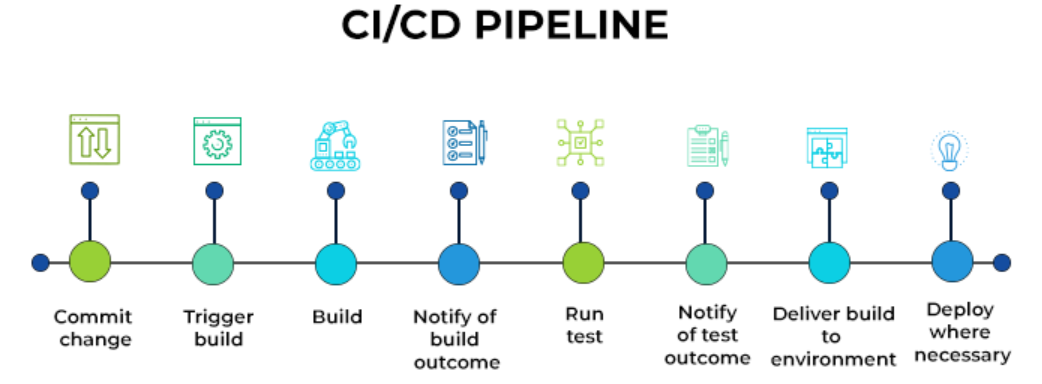
**CI/CD Pipeline for Monitoring Cloud Applications**

- Richa Vivek Savant

BL.EN.U4CSE21171

Introduction

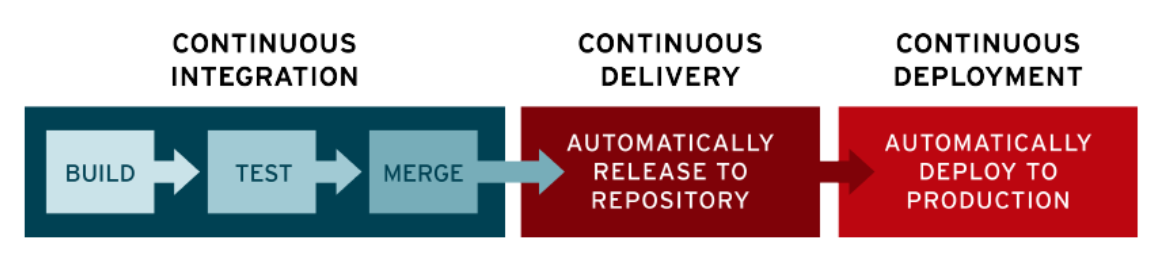
A pipeline for continuous integration and continuous deployment, or CI/CD, is a set of actions needed to provide a new software version. The goal of continuous integration and delivery (CI/CD) pipelines is to automate processes to improve software delivery across the software development life cycle.   
Organizations can produce higher-quality code more quickly and securely by automating continuous integration and continuous delivery (CI/CD) throughout the software development lifecycle's development, testing, production, and monitoring phases. The actual benefit of CI/CD pipelines is achieved by automation, even though each step of the pipeline can be carried out manually.



Continuous integration (CI), which optimizes code release cycles through automation capabilities, verifies every step of the development process, from merging codes to testing builds. Consequently, this reduces the likelihood of prolonged feature development cycles and associated problems such as merge disputes.   
The goal of continuous deployment (CD) is to quickly install a packaged artifact into a production environment. The entire distribution process—including deployment—is automated by it.   
In 2020, CI/CD workflows were utilized by 62% of teams who were in an advanced stage of their DevOps evolution process. The advantages and range that these procedures provide are the reasons why this number keeps going up.

The goal of automating the process is to reduce the possibility of human error and preserve a standardized procedure for software releases. The pipeline may contain tools for code compilation, unit testing, code analysis, security, and the generation of binaries. This pipeline would additionally include packaging the code as a container image for hybrid cloud deployment in containerized environments.

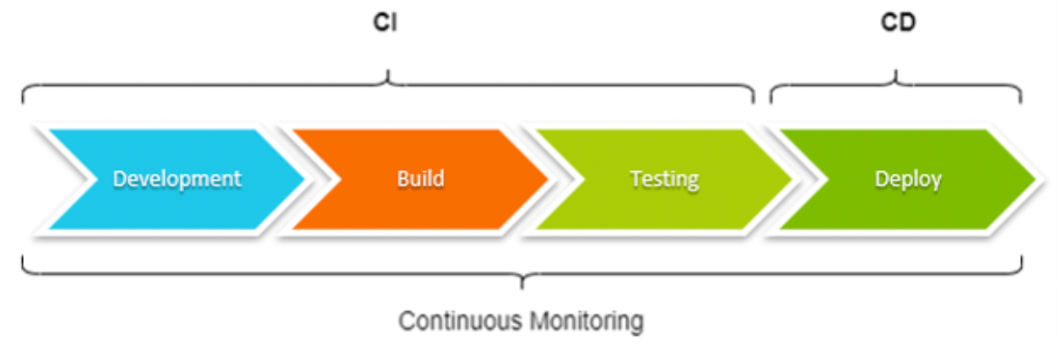
A DevOps methodology's foundation is CI/CD, which unites IT operations and development teams to deploy software. The speed at which code may be released has turned into a competitive differentiator as custom applications become essential to how businesses differentiate themselves.



Continuous Integration

Continuous Integration is the first step in the CI/CD process (CI). Developers can work independently with Continuous Integration (CI), starting their own code "branch" to apply minor modifications. The source code can be snapshotted by the developer while they are working, usually using a versioning tool like Git. Git makes it simple to roll back the codebase to a previous version in the event of an issue, leaving the developer free to focus on new features.  
Individuals' labor is then sent into an automated system that builds and tests the code modifications using scripts. A CI server assembles the source code modifications into the master code, or "trunk," following the build phase.

Teams can submit code changes more frequently using the CI/CD development process than they could with the traditional method of writing code independently and sending it to the master once a month, which can result in tedious work addressing bugs and inadequate version control. Continuous testing guarantees functionality, expedites bug repairs, and improves teamwork and software quality in the end.

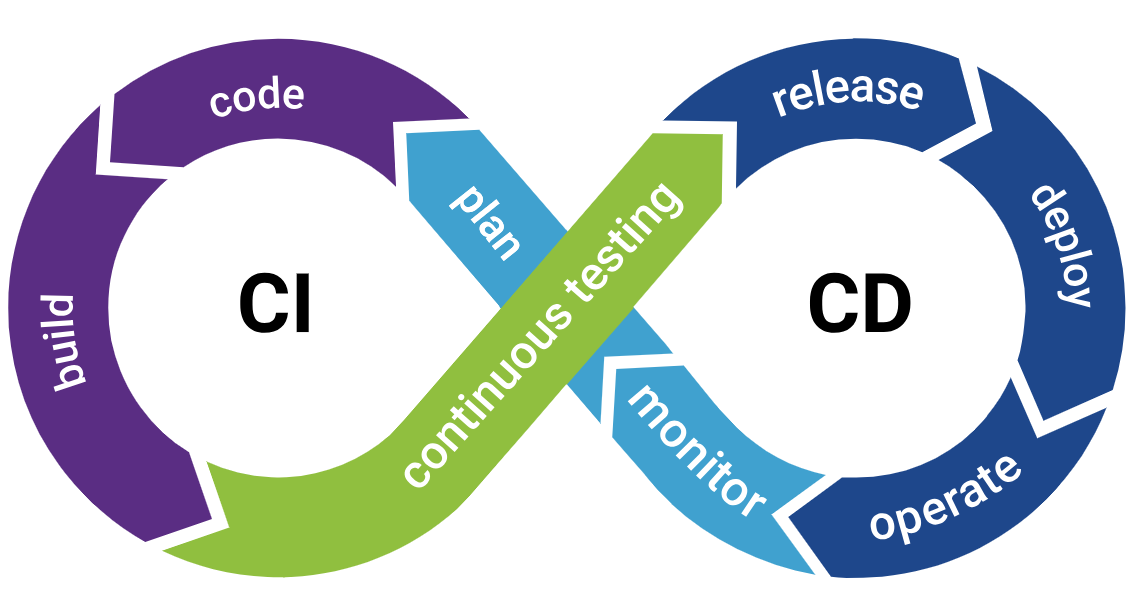


Continuous Delivery/Deployment

Continuous delivery (CD) is the process of deploying verified code modifications from continuous integration into specific code repositories or environments, like GitHub. They can now be implemented in a live production environment by the operations team. Errors are fixed automatically after the software and APIs are tested. The DevOps team manually sends the most recent build to the deploy stage after receiving notification about it in the final phase of the CD process.   
The stage of the continuous delivery pipeline aims to release new code as quickly as possible while retaining some degree of human control.

Continuous deployment is what the CD in the CI/CD method also stands for. After passing a set of specified tests, like integration tests that test code in a mimic environment to ensure code integrity, continuous deployment automatically distributes code changes to end users.   
The degree of automation employed in software or app releases is what separates continuous delivery from continuous deployment. When code is delivered continuously, it automatically migrates to environments similar to production settings for additional testing and quality control. Only when tests are successful can code be manually moved into production. Automation takes one step further in continuous deployment. There is no need for human approval for the code to be deployed to production once testing is complete.

The business demands of an organization determine how to implement the CI/CD pipeline and whether to employ deployment or continuous delivery. For DevOps teams that have a quick development lifecycle, such those creating SaaS systems and e-commerce websites, continuous deployment works best. Teams can release new or updated software as frequently and fast as feasible using continuous deployment. Only DevOps teams with a tested procedure usually employ this kind of continuous deployment pipeline, as modifications are automatically released to the public.   
Continuous delivery is usually the best choice for teams who would not need to deploy changes as frequently in their workflow, such those developing healthcare applications. It is slower but offers another layer of oversight to ensure functionality for the end-users.



Phases of the CI/CD Pipeline

From source code to production, these phases make up the development lifecycle and workflow of the CI/CD pipeline:

1. Build

Code creation and compilation are the tasks of this phase of the continuous integration process. Teams work together to integrate new code and improve upon existing code while promptly resolving any problems or disagreements.   
  
After obtaining the source code from a repository and creating links to pertinent modules, libraries, and dependencies, the build process assembles (builds) each of these parts into an executable (.exe) file. In addition to generating process logs, these tools also identify mistakes that need to be looked at and fixed and alert developers when the build is finished.

1. Similar to how source code is created, the programming language that is chosen usually determines the build tools. It is possible for a development team to use third-party tools to produce a build. Since many IDEs have build capabilities built in, they can efficiently handle both the source creation and building stages of a continuous integration/continuous development pipeline.   
     
   To convert an executable file into a packaged or deployable execution environment, like a virtual machine (VM) with an operating system and related components, or a container, like a Docker container with libraries and dependencies, a build phase may use additional tooling, such as scripts.
2. Test

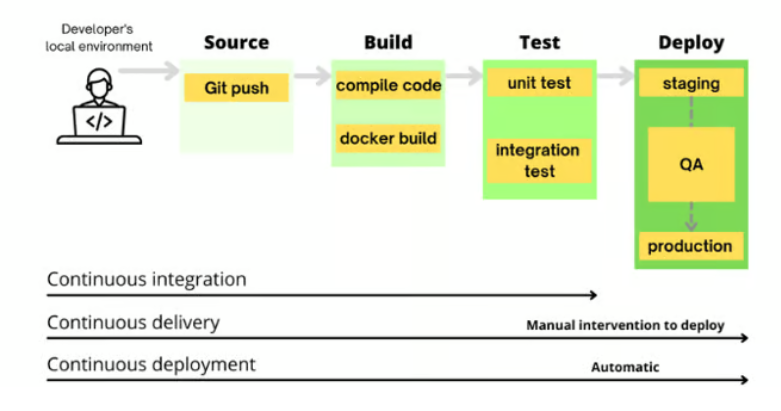
Teams test the code at this point. Both deployment and continuous delivery use automated testing. Regression, unit, and integration tests are a few examples of these tests.   
  
Although some static testing has previously been done on the source code, the finished build is now ready for the next CI/CD stage, which involves extensive dynamic testing. Basic functional or unit testing is typically the first step in this process to make sure new features and functions function as intended. Regression testing is then performed to make sure any new additions or changes don't unintentionally disrupt existing features that have already been in place. A series of tests is also performed on the build to evaluate its performance, user acceptability, and integration. If mistakes are found during testing, the findings are sent back to the developers for review and fixing in later releases.

During the CI/CD test phase, when a build is put through a vast array of tests and test cases to verify its functionality, automation is especially important. In most cases, human testing is too sluggish and prone to mistakes and oversights to guarantee accurate or unbiased test results. Although they develop thorough test cases and criteria, test specialists rely on test tools to carry out testing and validation in a busy pipeline.

1. Deliver/Deploy

Here, a production environment receives an approved codebase. Only with developer consent can this step be automated in continuous delivery, but it is automated in continuous deployment. Finally, the completed product is put into production after the modifications have been implemented. Products or code are submitted to repositories in continuous delivery, where they are subsequently approved by humans before being deployed or put into production. This stage in continuous deployment is automated.   
  
When the build successfully completes the testing stage, it is deemed ready for deployment into a live environment. It is sent to human stakeholders in a continuous delivery pipeline, who approve it before it is deployed. As soon as the build passes its test suite, it immediately deploys in a continuous deployment pipeline.

Creating a deployment environment, such as allocating resources and services within the data center, and transferring the build to its deployment target, like a server, are typical steps in the deployment process. Usually, automation tools' workflows or scripts are used to automate these processes. In order to identify unforeseen issues once the build is deployed and notify developers, deployments typically establish connections with error reporting and ticketing systems. Additionally, users can report perceived or actual issues with the release by creating bug tickets.



CI/CD Tools and Configuration

The optimization and automation of the software development process should be the primary considerations while choosing CI/CD systems. Open-source tools are used in an efficient CI/CD pipeline for integration, testing, and deployment. A successful software development pipeline is also impacted by properly configured continuous integration and delivery processes.

Jenkins is the most widely used open-source CI/CD tool. Jenkins is a Java-based automated continuous integration (CI) server that is used to automate reporting and CI/CD processes. Additional open-source integration technologies are CircleCI and Travis CI.

Platforms like GitLab aim to offer the integrated development environment (IDE) within a comprehensive platform that includes other tools. Integrated development environments (IDE), such as GitHub or AWS CodeCommit, assist developers in creating, maintaining, and tracking software packages.   
Teams that work in cloud environments utilize Kubernetes for orchestration and containers like Docker for packaging and shipping applications. Although Kubernetes is utilized in many CI/CD workflows, it isn't only for the CI/CD pipeline.

Implementing a CI/CD Pipeline

A CI/CD pipeline cannot be set up in a single method. The technique and tools used to accomplish them differ in specific sections, but this is intentional because the goal is to create a highly flexible pipeline that can adapt to the demands of the business and its projects.   
However, at every level of pipeline development, there are standard actions and choices that are applicable to every CI/CD process:

1. Select a version control system to maintain code repositories. Determine if you need a hosted version, or a hosting provider. Major cloud providers also offer options here.
2. Create repositories to house application source code and pipelines.
3. Determine what build, or CI, server to use. This can be self-hosted, such as Jenkins, or a third-party option such as GitHub Actions.
4. Implement a task in the pipeline that compiles application source code into a build. In some setups, this will generate a Docker image.
5. Run basic tests on the code (static analysis, style checks) to ensure its quality and consistency with organizational guidelines.
6. The build should now generate an artifact, or container image, published to a store or registry.
7. Initiate further testing on the build, as listed previously (functional, security, user acceptance, etc.). If predetermined thresholds are not met, fail the stage. Publish results of tests and code coverage so they are easily available.
8. Once the software build passes tests, it is ready for final preparations to production deployment.

Benefits of the CI/CD Pipeline

Automation of software releases — from initial testing to the final deployment — is the most significant benefit of the CI/CD pipeline. Additional benefits of the CI/CD process for development teams include the following:

1. Reducing time to deployment through automation: Automated testing makes the development process more efficient, reducing the length of the software delivery process. In addition, continuous deployment and automated provisioning allow a developer’s changes to a cloud application to go live within minutes of writing them.
2. Decreasing the costs associated with traditional software development: Fast development, testing and production (facilitated by automation) means less time spent in development and, therefore, less cost.
3. Continuous feedback for improvement: The CI/CD pipeline is a continuous cycle of build, test and deploy. Every time code is tested, developers can quickly take action on the feedback and improve the code.
4. Improving the ability to address error detection earlier in the development process: In continuous integration, testing is automated for each version of code built to look for issues integration. These issues are easier to fix the earlier in the pipeline that they occur.
5. Improving team collaboration and system integration: Everyone on the team can change code, respond to feedback and quickly respond to any issues that occur.
6. Reduced risk of defects: Finding and resolving defects late in the development process is costly and time-consuming. This is particularly true when problems arise with features already released to production. One can test and deploy code more frequently using a CI/CD pipeline, giving QA engineers the power to identify and fix errors as soon as they occur.
7. Faster product delivery: With a smooth CI/CD workflow, multiple daily releases can become a reality. Teams can automatically build, test, and deliver features with minimal manual intervention. Docker, Kubernetes, and Travis CI are some of the tools and frameworks that can be used to accomplish this.
8. Log generation: A way to track the system’s performance over time is pivotal to determine essential performance indicators. Observability is a technical tool that aids in this endeavour. Logging information plays a vital role in observability. Logs provide a large volume of information to decipher what’s happening beneath the UI and study program behaviour. A CI/CD pipeline generates a lot of logging data at every level of the software development process.
9. Quick rollback if required: One of the most exclusive benefits of a CI/CD pipeline is that it leads to the quick and easy rollback of code changes if there are any issues in the production environment after a release. If any new code change breaks a feature or general application, one can revert to its previous stable version right away and deploy the most recent successful build instantly to avoid production interruptions.
10. Efficient testing & monitoring: Testing entails automating each test case and experimenting with the program. Any cycle that needs to be repeated over time should be automated, and there are enough innovations available to achieve this goal. Using continuous monitoring, Ops teams can oversee and ensure that the application is running as expected and that the environment is stable. They must ensure that the applications perform optimally.
11. Cost-effectiveness: The CI/CD pipeline takes a different approach to software delivery. With automated testing hooks at every stage, developers can fix issues early and avoid critical issues in the production environment. With the implementation of a CI/CD pipeline, code quality improves drastically, which, in turn, improves the overall ROI.

Disadvantages of a CI/CD Pipeline

CI/CD implementation encounters some specific challenges, too. Here are two major ones:

1. Limited testing: A major issue with CI/CD pipelines is allocating and coordinating resources as well as intellectual investment to build test environments and create test cases. Multiple code contributions and simultaneous testing requirements are part of continuous development, which usually leads to configuration conflicts and few or forgotten test cases. This may cause errors to bypass the testing stage and reduce the effectiveness of the pipeline. Well-trained and informed software testers, along with thoroughly specified requirements and objectives, are indispensable.
2. Fixing bugs: The purpose of pipelines is to give developers feedback loops so they may address errors in a subsequent build. It's not too hard to find a bug, but it can be challenging to pinpoint the exact developer who has to fix that particular code segment. This can obscure the need for more training and make it more difficult to hold developers accountable for their work. Finding the bug's location and identifying the developers who will be working on fixing it can be aided by logging, team communication, and extensive documentation.

Importance of CI/CD Performance Monitoring

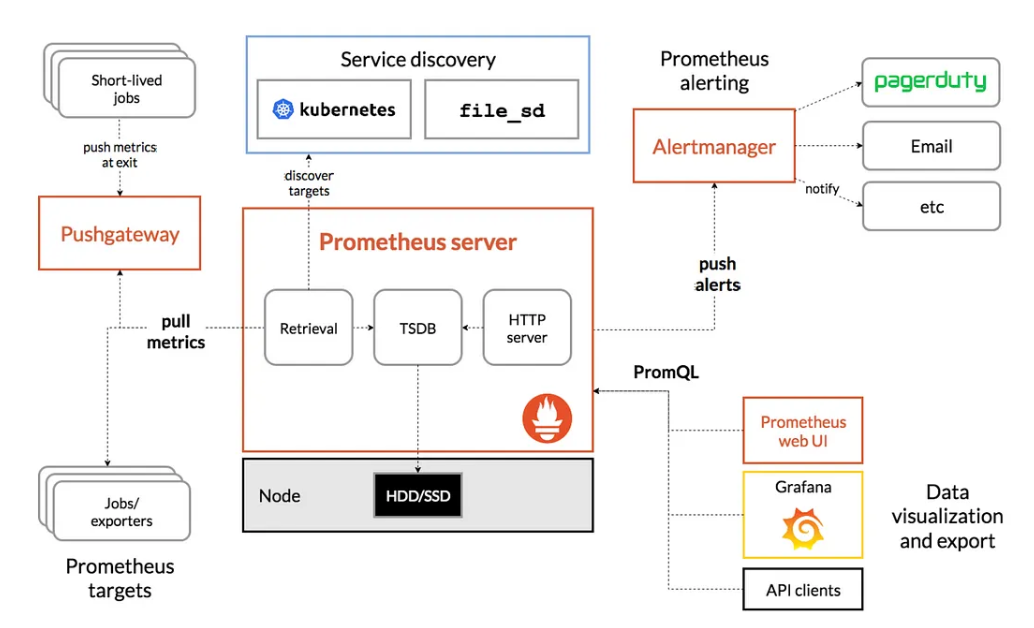
To guarantee the dependability, scalability, and effectiveness of software delivery procedures, applications running in a Continuous Integration/Continuous Deployment (CI/CD) pipeline must be monitored for performance issues. It is critical to comprehend how applications function at different phases of development and deployment in the dynamic world of contemporary software development, where quick iterations and frequent deployments are the norm.

The CI/CD workflow can be enhanced by adding monitoring technologies to give teams real-time insights into application behavior. This allows teams to find performance bottlenecks, make the most use of their resources, and proactively fix problems before they affect end users. This proactive approach promotes a culture of creativity and continuous improvement across the development lifecycle in addition to improving the overall quality of software releases.

Monitoring Tools

Prometheus is an open-source monitoring and alerting toolkit, widely recognized for its powerful querying language and efficient time series database. It’s particularly well-suited for monitoring dynamic cloud environments like Kubernetes due to its automatic service discovery capabilities, scalable data collection, and robust alerting features.

Grafana, on the other hand, is the leading open-source software for time series analytics, providing rich visualization capabilities that make it easy to create comprehensive dashboards. When integrated with Prometheus, Grafana allows you to visualize data in diverse and complex ways, turning raw data into actionable insights.

Prometheus: 

An open-source DevOps tool is called Prometheus. It offers real-time alerting and monitoring features for Kubernetes and other container orchestration platforms. The measurements from the platform are gathered and saved as time series data. It can monitor the container orchestration platform right out of the box. For other data visualization frameworks like Grafana, it serves as a data source.   
Prometheus gathers the following metrics from the Kubernetes cluster: performance metrics, server resources, information on possible performance bottlenecks, CPU and memory status, Kubernetes cluster health.

Setting up Prometheus to monitor a Kubernetes cluster is an essential activity for maintaining the functionality and health of applications in dynamic situations. Prometheus must be adjusted in order for it to automatically find and gather metrics from a variety of targets inside the Kubernetes cluster, including pods and services. Teams can simplify the monitoring process and adjust to the always changing Kubernetes settings by modifying the Prometheus configuration.

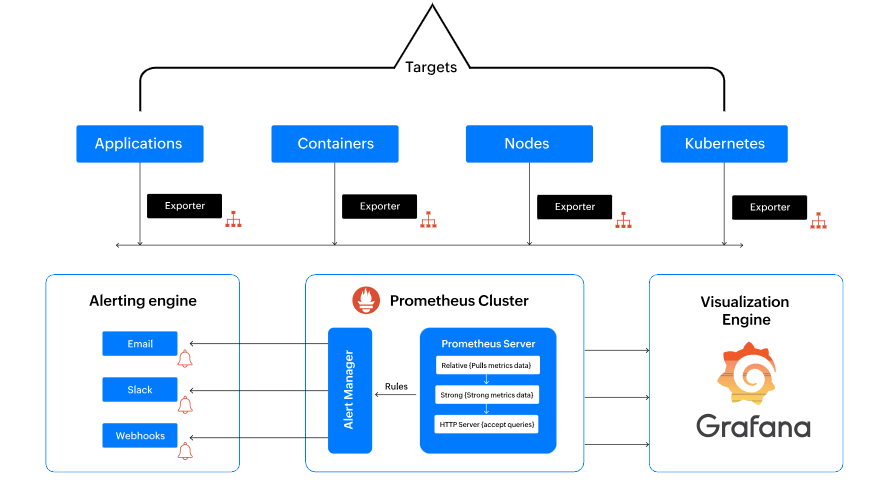
To configure Prometheus, one important thing to do is to allow service discovery in Kubernetes. As a result, Prometheus can now recognize pods and services on its own, simplifying monitoring in the midst of the steady influx of containerized apps. Teams may make sure that pertinent metrics are consistently gathered and offer insightful information about the cluster's condition by including Kubernetes service discovery into the Prometheus setup.

Prometheus's monitoring capabilities can also be further enhanced by customizing scrape settings for other services. Teams may guarantee that pertinent metrics are efficiently collected by specifying exact scrape setups for different services, such as web servers, databases, and bespoke apps. Teams can set metrics pathways and ports without getting bogged down in intricate technical details thanks to annotations included in pod or service specifications.   
In order to provide real-time notifications based on predetermined criteria, the setup also entails creating alerting rules within Prometheus. Teams can proactively resolve any issues before they affect end users by setting alerting rules that initiate notifications for particular measurement situations. By taking the initiative, this proactive strategy improves the Kubernetes cluster's dependability and resilience and promotes innovation and continuous improvement throughout the development lifecycle.

Ultimately, in order to guarantee the effectiveness of the Prometheus configuration, careful testing and verification are necessary. Teams may confirm that metrics are being gathered correctly and that alarms are firing as intended by reloading Prometheus's setup and testing scrape targets and alerts. By gaining access to the Prometheus dashboard, teams can make educated decisions and efficiently maintain the stability of their applications by learning about the health and performance of the cluster.   
Prometheus periodically gathers metrics from a variety of targets, including servers, apps, and services. These metrics usually include data on application performance, system resource use, and other pertinent operational information.

After it is gathered, Prometheus saves this data in its time-series database so that Prometheus Query Language (PromQL) can be used to query and analyze it. For monitoring and observability reasons, users can then integrate the data into bespoke dashboards or visualize it using tools such as Grafana.   
Prometheus may provide warnings based on preset rules and conditions in addition to giving data for monitoring and visualization. Prometheus has the ability to set out alerts that tell operators or automated systems about possible problems or irregularities in the monitored targets when particular thresholds are surpassed or conditions are satisfied.

Overall, the output of Prometheus consists of time-series metrics, query results, visualizations, and alerts, all of which contribute to providing insights into the health, performance, and behavior of the monitored systems and applications.



Grafana:

Grafana is an open-source, cross-platform web application for interactive visualization and analytics. It provides web-based alerts, graphs, and interactive dashboards when connected to compatible data sources such as Prometheus.   
Regardless of the data source, Grafana lets you query, visualize, and understand your metrics. It offers an intuitive user interface for tracking and evaluating metrics gathered by Prometheus, enabling proactive problem solving and well-informed decision making.

Adding Prometheus as a data source in the Grafana UI is the first step in configuring Grafana. This entails going to the Configuration area, gaining access to Data Sources, and entering the Prometheus server's URL. Through this connection, Grafana is able to access the metrics that Prometheus has gathered, which serves as the basis for creating dashboards that are educational. Furthermore, protecting important metrics data requires securing access to Grafana. Grafana can be secured against unwanted access by modifying the default admin password and putting authentication methods like OAuth or LDAP in place.

After Prometheus has been included as a data source, users can utilize dashboards that are already established or design new ones that are customized to meet their unique monitoring requirements. Grafana provides an extensive collection of community-driven dashboards that may be used for a variety of purposes, such as Kubernetes monitoring. To make sure these dashboards are linked to the Prometheus data source, users can import them into Grafana. Alternatively, users can start from scratch and build bespoke dashboards by choosing the types of visualization and setting up metric queries using the query language (PromQL) of Prometheus.

Grafana dashboards are more functional and easy to use when features and plugins are managed. Annotations, alert lists, and single statistics are just a few of the tools that users may experiment with in Grafana to add more context and insights to their dashboards. Furthermore, a multitude of plugins for sophisticated visualizations are available in Grafana's plugin marketplace, allowing users to further tailor their dashboards to meet their unique needs. The setup process can be streamlined and manual labor can be decreased by automating dashboard setting using Grafana's API or configuration management systems like Ansible or Terraform. This provides consistency and reproducibility across environments.

In conclusion, companies can construct educational and visually appealing dashboards for tracking and analyzing metrics data by integrating and maintaining Grafana with Prometheus. Businesses can obtain useful information on the functionality and condition of their systems and apps by setting Grafana to connect to Prometheus, controlling access, building or importing dashboards, and utilizing features and plugins. By empowering teams to see and resolve problems quickly, this proactive monitoring method improves the dependability and resilience of their infrastructure.

Benefits of setting up Prometheus and Grafana for Monitoring a Container Orchestration Platform

Setting up Prometheus and Grafana for monitoring gives us many benefits:

1. You get a complete end-to-end solution for observing and monitoring a Kubernetes cluster.
2. You can query the metrics using Prometheus PromQL query language.
3. If you have a microservice architecture, Prometheus keeps track of all your microservices simultaneously.
4. When a service fails, you get an immediate alert.
5. The Grafana dashboard provides performance and health reports for your clusters.

Conclusion

To sum up, the continuous integration and deployment (CI/CD) pipeline is a crucial structure in contemporary software development that expedites the workflow from code commit to deployment. Development teams benefit from its automation capabilities, which speed up the delivery of high-quality software and encourage teamwork and agility. But the effectiveness of a CI/CD pipeline depends on strong monitoring procedures, which are essential to guaranteeing the dependability and performance of deployed applications.

Prometheus and Grafana application monitoring within the CI/CD workflow provides unmatched insights into the performance and behavior of applications. Teams can monitor and analyze critical data in real-time by leveraging Grafana's user-friendly dashboards and seamlessly integrating Prometheus to dynamically collect metrics from Kubernetes environments. Teams can quickly detect and address problems thanks to this proactive monitoring method, which promotes a continuous improvement culture and makes it possible to confidently deliver dependable software to end users.