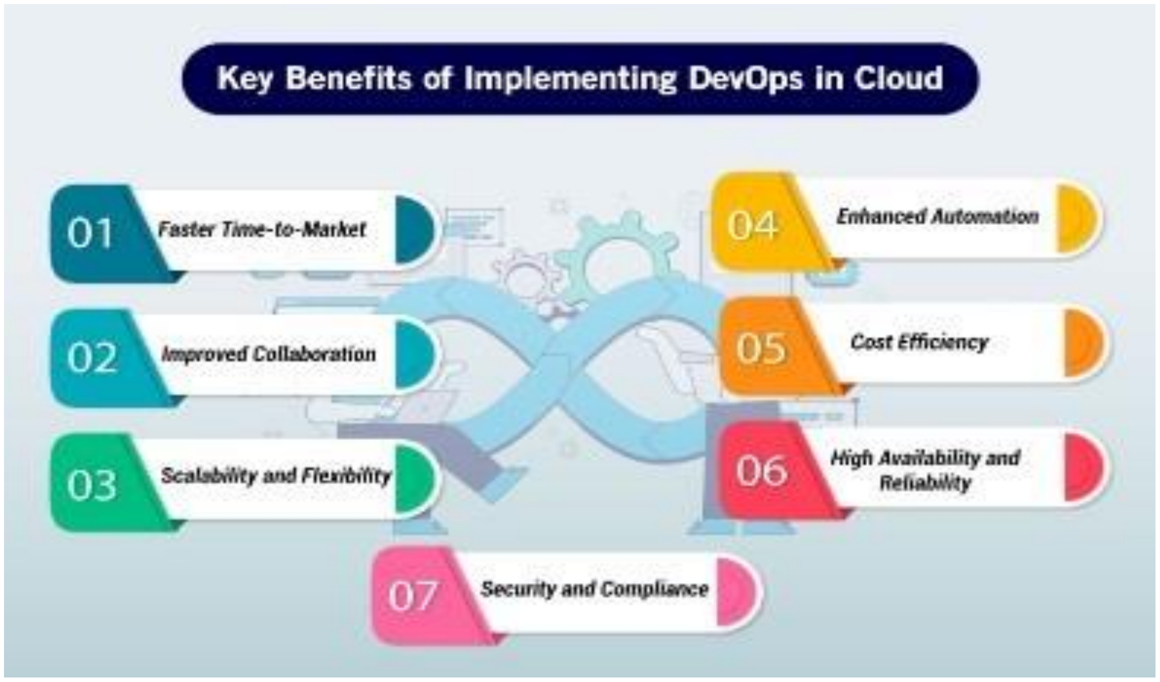
Key Best Practices for DevOps in the Cloud –

Automation is a cornerstone of successful cloud-based DevOps, with statistics emphasizing its significance. According to a 2021 Flexera report, 63% of organizations prioritize automation for cloud management. Automation reduces manual intervention, ensuring repeatability and consistency in provisioning and scaling cloud resources. It accelerates deployments, enhances reliability, and minimizes human errors, leading to a 50% reduction in the failure rate of changes, as observed in the 2022 State of DevOps report.

Infrastructure as Code (IaC) is another pivotal practice, with 87% of organizations adopting it, as reported by the 2020 State of DevOps report. IaC treats infrastructure configurations as code, enabling version control and automation in infrastructure provisioning. This practice enhances agility, allowing for the quick adaptation of resources to changing requirements while maintaining a record of historical changes and configurations.

Continuous Integration (CI) and Continuous Deployment (CD) practices are integral to cloud-based DevOps. A 2022 DORA report reveals that high-performing teams deploying CI/CD have 440 times faster lead times, significantly outpacing their peers. CI involves regularly integrating code changes into a shared repository, while CD automates the release of these changes into production. Cloud environments facilitate rapid, automated testing and deployment, enabling faster innovation and issue resolution.

Version control and collaboration tools are equally essential, enhancing transparency and teamwork. The use of version control systems like Git, which saw a 50% increase in adoption in 2021, and collaboration platforms like Slack or Microsoft Teams, fosters effective communication among development and operations teams. This leads to faster incident resolution and better knowledge sharing, ultimately supporting the goals of cloud-based DevOps.


Key Benefits of Implementing DevOps in Cloud
01 Faster Time-to-Market
02 Improved Collaboration
03 Scalability and Flexibility
04 Enhanced Automation
05 Cost Efficiency
06 High Availability and Reliability
07 Security and Compliance

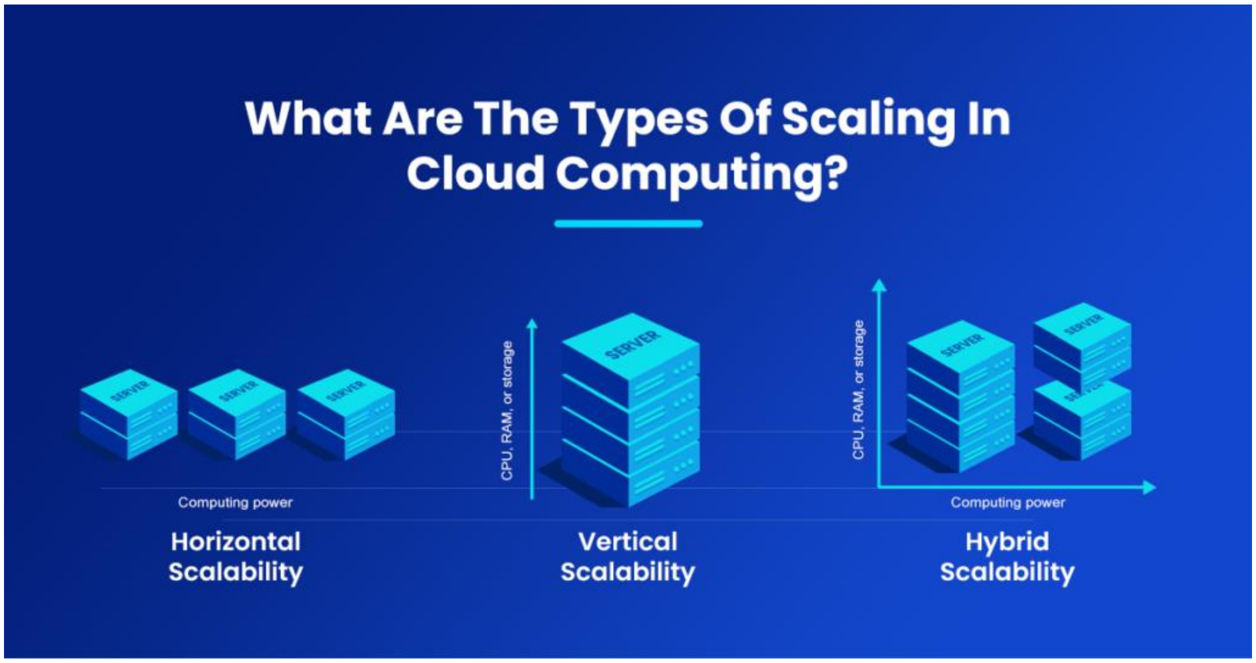
Cloud Scalability and it's types

Cloud scalability is the ability of a cloud computing system to adapt to changing computing requirements by either increasing or decreasing its resources, such as computing power, storage, or network capacity on demand. It allows the system to adjust its resources to the workload to meet the required performance levels. This scalability often involves increasing or decreasing the number of servers, storage, or other computing resources.

This type of scalability is essential because it allows organizations to quickly adjust to the changes in their computing needs while also providing efficient use of computing resources. The goal of cloud scalability is to make sure that the cloud service can scale cost-effectively and ensure that the service can handle greater loads by adding physical or virtual resources. And this scalability is a crucial advantage of cloud computing. It allows businesses to quickly and easily scale their operations as needed without making significant upfront investments in hardware and other infrastructure.

Overall, cloud scalability is a crucial aspect of cloud computing and an essential tool for any organization looking to manage its IT resources effectively.

There are three main types of cloud scalability: horizontal scalability, vertical scalability, and hybrid scalability. Each type offers unique benefits and is designed to meet different business needs.



What is Vertical Scaling in Cloud Computing?

Vertical scaling in cloud computing is adding resources to an existing instance or server to increase its capacity or capabilities. It automatically enables the system to allocate more or fewer resources to meet changing requirements. Vertical scaling is usually done by increasing the server's computing power, adding more RAM or CPU cores, or adding storage capacities, such as hard disks or solid-state drives.

Vertical scaling enables your applications to run faster and handle more load without purchasing a new server or instance. And vertical scaling is a popular choice for cloud computing because it is relatively easy to do and does not require any changes to the existing infrastructure.

What is Horizontal Scaling in Cloud Computing?

Horizontal scaling in cloud computing refers to the ability to scale out a system by adding more nodes, or servers, to the system. This scaling is often used to improve the cluster's processing power, allowing applications and services to handle more concurrent requests or to process more significant amounts of data.

In cloud computing, horizontal scaling is usually achieved by adding additional virtual machines (VMs), containers, or other resources to an existing cluster. This type of scaling is often used to improve performance or to handle increased traffic. When done correctly, horizontal scaling can be a very effective way to enhance the performance of a system.

What is Diagonal Scaling in Cloud Computing?

In cloud computing, diagonal scaling is a scaling in which the system is scaled vertically and horizontally, allowing for the addition of new nodes (machines) to both the columns and rows of cloud infrastructure simultaneously. This type of scaling is often used to improve performance and expand the system's capacity.

Diagonal scaling can be used in a cloud system to add more servers, storage, and networking resources. This type of scaling can also improve the system's performance by adding more resources. Additionally, diagonal scaling can improve the fail-over capability of cloud infrastructure by increasing the number of nodes that can be used in a distributed architecture.

Key Benefits and challenges of Cloud Scalability

Cloud scalability is the ability of a cloud-based system to automatically and dynamically adjust its capacity in response to changes in demand. This cloud scalability can significantly benefit businesses that experience spikes in demand or need to expand their operations rapidly. Cloud scalability is essential for any business that needs to quickly and efficiently manage large amounts of data.

What Are The Benefits Of Cloud Scalability?
Cost Savings
Improved Performance
Flexibility and Agility
Improved Availability and Reliability
Enhanced Security

Cost Efficiency: Cloud scalability allows businesses to scale up or down resources according to their needs. This efficiency helps enterprises pay only for the resources they use and adjust their costs as needed. Additionally, cloud scalability can help companies to reduce their IT costs by reducing the need for physical hardware and maintenance personnel.

Improved Performance: Cloud scalability allows businesses to increase performance and maintain higher levels of customer service by having the capacity to add more resources in response to rising demand immediately. And companies can scale up or down quickly and easily by using the cloud, allowing them to stay agile and competitive.

Increased Flexibility: With cloud scalability, businesses can easily adjust their resources to stay ahead of the competition and better meet customer needs.

Increased Reliability: By allowing businesses to scale their resources to meet customer demand, cloud scalability helps to ensure that services remain available and reliable.

Lower Maintenance Costs: With cloud scalability, businesses can lower their maintenance costs, as they no longer need to worry about managing and maintaining their hardware.

Improve Power: Scalable cloud services enable businesses to easily add or remove resources as needed, allowing them to adjust to changing demands quickly. This means companies can focus on their core operations and not worry about having enough computing power or storage capacity.

With cloud scalability, companies can quickly increase their computing power and storage capacity without investing in additional hardware. This scalability can be especially helpful for businesses that experience unpredictable or seasonal spikes in usage.

Enhanced Security:

Cloud scalability allows you to implement security measures such as load balancing, firewalls, and other security features, which can help you protect your data and applications from cyber threats.

Disaster Recovery:

Cloud scalability lets you easily bring more servers online for a secondary data center, and reduces costs during recovery by letting you scale to the resources you need, without paying for extra maintenance or hardware.

Overall, by adopting a scalable cloud infrastructure, you can position your business for success and achieve better overall business results.

Cloud Scalability Challenges-

In most cases, cloud scalability is easier than on-premises scaling. However, there are some challenges to be aware of:

Complexity—Cloud infrastructure is complex to scale, especially for larger organizations, due to increased resources,endpoints, and data to manage and secure. Lack of expertise can also be an issue for organizations.

Incompatibility—Legacy systems migrated to the cloud can cause issues later when scaled. Or if you use multiple cloud providers, they may have different approaches to scaling.

Service Interruptions—Systems should be optimized for scaling to help avoid service interruptions.

Data Security—Make sure your cloud service provides proper security and access controls.

Lack of Expertise—Anything can be a challenge if your team isn't familiar with it. Make sure to choose cloud technologies that are simple to implement and use, and provide your staff with the training to use them.

Continuous Integration, Deployment, and Delivery

The practice of Modern Development is incomplete without mentioning Continuous Integration, Continuous Deployment, and Continuous Delivery pipeline. This pipeline is one of the best DevOps practices to deliver the code changes frequently and safely.

Before we go on to learning and understanding the basics of the three practices, it is essential to understand the reasons why they are referred to as the most critical DevOps practice.

Why are CI and CD Referred to as the Most Crucial DevOps Practice-

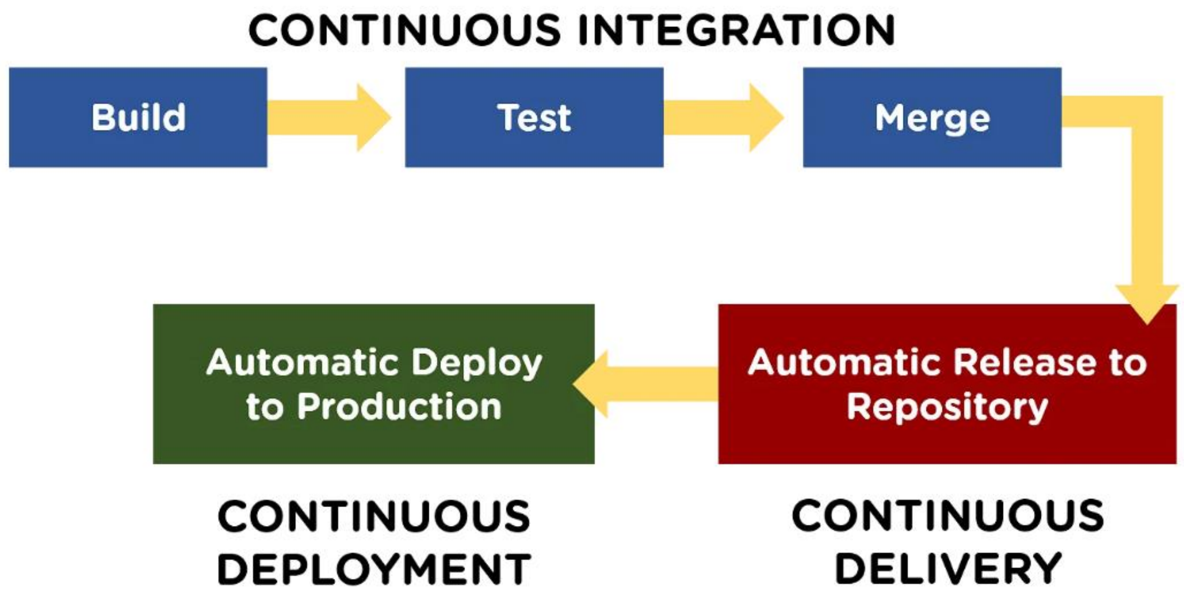
Continuous Integration and Continuous Delivery are among the most significant practices as they create an active process of integrating and delivering the product to the market.

Small code changes can be made in the software code, making the entire process simpler and more accessible.

CI and CD provide continuous feedback from the customers and the DevOps team, thus increasing the transparency of any problem within the team or outside it.

The overall process ensures the faster release of the product.

The failures can now be detected faster and hence fixed effortlessly and quickly, which increases the speed of release.

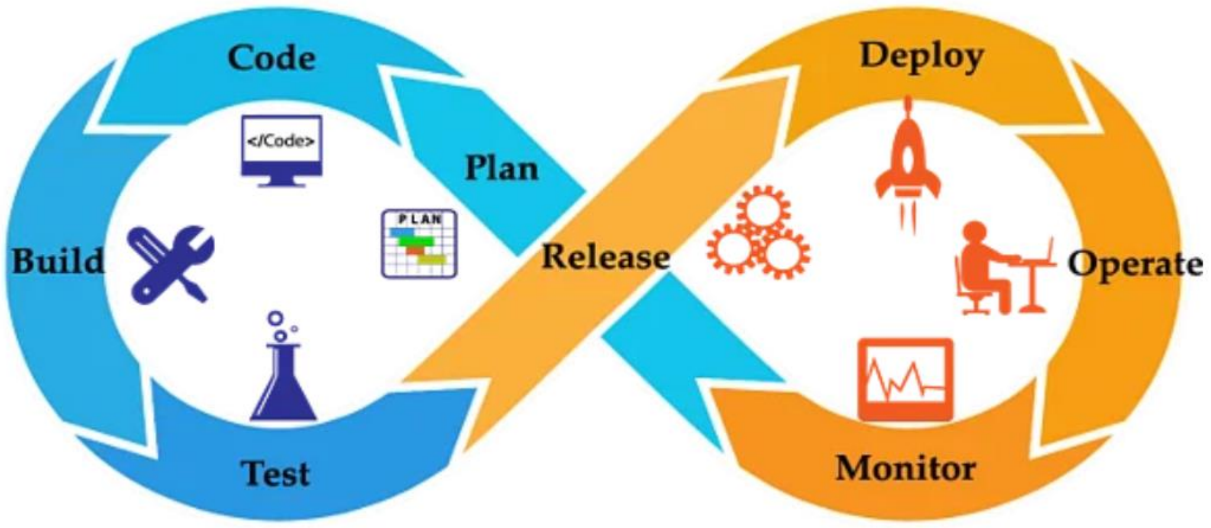


Next, let's talk about all the three processes in detail.

What is Continuous Integration?

Continuous Integration (CI) is a DevOps software development practice that enables the developers to merge their code changes in the central repository. That way, automated builds and tests can be run. The amendments by the developers are validated by creating a built and running an automated test against them.

In the case of Continuous Integration, a tremendous amount of emphasis is placed on testing automation to check on the application. This is to know if it is broken whenever new commits are integrated into the main branch.

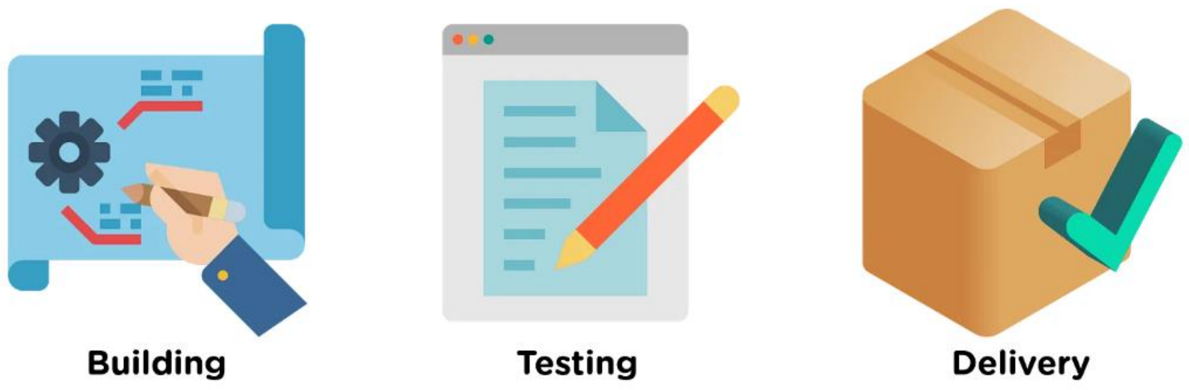


What is Continuous Delivery?

Continuous Delivery (CD) is a DevOps practice that refers to the building, testing, and delivering improvements to the software code. The phase is referred to as the extension of the Continuous Integration phase to make sure that new changes can be released to the customers quickly in a substantial manner.

This can be simplified as, though you have automated testing, the release process is also automated, and any deployment can occur at any time with just one click of a button.

Continuous Delivery gives you the power to decide whether to make the releases daily, weekly, or whenever the business requires it. The maximum benefits of Continuous Delivery can only be yielded if they release small batches, which are easy to troubleshoot if any glitch occurs.



What is Continuous Deployment?

When the step of Continuous Delivery is extended, it results in the phase of Continuous Deployment. Continuous Deployment (CD) is the final stage in the pipeline that refers to the automatic releasing of any developer changes from the repository to the production.

Continuous Deployment ensures that any change that passes through the stages of production is released to the end-users. There is absolutely no way other than any failure in the test that may stop the deployment of new changes to the output. This step is a great way to speed up the feedback loop with customers and is free from human intervention.

Continuous Integration	Continuous Delivery	Continuous Deployment
Automated build for every, commit	Automated build and UAT for every, commit	Automated build, UAT and release to production for every, commit
Independent of Continuous Delivery and Continuous Deployment	It is the next step after Continuous Integration	it is one step further Continuous Delivery
By the end, the application is not in a condition to be released to production	By the end, the application is in a condition to be released to the production.	The application is continuously deployed
Includes Whitebox testing	Includes Blackbox and Whitebox testing	It includes the entire process required to deploy the application

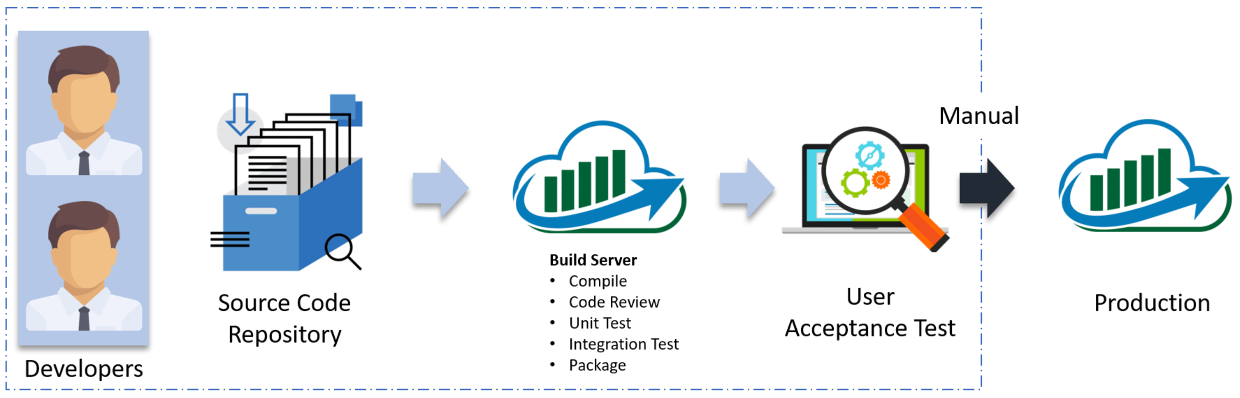
Let me summarize the differences using a table:

Continuous Delivery-stages

Continuous Delivery is a process, where code changes are automatically built, tested, and prepared for a release to production.

It is a process where you build software in such a way that it can be released to production at any time. A continuous delivery pipeline consists of five main phases—build/develop, commit, test, stage, and deploy.

Consider the diagram below:



Let me explain the above diagram:

Automated build scripts will detect changes in Source Code Management (SCM) like Git.

Once the change is detected, source code would be deployed to a dedicated build server to make sure build is not failing and all test classes and integration tests are running fine.

Then, the build application is deployed on the test servers (pre-production servers) for User Acceptance Test (UAT).

Finally, the application is manually deployed on the production servers for release.

There are many different types of testing, Broadly speaking there are two types of testing:

- **Blackbox Testing**: It is a testing technique that ignores the internal mechanism of the system and focuses on the output generated against any input and execution of the system. It is also called functional testing. It is basically used for validating the software.
- **Whitebox Testing**: is a testing technique that takes into account the internal mechanism of a system. It is also called structural testing and glass box testing. It is basically used for verifying the software..

Build/Develop -

A build/develop process performs the following:

Pulls source code from a public or private repository.

Establishes links to relevant modules, dependencies, and libraries.

Builds (compiles) all components into a binary artifact.

Depending on the programming language and the integrated development environment (IDE), the build process can include various tools. The IDE may offer build capabilities or require integration with a separate tool. Additional tools include scripts and a virtual machine (VM) or a Docker container.

Commit

The commit phase checks and sends the latest source code changes to the repository. Every check-in creates a new instance of the deployment pipeline. Once the first stage passes, a release candidate is created. The goal is to eliminate any builds unsuitable for production and quickly inform developers of broken applications.

Commit tasks typically run as a set of jobs, including:

Compile the source code

Run the relevant commit tests

Create binaries for later phases

Perform code analysis to verify health

Prepare artifacts like test databases for later phases

These jobs run on a build grid, a facility provided by continuous integration (CI) servers. It helps ensure that the commit stage completes quickly, in less than five minutes, ideally, and no longer than ten minutes.

Test - During the test phase, the completed build undergoes comprehensive dynamic testing. It occurs after the source code has undergone static testing. Dynamic tests commonly include:

Unit or functional testing—helps verify new features and functions work as intended.

Regression testing—helps ensure new additions and changes do not break previously working features.

Additionally, the build may include a battery of tests for user acceptance, performance, and integration. When testing processes identify errors, they loop the results back to developers for analysis and remediation in subsequent builds.

Since each build undergoes numerous tests and test cases, an efficient CI/CD pipeline employs automation. Automated testing helps speed up the process and free up time for developers. It also helps catch errors that might be missed and ensure objective and reliable testing.

Stage - The staging phase involves extensive testing for all code changes to verify they work as intended, using a staging environment, a replica of the production (live) environment. It is the last phase before deploying changes to the live environment.

The staging environment mimics the real production setting, including hardware, software, configuration, architecture, and scale. You can deploy a staging environment as part of the release cycle and remove it after deployment in production.

The goal is to verify all assumptions made before development and ensure the success of your deployment. It also helps reduce the risk of errors that may affect end users, allowing you to fix bugs, integration problems, and data quality and coding issues before going live.

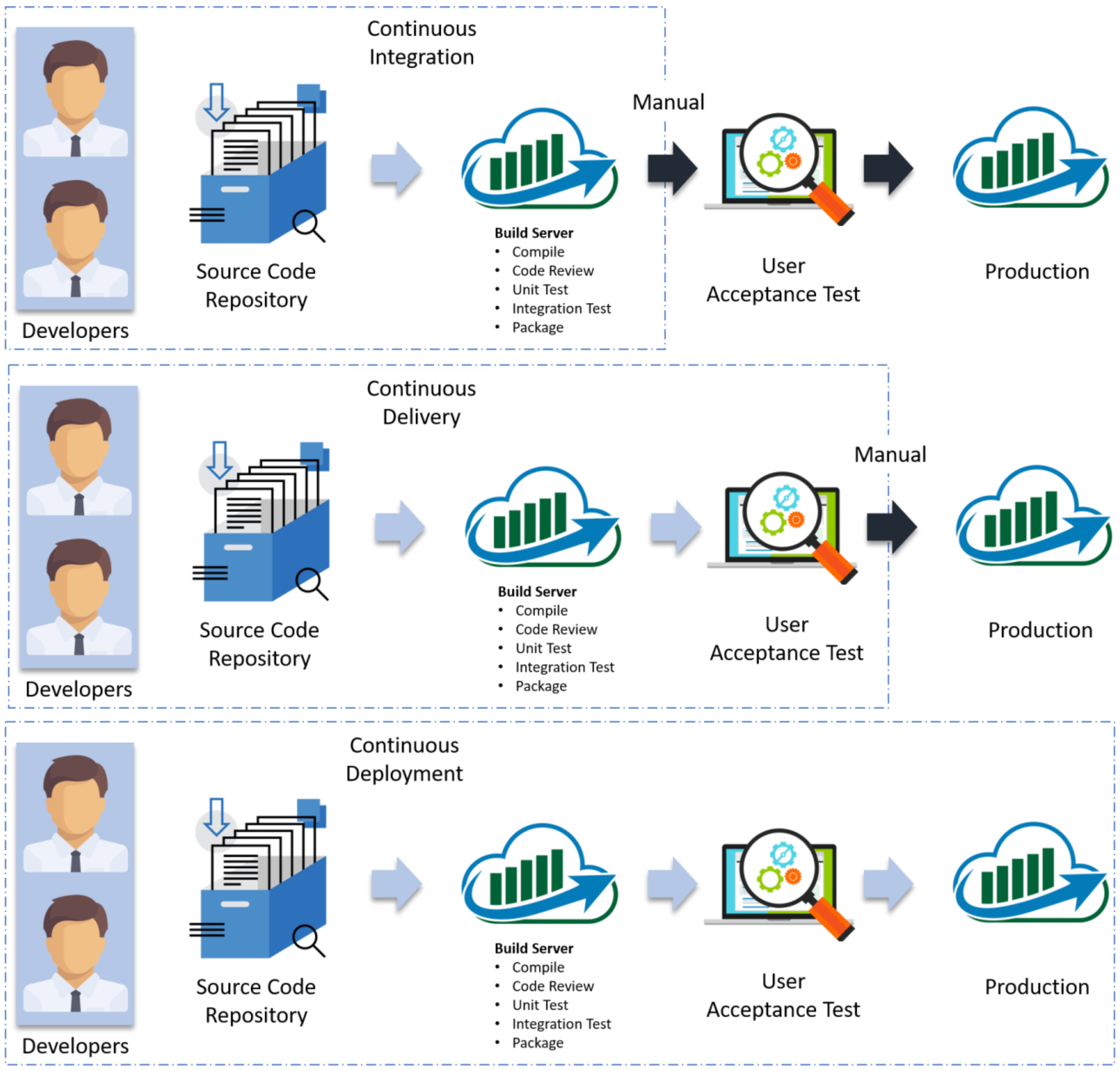
Deploy - The deployment phase occurs after the build passes all testing and becomes a candidate for deployment in production. A continuous delivery pipeline sends the candidate to human teams for approval and deployment. A continuous deployment pipeline deploys the build automatically after it passes testing.

Deployment involves creating a deployment environment and moving the build to a deployment target. Typically, developers automate these steps with scripts or workflows in automation tools. It also requires connecting to error reporting and ticketing tools. These tools help identify unexpected errors post-deployment and alert developers, and allow users to submit bug tickets.

In most cases, developers do not deploy candidates fully as is. Instead, they employ precautions and live testing to roll back or curtail unexpected issues. Common deployment strategies include beta tests, blue/green tests, A/B tests, and other crossover periods.

Differences between Continuous Integration, Delivery And Deployment

Visual content reaches an individual's brain in a faster and more understandable way than textual information. So I am going to start with a diagram which clearly explains the difference:



In Continuous Integration, every code commit is build and tested, but, is not in a condition to be released. I mean the build application is not automatically deployed on the test servers in order to validate it using different types of Blackbox testing like – User Acceptance Testing (UAT).

In Continuous Delivery, the application is continuously deployed on the test servers for UAT. Or, you can say the application is ready to be released to production anytime. So, obviously Continuous Integration is necessary for Continuous Delivery.

Continuous Deployment is the next step past Continuous Delivery, where you are not just creating a deployable package, but you are actually deploying it in an automated fashion.

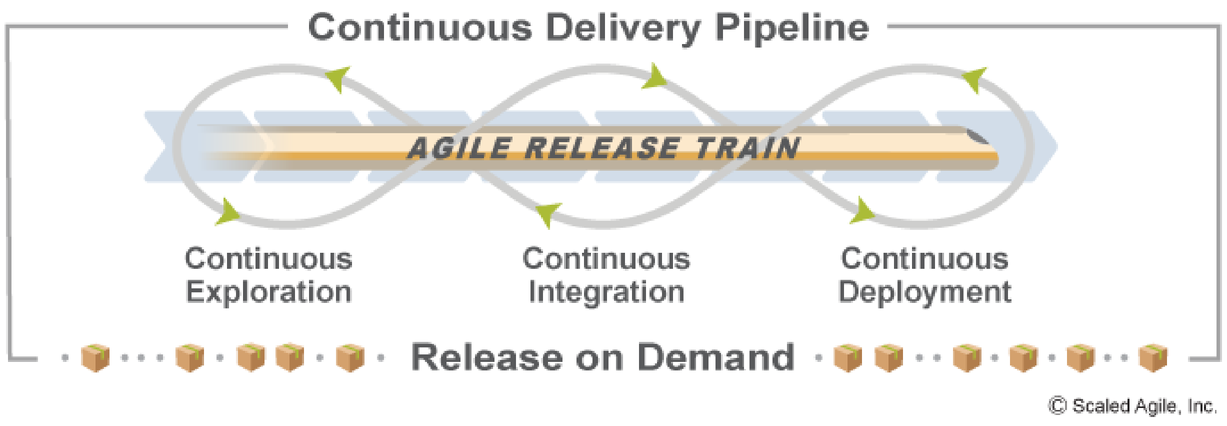
Let me summarize the differences using a table:

Continuous Integration	Continuous Delivery	Continuous Deployment
Automated build for every, commit	Automated build and UAT for every, commit	Automated build, UAT and release to production for every, commit
Independent of Continuous Delivery and Continuous Deployment	It is the next step after Continuous Integration	it is one step further Continuous Delivery
By the end, the application is not in a condition to be released to production	By the end, the application is in a condition to be released to the production.	The application is continuously deployed
Includes Whitebox testing	Includes Blackbox and Whitebox testing	It includes the entire process required to deploy the application

In simple terms, Continuous Integration is a part of both Continuous Delivery and Continuous Deployment. And Continuous Deployment is like Continuous Delivery, except that releases happen automatically.

Continuous Delivery Pipeline

The Continuous Delivery Pipeline (CDP) represents the workflows, activities, and automation needed to shepherd a new piece of functionality from ideation to an on-demand release of value to the end user. As illustrated in Figure 1, the pipeline consists of four aspects: Continuous Exploration (CE), Continuous Integration (CI), Continuous Deployment (CD), and Release on Demand, each of which is described in its own article.



The pipeline is a significant element of the Agile Product Delivery competency. Each Agile Release Train (ART) builds and maintains, or shares, a pipeline with the assets and technologies needed to deliver solution value as independently as possible. The first three elements of the pipeline (CE, CI, and CD) work together to support the delivery of small batches of new functionality, which are then released to fulfill market demand. Details Building and maintaining a Continuous Delivery Pipeline provides each ART with the ability to deliver new functionality to users far more frequently than with traditional processes. For some, 'continuous' may mean daily releases or even releasing multiple times per day. For others, continuous may mean weekly or monthly releases—whatever satisfies market demands and the goals of the enterprise.

Traditional practices tend to perceive releases as large monolithic chunks. However, the reality is that releasing value need not translate to an 'all-or-nothing' approach. Using a satellite as an example, the elements of the system are comprised of the satellite, the ground station, and a web farm that feeds the acquired satellite data to end-users. Some elements may be released daily—perhaps the web farm functionality. Other elements, like the hardware components of the satellite itself, may only be released every launch cycle.

Decoupling the web farm functionality from the physical launch eliminates the need for a monolithic release. It also increases Business Agility by allowing components of the solution to be delivered in response to frequent changes in market need.

The Four Aspects of the Continuous Delivery Pipeline The SAFe continuous delivery pipeline contains four aspects: continuous exploration, continuous integration, continuous deployment, and release on demand. The CDP enables organizations to map their current pipeline into a new structure and then use relentless improvement to deliver value to customers. Feedback loops that exist internally within and between the aspects, and externally between the customers and the enterprise, fuel improvements. Internal feedback loops often center on process improvements, while external feedback often centers on solution improvements. Collectively, the improvements create synergy in ensuring the enterprise is 'building the right thing, the right way' and delivering value to the market frequently. The paragraphs below describe each aspect.

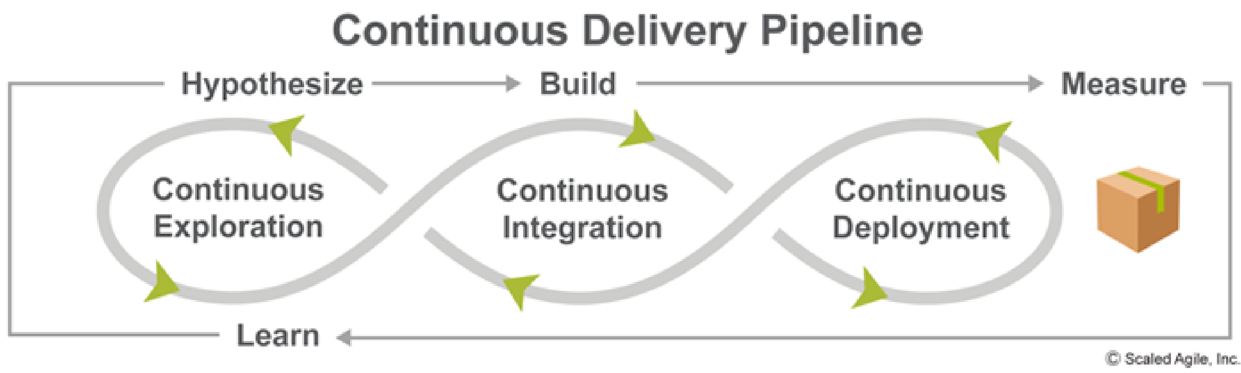
Continuous Exploration (CE) focuses on creating alignment on what needs to be built. In CE, design thinking is used to ensure the enterprise understands the market problem / customer need and the solution required to meet that need. It starts with an idea or a hypothesis of something that will provide value to customers, typically in response to customer feedback or market research. Ideas are then analyzed and further researched, leading to the understanding and convergence of what is needed as either a Minimum Viable Product (MVP) or Minimum Marketable Feature (MMF). These feed the solution space of exploring how existing architectures and solutions can, or should, be modified. Finally, convergence occurs by understanding which Capabilities and Features, if implemented, are likely to meet customer and market needs. Collectively, these are defined and prioritized in the Program Backlog.

Continuous Integration (CI) focuses on taking features from the Program backlog and implementing them. In CI, the application of design thinking tools in the problem space focuses on refinement of features (e.g., designing a user story map), which may motivate more research and the use of solution space tools (such as user feedback on a paper prototype). After specific features are clearly understood, Agile Teams implement them. Completed work is committed to version control, built and integrated into a full system or solution, and tested end-to-end before being validated in a staging environment.

Continuous Deployment (CD) takes the changes from the staging environment and deploys them to production. At that point, they're verified and monitored to make sure they are working properly. This step makes the features available in production, where the business determines the appropriate time to release them to customers. This aspect also allows the organization to respond, rollback, or fix forward when necessary.

Release on Demand (RoD) is the ability to make value available to customers all at once, or in a staggered fashion based on market and business needs. This provides the business the opportunity to release when market timing is optimal and carefully control the amount of risk associated with each release. Release on Demand also encompasses critical pipeline activities that preserve the stability and ongoing value of solutions long after release.

Although it is described sequentially, the pipeline isn't strictly linear. Rather, it's a learning cycle that allows teams to establish one or more hypotheses, build a solution to test each hypothesis, and learn from that work, as Figure 2 illustrates.



Continuous Integration - it's activities in Agile Release Train

Continuous Integration (CI) is an aspect of the Continuous Delivery Pipeline in which new functionality is developed, tested, integrated, and validated in preparation for deployment and release. CI is the second aspect in the four-part Continuous Delivery Pipeline of Continuous Exploration (CE), Continuous Integration (CI), Continuous Deployment (CD), and Release on Demand (Figure 1).

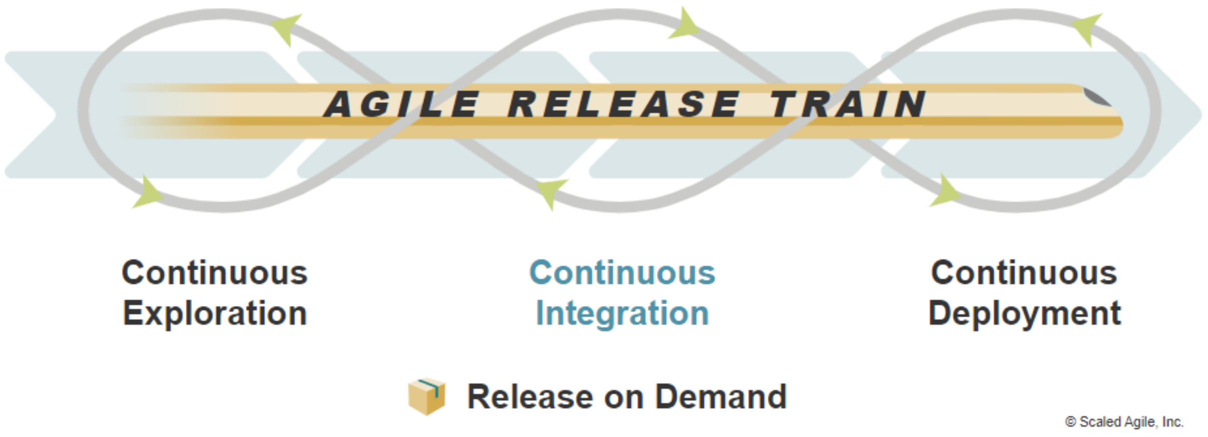


Figure 1. Continuous integration in the context of the continuous delivery pipeline.

Details Continuous integration is a critical technical practice for each Agile Release Train (ART). It improves quality, reduces risk, and establishes a fast, reliable, and sustainable development pace. With continuous integration, the system always runs, meaning it's potentially deployable, even during development. CI is most easily applied to software solutions where small, tested vertical threads can deliver value independently. In larger, multi-platform software systems, the challenge is harder. Each platform has technical constructs which need continuous integration to validate new functionality. CI is even more complicated when systems comprise software, hardware, components, and services provided by suppliers. But the fact remains that frequently integrating and testing features together is the only practical way to validate a solution fully.

As a result, teams need a balanced approach that allows them to build-in quality and gets fast feedback on their integrated work. For purely software-based solutions, continuous integration is relatively easy to achieve with modern tools. For more complex systems with hardware and software, a continuous integration approach is required (see the Enterprise Solution Delivery article) to balance the economic trade-offs between frequency, the scope of integration, and testing.

The **Four Activities** of Continuous Integration As illustrated in Figure 2, SAFe describes four activities associated with continuous integration:

1. **Develop** describes the practices necessary to implement stories and commit the code and components to version control
2. **Build** describes the techniques needed to create deployable binaries and merge development branches into the trunk
3. **Test end-to-end** describes the practices necessary to validate the solution
4. **Stage** describes the steps required to host and validate solutions in a staging environment before production

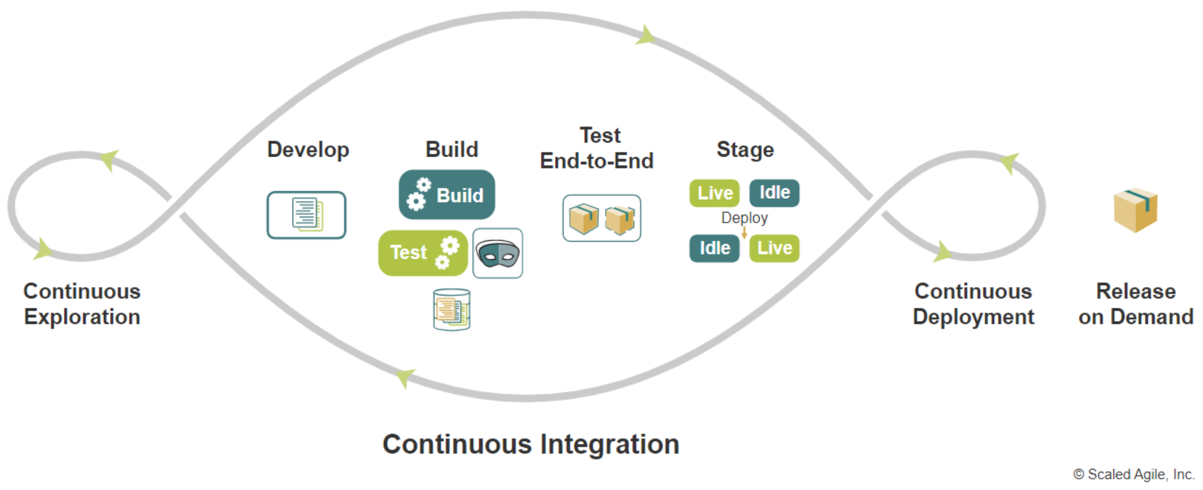


Figure 2. Four activities of continuous integration

Deployment Pipeline and the main Stages of a Deployment Pipeline

In software development, a deployment pipeline is a system of automated processes designed to quickly and accurately move new code additions and updates from version control to production. In past development environments, manual steps were necessary when writing, building, and deploying code. This was extremely time consuming for both developers and operations teams, as they were responsible for performing tedious manual tasks such as code testing and code releases.

The introduction of automation in a deployment pipeline allowed development teams to focus more on innovating and improving the end product for the user. By reducing the need for any manual tasks, teams are able to deploy new code updates much quicker and with less risk of any human error.

In this article, we will break down the different stages of a deployment pipeline, how one is built, the benefits of a deployment pipeline for software development, as well as some helpful tools to get the most out of your system.

Main Stages of a Deployment Pipeline

There are four main stages of a deployment pipeline:

Version Control

Acceptance Tests

Independent Deployment

Production Deployment

Version Control is the first stage of the pipeline. This occurs after a developer has completed writing a new code addition and committed it to a source control repository such as GitHub. Once the commit has been made, the deployment pipeline is triggered and the code is automatically compiled, unit tested, analyzed, and run through installer creation. If and when the new code passes this version control stage, binaries are created and stored in an artifact repository. The validated code then is ready for the next stage in the deployment pipeline.

In the Acceptance Tests stage of the deployment pipeline, the newly compiled code is put through a series of tests designed to verify the code against your team's predefined acceptance criteria. These tests will need to be custom-written based on your company goals and user expectations for the product. While these tests run automatically once integrated within the deployment pipeline, it's important to be sure to update and modify your tests as needed to consistently meet rising user and company expectations.

Once code is verified after acceptance testing, it reaches the Independent Deployment stage where it is automatically deployed to a development environment. The development environment should be identical (or as close as possible) to the production environment in order to ensure an accurate representation for functionality tests. Testing in a development environment allows teams to squash any remaining bugs without affecting the live experience for the user.

The final stage of the deployment pipeline is Production Deployment. This stage is similar to what occurs in Independent Deployment, however, this is where code is made live for the user rather than a separate development environment. Any bugs or issues should have been resolved at this point to avoid any negative impact on user experience. DevOps or operations typically handle this stage of the pipeline, with an ultimate goal of zero downtime. Using Blue/Green Drops or Canary Releases allows teams to quickly deploy new updates while allowing for quick version rollbacks in case an unexpected issue does occur.

Deployment Pipeline Tools available in the industry

Making use of available tools will help to fully automate and get the most out of your deployment pipeline. When first building a deployment pipeline, there are several essential tool categories that must be addressed, including source control, build/compilation, containerization, configuration management, and monitoring.

A development pipeline should be constantly evolving, improving and introducing new tools to increase speed and automation. Some favorite tools for building an optimal deployment pipeline include:

1. Jenkins
2. Azure DevOps
3. GitHub Actions
4. CircleCI
5. Gitlab CI/CD
6. Travis CI
7. Bitbucket Pipeline
8. TeamCity
9. Semaphore
10. Harness
11. CodeShip
12. PagerDuty

Suggest doing online research on some of the above tools as per individual preferences for the explanation.

UNIT-5

Steps involved in setting Up a Server Environment for Jenkins CI/CD.

Setting up a reliable server environment is the foundation for successful CI/CD (Continuous Integration and Continuous Deployment) with Jenkins. In this guide, we will walk you through the essential steps to prepare your server environment, ensuring it meets the requirements for hosting Jenkins and your CI/CD pipeline.

Step 1: Choose Your Server

The first decision you need to make is where you'll host Jenkins and your CI/CD processes. You have several options:

On-Premises Server: If you have your own data center or a physical server, you can install Jenkins on this hardware. Make sure it meets the hardware and software requirements outlined in the Jenkins documentation.

Cloud Infrastructure: Cloud providers like AWS, Azure, and Google Cloud offer scalable infrastructure options. You can set up a virtual machine (VM) or container in the cloud to host Jenkins.

Dedicated Hosting: You can choose a dedicated hosting provider to rent server space and resources. This option provides more control than shared hosting.

Docker Container: For smaller-scale setups or testing purposes, you can run Jenkins in a Docker container on your local machine or a server.

Step 2: Operating System

The choice of the operating system (OS) for your server is critical. Jenkins is platform-agnostic and can run on various operating systems, including Linux, macOS, and Windows. However, Linux is the most commonly used OS for Jenkins due to its stability and performance.

Recommended Linux Distributions:

Ubuntu: Ubuntu LTS releases (e.g., 18.04, 20.04) are well-supported and user-friendly.

CentOS: CentOS 7 and 8 are popular choices for server environments.

Red Hat Enterprise Linux (RHEL): RHEL is known for its stability and is suitable for enterprise-grade installations.

Choose the Linux distribution that aligns with your organization's preferences and policies.

Step 3: Hardware Requirements

Ensure that your server hardware meets the minimum requirements for hosting Jenkins:

CPU: A multi-core processor (e.g., 2 cores) is sufficient for most Jenkins setups.

RAM: Jenkins recommends a minimum of 256MB of RAM, but for practical purposes, aim for at least 1GB or more, depending on the complexity of your CI/CD tasks.

Disk Space: Jenkins itself doesn't require much disk space, but you'll need additional space for storing build artifacts, logs, and other data generated by your CI/CD pipeline.

Step 4: Network Configuration

Proper network configuration is essential for connecting to Jenkins, especially if your server is in a private network or the cloud:

Firewalls: Open the necessary ports (e.g., 80, 8080, 443) to allow incoming traffic to reach Jenkins. Configure your firewall settings to permit external access if needed.

DNS: Set up domain name resolution to assign a domain or subdomain to your Jenkins server. This makes it easier for users and tools to access your Jenkins instance.

Step 5: Security Considerations

Security is paramount when setting up your server environment:

SSH Access: Secure SSH access to your server with strong authentication methods like SSH keys.

Firewall Rules: Implement strict firewall rules to restrict access to only trusted IP addresses.

Regular Updates: Keep your server's OS and software packages up to date to patch security vulnerabilities.

Conclusion

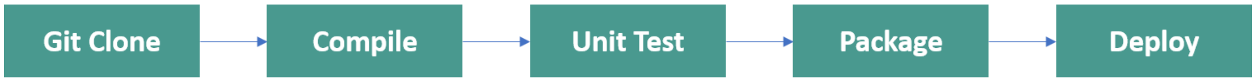
Setting up the server environment is the crucial first step towards building a robust CI/CD pipeline with Jenkins. Choose your server type, operating system, and hardware carefully, and pay close attention to security measures to protect your Jenkins instance and the applications it manages.

Build a Continuous Delivery Pipeline Using Jenkins

Jenkins can be used to create a Continuous Delivery Pipeline, which will include the following tasks:

- Fetching the code from GitHub
- Compiling the source code
- Unit testing and generating the JUnit test reports

- Packaging the application into a WAR file and deploying it on the Tomcat server

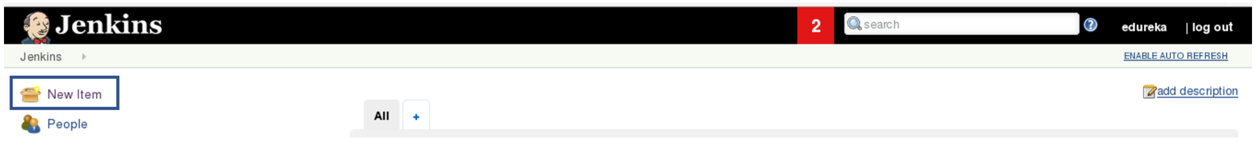


Pre-requisites:

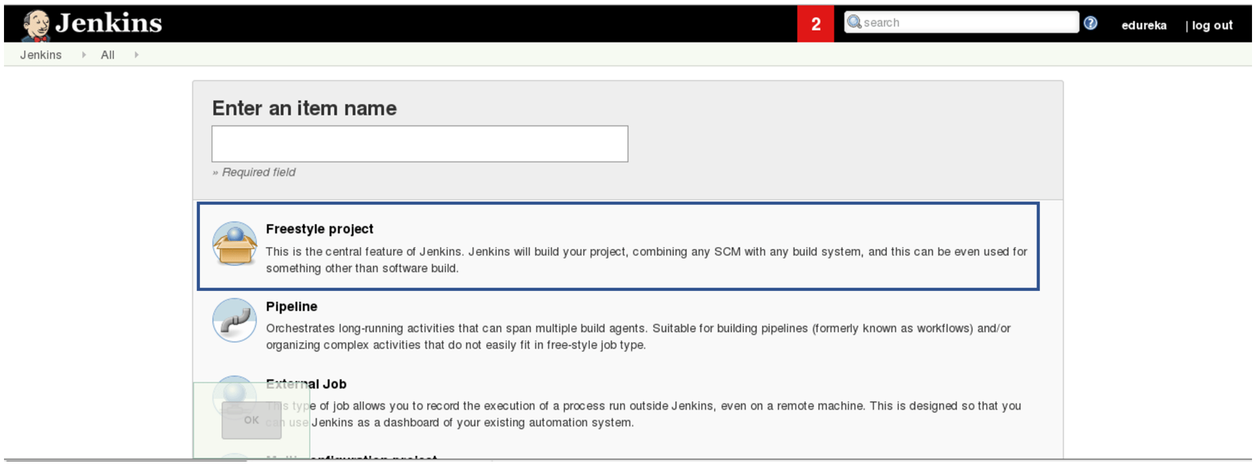
- CentOS 7 Machine
- Jenkins 2.121.1
- Docker
- Tomcat 7

Step – 1 Compiling The Source Code:

Let's begin by first creating a Freestyle project in Jenkins. Consider the below screenshot:



Give a name to your project and select Freestyle Project:



Enter an item name

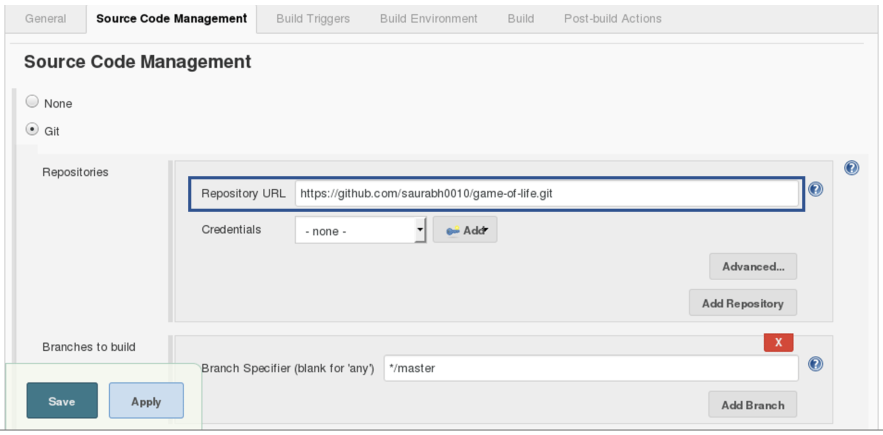
» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

External Job
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.

When you scroll down you will find an option to add source code repository, select git and add the repository URL, in that repository there is a pom.xml fine which we will use to build our project. Consider the below screenshot:



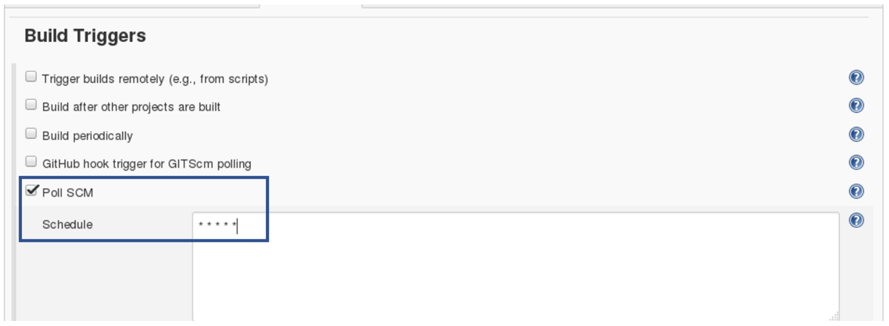
Source Code Management

None
Git [✓]

Repositories
Repository URL: https://github.com/saurabh0010/game-of-life.git
Credentials: - none -

Branches to build
Branch Specifier (blank for 'any'): */master

Now we will add a Build Trigger. Pick the poll SCM option, basically, we will configure Jenkins to poll the GitHub repository after every 5 minutes for changes in the code. Consider the below screenshot:


Build Triggers

- Trigger builds remotely (e.g., from scripts)
- Build after other projects are built
- Build periodically
- GitHub hook trigger for GITScm polling
- [✓] Poll SCM

Schedule: * * * * *

Before I proceed, let me give you a small introduction to the Maven Build Cycle.

Each of the build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

Following is the list of build phases:

- validate – validate the project is correct and all necessary information is available
- compile – compile the source code of the project
- test – test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- package – take the compiled code and package it in its distributable format, such as a JAR.
- verify – run any checks on results of integration tests to ensure quality criteria are met
- install – install the package into the local repository, for use as a dependency in other projects locally
- deploy – done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

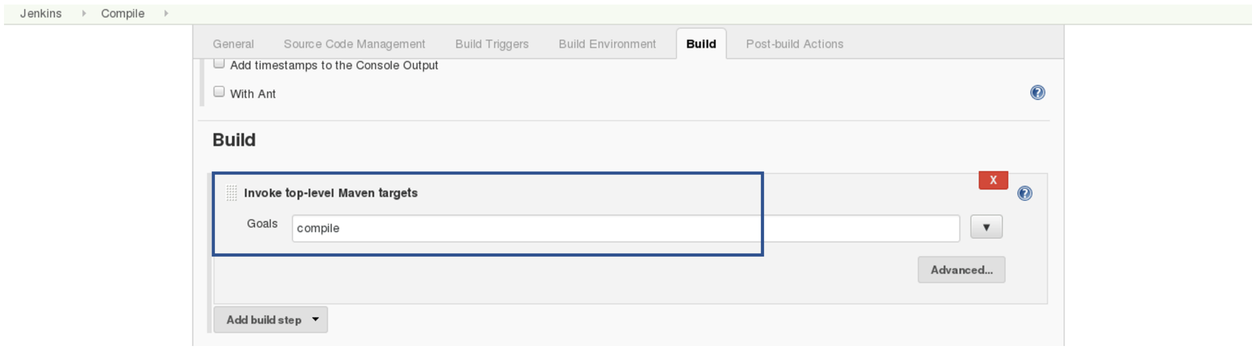
I can run the below command, for compiling the source code, unit testing and even packaging the application in a war file:

```
mvn clean package
```

You can also break down your build job into a number of build steps. This makes it easier to organize builds in clean, separate stages.

So we will begin by compiling the source code. In the build tab, click on invoke top level maven targets and type the below command: `compile`

Consider the below screenshot:



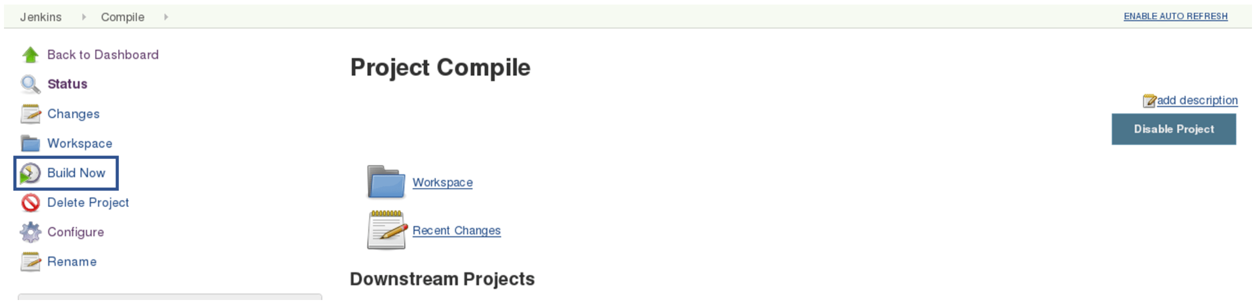
Build

Invoke top-level Maven targets

Goals: compile

This will pull the source code from the GitHub repository and will also compile it (Maven Compile Phase).

Click on Save and run the project.



Project Compile

Downstream Projects

Step – 2 Unit Testing:

Now we will create one more Freestyle Project for unit testing.
Add the same repository URL in the source code management tab, like we did in the previous job.
Now, in the “Buid Trigger” tab click on the “build after other projects are built”. There type the name of the previous project where we are compiling the source code, and you can select any of the below options:

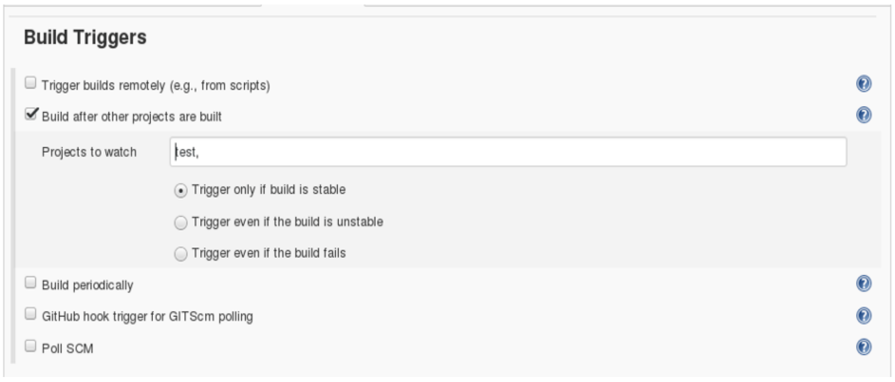
- Trigger only if the build is stable
- Trigger even if the build is unstable
- Trigger even if the build fails

Do the required changes and again save it and click on Build Now.
Now, the JUnit report is written to /var/lib/jenkins/workspace/test/gameoflife-core/target/surefire-reports/TEST-behavior.

Step – 3 Creating A WAR File And Deploying On The Tomcat Server:

Now, the next step is to package our application in a WAR file and deploy that on the Tomcat server for User Acceptance test.Create one more freestyle project and add the source code repository URL.

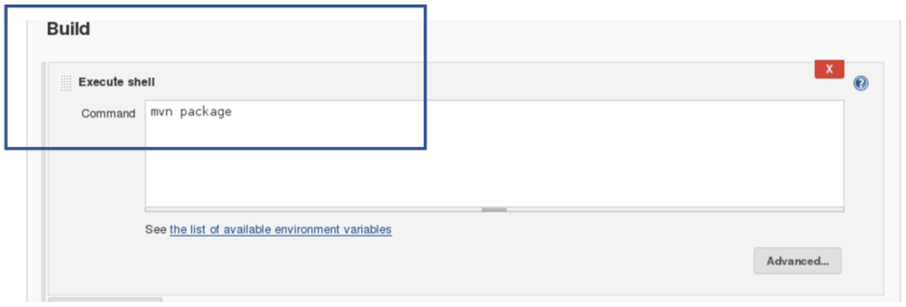
Then in the build trigger tab, select build when other projects are built, consider the below screenshot:


Build Triggers

Trigger builds remotely (e.g., from scripts)
[✓ Build after other projects are built]
Projects to watch: test,
[✓ Trigger only if build is stable]
Trigger even if the build is unstable
Trigger even if the build fails
Build periodically
GitHub hook trigger for GITScm polling
Poll SCM

Basically, after the test job, the deployment phase will start automatically.

In the build tab, select shell script. Type the below command to package the application in a WAR file: *mvn package*

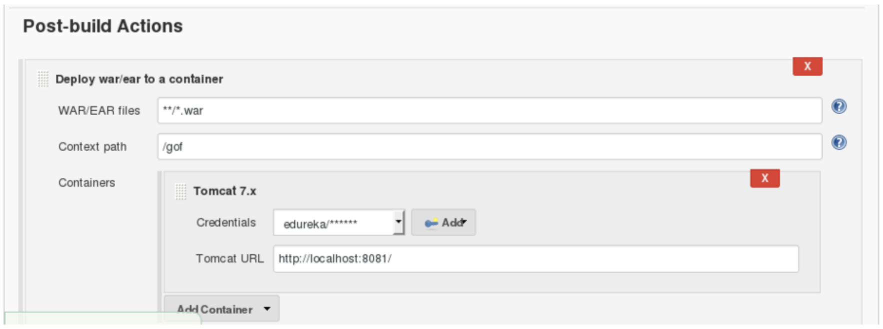

Build

Execute shell
Command:

```
mvn package
```

See the list of available environment variables

Next step is to deploy this WAR file to the Tomcat server. In the "Post-Build Actions" tab select deploy war/ear to a container. Here, give the path to the war file and give the context path. Consider the below screenshot:


Post-build Actions

Deploy war/ear to a container

WAR/EAR files: **/*.war

Context path: /gof

Containers

Tomcat 7.x

Credentials: edureka/******

Tomcat URL: http://localhost:8081/

Click on Save and then select Build Now.
Please note the above steps understanding requires a practice on a laptop.

Types of Jenkins pipelines -Jenkins Multibranch Pipeline

Jenkins is an open-source automation server that is widely used for continuous integration and continuous delivery (CI/CD). One of the key features of Jenkins is its support for pipelines, which enable developers to create automated workflows for building, testing, and deploying their applications.

Jenkins Pipelines can be either - a Declarative pipeline or a Scripted Pipeline.

Declarative pipeline makes use of numerous, generic, predefined build steps/stages (i.e. code snippets) to build our job according to our build/automation needs whereas, with Scripted pipelines, the steps/stages can be custom-defined & used using a groovy syntax which provides better control & fine-tuned execution levels.

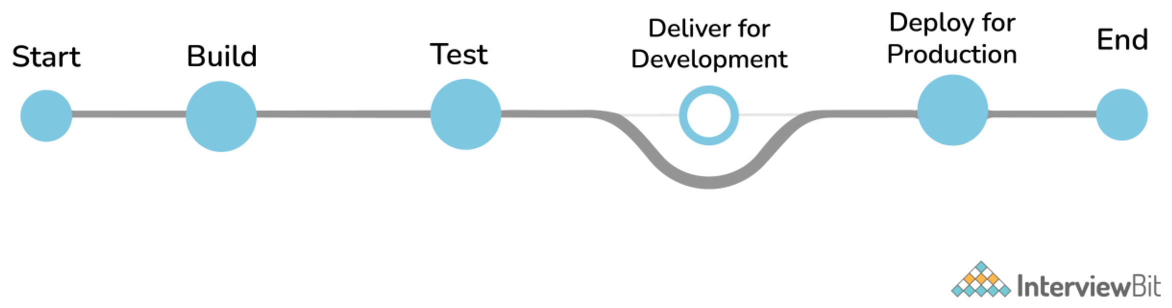
Declarative and Scripted Pipelines are constructed fundamentally differently. Declarative Pipeline is a more recent feature of Jenkins Pipeline which: provides richer syntactical features over Scripted Pipeline syntax, and.

Scripted Pipeline -

Scripted Pipeline is the original pipeline syntax for Jenkins, and it is based on Groovy scripting language. In Scripted Pipeline, the entire workflow is defined in a single file called a Jenkinsfile. The Jenkinsfile is written in Groovy and is executed by the Jenkins Pipeline plugin. Scripted Pipeline provides a lot of flexibility and control over the workflow, but it can be more complex and verbose than Declarative Pipeline.

Declarative Pipeline -

Declarative Pipeline is a more recent addition to Jenkins and provides a more structured and simpler syntax for defining pipelines. Declarative Pipeline is based on the Groovy programming language, but it uses a Groovy-based DSL (Domain-Specific Language) for pipeline configuration.. The main benefit of Declarative Pipeline is its readability and ease of use, as it is designed to be more intuitive and less verbose than Scripted Pipeline.



Jenkins Multibranch Pipeline

It is a pipeline job that can be configured to Create a set of Pipeline projects according to the detected branches in one SCM repository. This can be used to configure pipelines for all branches of a single repository e.g. if we maintain different branches (i.e. production code branches) for different configurations like locales, currencies, countries, etc.

A multi-branch pipeline is a concept of automatically creating Jenkins pipelines based on Git branches. It can automatically discover new branches in the source control (Github) and automatically create a pipeline for that branch. When the pipeline build starts, Jenkins uses the Jenkinsfile in that branch for build stages.

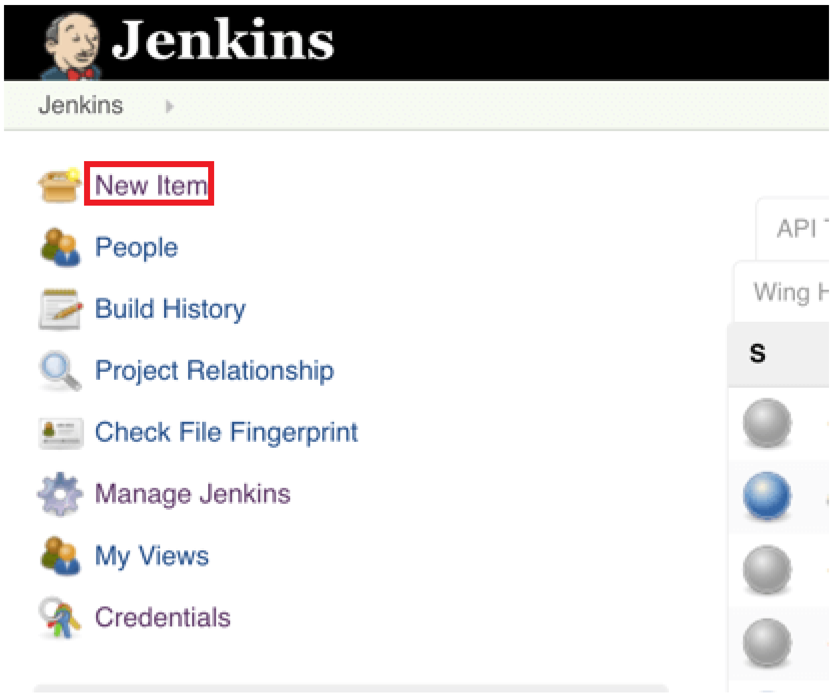
Create CI/CD Pipeline in Jenkins- step by step

Jenkins is one of the most widely used open-source CI/CD DevOps tools. It enables developers to implement CI/CD pipelines within the development environment in a comprehensive manner. Jenkins is written in Java and supports various version control tools including Git and Maven.

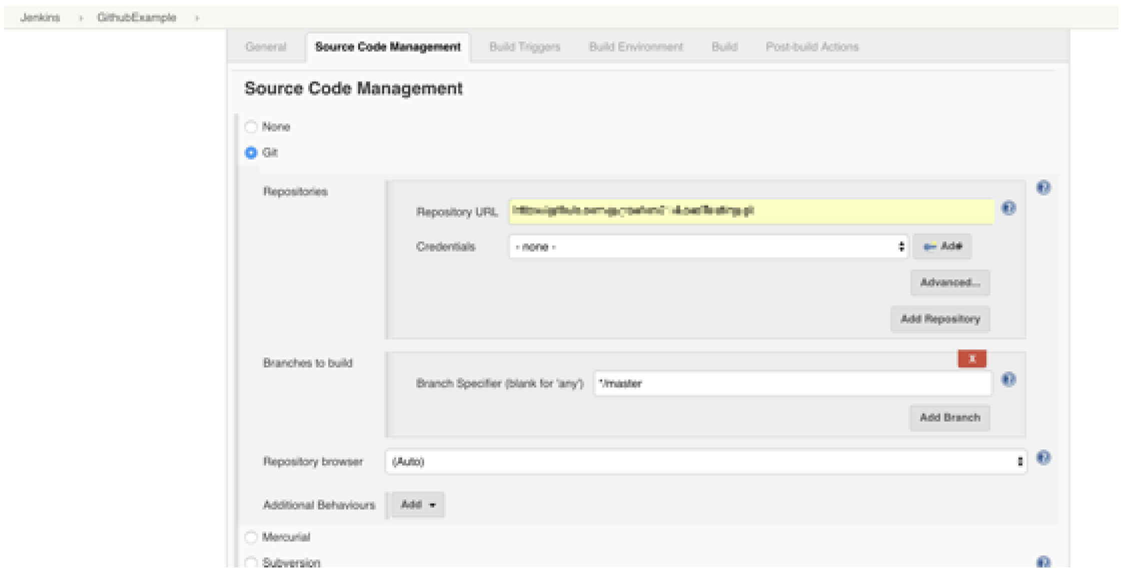
Its popularity is based on the fact that Jenkins is an open source repository of rich integrations and plugins that are well documented and extensible, based on an extended community of developers who have developed hundreds of plugins to accomplish almost any task.

Jenkins runs on the server and requires constant maintenance and updates. The availability of Jenkins as a cross-platform tool for Windows, Linux, and various operating systems makes it stand out among other DevOps tools. Moreover, it can easily be configured using CLI or GUI.

On the Jenkins dashboard, click on “New Item”. After that, enter the name of your project, choose “Freestyle Project” from the list, and hit the enter key.



The next step is to configure the project on the Jenkins dashboard. In the General tab, choose your GitHub Project, and enter the GitHub Repository URL. Now, head over to the Source Code Management tab to add the credentials.



Repository URL: [illegible]

Credentials: - none -

Branch Specifier (blank for 'any'): */master

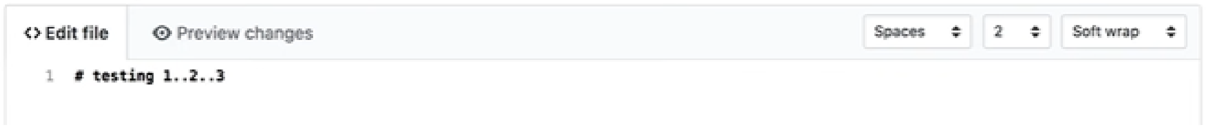
Repository browser: (Auto)

For this step, press the Add button and type Username and Password. Press Add again to close the dialogue box. Be sure to select the main branch in the Branch Specifier.

Now, head over to the Build Triggers section to set the triggers for Jenkins. The purpose of triggers is to necessarily indicate to Jenkins when to build the project. Select Poll SCM section, which queues VCS on the predefined schedule. The predefined schedule is ***** which represents every minute. That means, our Jenkins job will check for the change in our repository every single minute.

Build Jenkins Project-

Now go back to the repository that you configured, and make a sample file, or edit any existing file and commit the changes.



```
# testing 1..2..3
```

After that, head over to Jenkins and click on the “Build Now” button from the navigation bar.

Stages of Jenkins

In Jenkins, we have mainly defined our three stages: Build, Test, and Deploy. Each stage includes individual actions defined inside the steps section.

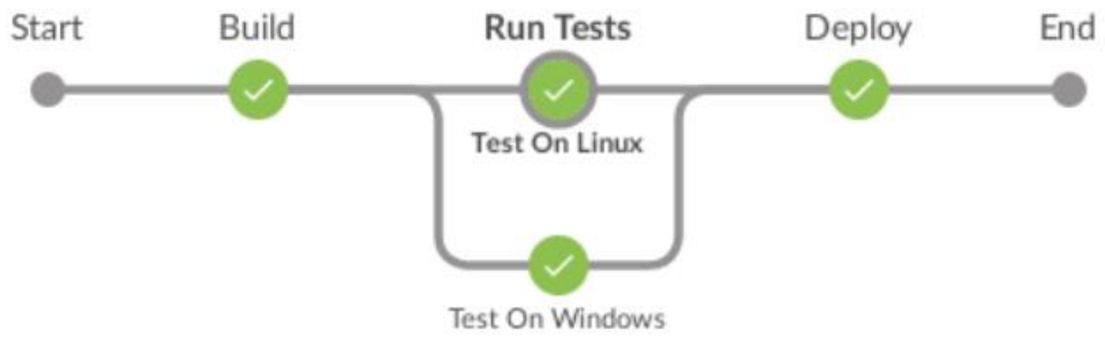
Below snippet of Jenkinsfile explain the stages used in scripted pipeline -

Jenkinsfile (Scripted Pipeline)

```
node {
  stage('Build') {
    //
  }
  stage('Test') {
    //
  }
  stage('Deploy') {
    //
  }
}
```

Also there are some more stages like ‘*Cleanup Workspace*’, ‘*Code Checkout*’ and ‘*Code Analysis*’, *‘Security scans’* etc. are available. Suggest to do some online research to get more insights of the above mentioned stages in Jenkins pipelines.

Below is the execution of stages in Pipeline on a Blue Ocean plugin tool.



Stages can be configured either sequential or parallel. In the above flow diagram, we can notice that the 2 stages in ‘Run Tests’ ran in parallel.

Alternates to Jenkins in the industry - some of the best practices in Jenkins

Jenkins is an open-source automation server that has been around for more than a decade. At the time of its release, it was a tool that completely revolutionized the software development world.

This highly efficient tool has been top-rated for developing software applications, testing projects, and making continuous integration pipelines possible. Jenkins automates the build, test, and deploy processes to make streamlined continuous integration and continuous delivery pipelines.

Additionally, Jenkins supports several plugins with highly versatile uses, enabling teams to use it with languages other than Java. However, Jenkins isn’t perfect, and after over a decade in the industry, it’s inevitable that Jenkins has accumulated several competitors.

Below are few out of many other Jenkins Alternatives used in the Industry.

1. TeamCity

2. AWS CodePipeline

3. Bamboo

4. CircleCI

5. Gitlab CI

6. Travis CI

7. Codefresh

Below are some of the best practices in Jenkins:

Don’t Use Groovy Code for Main Pipeline Functionality –

Prefer to use single steps (such as shell commands), and not Groovy code, for each step of the pipeline. Use Groovy code to chain your steps together. This will reduce the complexity of your pipeline, and ensure that as the number of steps grows, the pipeline can still run without major resources on the controller.

Use Shared Libraries -

As an organization uses Jenkins Pipeline for more projects, different teams will probably create similar pipelines. It is useful to share parts of the pipeline between different projects to reduce duplication. To this end, Jenkins Pipeline lets you create shared libraries, which you can define in an external source control repo and load into your existing pipelines.

Use Docker Images in Your Pipeline –

Many organizations use Docker to set up build and test environments and deploy applications. From Jenkins 2.5 onwards, Jenkins Pipeline has built-in support for interacting with Docker within Jenkins files.

Jenkins Pipeline lets you use Docker images for a single stage of the pipeline or the entire pipeline. This lets you define the tools you need directly in the Docker image, without having to configure agents. You can use Docker containers with minor modifications to a Jenkinsfile.

The code looks like this. When this pipeline runs, Jenkins will automatically start the required container.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent {
        docker { image 'node:16.13.1-alpine' }
    }
    stages {
        stage('Test') {
            steps {
                sh 'node --version'
            }
        }
    }
}
```

Use Multi-Branch Pipelines -

When working with Jenkins, you may need to build an application using several branches in one Git repository. Normally, you would create new Jenkins jobs for every new branch—a multi-branch pipeline job can make this process less time-consuming. All the pipeline information, including the Jenkinsfile, must be in the same place (i.e., the Git repo).

Backup Every Jenkins Instance -

It is also crucial to have strong backups of every Jenkins instance to enable disaster recovery and restore corrupted or accidentally deleted files. Backup creation schemes include snapshots of the file system, backup plugins, and shell scripts that back up Jenkins instances. The number of backed-up files affects the overall backup size, complexity, and recovery time.

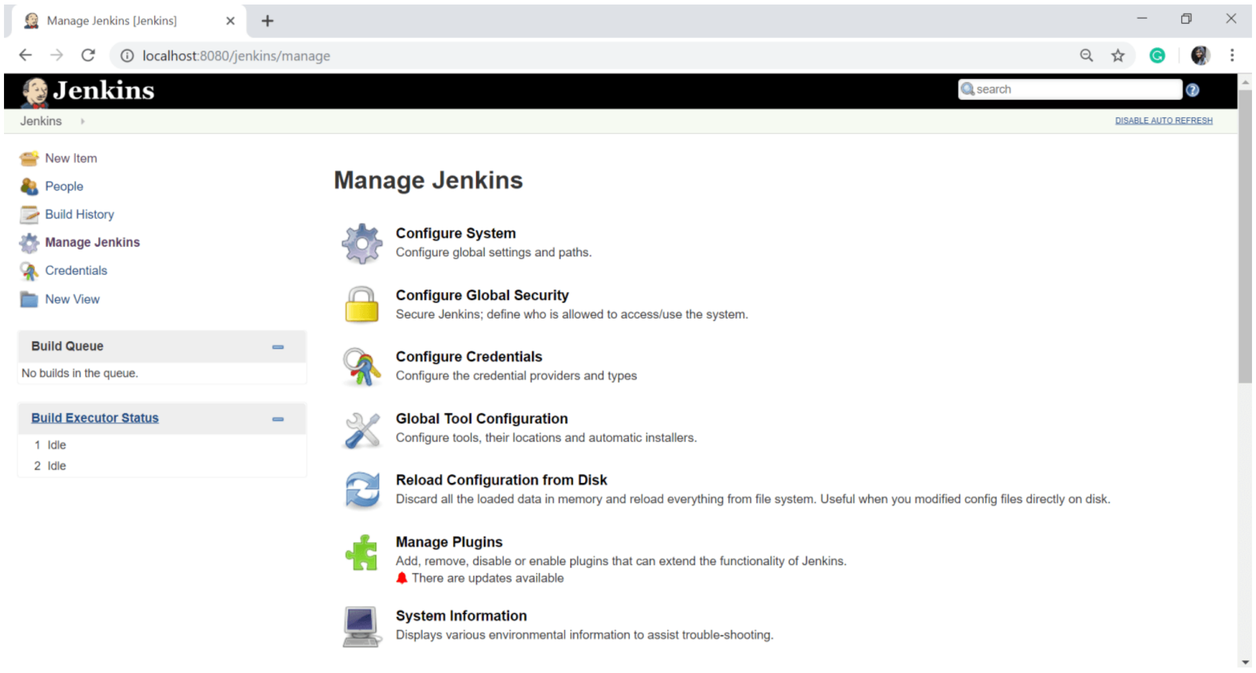
Cleaning Up Old Jenkins Builds -

Jenkins admins can delete old and unwanted builds, and this will not affect operations of the Jenkins controller. However, if you don't delete old builds, you will eventually run out of resources for new releases. The buildDiscarderdirective in pipeline jobs can help you define a policy for automatically removing older builds.

Manage Jenkins - Some of the options in Jenkins Management

To manage Jenkins, click on the "Manage Jenkins" option from the left hand of the Jenkins Dashboard page.

When you click on the Manage Jenkins, you will get the various options to manage the Jenkins:



No builds in the queue.

1 Idle
2 Idle

Manage Jenkins

Configure System
Configure global settings and paths.

Configure Global Security
Secure Jenkins; define who is allowed to access/use the system.

Configure Credentials
Configure the credential providers and types

Global Tool Configuration
Configure tools, their locations and automatic installers.

Reload Configuration from Disk
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly on disk.

Manage Plugins
Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
There are updates available

System Information
Displays various environmental information to assist trouble-shooting.

We have a lot of options in Jenkins Management page like:

- Configure System
- Configure Global Security
- Configure Credentials
- Global Tool Configuration
- Reload Configuration from Disk
- Manage Plugins
- System Information
- System Log
- Load Statistics
- Jenkins CLI
- Script Console
- Manage Nodes
- About Jenkins
- Manage Old Data and
- Prepare for Shutdown

Let's see some above management options:

Configure System:
In this, we can manage paths to the various tools to use in builds, such as the versions of Ant and Maven, as well as security options, email servers, and other system-wide configuration details. Jenkins will add the required configuration fields dynamically when new plugins are installed.

Configure Global Security:
Configure Global Security option provides the ability to set up users and their relevant permissions on the Jenkins instance. By default, you will not want everyone to be able to define builds or other administrative tasks in Jenkins. So Jenkins provides the ability to have a security configuration in place.

Reload Configuration from Disk:
Jenkins stores all its system and job configuration details as XML files and all XML files are stored in the Jenkins home directory. Jenkins also stores all of the build histories.

If you are moving build jobs from one Jenkins instance to another or archiving old build jobs, you will have to insert or remove the corresponding build job directories to Jenkins's builds directory. You do not have to take Jenkins offline to do this?you can simply use the "Reload Configuration from Disk" option to reload the Jenkins system and build job configurations directly.

Manage Plugin:
Here you can install a wide or different variety of third-party plugins from different Source code management tools such as Git, ClearCase or Mercurial, to code quality and code coverage metrics reporting. We can download, install, update, or remove the plugins from the Manage Plugins screen.
System Information:

This option displays a list of all the current Java system properties and system environment variables. Here you can check what version of Java is currently running in, what user it is running under, and so forth.

System Log:
The System Log option is used to view the Jenkins log files in real time. As well as, the main use of this option is for troubleshooting.

Load Statistics:
This option is used to see the graphical data on how busy the Jenkins instance is in terms of the number of concurrent builds and the length of the build queue which provides an idea of how long your builds need to wait before being executed. These statistics can show you a good idea of whether extra capacity or extra build nodes are required from an infrastructure perspective.

Jenkins CLI:
Using this option, you can access various features in Jenkins through a command-line. To run the Jenkins through cli, first you have to download the Jenkins-cli.jar file and run it on the system.

Script Console:
This option allows you to run Groovy scripts on the server. This option is very useful for advanced troubleshooting since it requires a strong knowledge of the internal Jenkins architecture.

Manage nodes:
Jenkins can handle parallel and distributed builds. In this page, you can configure how many builds you want. Jenkins runs concurrently, and, if you are using distributed builds, set up builds nodes. A build node (slave) is another machine that Jenkins can use to execute its builds.

About Jenkins:
This option will show the version and license information of the Jenkins you are running. As well as it displays the list of all third party libraries.

Prepare for Shutdown:
If you want to shut down Jenkins, or the server Jenkins is running on, it is best not to do so when a build is being executed. To shut down Jenkins cleanly, you can use the Prepare for Shutdown link, which prevents any new builds from being started. Eventually, when all of the current builds have completed, you will be able to shut down Jenkins cleanly.

The Jekins Maintenance activities

The following are some of the basic activities you will carry out, some of which are best practices for Jenkins server maintenance

URL Options
The following commands when appended to the Jenkins instance URL will carry out the relevant actions on the Jenkins instance, being the 8080 is standard http port.

http://localhost:8080/jenkins/exit − shutdown jenkins

http://localhost:8080/jenkins/restart − restart jenkins

http://localhost:8080/jenkins/reload − to reload the configuration

We could also do the backup Jenkins Home -
The Jenkins Home directory is nothing but the location on your drive where Jenkins stores all information for the jobs, builds etc. The location of your home directory can be seen when you click on Manage Jenkins → Configure system.

Set up Jenkins on the partition that has the most free disk-space – Since Jenkins would be taking source code for the various jobs defined and doing continuous builds, always ensure that Jenkins is setup on a drive that has enough hard disk space. If you hard disk runs out of space, then all builds on the Jenkins instance will start failing.

Another best practice is to write cron jobs or maintenance tasks that can carry out clean-up operations to avoid the disk where Jenkins is setup from becoming full.

Clean-up or achieving the old jobs periodically.

Handling users, credentials stored in plugin and security aspects of Jenkins server would also come under maintenance.

Jenkins server related patches, updates are the regular maintenance activities too, that includes plugin upgrades ss well.

Maintenance Jobs Scheduler - Perform deleting or disabling of old jobs based on some cron tasks. You can configure this plugin globally based on some specific scheduler, excluding jobs with some regex, add some description in each disabling job (for tracking purposes), apply that filter for those jobs older than X days.

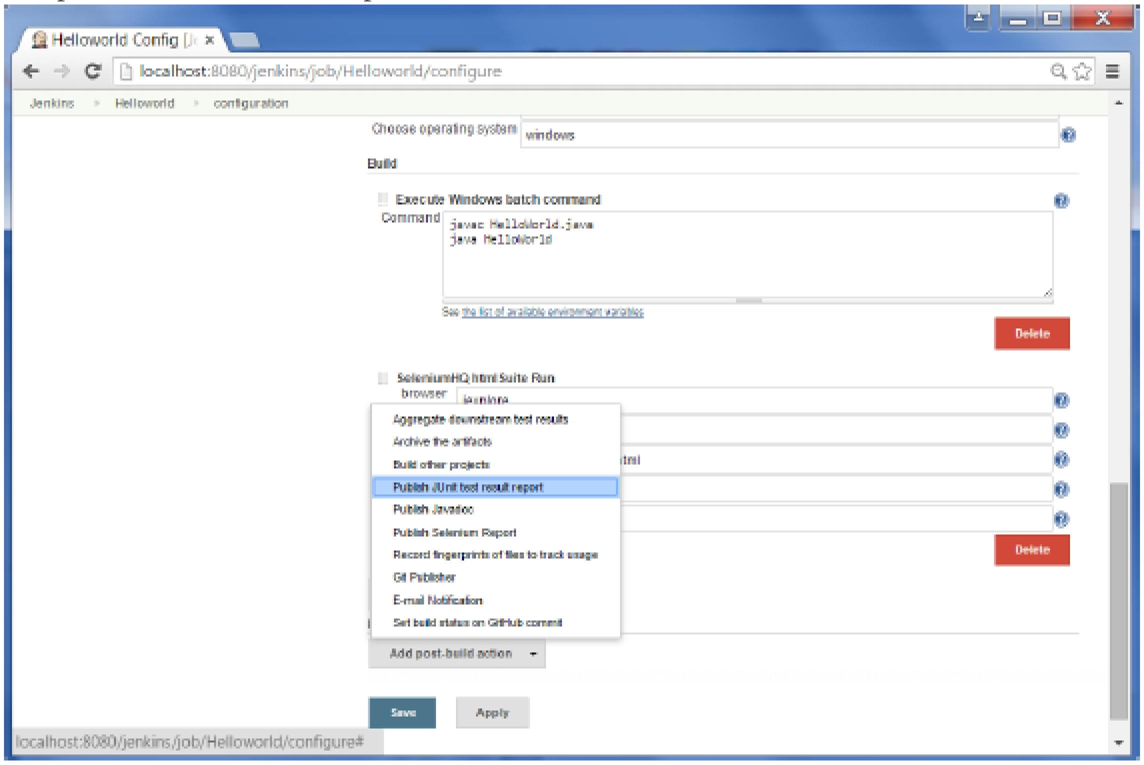
Agent maintenance is crucial to run the deployments on the application servers, Keep monitoring enabled on the agent nodes to ensure all the agents are up & running all the times.

Reporting and Security aspects in Jenkins

Jenkins can generate reports that summarize and analyze the results of your CI jobs, such as test reports, code coverage reports, and code quality reports. Test reports show the number of tests, failures, errors, and skipped tests with the option to drill down to the details of each test case.

There are many reporting plugins available with the simplest one being the reports available for jUnit tests.

In the Post-build action for any job, you can define the reports to be created. After the builds are complete, the Test Results option will be available for further drill-down.



Choose operating system: windows

Execute Windows batch command

Command:

```
javac HelloWorld.java
java HelloWorld
```

SeleniumHQ htmlSuite Run

browser: iexplore

Also navigate to your job in Jenkins, you will see any link to the reports, but once you do a Build Now, you would see a new link generated "Latest Test Result" and a chart showing the test failures. We can see the xml which contains the JUnit report where all the pass and fail records available based on the test cases.

Managing Security:

Jenkins is used everywhere from workstations on corporate intranets, to high-powered servers connected to the public internet. To safely support this wide spread of security and threat profiles, Jenkins offers many configuration options for enabling, editing, or disabling various security features.

As of Jenkins 2.0, many of the security options were enabled by default to ensure that Jenkins environments remained secure unless an administrator explicitly disabled certain protections.

This section will introduce the various security options available to a Jenkins administrator, explaining the protections offered, and trade-offs to disabling some of them.

Enabling Security -
Beginning with Jenkins 2.214 and Jenkins LTS 2.222.1, the "Enable Security" checkbox has been removed. Jenkins own user database is used as the default security realm.

Access Control

Access Control is the primary mechanism for securing a Jenkins environment against unauthorized usage. Two facets of configuration are necessary for configuring Access Control in Jenkins:

1. A Security Realm which informs the Jenkins environment how and where to pull user (or identity) information from. Also commonly known as "authentication."

2. Authorization configuration which informs the Jenkins environment as to which users and/or groups can access which aspects of Jenkins, and to what extent.

Using both the Security Realm and Authorization configurations it is possible to configure very relaxed or very rigid authentication and authorization schemes in Jenkins.

Security Realm
By default Jenkins includes support for a few different Security Realms:

Delegate to servlet container-
For delegating authentication a servlet container running the Jenkins controller, such as Jetty. If using this option, please consult the servlet container’s authentication documentation.

Jenkins’ own user database-
Use Jenkins’s own built-in user data store for authentication instead of delegating to an external system. This is enabled by default with new Jenkins 2.0 or later installations and is suitable for smaller environments.

LDAP-
Delegate all authentication to a configured LDAP server, including both users and groups. This option is more common for larger installations in organizations which already have configured an external identity provider such as LDAP. This also supports Active Directory installations.

Unix user/group database
Delegates the authentication to the underlying Unix OS-level user database on the Jenkins controller. This mode will also allow re-use of Unix groups for authorization. For example, Jenkins can be configured such that "Everyone in the developers group has administrator access."

Authorization-
The Security Realm, or authentication, indicates who can access the Jenkins environment. The other piece of the puzzle is Authorization, which indicates what they can access in the Jenkins environment. By default Jenkins supports a few different Authorization options:

Anyone can do anything
Everyone gets full control of Jenkins, including anonymous users who haven’t logged in. Do not use this setting for anything other than local test Jenkins controllers.

Legacy mode

Behaves exactly the same as Jenkins <1.164. Namely, if a user has the "admin" role, they will be granted full control over the system, and otherwise (including anonymous users) will only have the read access. Do not use this setting for anything other than local test Jenkins controllers.

Logged in users can do anything
In this mode, every logged-in user gets full control of Jenkins. Depending on an advanced option, anonymous users get read access to Jenkins, or no access at all. This mode is useful to force users to log in before taking actions, so that there is an audit trail of users' actions.

Matrix-based security
This authorization scheme allows for granular control over which users and groups are able to perform which actions in the Jenkins environment (see the screenshot below).

Project-based Matrix Authorization Strategy
This authorization scheme is an extension to Matrix-based security which allows additional access control lists (ACLs) to be defined for each project separately in the Project configuration screen. This allows granting specific users or groups access only to specified projects, instead of all projects in the Jenkins environment.

Also there are few other items involved around security in Jenkins like Markup Formatter, CSRF Protection, Agent/Master Access Control etc.

Few of the plugins that can be installed on Jenkins

Plugins are the primary means of enhancing the functionality of a Jenkins environment to suit organization- or user-specific needs. There are over a thousand different plugins which can be installed on a Jenkins controller and to integrate various build tools, cloud providers, analysis tools, and much more.

Minimum hardware requirements: 256 MB of RAM. 1 GB of drive space (although 10 GB is a recommended minimum if running Jenkins as a Docker container)

The Jenkins project only publishes free and open source plugins distributed under an OSI-approved license. The canonical source code repository is in the jenkinsci GitHub organization to ensure source code access and project continuity in case previous maintainers move on.

More than 1,800 plugins are available to install on Jenkins to integrate different build tools, analytic tools, and other cloud providers. These plugins cover everything from managing the source code to app administration, platform support, determining UI/UX, and building management.

Jenkins has a powerful extension and plugin system that allows developers to write plugins affecting nearly every aspect of Jenkins' behavior.
Create a Plugin
Step 1: Preparing for Plugin Development.
Step 2: Create a Plugin.
Step 3: Build and Run the Plugin.

Step 4: Extend the Plugin.

Managing Plugins-
Plugins are the primary means of enhancing the functionality of a Jenkins environment to suit organization- or user-specific needs. There are over a thousand different plugins which can be installed on a Jenkins controller and to integrate various build tools, cloud providers, analysis tools, and much more.

Plugins can be automatically downloaded, with their dependencies, from the Update Center. The Update Center is a service operated by the Jenkins project which provides an inventory of open source plugins which have been developed and maintained by various members of the Jenkins community.

This section covers everything from the basics of managing plugins within the Jenkins web UI, to making changes on the controller's file system.

Installing a plugin-
Jenkins provides two methods for installing plugins on the controller:

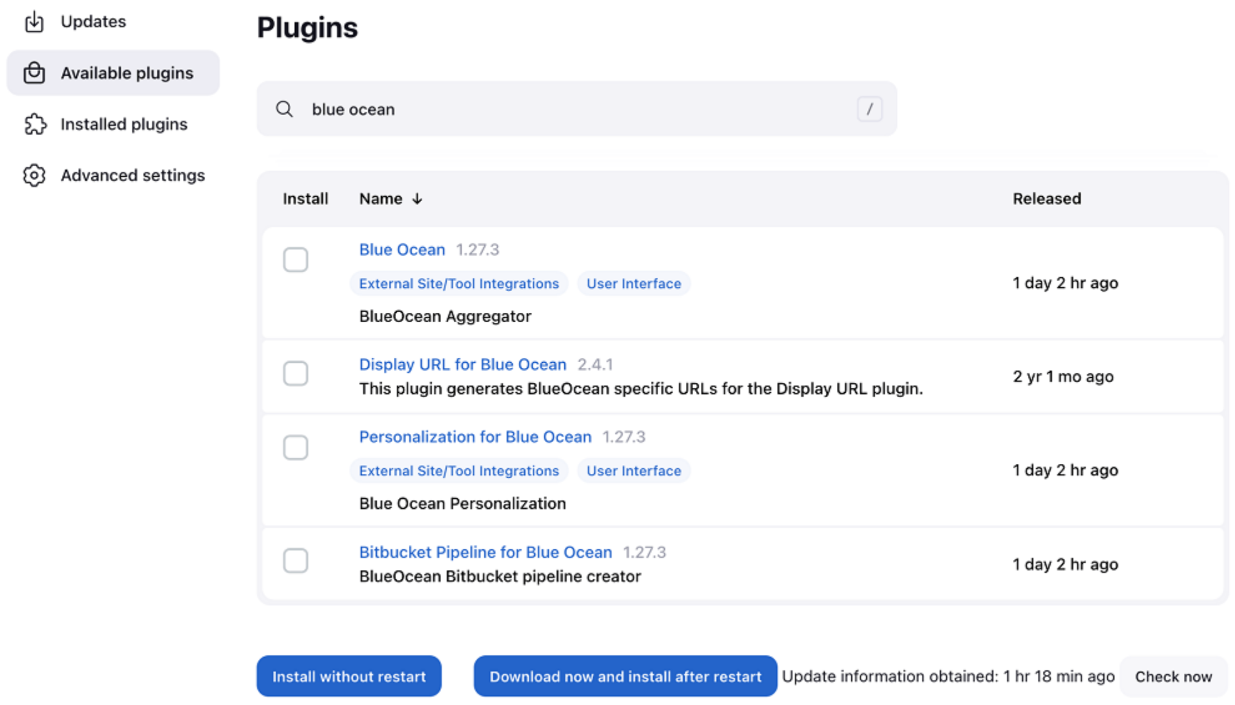
1. Using the "Plugin Manager" in the web UI.
2. Using the Jenkins CLI install-plugin command.

Each approach will result in the plugin being loaded by Jenkins but may require different levels of access and trade-offs in order to use.

The two approaches require that the Jenkins controller be able to download meta-data from an Update Center, whether the primary Update Center operated by the Jenkins project [1], or a custom Update Center.

The plugins are packaged as self-contained .hpi files, which have all the necessary code, images, and other resources which the plugin needs to operate successfully.

From the web UI –
The simplest and most common way of installing plugins is through the Manage Jenkins > Plugins view, available to administrators of a Jenkins environment.
Under the Available tab, plugins available for download from the configured Update Center can be searched and considered:



Plugins

blue ocean

Install	Name ↓	Released
☐	Blue Ocean 1.27.3 External Site/Tool Integrations · User Interface BlueOcean Aggregator	1 day 2 hr ago
☐	Display URL for Blue Ocean 2.4.1 This plugin generates BlueOcean specific URLs for the Display URL plugin.	2 yr 1 mo ago
☐	Personalization for Blue Ocean 1.27.3 External Site/Tool Integrations · User Interface Blue Ocean Personalization	1 day 2 hr ago
☐	Bitbucket Pipeline for Blue Ocean 1.27.3 BlueOcean Bitbucket pipeline creator	1 day 2 hr ago

Update information obtained: 1 hr 18 min ago

Most plugins can be installed and used immediately by checking the box adjacent to the plugin and clicking *Install without restart.*