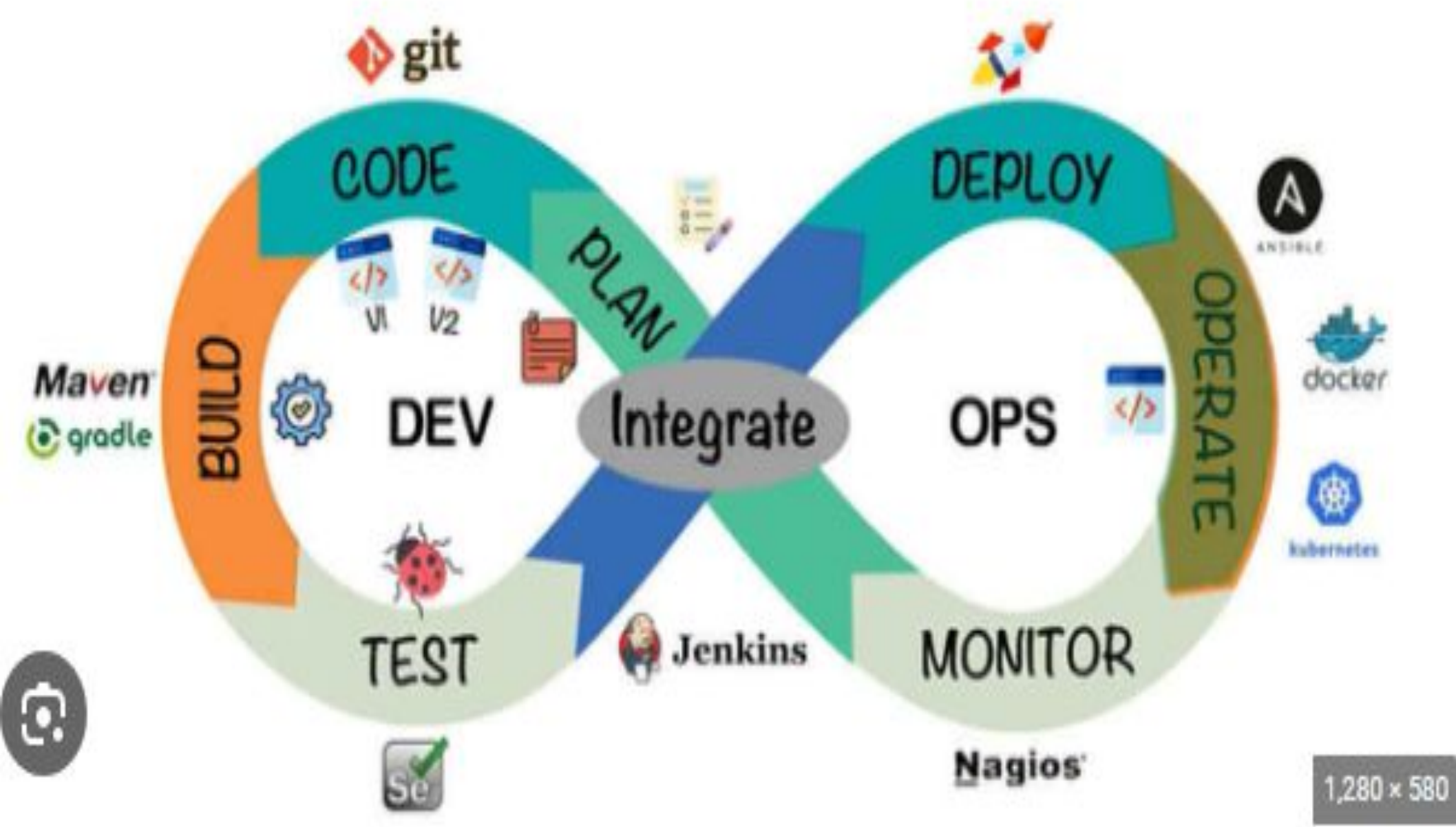
U20ITCM08
Automation Techniques and Tools-DevOps

Dr. K. Lakshmi, Professor, Department of IT,SMVEC



Course Outcomes

- CO1 - Explains about traditional software methodologies and about software project estimation, roles of developers and IT operations conflicts (K2)
- CO2 - Realize the importance of agile software development practices in determining the requirements for a software system and about agile manifesto, values and principles (K3)
- CO3 - Provides basic ideas of devops and its role in terms of version control, automated testing, continuous integration and delivery (K3)
- CO4 - Illustrates the purposes of devops with MVP, continuous integration and delivery (K2)
- CO5 - Explains the role of CAMS in devops (K3)

Syllabus

- **UNIT I- TRADITIONAL SOFTWARE DEVELOPMENT**

  The Advent of Software Engineering - Software Process, Perspective and Specialized Process Models - Software Project Management: Estimation - Developers vs IT Operations conflict.

- **UNIT II- RISE OF AGILE METHODOLOGIES**

  Agile movement in 2000 - Agile Vs Waterfall Method - Iterative Agile Software Development - Individual and team interactions over processes and tools – Working software over comprehensive documentation -Customer collaboration over contract negotiation - Responding to change over following a plan

- **UNIT III- INTRODUCTION DEVOPS**

  Introduction to DevOps - Version control - Automated testing - Continuous integration - Continuous delivery -Deployment pipeline - Infrastructure management – Databases

- **UNIT IV – PURPOSE OF DEVOPS**

  Minimum Viable Product- Application Deployment- Continuous Integration- Continuous Delivery

- **UNIT V – CAMS (CULTURE, AUTOMATION, MEASUREMENT AND SHARING)**

  CAMS – Culture, CAMS – Automation, CAMS – Measurement, CAMS – Sharing, Test-Driven Development, Configuration Management-Infrastructure Automation- Root Cause Analysis- Blamelessness- Organizational Learning

Text Books and References

- **Text Books:**
  - Dev Ops – Volume 1 , Pearson and Xebia Press
  - Grig Gheorghiu, Alfredo Deza, Kennedy Behrman, Noah Gift, Python for DevOps,2019
- **Reference Books:**
  - The DevOps Handbook - Book by Gene Kim, Jez Humble, Patrick Debois, and Willis Willis
  - What is DevOps? - by Mike Loukides
  - Joakim Verona, Practical DevOps ,2016.
- **Websites:**
  - www.ibm.com/cloud/devops.
  - www.softwaretestinghelp.com>devops-automation.

UNIT IV

UNIT IV- PURPOSE OF DEVOPS

Minimum Viable Product- Application Deployment- Continuous Integration- Continuous Delivery

Minimum Viable Product

Minimum Viable Product

- **The minimum viable product, or MVP, is the simplest version of a product that you need to build to sell it to a market.**
- **Minimum Viable Product** (MVP) is a product with basic features which is launched to gain users and shape the future features based on user feedback. This enables companies to bring in new products regularly.

- For **example**, let's look at **WhatsApp**, an application used by millions of users across the world. WhatsApp consists of multiple components such as chat, photo/video sharing, video calling, location sharing, payments etc., But did WhatsApp have all of these components right from the beginning?
- How about Facebook? The product that we use today is much more than what it was, when they started close to a decade back. Facebook's first product release **did not have a facebook wall or news feed**.
- The idea is to roll out a basic version of the final product, get customer feedback and shape the features according to the feedback.
- Thus minimum viable product or MVP is that version of the product with basic features, just enough to get the attention of the customers. The final product will be shaped and released after getting sufficient feedback from the initial set of users.

- For example, imagine a company who is building a recruitment software.
- As MVP, they might just release a job board, where all the jobs are listed and applicants can send resumes through email.
- Upon getting sufficient feedback from customers, they can introduce features like applicant tracking systems, job recommendation engine, job assessments, live interviews, etc. as different versions of the product.
- Eventually, the company will add more and more features to the product. By releasing the MVP, they will get to know the customer mindset, the demand and reception for the product and more importantly they get the early mover advantage. They will be able to establish themselves as a brand that people prefer.
- Businesses should also be careful while adding features to the MVP. In the process of building the perfect product, some organizations, lose focus and load the MVP with large number of irrelevant features. Customers will start losing interest and eventually, the business will fail.

How to develop a Minimum Viable Product?

How to develop a Minimum Viable Product?

- 1. Establish Your Objective and Target Market:
  - Clearly define the goal of your MVP. What problem does it solve, and for whom?
  - Determine who your target market is and what problems they are facing.

- 2. Identify Core Features:
  - List the absolute minimum features required to address the identified problem or need.
  - Focus on features that provide the most value to your users.

- 3. Prioritize Features:
  - Prioritize features based on their importance to the core functionality of the product.
  - Identify features that are essential for the initial release and can be built in a short timeframe.

How to develop a Minimum Viable Product?

- 4. Create a User Flow:
  - Outline the user journey through your product.
  - Define how users will interact with the MVP, from onboarding to completing key actions.

- 5. Design the User Interface (UI):
  - Develop a simple and intuitive UI that aligns with your user flow.
  - Use wireframes or low-fidelity prototypes to visualize the user experience.

- 6 Develop the MVP:
  - Start the development process, focusing on implementing the prioritized features.
  - Use agile development methodologies to enable quick iterations and flexibility.

Role of DevOps in MVP development:

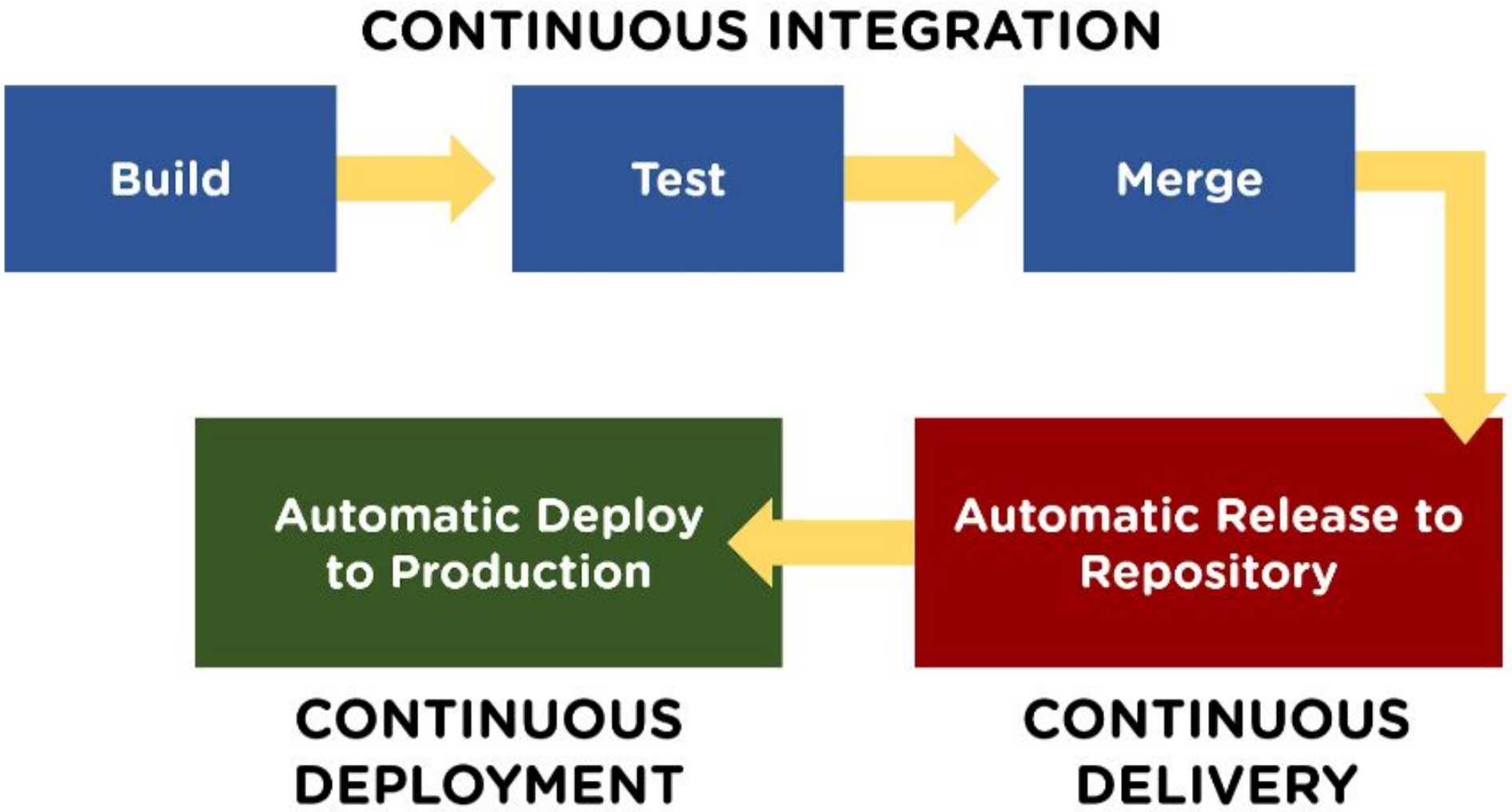
- **Continuous Updates:** DevOps enables continuous integration and delivery, allowing quick updates to the MVP based on dynamic market conditions and customer feedback.
- **Microservices:** Breaking down the application into microservices simplifies maintenance, updates, and bug-fixing, fostering seamless collaboration among teams and accelerating MVP development.
- **Automation:** DevOps automation streamlines routine tasks, expediting feature development and deployment. Infrastructure as Code (IaC) facilitates rapid scalability and deployment of MVPs.

Benefits of Minimum Viable Product

- Faster Time to Market: By focusing on essential features, an MVP allows for quicker development and deployment. This rapid time to market is crucial for gaining a competitive edge and responding promptly to market demands.
- Cost Efficiency: Developing an MVP involves allocating resources efficiently to build only the core features necessary for the initial release. This approach reduces development costs and minimizes the risk of investing heavily in a product that may not succeed.
- Market Validation: An MVP serves as a tool for testing assumptions and validating the market demand for a product. By releasing a basic version and collecting user feedback, businesses can assess whether there is genuine interest in their solution.
- User Feedback and Iteration: The release of an MVP allows for the collection of valuable user feedback. This feedback is essential for understanding user preferences, identifying issues, and making informed decisions for future iterations. The iterative development process ensures that the product evolves based on real-world user experiences.
- Risk Mitigation: Developing a full-featured product without market validation poses a significant risk. By starting with an MVP, businesses can test their ideas with minimum investment and adjust their strategy based on real user data, reducing the risk of failure.

Continuous Integration

Continous Integration

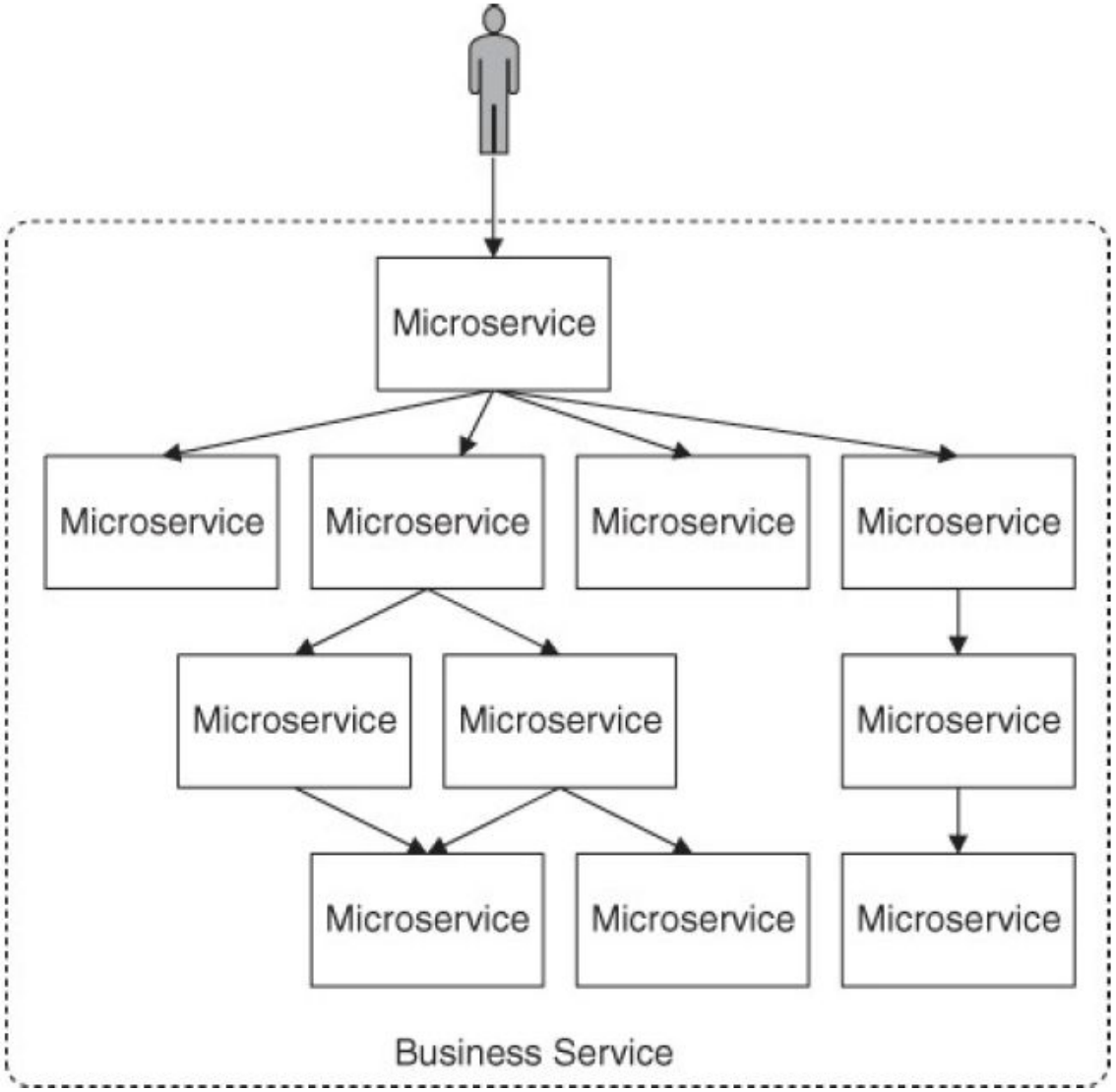


https://www.simplilearn.com/tutorials/devops-tutorial/continuous-integration

Microservice architecture

- Microservice architecture is an architectural style that satisfies these requirements. This style is used in practice by organizations that have adopted or inspired many DevOps practices.
- A microservice architecture consists of a collection of services where each service provides a small amount of functionality and the total functionality of the system is derived from composing multiple services.

Microservice architecture



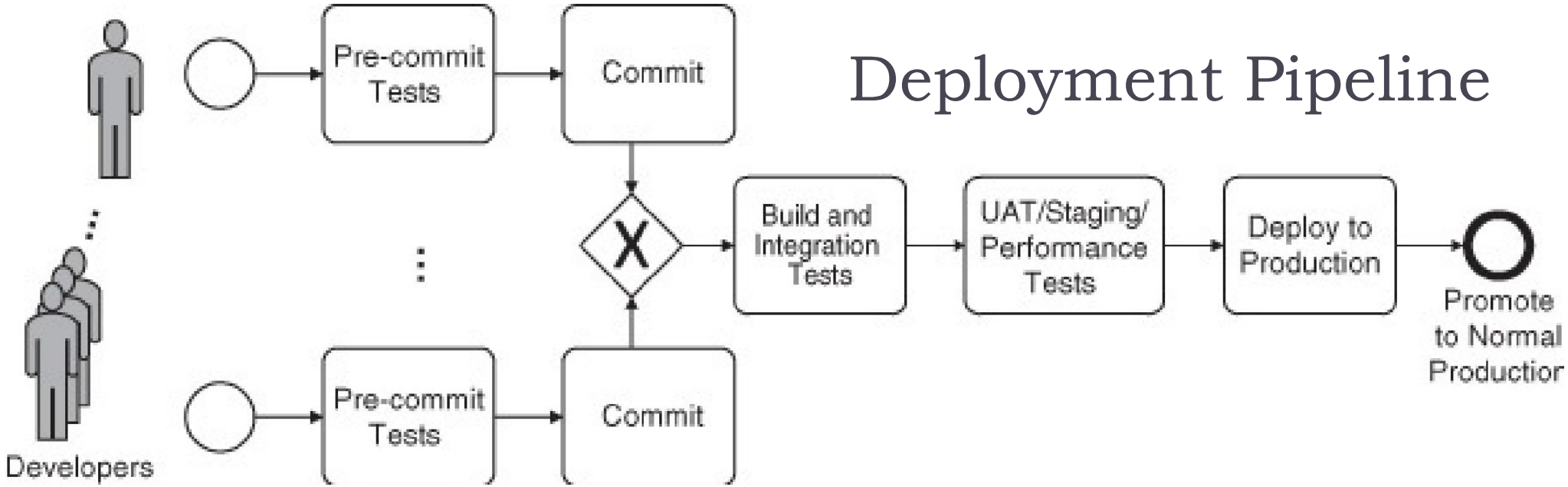
Microservice architecture

- Figure 4.1 describes the situation that results from using a microservice architecture.
- A user interacts with a single consumer-facing service. This service, in turn, utilizes a collection of other services.
- We use the terminology service to refer to a component that provides a service and client to refer to a component that requests a service.
- A single component can be a client in one interaction and a service in another.

Infrastructure requirements

- Team members can work on different versions of the system concurrently.
- Code developed by one team member does not overwrite the code developed by another team member by accident.
- Work is not lost if a team member suddenly leaves the team.
- Team members' code can be easily tested.
- Team members' code can be easily integrated with the code produced by other members of the same team.
- The code produced by one team can be easily integrated with code produced by other teams.
- An integrated version of the system can be easily deployed into various environments (e.g., testing, staging, and production).
- An integrated version of the system can be easily and fully tested without affecting the production version of the system.
- A recently deployed new version of the system can be closely supervised.
- Older versions of the code are available in case a problem develops once the code has been placed into production.
- Code can be rolled back in the case of a problem.

Deployment Pipeline



- The deployment pipeline begins when a developer commits code to a joint versioning system.
- Prior to doing this commit, the developer will have performed a series of pre-commit tests on their local environment;
- A commit then triggers an integration build of the service being developed.
- This build is tested by integration tests. If these tests are successful, the build is promoted to a quasi-production environment—the staging environment—where it is tested once more.
- Then, it is promoted to production under close supervision. After another period of close supervision, it is promoted to normal production.

Deployment Pipeline

- One way to define continuous integration is to have automatic triggers between one phase and the next, up to integration tests. That is, if the build is successful then integration tests are triggered. If not, the developer responsible for the failure is notified.
- Continuous delivery is defined as having automated triggers as far as the staging system.
- Continuous deployment means that the next to last step (i.e., deployment into the production system) is automated as well.
- Once a service is deployed into production it is closely monitored for a period and then it is promoted into normal production.

Moving a System Through the Deployment Pipeline

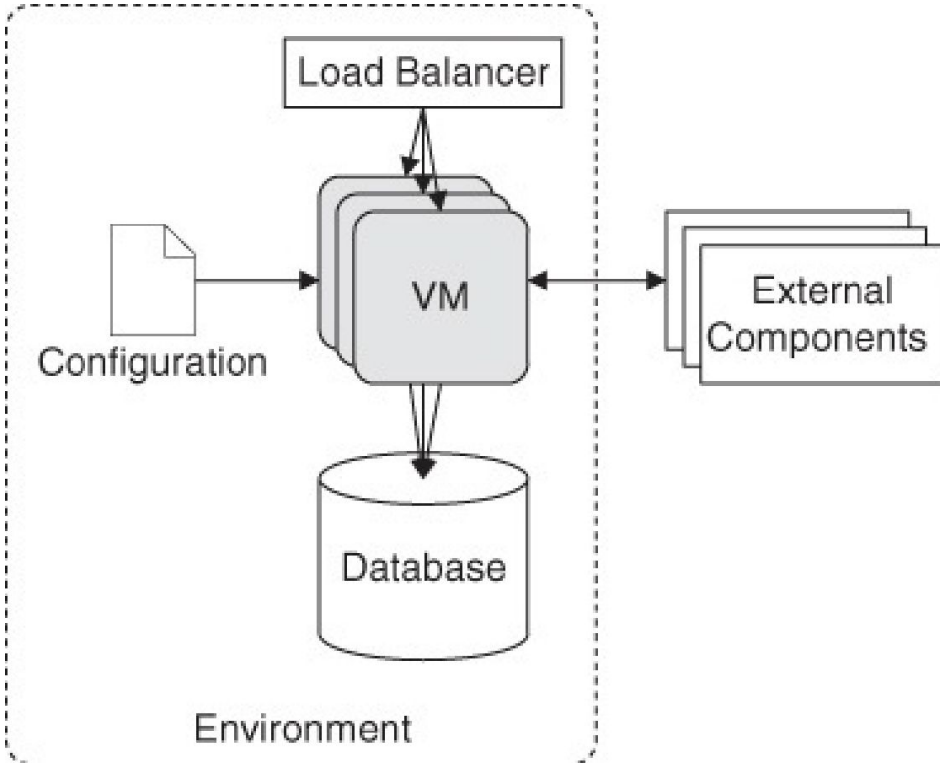
- Code can't move automatically on the pipeline.
- It is controlled by the developer or developor command or triggers or scripts in the pipeline.
- Two important things associated in movement.
  - Traceability
  - Environment associated in each step of the pipeline

Traceability

- Traceability means that, for any system in production, it is possible to determine exactly how it came to be in production. This means keeping track not only of source code but also of all the commands to all the tools that acted on the elements of the system.
- Controlling tools by scripts is far better than controlling tools by commands.
- The scripts and associated configuration parameters should be kept under version control, just as the application code.
- Tests are also maintained in version control.
- Configuration parameters can be kept as files that are stored in version control or handled through dedicated configuration management systems.
- Treating infrastructure-as-code means that this code should be subject to the same quality control as application source code. That is, this code should be tested, changes to it should be regulated in some fashion, and its different parts should have owners.
- Keeping everything in version control and configuration management systems allows you to re-create the exact environments used anywhere, from local development to production.

The Environment

- An executing system can be viewed as a collection of executing code, an environment, configuration, systems outside of the environment with which the primary system interacts, and data.



The Environment

Pre-commit.

- The code is the module of the system on which the developer is working. Building this code into something that can be tested requires access to the appropriate portions of the version control repository that are being created by other developers.
- The environment is typically a laptop or a desktop, the external systems are stubbed out or mocked, and only limited data is used for testing.
- Configuration parameters should reflect the environment and also control the debugging level.

The Environment

Build and integration testing.

- The environment is usually a continuous integration server. The code is compiled, and the component is built and baked into a VM image. The image can be either heavily or lightly baked. This VM image does not change in subsequent steps of the pipeline.
- During integration testing, a set of test data forms a test database. This database is not the production database, rather, it consists of a sufficient amount of data to perform the automated tests associated with integration.
- The configuration parameters connect the built system with an integration testing environment.

The Environment

UAT/staging/performance testing.

- The environment is as close to production as possible. Automated acceptance tests are run, and stress testing is performed through the use of artificially generated workloads.
- The database should have some subset of actual production data in it. With very large data sets, it may not be possible to have a complete copy of the actual data, but the subset should be large enough to enable the tests to be run in a realistic setting.
- Configuration parameters connect the tested system with the larger test environment.
- Access to the production database should not be allowed from the staging environment.

Production.

- The production environment should access the live database and have sufficient resources to adequately handle its workload.
- Configuration parameters connect the system with the production environment.

Development and Pre-commit Testing

Version Control and Branching

- Core features of version control are: the ability to identify distinct versions of the source code, sharing code revisions between developers, recording who made a change from one version to the next, and recording the scope of a change.
- CVS and SVN are centralized solutions, where each developer checks out code from a central server and commits changes back to that server.
- Git is a distributed version control system: Every developer has a local clone (or copy) of a Git repository that holds all contents. Commits are done to the local repository.
- A set of changes can be synchronized against a central server, where changes from the server are synchronized with the local repository (using the pull command) and local changes can be forwarded to the server (using the push command). Push can only be executed if the local repository is up-to-date, hence a push is usually preceded by a pull.
- Almost all version control systems support the creation of new branches. A branch is essentially a copy of a repository (or a portion) and allows independent evolution of two or more streams of work.

Development and Pre-commit Testing

- For example, an error has been occured in the production software, the code is taken as a branch, error is fixed and merged to the main trunk.
- Problems in branching:
  1. Too many branches will give a confusion on which branch a new change is to be updated. For this reason, short-lived tasks should not create a new branch.
  2. Merging two branches can be difficult. Different branches evolve concurrently, and often developers touch many different parts of the code.
- Alternate method to the branching is to do updates in the main trunk only. But the problem is this method does not allow the concurrent development of new feature and the existance of old feature/code.
- Solution for this problem is **feature toggle.**

Development and Pre-commit Testing

Feature Toggle

- A feature toggle (also called a feature flag or a feature switch) is an "if" statement around immature code. Example.

```
If (Feature_Toggle) then
    new code
else
    old code
end;
```

- A new feature that is not ready for testing or production is disabled in the source code itself, for example, by setting a global Boolean variable.
- Common practice places the switches for features into configuration.
- Feature toggling allows you to continuously deliver new releases, which may include unfinished new features—but these do not impact the application, since they are still switched off.
- The switch is toggled in production (i.e., the feature is turned on) only once the feature is ready to be released and has successfully passed all necessary tests.

Development and Pre-commit Testing

- When there are many feature toggles, managing them becomes complicated.
- It would be useful to have a specialized tool or library that knows about all of the feature toggles in the system, is aware of their current state, can change their state, and can eventually remove the feature toggle from your code base.

Development and Pre-commit Testing

Configuration Parameters

- A configuration parameter is an externally settable variable that changes the behavior of a system.
- A configuration setting may be: the language you wish to expose to the user, the location of a data file, the thread pool size, the color of the background on the screen, or the feature toggle settings.
- The number of configuration parameters should be kept at a manageable level.
- More configuration parameters usually result in complex connections between them, and the set of compatible settings to several parameters will only be known to experts in the configuration of the software.
- Now a days, tools are available to check the configuration setting which includes, correct values, ranges, format, correct url, compatibility with other configuration settings.

Development and Pre-commit Testing

- It is important to decide whether to have same configuration file for different stages. You need to decide on the following
  - ***Values are the same in multiple environments.*** Feature toggles and performance-related values (e.g., database connection pool size) should be the same in performance testing/UAT/staging and production, but may be different on local developer machines.
  - ***Values are different depending on the environment.*** The number of virtual machines (VMs) running in production is likely bigger than that number for the testing environments.
  - ***Values must be kept confidential.*** The credentials for accessing the production database or changing the production infrastructure must be kept confidential and only shared with those who need access to them.

Development and Pre-commit Testing

Testing During Development and Pre-commit Tests

- Two types of development approach is followed
  - **Test-driven development**: before writing the actual code for a piece of functionality, you develop an automated test for it. Then the functionality is developed, with the goal of fulfilling the test. Once the test passes, the code can be refactored to meet higher-quality standards.
  - **Unit tests.** Unit tests are code-level tests, each of which is testing individual classes and methods. The unit test suite should have exhaustive coverage and run very fast. Typical unit tests check functionality that relies solely on the code in one class and should not involve interactions with the file system or the database. A common practice is to write the code in a way that complicated but required artifacts (such as database connections) form an input to a class—unit tests can provide mock versions of these artifacts, which require less overhead and run faster.
- A modern practice enforce pre-commit tests. These tests are run automatically before a commit is executed. Typically they include a relevant set of unit tests, as well as a few smoke tests. Smoke tests are specific tests that check in a fast (and incomplete) manner that the overall functionality of the service can still be performed. The goal is that any bugs that pass unit tests but break the overall system can be found long before integration testing.

Build and Integration Testing

- Build is the process of creating an executable artifact from input such as source code and configuration. As such, it primarily consists of compiling source code and packaging all files that are required for execution (e.g., the executables from the code, interpretable files like HTML, JavaScript, etc.).
- Once the build is complete, a set of automated tests are executed that test whether the integration with other parts of the system uncovers any errors.
- The unit tests can be repeated here to generate a history available more broadly than to a single developer.

Build Scripts

- The build and integration tests are performed by a continuous integration (CI) server.
- The input to this server should be scripts that can be invoked by a single command "Build". All the rest of the activities are controlled by the script.
- This practice ensures that the build is repeatable and traceable.

Build and Integration Testing

Packaging

- The goal of building is to create something suitable for deployment. There are several standard methods of packaging the elements of a system for deployment.
- The appropriate method of packaging will depend on the production environment. Some packaging options are:
- **Runtime-specific packages**, such as Java archives, web application
- **Operating system packages.** If the application is packaged into software packages of the target OS
- **VM images** can be created from a template image, to include the changes from the latest revision. VMware images require a VMware hypervisor; Amazon Web Services can only run Amazon Machine Images; and so forth. This implies that the test environments must use the same cloud service. If not, the deployment needs to be adapted accordingly, which means that the deployment to test environments does not necessarily test the deployment scripts for production

Build and Integration Testing

- **Lightweight containers** are a new phenomenon. Like VM images, lightweight containers can contain all libraries and other pieces of software necessary to run the application, while retaining isolation of processes, rights, files, and so forth. In contrast to VM images, lightweight containers do not require a hypervisor on the host machine, nor do they contain the whole operating system, which reduces overhead, load, and size. Lightweight containers can run on local developer machines, on test servers owned by the organization, and on public cloud resources—but they require a compatible operating system. Ideally the same version of the same operating system should be used, because otherwise, as before, the test environments do not fully reflect the production environment.

Continuous Integration and Build Status

- Once building is set up as a script callable as a single command, continuous integration can be done as follows:
  - The CI server gets notified of new commits or checks periodically for them.
  - When a new commit is detected, the CI server retrieves it.
  - The CI server runs the build scripts.
  - If the build is successful, the CI server runs the automated tests.
  - The CI server provides results from its activities to the development team (e.g., via an internal web page or e-mail).
- A commit is said to break the build if the compilation/build procedure fails, or if the automatic tests that are triggered by it violate a defined range of acceptable values for some metrics.
  - For instance, forgetting to add a new file in a commit but changing other files that assume the presence of a new file will break the build.
- Tests can be roughly categorized into critical (a single failure of a test would result in breaking the build) and less critical (only a percentage of failed tests larger than a set threshold would result in breaking the build).

Integration Testing

- Integration testing is the step in which the built executable artifact is tested.
- The environment includes connections to external services, such as a surrogate database.
- Including other services requires mechanisms to distinguish between production and test requests, so that running a test does not trigger any actual transactions, such as production, shipment, or payment.
- This distinction can be achieved by providing mock services, by using a test version provided by the owner of the service, or—if dealing with test-aware components—by marking test messages as such by using mechanisms built into the protocol used to communicate with that service.
- If mock versions of services are used, it is good practice to separate the test network from the real services (e.g., by firewall rules) to make absolutely sure no actual requests are sent by running the tests.

Continuous Delivery

UAT/Staging/Performance Testing

- Staging is the last step of the deployment pipeline prior to deploying the system into production.
- The staging environment mirrors, as much as possible, the production environment.
- The types of tests that occur at this step are the following:
  - User acceptance tests (UATs) are tests where prospective users work with a current revision of the system through its UI and test it, either according to a test script or in an exploratory fashion.Some confidential data may be removed or replaced in the UAT environment, where test users or UAT operators do not have sufficient levels of authorization. UATs are valuable for aspects that are hard or impossible to automate, such as consistent look and feel, usability, or exploratory testing.

UAT/Staging/Performance Testing

- **Automated acceptance tests** are the automated version of repetitive UATs. Such tests control the application through the UI, trying to closely mirror what a human user would do. As such, automated acceptance tests enable a higher rate of repetition than is possible with relatively expensive human testers, at odd times of the day or night. Due to the relatively high effort to automate acceptance tests, they are often done only for the most important checks, which need to be executed repetitively and are unlikely to require a lot of maintenance.Automated acceptance tests are relatively slow to execute and require proper setup.

UAT/Staging/Performance Testing

- **Smoke tests,** are a subset of the automated acceptance tests that are used to quickly analyze if a new commit breaks some of the core functions of the application. The name is believed to have originated in plumbing: A closed system of pipes is filled with smoke, and if there are any leaks, it is easy to detect them. One rule of thumb is to have a smoke test for every user story, following the happy path in it. Smoke tests should be implemented to run relatively fast, so that they can be run even as part of the pre-commit tests.

UAT/Staging/Performance Testing

- **Nonfunctional tests** test aspects such as performance, security, capacity, and availability. Proper performance testing requires a suitable setup, using resources comparable to production and very similar every time the tests are run. This ensures that changes from the application, not background noise, are measured. As with the setup of other environments, virtualization and cloud technology make things easier. However, especially when it comes to public cloud resources, one needs to be careful in that regard because public clouds often exhibit performance variability.

Deployment

Introduction

- Deployment is the process of placing a version of a service into production.
- The initial deployment of a service can be viewed as going from no version of the service to the initial version of the service.
- The overall goal of a deployment is to place an upgraded version of the service into production with minimal impact to the users of the system, be it through failures or downtime.
- There are three reasons for changing a service—to fix an error, to improve some quality of the service, or to add a new feature.

Strategies for Managing a Deployment

The goal of a deployment is to move from the current state that has N VMs of the old version, A, of a service executing, to a new state where there are N VMs of the new version, B, of the same service in execution.

The following assumptions are made in the deployment

- Service to the clients should be maintained while the new version is being deployed. Maintaining service to the clients with no downtime is essential for many Internet e-commerce businesses. Even if the system permits downtime, it demands the odd working time of the system administrators.
- Any development team should be able to deploy a new version of their service at any time without coordinating with other teams. This may certainly have an impact on client services developed by other teams.

Six Strategies for Application Deployment

- There are a variety of techniques to deploy new applications to production, so choosing the right strategy is an important decision, weighing the options in terms of the impact of change on the system, and on the end-users.
- In this post, we are going to talk about the following strategies:
- **Recreate**: Version A is terminated then version B is rolled out.
- **Ramped** (also known as rolling-update or incremental): Version B is slowly rolled out and replacing version A.
- **Blue/Green**: Version B is released alongside version A, then the traffic is switched to version B.
- **Canary**: Version B is released to a subset of users, then proceed to a full rollout.
- **A/B testing**: Version B is released to a subset of users under specific condition.
- **Shadow**: Version B receives real-world traffic alongside version A and doesn't impact the response.

- **Recreate**
- The recreate strategy is a dummy deployment which consists of shutting down version A then deploying version B after version A is turned off. This technique implies downtime of the service that depends on both shutdown and boot duration of the application.
- **Pros:**
  - Easy to setup.
  - Application state entirely renewed.
- **Cons:**
  - High impact on the user, expect downtime that depends on both shutdown and boot duration of the application.

- **Ramped**
- The ramped deployment strategy consists of slowly rolling out a version of an application by replacing instances one after the other until all the instances are rolled out. It usually follows the following process: with a pool of version A behind a load balancer, one instance of version B is deployed. When the service is ready to accept traffic, the instance is added to the pool. Then, one instance of version A is removed from the pool and shut down.
- Depending on the system taking care of the ramped deployment, you can tweak the following parameters to increase the deployment time:
- Parallelism, max batch size: Number of concurrent instances to roll out.
- Max surge: How many instances to add in addition of the current amount.
- Max unavailable: Number of unavailable instances during the rolling update procedure.
- **Pros:**
  - Easy to set up.
  - Version is slowly released across instances.
  - Convenient for stateful applications that can handle rebalancing of the data.
- **Cons:**
  - Rollout/rollback can take time.
  - Supporting multiple APIs is hard.
  - No control over traffic.

- **Blue/Green**
- A blue/green deployment (sometimes called big flip or red/black deployment) consists of maintaining the N VMs containing version A in service while provisioning N VMs of virtual machines containing version B. Once N VMs have been provisioned with version B and are ready to service requests, then client requests can be routed to version B. This is a matter of instructing the domain name server (DNS) or load balancer to change the routing of messages. This routing switch can be done in a single stroke for all requests. After a supervisory period, the N VMs provisioned with version A are removed from the system. If anything goes wrong during the supervisory period, the routing is switched back, so that the requests go to the VMs running version A again.
- **Pros:**
  - Instant rollout/rollback.
  - Avoid versioning issue, the entire application state is changed in one go.
- **Cons:**
  - Expensive as it requires double the resources.
  - Proper test of the entire platform should be done before releasing to production.
  - Handling stateful applications can be hard.

- **Canary**
- A canary deployment consists of gradually shifting production traffic from version A to version B. Usually the traffic is split based on weight. For example, 90 percent of the requests go to version A, 10 percent go to version B.
- This technique is mostly used when the tests are lacking or not reliable or if there is little confidence about the stability of the new release on the platform.
- **Pros:**
  - Version released for a subset of users.
  - Convenient for error rate and performance monitoring.
  - Fast rollback.
- **Con:**
  - Slow rollout.

A/B testing

- A/B testing deployments consists of routing a subset of users to a new functionality under specific conditions. It is usually a technique for making business decisions based on statistics, rather than a deployment strategy. However, it is related and can be implemented by adding extra functionality to a canary deployment so we will briefly discuss it here.
- This technique is widely used to test conversion of a given feature and only roll-out the version that converts the most.
- Here is a list of conditions that can be used to distribute traffic amongst the versions:
  - By browser cookie
  - Query parameters
  - Geolocalisation
  - Technology support: browser version, screen size, operating system, etc.
  - Language
-

Pros:

- Several versions run in parallel.
- Full control over the traffic distribution.

Cons:

- Requires intelligent load balancer.
- Hard to troubleshoot errors for a given session, distributed tracing becomes mandatory.

- **Shadow**
- A shadow deployment consists of releasing version B alongside version A, fork version A's incoming requests and send them to version B as well without impacting production traffic. This is particularly useful to test production load on a new feature. A rollout of the application is triggered when stability and performance meet the requirements.
- This technique is fairly complex to setup and needs special requirements, especially with egress traffic. For example, given a shopping cart platform, if you want to shadow test the payment service you can end-up having customers paying twice for their order. In this case, you can solve it by creating a mocking service that replicates the response from the provider.
- **Pros:**
  - Performance testing of the application with production traffic.
  - No impact on the user.
  - No rollout until the stability and performance of the application meet the requirements.
- **Cons:**
  - Expensive as it requires double the resources.
  - Not a true user test and can be misleading.
  - Complex to setup.
  - Requires mocking service for certain cases.

Strategy	ZERO DOWNTIME	REAL TRAFFIC TESTING	TARGETED USERS	CLOUD COST	ROLLBACK DURATION	NEGATIVE IMPACT ON USER	COMPLEXITY OF SETUP
RECREATE version A is terminated then version B is rolled out	✗	✗	✗	■□□	■■■	■■■	□□□
RAMPED version B is slowly rolled out and replacing version A	✓	✗	✗	■□□	■■■	■□□	■□□
BLUE/GREEN version B is released alongside version A, then the traffic is switched to version B	✓	✗	✗	■■■	□□□	■■□	■■□
CANARY version B is released to a subset of users, then proceed to a full rollout	✓	✓	✗	■□□	■□□	■□□	■■□
A/B TESTING version B is released to a subset of users under specific condition	✓	✓	✓	■□□	■□□	■□□	■■■
SHADOW version B receives real world traffic alongside version A and doesn't impact the response	✓	✓	✗	■■■	□□□	□□□	■■■

Continuous Integration

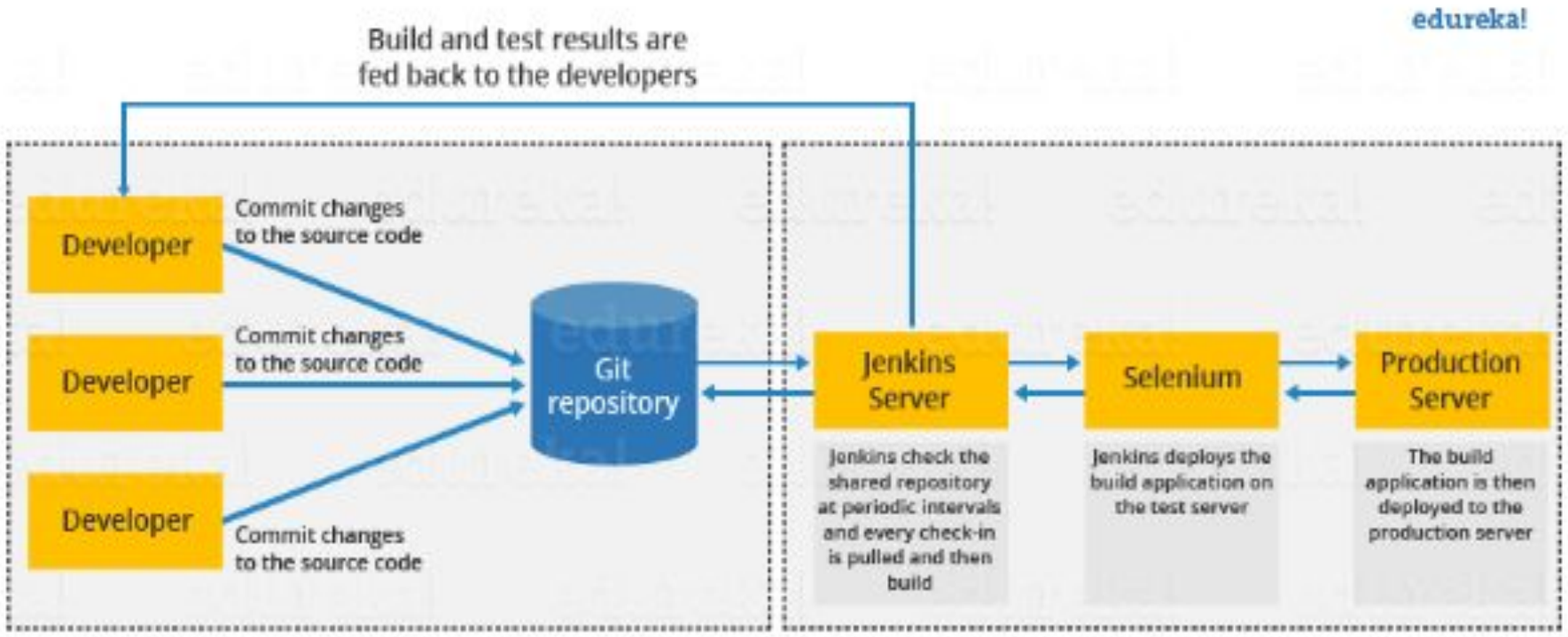
Continuous Integration

- Continuous Integration *(CI)* is a development practice in which the developers are needs to commit changes to the source code in a shared repository at regular intervals. Every commit made in the repository is then built. This allows the development teams to detect the problems early.
- Continuous integration requires the developers to have regular builds. The general practice is that whenever a code commit occurs, a build should be triggered.
- **What is Jenkins?**
- Jenkins is an open source automation tool written in Java programming language that allows continuous integration.
- Jenkins **builds** and **tests** our software projects which continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.
- It also allows us to continuously **deliver** our software by integrating with a large number of testing and deployment technologies.

Continuous Integration with Jenkins

- Let's consider a scenario where the complete source code of the application was built and then deployed on test server for testing. It sounds like a perfect way to *develop software*, but this process has many problems.
  - Developer teams have to wait till the complete software is developed for the test results.
  - There is a high prospect that the test results might show multiple bugs. It was tough for developers to locate those bugs because they have to check the entire source code of the application.
  - It slows the software delivery process.
  - Continuous feedback pertaining to things like architectural or coding issues, build failures, test status and file release uploads was missing due to which the quality of software can go down.
  - The whole process was manual which increases the threat of frequent failure.

- It is obvious from the above stated problems that not only the software delivery process became slow but the quality of software also went down. This leads to customer dissatisfaction.
- So to overcome such problem there was a need for a system to exist where developers can continuously trigger a build and test for every change made in the source code.
- This is what Continuous Integration (CI) is all about. Jenkins is the most mature Continuous Integration tool available so let us see how Continuous Integration with Jenkins overcame the above shortcomings.
- Let's see a generic flow diagram of Continuous Integration with Jenkins:



- **Let's see how Jenkins works**. The above diagram is representing the following functions:
  - First of all, a developer commits the code to the source code repository. Meanwhile, the Jenkins checks the repository at regular intervals for changes.
  - Soon after a commit occurs, the Jenkins server finds the changes that have occurred in the source code repository. Jenkins will draw those changes and will start preparing a new build.
  - If the build fails, then the concerned team will be notified.
  - If built is successful, then Jenkins server deploys the built in the test server.
  - After testing, Jenkins server generates a feedback and then notifies the developers about the build and test results.
  - It will continue to verify the source code repository for changes made in the source code and the whole process keeps on repeating.

Advantages and Disadvantages of using Jenkins

Advantages of Jenkins

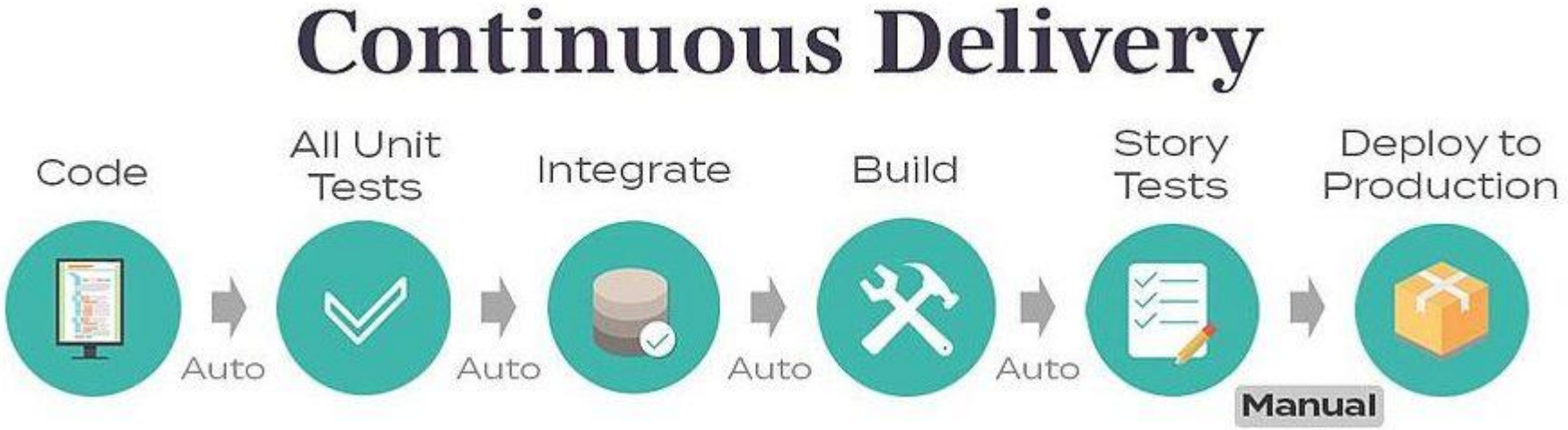
- It is an open source tool.
- It is free of cost.
- It does not require additional installations or components. Means it is easy to install.
- Easily configurable.
- It supports 1000 or more plugins to ease your work. If a plugin does not exist, you can write the script for it and share with community.
- It is built in java and hence it is portable.
- It is platform independent. It is available for all platforms and different operating systems. Like OS X, Windows or Linux.
- Easy support, since it open source and widely used.
- Jenkins also supports cloud based architecture so that we can deploy Jenkins in cloud based platforms.

Disadvantages of Jenkins

- Its interface is out dated and not user friendly compared to current user interface trends.
- Not easy to maintain it because it runs on a server and requires some skills as server administrator to monitor its activity.
- CI regularly breaks due to some small setting changes. CI will be paused and therefore requires some developer's team attention.

Continuous Delivery

- Continuous delivery (CD) is a software development methodology in which code modifications are automatically packaged and deployed to production. It aims to accelerate development, cut expenses, and lower risks without losing code quality.
- CD is attained by designing a simple release procedure that is easily reproducible and restricts manual activities. In an ideal CD procedure, only application deployment into production requires human participation.

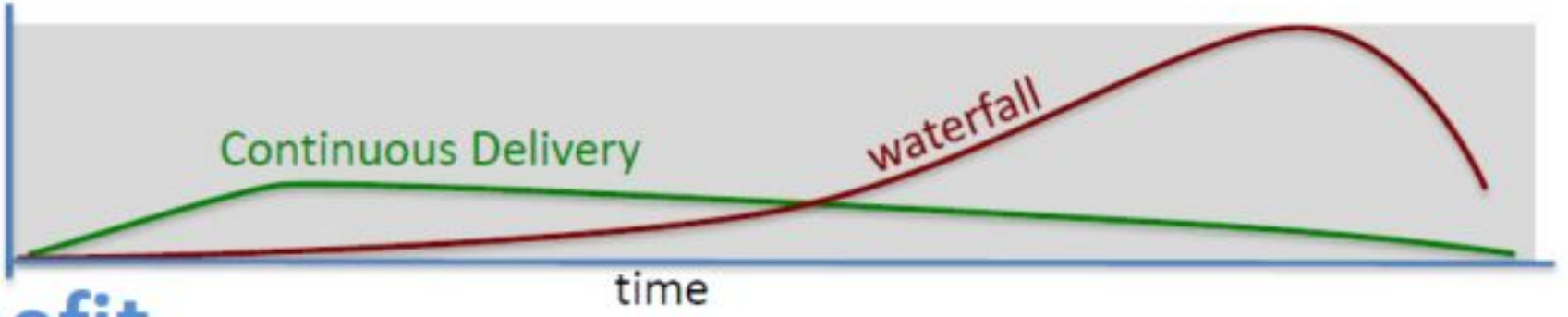


- Once a development team has accomplished continuous integration, CD is the next step in software pipeline automation (CI). CI automates the merging and testing of code modifications, focusing particularly on unit testing.
- The application is deployed to a staging environment for additional testing once the code has passed evaluations.
- These assessments consist of integration testing, performance testing, user interface testing, and more.
- CI and CD constitute the CI/CD pipeline, which transfers code from the computers of individual developers via automated testing to a production-ready build.
- At this point, all that is required is for a team member to deploy the latest version manually, often at regular intervals.
- With the emphasis on automation and velocity, CI/CD is a pillar of the DevOps concept.

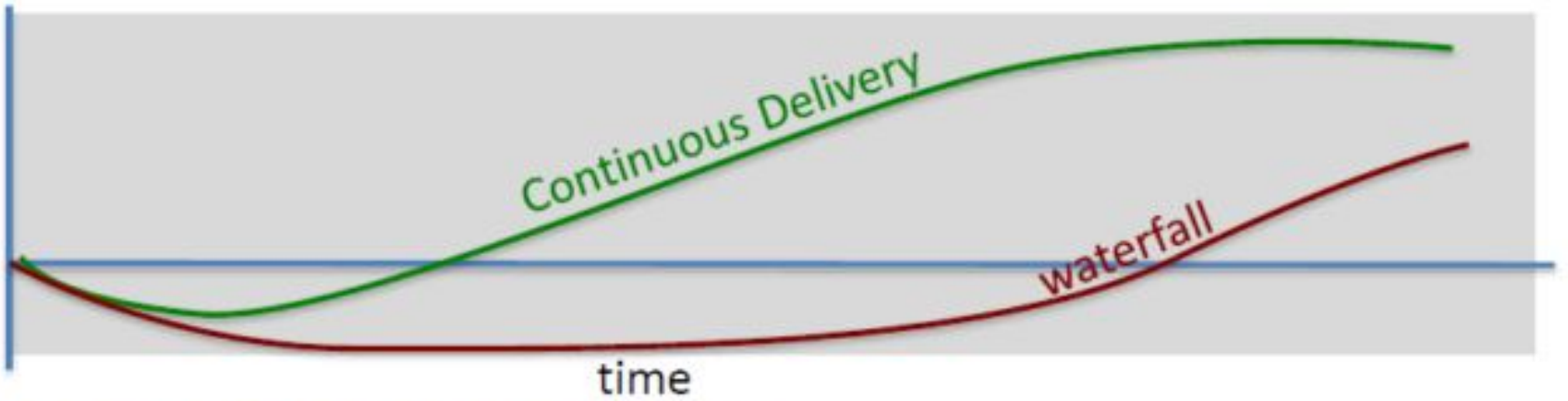
Why is it important?

- In a typical waterfall lifecycle, a customer release (minor or major) is a significant milestone that requires months of work. The entire crew is focused on achieving this objective.
- The release's features are developed, tested, and merged into the main branch as a single unit. There is a huge cost associated with failed releases and intense pressure to achieve deadlines.
- Consumers also wait lengthy durations for resolutions to their problems. Significant downtime occurs when the release patch is applied.
- Contrast this with the Agile model's shorter release lifecycle. Here, features can be checked into the central repository every day (Continuous Integration).
- Automated suites of unit, regression, and system tests ensure that freshly contributed code is quickly validated. Effect on relevant characteristics is examined. If this code can then be deployed promptly to customers, the strain surrounding a customer release is eliminated.
- Releasing becomes a common, low-risk, and automated occurrence. Manual errors are eliminated from the releasing procedure.
- Even during holidays, urgent fixes and updates can be issued with minimum manpower. This led to the evolution of Continuous Delivery.

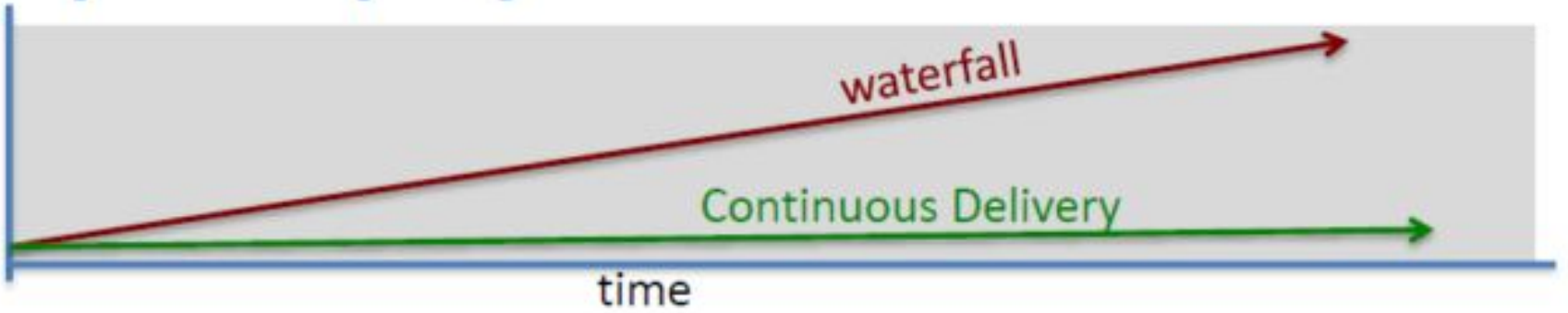
Bugs



Benefit

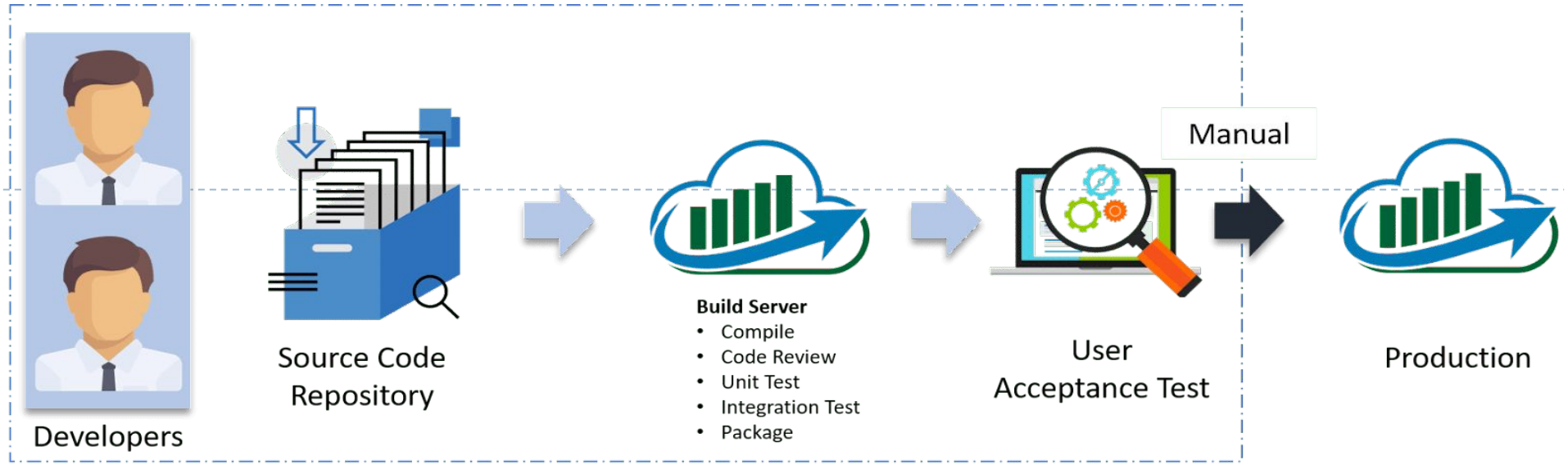


Cost per Deployment



How to Build a Continuous Delivery Pipeline

- Continuous delivery pipelines extend the operations and technologies currently implemented for a CI pipeline.
- Throughout the CI process, the code has been compiled to construct the application and unit-tested to ensure its functionality and quality.
- If the application fails to build or the unit tests fail, the code is returned to the developers to remediate and test again.
- Now, the application is prepared for additional testing. You will need a method to generate a staging environment identical to the production setting.
- Development teams generally resort to cloud services to provide a multistage environment to host the application and organise the testing workflow, since cloud hosting can expand to match processing demands.



- A CD pipeline may contain quality gates that establish success criteria. Before the program may advance to the subsequent phase, the performance, integration, and user interface (UI) tests must achieve these criteria. AI may be effective for identifying failure causes and viable solutions.
- You should automate as many tests and processes as possible as part of this testing strategy. This reduces the risk of human mistakes associated with manual methods, such as executing tests in a different order, as well as enhances speed. Everything related to Continuous Delivery in DevOps should be uniform and repeatable.
- Lastly, invest in observation and monitoring systems, as a failure in one segment of the pipeline could cause the entire system to fail. Including automated alarms and redundancies will ensure that, for instance, the failure of a single testing tool does not affect client delivery.

Benefits of Continuous Delivery in DevOps

1. Quicker detection of defects

- Continuous Delivery in DevOps is based on a robust testing technique, automatically testing an application against expected behaviour after deployment in the "real world." This allows developers to uncover flaws before pushing the code to production, where they may cause user disruptions and irritation.
- Not only does Continuous Delivery in DevOps boost user satisfaction, but it also enables the development team to anticipate these flaws in future releases, whereas a fault that does not actively disrupt the program may never be found and hence never be repaired.

2. Reduction of Costs

- By removing manual processes, Continuous Delivery in DevOps reduces the cost of delivering new software and upgrades, allowing developers to spend more time on higher-order tasks.
- Also, the speed of a CI/CD pipeline allows for the quicker delivery of additional features. This boosts the development team's output and frees up bandwidth to explore more projects without the need to hire additional engineers.

- 3. Improve quality.
- Continuous Delivery in DevOps is used to standardise an application's requirements by embodying them in test cases, hence increasing the likelihood that the result will meet users' needs.
- Continuous Delivery in DevOps also enables development teams to offer a minimally viable product (MVP) more quickly, allowing the customer to provide immediate feedback on improvement areas. Developers want feedback in order to continue offering consumer value.
- Generally speaking, an agile approach is superior to a waterfall process in which the customer does not see the final product until its completion. All the effort spent designing the final product is now a sunk cost if the customer decides after delivery that it does not match their requirements.

- 4. Reduce Risk
- The primary objective of every software deployment should be "do no harm." The second objective is to provide value to the client, but they cannot appreciate an improved UI, for instance, if the program is unavailable due to the upgrade.
- By standardising the release process and implementing test validations to discover defects before they are published into production, Continuous Delivery (CD) reduces the risk associated with each deployment and instils greater confidence in the application among developers.

- 5. Increase employee satisfaction.
- It is hardly a secret that people prefer cognitive jobs to manual, repetitive ones. By establishing an automated CD workflow, a business removes pain points for its developers and frees them to focus on strategy and optimization.
- In addition, the speed of CD pipelines expedites the deployment of engineers' code, allowing them to observe the impact of their work and how it assists customers in achieving their objectives.
- 6. Hasten Product Delivery
- CD eliminates obstacles in the development process so that updates can be deployed as soon as they have been validated. Its efficiency enables the engineering team to put out new features quickly to meet consumer demands.
- When an important issue arises, this speed advantage pays greater returns since it enables developers to deploy security updates and other solutions quickly.

End of Unit IV