**1.What is required DevOps Behaviours?**

The Behaviour of people shows the culture of an organization. Behaviour in terms of Make it observable, measurable. Making it small and tangible, will enable the teams to visibly grow towards the new behaviours in incremental steps.

**Examples of DevOps behaviours include:**

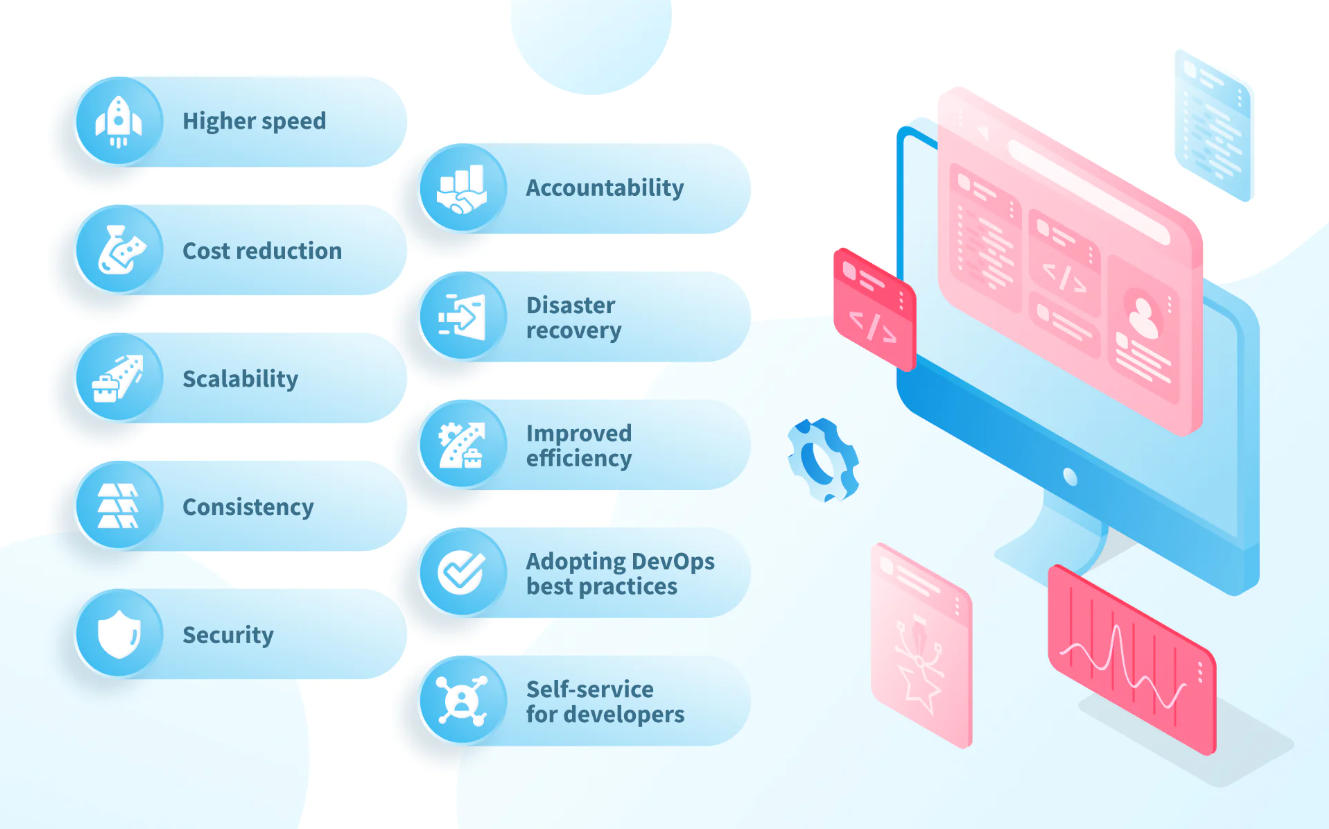
* **Multidisciplinary**: “Team members actively pick up each other’s tasks when they see a team member is too busy or unavailable”
* **Transparency**: “The team publishes real-time progress reports on the floor monitors”
* **Ownership**: “The Product Owner drives the demos and adequately informs all his/her stakeholders regarding progress and deliverables”
* **Prioritization**: “The PO and TMs jointly discuss at least once per sprint the prioritization of the work on both functional and non-functional work items”
* **Improve**: “Every severe incident is followed by a blameless post mortem, where everyone involved shares ideas & opportunities for improvement”
* **Fail safe**: “All managers openly reward the identification, prevention and resolution of errors.”

**2. What is Infrastructure as Code and What are its Benefits?**

Infrastructure as a Code **(IaC)** is a type of infrastructure management that utilizes automated processes and coding as opposed to manual inputs. Unlike basic solutions that automate some mundane IT processes, Infrastructure as a Code can handle more complex and convoluted steps. It allows developers to create instructions for storage and network requirements as well as edit configurations while preserving the current state of the infrastructure. As a result, businesses can benefit from improved process automation and recovery plans by integrating their infrastructure into version control.

**IaC** tools operate using either declarative or imperative approaches. An imperative model allows developers to specify the steps that the system needs to take to make a certain change, and the machine will follow each step. A declarative approach takes an opposite route and enables users to define the end requirement without elaborating on the steps, and the system figures out which step to take to achieve the set goal. Most **IaC** tools opt for the declarative approach due to its higher flexibility and automation.

The traditional IT infrastructure management processes are manual, which inevitably leads to human errors and higher costs. Infrastructure managers manually configure the servers to meet the requirements of the operating system and applications before deploying the app. The first downside of the traditional approach is costs and resources, as a company needs to hire professionals who can perform these tedious yet complex tasks.  manual configuration is much slower than automated processes, which leads to delays and later to unavailability of the application. Organizations lose much-needed scalability, especially during peaks, and risk keeping their apps unavailable for longer periods of time. On top of that, whenever you experience an issue pertaining to the infrastructure, it becomes extremely difficult to identify the cause. Without continuous monitoring and reporting that **IaC** tools deliver, professionals might spend a lot of time trying to find a root cause. This will eventually lead to churn and poor reputation among users.



**3. What is Continuous Integration? (Get a Good Conceptual Understanding of it)**

(CI) is a practice that integrates Code in a shared repository In software engineering, continuous integration is the practice of merging all developers' working copies to a shared mainline several times a day. This process makes the team more Agile, productive and confident.

Jenkins is an open source and automation tool used to built and test Software Projects and Jenkins turns out to be a perfect fit for building a CI/CD pipeline because of its flexibility, openness, plug-in capabilities, and simple to use nature.

**4. What is Continuous Delivery?**

Continuous delivery (CD) is the process of automating build, test, configuration, and deployment from a build to a production environment. A release pipeline can create multiple testing or staging environments to automate infrastructure creation and deploy new builds.

**5. What is the Difference Between DevOps and Site Reliability Engineering?**

DevOps is an approach to managing software development processes that collaborates between operations teams and developers. SRE focuses on the system engineer position in core infrastructure and is more appropriate in a production setting.

**6. What is the Organizational Impact of DevOps?**

The impact of Devops in organizations is development and operations teams work outside silos, releasing products faster, thereby increasing operational efficiency.

**7. What is Cloud Computing?**

Cloud computing is the on-demand availability of computer system resources, especially data storage and computing power, without direct active management by the user.

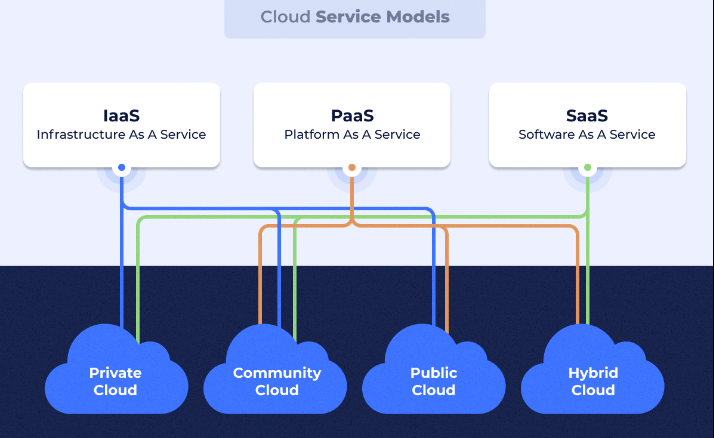
**8. What are the Essential Characteristics of Cloud Computing?**

Cloud model is composed of five essential characteristics:

1. On-demand self-service
2. Broad network access
3. Resource pooling
4. Rapid elasticity
5. Measured Service

**9. What are Cloud Deployment Models?**

There are four cloud deployment models: public, private, community, and hybrid. Each deployment model is defined according to where the infrastructure for the environment is located. There are three main cloud service models: Software as a Service, Platform as a Service, and Infrastructure as a Service.



**10. Cloud Service Models and Their Benefits.**

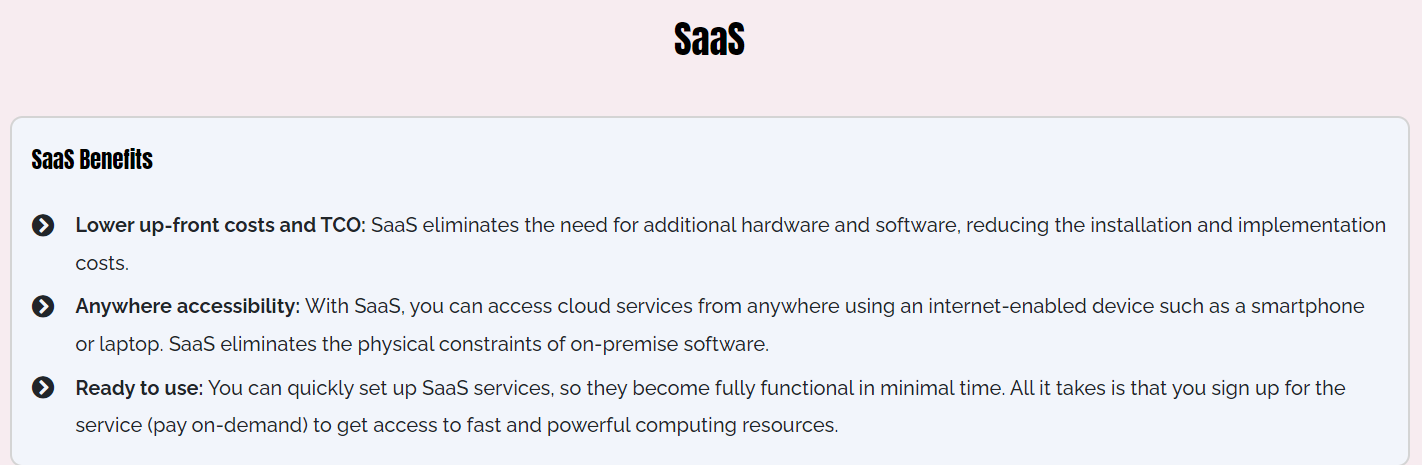
**IaaS** is the cloud service model that offers a higher level of control compared with an on-premises solution. **PaaS** makes the process of developing and deploying applications simpler and more cost-effective. **SaaS** makes collaboration among teams and access to software applications easier.

Application

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence



**Article on CI/CD**

**CI/CD defined.**

Continuous integration is a coding philosophy and set of practices that drive development teams to frequently implement small code changes and check them in to a version control repository. Most modern applications require developing code using a variety of platforms and tools, so teams need a consistent mechanism to integrate and validate changes. Continuous integration establishes an automated way to build, package, and test their applications. Having a consistent integration process encourages developers to commit code changes more frequently, which leads to better collaboration and code quality.

**[ Also on InfoWorld: 5 best practices for securing CI/CD pipelines ]**

*Continuous delivery* picks up where continuous integration ends, and automates application delivery to selected environments, including production, development, and testing environments. Continuous delivery is an automated way to push code changes to these environments.

**Automating the CI/CD pipeline**

CI/CD tools help store the environment-specific parameters that must be packaged with each delivery. CI/CD automation then makes any necessary service calls to web servers, databases, and other services that need restarting. It can also execute other procedures following deployment.

Because the objective is to deliver quality code and applications, CI/CD also requires continuous testing. In continuous testing, a set of automated regression, performance, and other tests are executed in the CI/CD pipeline.

A mature devops team with a robust CI/CD pipeline can also implement *continuous deployment*, where application changes run through the CI/CD pipeline and passing builds are deployed directly to the production environment. Some teams practicing continuous deployment elect to deploy daily or even hourly to production, though continuous deployment isn’t optimal for every business application.

Nominations are open for the 2024 Best Places to Work in IT

Organizations that implement a CI/CD pipeline often have several devops best practices in place, including microservices development, serverless architecture, continuous testing, infrastructure as code, and deployment containers. Each of these practices improves process automation and increases the robustness of cloud computing environments. Together, these practices provide a strong foundation to support continuous deployment.

**How continuous integration improves collaboration and code quality.**

Continuous integration is a development philosophy backed by process mechanics and automation. When practicing continuous integration, developers commit their code into the version control repository frequently; most teams have a standard of committing code at least daily. The rationale is that it’s easier to identify defects and other software quality issues on smaller code differentials than on larger ones developed over an extensive period. In addition, when developers work on shorter commit cycles, it is less likely that multiple developers will edit the same code and require a merge when committing.

Teams implementing continuous integration often start with the version control configuration and practice definitions. Although checking in code is done frequently, agile teams develop features and fixes on shorter and longer timeframes. Development teams practicing continuous integration use different techniques to control what features and code are ready for production.

Many teams use feature flags, a configuration mechanism to turn features and code on or off at runtime. Features that are still under development are wrapped with feature flags in the code, deployed with the main branch to production, and turned off until they are ready to be used. In recent research, devops teams using feature flags had a ninefold increase in development frequency. Feature flagging tools such as CloudBees, Optimizely Rollouts, and LaunchDarkly integrate with CI/CD tools to support feature-level configurations.

**Automated builds**

In an automated build process, all the software, database, and other components are packaged together. For example, if you were developing a Java application, continuous integration would package all the static web server files such as HTML, CSS, and JavaScript along with the Java application and any database scripts.

Continuous integration not only packages all the software and database components, but the automation will also execute unit tests and other types of tests. Testing provides vital feedback to developers that their code changes didn’t break anything.

Most CI/CD tools let developers kick off builds on demand, triggered by code commits in the version control repository, or on a defined schedule. Teams need to determine the build schedule that works best for the size of the team, the number of daily commits expected, and other application considerations. A best practice is to ensure that commits and builds are fast; otherwise, these processes may impede teams trying to code quickly and commit frequently.

**Continuous testing and security automation**

Automated testing frameworks help quality assurance engineers define, execute, and automate various types of tests that can help development teams know whether a software build passes or fails. They include functionality tests developed at the end of every sprint and aggregated into a *regression test* for the entire application. The regression test informs the team whether a code change failed one or more of the tests developed across the functional areas of the application where there is test coverage.

A best practice is to enable and require developers to run all or a subset of regression tests in their local environments. This step ensures developers only commit code to version control after code changes have passed regression tests.

Regression tests are just the beginning, however. Devops teams also automate performance, API, browser, and device testing. Today, teams can also embed static code analysis and security testing in the CI/CD pipeline for shift-left testing. Agile teams can also test interactions with third-party APIs, SaaS, and other systems outside of their control using service virtualization. The key is being able to trigger these tests through the command line, a webhook, or a web service, and get a success or failure response.

Continuous testing implies that the CI/CD pipeline integrates test automation. Some unit and functionality tests will flag issues before or during the continuous integration process. Tests that require a full delivery environment, such as performance and security testing, are often integrated into continuous delivery and done after a build is delivered to its target environments.

**Stages in the continuous delivery pipeline**

Continuous delivery is the automation that pushes applications to one or more delivery environments. Development teams typically have several environments to stage application changes for testing and review. A devops engineer uses a CI/CD tool such as Jenkins, CircleCI, AWS CodeBuild, Azure DevOps, Atlassian Bamboo, Argo CD, Buddy, Drone, or Travis CI to automate the steps and provide reporting.

For example, Jenkins users define their pipelines in a Jenkinsfile that describes different stages such as build, test, and deploy. Environment variables, options, secret keys, certifications, and other parameters are declared in the file and then referenced in stages. The post section handles error conditions and notifications.

A typical continuous delivery pipeline has build, test, and deploy stages. The following activities could be included at different stages:

* Pulling code from version control and executing a build.
* Enabling stage gates for automated security, quality, and compliance checks and supporting approvals when required.
* Executing any required infrastructure steps automated as code to stand up or tear down cloud infrastructure.
* Moving code to the target computing environment.
* Managing environment variables and configuring them for the target environment.
* Pushing application components to their appropriate services, such as web servers, APIs, and database services.
* Executing any steps required to restart services or call service endpoints needed for new code pushes.
* Executing continuous tests and rollback environments if tests fail.
* Providing log data and alerts on the state of the delivery.
* Updating configuration management databases and sending alerts to IT service management workflows on completed deployments.

A more sophisticated continuous delivery pipeline might have additional steps such as synchronizing data, archiving information resources, or patching applications and libraries.

Teams using continuous deployment to deliver to production may use different cutover practices to minimize downtime and manage deployment risks. One option is configuring canary deployments with an orchestrated shift of traffic usage from the older software version to the newer one.

**CI/CD tools and plugins**

CI/CD tools typically support a marketplace of plugins. For example, Jenkins lists more than 1,800 plugins that support integration with third-party platforms, user interface, administration, source code management, and build management.

Once the development team has selected a CI/CD tool, it must ensure that all environment variables are configured outside the application. CI/CD tools allow development teams to set these variables, mask variables such as passwords and account keys, and configure them at the time of deployment for the target environment.

Continuous delivery tools also provide dashboard and reporting functions, which are enhanced when devops teams implement observable CI/CD pipelines. Developers are alerted if a build or delivery fails. The dashboard and reporting functions integrate with version control and agile tools to help developers determine what code changes and user stories made up the build.

**Measuring CI/CD success with devops KPIs**

The impact of implementing CI/CD pipelines can be measured as a devops key performance indicator (KPI). Indicators such as deployment frequency, change lead time, and incident meantime to recovery (MTTR) are often improved by implementing CI/CD with continuous testing. However, CI/CD is just one process that can drive these improvements, and there are [other prerequisites](https://blogs.starcio.com/2019/10/prerequisites-deployment-frequencies-devops.html) to improving deployment frequencies.

**CI/CD with Kubernetes and serverless architectures**

Many teams operating CI/CD pipelines in cloud environments also use containers such as Docker and orchestration systems such as Kubernetes. Containers allow for packaging and shipping applications in a standard, portable way. Containers make it easy to scale up or tear down environments with variable workloads.

There are many approaches to using containers, infrastructure as code (IaC), and CI/CD pipelines together. Free tutorials such as Kubernetes with Jenkins or Kubernetes with Azure DevOps can help you explore your options.

Another option is to use a serverless architecture to deploy and scale your applications. In a serverless environment, the cloud service provider manages the infrastructure, and the application consumes resources as needed based on its configuration. On AWS, for example, serverless applications run as Lambda functions and deployments can be integrated into a Jenkins CI/CD pipeline with a plugin. Azure serverless and GPS serverless computing are similar services.