DEPLOYMENT PIPELINE

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

The introduction of automation in a deployment pipeline allowed development teams to focus more on innovating and

improving the end product for the user. By reducing the need for any manual tasks, teams are able to deploy new code updates much quicker and with less risk of any human error.

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

Main Stages of a Deployment Pipeline

There are four main stages of a deployment pipeline:

1. Version Control
2. Acceptance Tests
3. Independent Deployment
4. Production Deployment

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

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

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

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

Benefits of a Deployment Pipeline

Building a deployment pipeline into your software engineering system offers several advantages for your internal team, stakeholders, and the end user. Some of the primary benefits of an integrated deployment pipeline include:

- Teams are able to release new product updates and featuresmuch faster.
- There is less chance of human error by eliminating manual steps.
- Automating the compilation, testing, and deployment of code allows developers and other DevOps team members to focus more on continuously improving and innovating a product.
- Troubleshooting is much faster, and updates can be easily rolledback to a previous working version.
- Production teams can better respond to user wants and needs with faster, more frequent updates by focusing on smaller

releases as opposed to large, waterfall updates of past production systems.

How to Build a Deployment Pipeline

A company's deployment pipeline must be unique to their company and user needs and expectations, and will vary based on their type of product or service. There is no one-size-fits-all approach to creating a deployment pipeline, as it requires a good amount of upfront planning and creation of tests.

When planning your deployment pipeline, there are three essential components to include:

- **Build Automation (Continuous Integration):** Build automation, also referred to as Continuous Integration or CI for short, are automated steps within development designed for continuous integration – the compilation, building, and mergingof new code.
- **Test Automation:** Test automation relies on the creation of custom-written tests that are automatically triggered throughout a deployment pipeline and work to verify new compiled code
  against your organization's predetermined acceptance criteria.
- **Deploy Automation (Continuous Deployment/Delivery):** Like continuous integration, deploy automation with Continuous Deployment/Delivery (CD for short) helps expedite code delivery by automating the process of releasing code to a shared repository, and then automatically deploying the updates to a development or production environment.

When building your deployment pipeline, the primary goal should be to eliminate the need for any manual steps or intervention. This meanswriting custom algorithms for automatically compiling/building, testing, and deploying new code from development. By taking these otherwise tedious and repetitive steps off developers and other DevOps team members, they can focus more on creating new,
innovative product updates and features for today's highly
competitive user base.

What Are Continuous Integration (CI) and Continuous Delivery (CD) Pipelines?

A Continuous Integration (CI) and Continuous Delivery (CD) Pipeline works b continuously compiling, validating, and deploying new code updates as they ar written – rather than waiting for specificmerge or release days. This allows team to make faster, more frequent updates to a product with improved accuracy fron introducing automated steps. CI/CD Pipelines are a key component of an efficien full deployment pipeline.

Deployment Pipeline Tools

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

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

- Jenkins
- Azure DevOps
- CodeShip
- PagerDuty

DEPLOYMENT TOOLS

DevOps tools make it convenient and easier for companies to reduce the probability of errors and maintain continuous integration in operations. It addresses the key aspects of a company. DevOps tools automate the whole process and automatically build, test, and deploy the features.

DevOps tools make the whole deployment process and easy going

one and they can help you with the following aspects:

- Increased development.
- Improvement in operational efficiency.
- Faster release.
- Non-stop delivery.
- Quicker rate of innovation.
- Improvement in collaboration.

- Seamless flow in the process chain.

The DevOps tools play a very vital role in every organization, all of which are discussed here. Read along to know all about it.

DEPLOYMENT TOOLS

DevOps tools make it convenient and easier for companies to reduce the probability of errors and maintain continuous integration in operations. It addresses the key aspects of a company. DevOps tools automate the whole process and automatically build, test, and deploy the features.

DevOps tools make the whole deployment process and easy going one and they can help you with the following aspects:

- Increased development.
- Improvement in operational efficiency.
- Faster release.
- Non-stop delivery.
- Quicker rate of innovation.
- Improvement in collaboration.

- Seamless flow in the process chain.

The DevOps tools play a very vital role in every organization, all of which are discussed here. Read along to know all about it.

Architecture

The following diagram shows the flow of data in a deployment pipeline. It illustrates how you can turn your artifacts into resources.

Deployment pipelines are often part of a larger continuous integration/continuous deployment (CI/CD) workflow and are typically implemented using one of the following models:

- **Push model:** In this model, you implement the deployment pipeline using a central CI/CD system such as Jenkins or GitLab. This CI/CD system might run on Google Cloud, on-premises, or on a different cloud environment. Often, the same CI/CD system is used to manage multiple deployment pipelines.

  The push model leads to a centralized architecture with a few CI/CD systems that are used for managing a potentially large number of resources or applications. For example, you might use a single Jenkins or GitLab instance to manage your entire production environment, including all its projects and applications.
- **Pull model:** In this model, the deployment process is implemented by an agent that is deployed alongside the resource–for example, in the same Kubernetes cluster. The agent pulls artifacts or source code from a centralized location, and deploys them locally. Each agent manages one or two resources.

  The pull model leads to a more decentralized architecture with a potentially large number of single-purpose agents.

  Compared to manual deployments, consistently using deployment pipelines can have the following benefits:

- Increased efficiency, because no manual work is required.
- Increased reliability, because the process is fully automated and repeatable.

- Increased traceability, because you can trace all deployments to changes in code or to input artifacts.

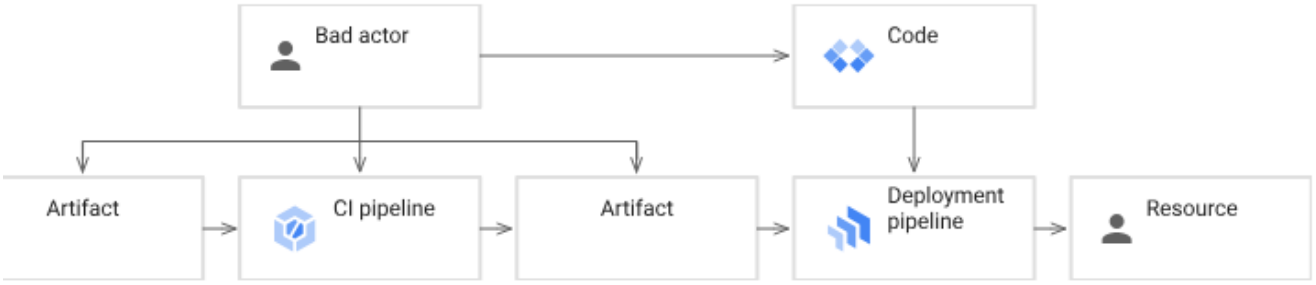
To perform, a deployment pipeline requires access to the resources it manages:

- A pipeline that deploys infrastructure by using tools like Terraform might need to create, modify, or even delete resources like VM instances, subnets, or Cloud Storage buckets.
- A pipeline that deploys applications might need to upload new container images to Artifact Registry, and deploy new application versions to App Engine, Cloud Run, or Google Kubernetes Engine (GKE).
- A pipeline that manages settings or deploys configuration files might need to modify VM instance metadata, Kubernetes configurations, or modify data in Cloud Storage.

If your deployment pipelines aren't properly secured, their access to Google Cloud resources can become a weak spot in your security posture. Weakened security can lead to several kinds of attacks, including the following:

- **Pipeline poisoning attacks:** Instead of attacking a resource directly, a bad actor might attempt to compromise the deployment pipeline, its configuration, or its underlying infrastructure. Taking advantage of the pipeline's access to Google Cloud, the bad actor could make the pipeline perform malicious actions on Cloud resources, as shown in the following diagram:

- **Supply chain attacks:** Instead of attacking the deployment pipeline, a bad actor might attempt to compromise or replace pipeline input—including source code, libraries, or container images, as shown in the following diagram:



To determine whether your deployment pipelines are appropriately secured, it's insufficient to look only at the allow policies and deny policies of Google Cloud resources in isolation. Instead, you must consider the entire graph of systems that directly or indirectly grant access to a resource. This graph includes the following information:

- The deployment pipeline, its underlying CI/CD system, and its underlying infrastructure
- The source code repository, its underlying servers, and its underlying infrastructure
- Input artifacts, their storage locations, and their underlying infrastructure
- Systems that produce the input artifacts, and their underlying infrastructure

Complex input graphs make it difficult to identify user access to resources and systemic weaknesses.

The following sections describe best practices for designing deployment pipelines in a way that helps you manage the size of the graph, and reduce the risk of lateral movement and supply chain attacks.

Assess security objectives

Your resources on Google Cloud are likely to vary in how sensitive they are. Some resources might be highly sensitive because they're business critical or confidential. Other resources might be less sensitive because they're ephemeral or only intended for testing purposes.

To design a secure deployment pipeline, you must first understand the resources the pipeline needs to access, and how sensitive these resources are. The more sensitive your resources, the more you should focus on securing the pipeline.

The resources accessed by deployment pipelines might include:

- Applications, such as Cloud Run or App Engine
- Cloud resources, such as VM instances or Cloud Storage buckets

- Data, such as Cloud Storage objects, BigQuery records, or files

Some of these resources might have dependencies on other resources, for example:

- Applications might access data, cloud resources, and other applications.
- Cloud resources, such as VM instances or Cloud Storage buckets, might contain applications or data.

As shown in the preceding diagram, dependencies affect how sensitive a resource is. For example, if you use an application that accesses highly sensitive data, typically you should treat that application as highly sensitive. Similarly, if a cloud resource like a Cloud Storage bucket contains sensitive data, then you typically should treat the bucket as sensitive.

Because of these dependencies, it's best to first assess the sensitivity of your data. Once you've assessed your data, you can examine the dependency chain and assess the sensitivity of your Cloud resources and applications.

Categorize the sensitivity of your data

To understand the sensitivity of the data in your deployment pipeline, consider the following three objectives:

- **Confidentiality**: You must protect the data from unauthorized access.
- **Integrity**: You must protect the data against unauthorized modification or deletion.

- **Availability**: You must ensure that authorized people and systems can access the data in your deployment pipeline.

For each of these objectives, ask yourself what would happen if your pipeline was breached:

- **Confidentiality:** How damaging would it be if data was disclosed to a bad actor, or leaked to the public?
- **Integrity:** How damaging would it be if data was modified or deleted by a bad actor?
- **Availability:** How damaging would it be if a bad actor disrupted your data access?

To make the results comparable across resources, it's useful to introduce security categories. Standards for Security Categorization (FIPS-199) suggests using the following four categories:

- **High:** Damage would be severe or catastrophic
- **Moderate:** Damage would be serious
- **Low:** Damage would be limited
- **Not applicable:** The standard doesn't apply

Depending on your environment and context, a different set of categories could be more appropriate.

The confidentiality and integrity of pipeline data exist on a spectrum, based on the security categories just discussed. The following subsections contain examples of resources with different confidentiality and integrity measurements:

Resources with low confidentiality, but low, moderate, and high integrity

The following resource examples all have low confidentiality:

- **Low integrity:** Test data
- **Moderate integrity:** Public web server content, policy constraints for your organization

- **High integrity:** Container images, disk images, application configurations, access policies (allow and deny lists), liens, access-level data

Resources with medium confidentiality, but low, moderate, and high integrity

The following resource examples all have medium confidentiality:

- **Low integrity:** Internal web server content
- **Moderate integrity:** Audit logs
- **High integrity:** Application configuration files

Resources with high confidentiality, but low, moderate, and high integrity

The following resource examples all have high confidentiality:

- **Low integrity:** Usage data and personally identifiable information
- **Moderate integrity:** Secrets
- **High integrity:** Financial data, KMS keys

Categorize applications based on the data that they access

When an application accesses sensitive data, the application and the deployment pipeline that manages the application can also become sensitive. To qualify that sensitivity, look at the data that the application and the pipeline need to access.

Once you've identified and categorized all data accessed by an application, you can use the following categories to initially categorize the application before you design a secure deployment pipeline:

- **Confidentiality**: Highest category of any data accessed
- **Integrity**: Highest category of any data accessed
- **Availability**: Highest category of any data accessed

This initial assessment provides guidance, but there might be additional factors to consider—for example:

- Two sets of data might have low-confidentiality in isolation. But when combined, they could reveal new insights. If an application has access to both sets of data, you might need to categorize it as medium- or high-confidentiality.
- If an application has access to high-integrity data, then you should typically categorize the application as high-integrity. But if that access is read only, a categorization of high-integrity might be too strict.

For details on a formalized approach to categorize applications, see Guide for Mapping Types of Information and Information Systems to Security Categories (NIST SP 800-60 Vol. 2 Rev1).

Categorize cloud resources based on the data and applications they host

Any data or application that you deploy on Google Cloud is hosted by a Google Cloud resource:

- An application might be hosted by an App Engine service, a VM instance, or a GKE cluster.
- Your data might be hosted by a persistent disk, a Cloud Storage bucket, or a BigQuery dataset.

When a cloud resource hosts sensitive data or applications, the resource and the deployment pipeline that manages the resource can also become sensitive. For example, you should consider a Cloud Run service and its deployment pipeline to be as sensitive as the application that it's hosting.

After categorizing your data and your applications, create an initial security category for the application. To do so, determine a level from the following categories:

- **Confidentiality**: Highest category of any data or application hosted
- **Integrity**: Highest category of any data or application hosted
- **Availability**: Highest category of any data or application hosted

When making your initial assessment, don't be too strict—for example:

- If you encrypt highly confidential data, treat the encryption key as highly confidential. But, you can use a lower security category for the resource containing the data.

- If you store redundant copies of data, or run redundant instances of the same applications across multiple resources, you can make the category of the resource lower than the category of the data or application it hosts.

Constrain the use of deployment pipelines

If your deployment pipeline needs to access sensitive Google Cloud resources, you must consider its security posture. The more sensitive the resources, the better you need to attempt to secure the pipeline. However, you might encounter the following practical limitations:

- When using existing infrastructure or an existing CI/CD system, that infrastructure might constrain the security level you can realistically achieve. For example, your CI/CD system might only support a limited set of security controls, or it might be running on infrastructure that you consider less secure than some of your production environments.
- When setting up new infrastructure and systems to run your deployment pipeline, securing all components in a way that meets your most stringent security requirements might not be cost effective.

To deal with these limitations, it can be useful to set constraints on what scenarios should and shouldn't use deployment pipelines and a particular CI/CD system. For example, the most sensitive deployments are often better handled outside of a deployment pipeline. These deployments could be manual, using a privileged session management system or a privileged access management system, or something else, like tool proxies.

To set your constraints, define which access controls you want to enforce based on your resource categories. Consider the guidance offered in the following table:

Category of resource	**Access controls**
Low	No approval required
Moderate	Team lead must approve
High	Multiple leads must approve and actions must be recorded

Contrast these requirements with the capabilities of your source code management (SCM) and CI/CD systems by asking the following questions and others:

- Do your SCM or CI/CD systems support necessary access controls and approval mechanisms?
- Are the controls protected from being subverted if bad actors attack the underlying infrastructure?
- Is the configuration that defines the controls appropriately secured?

Depending on the capabilities and limitations imposed by your SCM or CI/CD systems, you can then define your data and application constraints for your deployment pipelines. Consider the guidance offered in the following table:

Category of resource	Constraints
Low	Deployment pipelines can be used, and developers can self-approve deployments.
Moderate	Deployment pipelines can be used, but a team lead has to approve every commit and deployment.
High	Don't use deployment pipelines. Instead, administrators have to use a privileged access management system and session recording.

Maintain resource availability

Using a deployment pipeline to manage resources can impact the availability of those resources and can introduce new risks:

- **Causing outages:** A deployment pipeline might push faulty code or configuration files, causing a previously working system to break, or data to become unusable.
- **Prolonging outages:** To fix an outage, you might need to rerun a deployment pipeline. If the deployment pipeline is broken or unavailable for other reasons, that could prolong the outage.

A pipeline that can cause or prolong outages poses a denial of service risk: A bad actor might use the deployment pipeline to intentionally cause an outage.

Create emergency access procedures

When a deployment pipeline is the only way to deploy or configure an application or resource, pipeline availability can become critical. In extreme cases, where a deployment pipeline is the only way to manage a business-critical application, you might also need to consider the deployment pipeline business-critical.

Because deployment pipelines are often made from multiple systems and tools, maintaining a high level of availability can be difficult or uneconomical.

You can reduce the influence of deployment pipelines on availability by creating emergency access procedures. For example, create an alternative access path that can be used if the deployment pipeline isn't operational.

Creating an emergency access procedure typically requires most of the following processes:

- Maintain one of more user accounts with privileged access to relevant Google Cloud resources.
- Store the credentials of emergency-access user accounts in a safe location, or use a privileged access management system to broker access.
- Establish a procedure that authorized employees can follow to access the credentials.
- Audit and review the use of emergency-access user accounts.

Ensure that input artifacts meet your availability demands

Deployment pipelines typically need to download source code from a central source code repository before they can perform a deployment. If the source code repository isn't available, running the deployment pipeline is likely to fail.

Many deployment pipelines also depend on third-party artifacts. Such artifacts might include libraries from sources such as npm, Maven Central, or the NuGet Gallery, as well as container base images, and .deb, and .rpm packages. If one of the third-party sources is unavailable, running the deployment pipeline might fail.

To maintain a certain level of availability, you must ensure that the input artifacts of your deployment pipeline all meet the same or higher availability requirements. The following list can help you ensure the availability of input artifacts:

- Limit the number of sources for input artifacts, particularly third-party sources
- Maintain a cache of input artifacts that deployment pipelines can use if source systems are unavailable

Treat deployment pipelines and their infrastructure like production systems

Deployment pipelines often serve as the connective tissue between development, staging, and production environments. Depending on the environment, they might implement multiple stages:

- In the first stage, the deployment pipeline updates a development environment.
- In the next stage, the deployment pipeline updates a staging environment.
- In the final stage, the deployment pipeline updates the production environment.

When using a deployment pipeline across multiple environments, ensure that the pipeline meets the availability demands of each environment. Because production environments typically have the highest availability demands, you should treat the deployment pipeline and its underlying infrastructure like a production system. In other words, apply the same access control, security, and quality standards to the infrastructure running your deployment pipelines as you do for your production systems.

OVERALL ARCHITECTURE BUILDING AND TESTING