# 1 Introduction

Cloud-native applications are made up of multiple loosely coupled components (called microservices typically implemented as containers), operate in perimeter-less network environments requiring zero trust concepts (on-premises or cloud), and are accessed by users from a diverse set of locations (e.g., campus, home office, etc.). Cloud-native applications do not just refer to applications that run in the cloud. They also refer to the class of applications with design and runtime architectures, such as microservices, and a dedicated infrastructure for providing all application services, including security. The incorporation of zero trust principles [1] into this class of application provides techniques wherein access to all protected resources is enforced through identity-based protection and network-based protections (e.g., micro-segmentation), where applicable.

Cloud-native applications require agile and secure updates and deployment techniques for business reasons as well as the necessary resilience to respond to cybersecurity events. Hence, they call for a different application development, deployment, and runtime monitoring paradigm (collectively called the software life cycle paradigm) than the ones used for traditional monolithic or multi-tier applications. DevSecOps (Development, Security, and Operations) is a facilitating paradigm for this class of applications since it facilitates agile and secure development, delivery, deployment, and operations through (a) primitives, such as continuous integration, continuous delivery/continuous deployment (CI/CD) pipelines (explained in section 3); (b) security testing throughout the life cycle; and (c) continuous monitoring during runtime, all of which are supported by automation tools. In fact, the paradigm that meets the above objectives was originally given the term DevOps to indicate the fact that it seeks to remove the silos between development and operations groups and promotes (or drives) increased collaboration. The term DevSecOps was later coined by a portion of the community to emphasize the role of the security team in the whole process. Thus, DevSecOps is the term that denotes a culture and set of practices with automation tools to drive increased collaboration, trust, shared responsibility, transparency, autonomy, agility, and automation across the key stakeholders responsible for delivering software, including development, operations, and security organizations. DevSecOps has the necessary primitives and other building blocks to meet the design goals of cloud-native applications.

It should be noted that there is no community-wide consensus on the term “DevSecOps.” As already stated, the term was primarily coined to emphasize the fact that security must be tested and incorporated in all stages of the software development life cycle (i.e., build, test, package, deploy, and operate). A portion of the community continues to use the term “DevOps” based on the argument that there is no need to define a new term since security must be an integral part of any software life cycle process.

## 1.1 Scope

In theory, DevSecOps primitives can be applied to many application architectures but are best suited to microservices-based ones, which permit agile development paradigms due to the fact that the application is made up of relatively small, loosely coupled modules called microservices. Even within microservices-based architectures, the implementation of DevSecOps primitives can

take on different forms, depending on the platform. In this document, the chosen platform is a container orchestration and resource management platform (e.g., Kubernetes). The platform is an abstraction layer over a physical (bare metal) or virtualized (e.g., virtual machines, containers) infrastructure. To unambiguously refer to this platform or application environment throughout this document, it is called the *Reference Platform for DevSecOps Primitives*, or simply the *reference platform*.

Before describing the implementation of DevSecOps primitives for the reference platform, it is assumed that the following due diligence is applied with respect to deployment of the service mesh component [2]:

- Secure design patterns for deploying and managing service mesh-based components for infrastructure (e.g., network routing), policy enforcement, and monitoring
- Tests to prove that these service mesh components work as intended in a variety of scenarios for all aspects of the application, such as ingress, egress, and inside services

The guidance provided for implementation of DevSecOps primitives for the reference platform is agnostic to (a) the tools used in DevSecOps pipelines and (b) the service mesh software, which provides application services, though examples from service mesh offerings, such as Istio, are used to link them to real-world application artifacts (e.g., containers, policy enforcement modules, etc.).

A slightly more detailed description of the code types (referred to in the executive summary) in the entire application environment presented by the reference platform is given below. Please note that these code types include those that support implementation of DevSecOps primitives.

1. Application code – embodies the application logic for carrying out one or more business functions) and is made up of code describing the business transactions and database access
2. Application services code (e.g., service mesh code) – provides various services for the application, such as service discovery, establishing network routes, network resiliency services (e.g., load balancing, retries), and security services (e.g., enforcing authentication, authorization etc. based on policies (item 4 below))
3. Infrastructure as code – expresses the computing, networking, and storage resources needed to run the application in the form of a declarative code
4. Policy as code – contains declarative code for generating the rules and configuration parameters for realizing security objectives, such as zero trust through security controls (e.g., authentication, authorization) during runtime
5. Observability as code – triggers software related to logging (recording all transactions) and tracing (communication pathways involved in executing application requests) and monitors (keeping track of application states during runtime)

Code types 3, 4, and 5 may have overlap with code type 2.

This document covers the implementation of pipelines or workflows associated with all five code types listed above. Thus, the entire application environment (not just the application code) benefits from all of the best practices that exist for application code (e.g., agile iterative development, version control, governance, etc.). Infrastructure as code, policy as code, and observability as code belong to a special class called declarative code. When using 'as code' techniques, the code that is written (e.g., for provisioning a resource) is managed similar to application source code. This implies that it is versioned, documented, and has access controls defined similar to what is done for application source code repository. Often, domain-specific declarative languages are used: the requirements are declared, and an associated tool converts them into artifacts that make up a runtime instance. For example, in the case of infrastructure as code (IaC), the declarative language models the infrastructure as a series of resources. The associated configuration management tool pulls together these resources and generates what are known as *manifests* that define the final shape and state of the platform (runtime instance) associated with the defined resources. These manifests are stored in servers associated with a configuration management tool and are used by the tool to create compiled configuration instructions for the runtime instance on the designated platform. Manifests are generally encoded in platform-neutral representations (e.g., JSON) and fed to platform resource provisioning agents through REST APIs.

### 1.2 Related DevSecOps Initiatives

There are several DevSecOps initiatives in various agencies of the Federal Government with varying emphasis and focus, depending on the processes enabled by software and mission needs. Though not exhaustive, here is a brief overview of those initiatives [3].

- DevSecOps pipelines are involved in building, checking in, and checking out from a container registry called Iron Bank, a repository of DOD-vetted hardened container images.
- The Air Force's Platform One, the DevSecOps platform that enabled the concept of continuous authority to operate (C-ATO), which in turn streamlined the DOD's authorization process to accommodate the speed and frequency of modern continuous software deployments.
- The National Geospatial-Intelligence Agency (NGA) outlined its DevSecOps strategy in "The NGA Software Way," where three key metrics – availability, lead time, and deployment frequency – are laid out for each of its software products, along with specification of seven distinct product lines for enabling DevSecOps pipelines, including messaging and workflow tools.
- The Centers for Medicare and Medicaid Services (CMS) is adopting a DevSecOps approach where one emphasis is on laying the groundwork for the software bill of materials (SBOM) – a formal record that contains the details and supply chain relationships of various components used in building software. The purpose of producing SBOMs is to meet the goals established under the Continuous Diagnostics and Mitigation (CDM) program.

- At the Naval Surface Warfare Center (NSWC), the implementation methodology of DevSecOps primitives is used to teach and train the software workforce on various software metrics and the role of automation as enablers for achieving those metrics.
- The Army's DevSecOps initiative is called the Army Software Factory and is focused on building skillsets rather than building software. It utilizes DevSecOps capabilities (pipelines and platform-as-a-service features) as a technology accelerator to gain efficiency and proficiency in product management, user experience, user interface (UI/UX) design, platform, and software engineering.

### 1.3 Target Audience

Since DevSecOps primitives span development (build and test for security, package), delivery/ deployment and continuous monitoring (to ensure secure states during runtime), the target audience for the recommendations in this document includes software development, operations, and security teams.

### 1.4 Relationship to Other NIST Guidance Documents

Since the reference platform is made up of container orchestration and resource management platform and service mesh software, the following publications offer guidance for securing this platform as well as provide background information for the contents of this document:

- Special Publication (SP) 800-204, *Security Strategies for Microservices-based Application Systems* [4], discusses the characteristics and security requirements of microservices-based applications and the overall strategies for addressing those requirements.
- SP 800-204A, *Building Secure Microservices-based Applications Using Service-Mesh Architecture* [5], provides deployment guidance for various security services (e.g., establishment of secure sessions, security monitoring, etc.) for a microservices-based application using a dedicated infrastructure (i.e., a service mesh) based on service proxies that operate independent of the application code.
- SP 800-204B [6], *Attribute-based Access Control for Microservices-based Applications Using a Service Mesh*, provides deployment guidance for building an authentication and authorization framework within the service mesh that meets the security requirements, such as (1) zero trust by enabling mutual authentication in communication between any pair of services and (2) a robust access control mechanism based on an access control model, such as attribute-based access control (ABAC) model, that can be used to express a wide set of policies and is scalable in terms of user base, objects (resources), and deployment environment.
- SP 800-190 [7], *Application Container Security Guide*, explains the security concerns associated with container technologies and makes practical recommendations for addressing those concerns when planning for, implementing, and maintaining containers. The recommendations are provided for each tier within the container technology architecture.

### 1.5 Organization of This Document

This document is organized as follows:

- Chapter 2 gives a brief description of the reference platform for which guidance for implementation of DevSecOps primitives is provided.
- Chapter 3 introduces the DevSecOps primitives (i.e., pipelines), the methodology for designing and executing the pipelines, and the role that automation plays in the execution.
- Chapter 4 covers all facets of pipelines, including (a) common issues to be addressed for all pipelines, (b) descriptions of the pipelines for the five code types in the reference platform that are listed in Section 1.1, and (c) the benefit of DevSecOps for security assurance for the entire application environment (the reference platform with five code types, thus carrying the DevSecOps implementation) during the entire life cycle, including the "Continuous Authority to Operate (C-ATO)."
- Chapter 5 provides summary and conclusion.

# 2 Reference Platform for the Implementation of DevSecOps Primitives

As stated in Section 1.1, the reference platform is a container orchestration and management platform. In modern application environments, the platform is an abstraction layer over a physical (bare metal) or virtualized (e.g., virtual machines, containers) infrastructure. Before the implementation of DevSecOps primitives, the platform simply contains the application code, which contains the application logic and the service mesh code, which in turn provides application services. This section will consider the following:

- A container orchestration and resource management platform that houses both the application code and most of the service mesh code
- The service mesh software architecture

## 2.1 Container Orchestration and Resource Management Platform

Since microservices are typically implemented as containers, a container orchestration and resource management platform are used for deployment, operations, and maintenance of services.

A typical orchestration and resource management platform consists of various logical (forming the abstraction layer) and physical artifacts for the deployment of containers. For example, in Kubernetes, containers run inside the smallest unit of deployment called a pod. A pod can theoretically host a group of containers, though usually, only one container runs inside a pod. A group of pods are defined inside what is known as a node, where a node can be either a physical or virtual machine (VM). A group of nodes constitutes a cluster. Usually, multiple instances of a single microservice are needed to distribute the workload to achieve the desired performance level. A cluster is a pool of resources (nodes) that is used to distribute the workload of microservices. One of the techniques used is horizontal scaling, where microservices that are accessed more frequently are allocated more instances or allocated to nodes with more resources (e.g., CPUs and/or memory).

### 2.1.1 Security Limitations of Orchestration Platform

Microservices-based applications require several application services, including security services such as authentication and authorization, as well as the generation of metrics for individual pods (monitoring), consolidated logging (to ascertain causes of failures of certain requests), tracing (sequence of service calls for an application request), traffic control, caching, secure ingress, service-to-service (east/west traffic), and egress communication.
Taking the example of secure communications between containers, specific code needs to be added in order to secure the communications between pods in a platform such as Kubernetes (e.g., with mTLS). Pods that communicate do not apply identity and access management between themselves. Though there are tools that can be implemented to act as a firewall between pods, such as the Kubernetes Network Policy [8], this is a layer 3 solution rather than a layer 7 solution, which is what most modern firewalls are. This means that while one can know the source of traffic, one cannot peek into the data packets to understand what they contain. Further, it does not allow for making vital metadata-driven decisions, such as routing on a new version of a pod

based on an HTTP header. There are Kubernetes ingress objects that provide a reverse proxy based on layer 7, but they do not offer anything more than simple traffic routing. Kubernetes does offer different ways of deploying pods that some form of A/B testing or canary deployments, but they are done at the connection level and provide no fine-grained control or fast failback. For example, if a developer wants to deploy a new version of a microservice and pass 10 % of traffic through it, they will have to scale the containers to at least 10 – nine for the old version and one for the new version. Further, Kubernetes cannot split the traffic intelligently and instead balances loads between pods in a round-robin fashion. Every Kubernetes container within a pod has a separate log, and a custom solution over Kubernetes must be implemented to capture and consolidate them.

Although the Kubernetes dashboard offers monitoring features on individual pods and their states, it does not expose metrics that describe how application components interact with each other or how much traffic flows through each of the pods. Consolidated logging is required to determine error conditions that cause an application request or transaction to fail. Tracing is required to trace the sequence of containers that are invoked as part of any application request based on the application logic that underlies a transaction. Since traffic flow cannot be traced through Kubernetes pods out of the box, it is unclear where on the chain the failure for the request occurred.

This is where the service mesh software can provide the needed application services and much more.

## 2.2 Service Mesh Software Architecture

Having looked at the various application services required by microservices-based applications, consider the architecture of service mesh software that provides those services. The service mesh software consists of two main components: the control plane and the data plane.

### 2.2.1 Control Plane

The control plane has several components. While the data plane of the service mesh mainly consists of proxies running as containers within the same pod as application containers, the control plane components run in their own pods, nodes, and associated clusters. The following are the various functions of the control plane [9]:

1. Service discovery and configuration of the Envoy sidecar proxies
2. Automated key and certificate management
3. API for policy definition and the gathering of telemetry data
4. Configuration ingestion for service mesh components
5. Management of an inbound connection to the service mesh (Ingress Gateway)
6. Management of an outbound connection from the service mesh (Egress Gateway)

7. Inject sidecar proxies into those pods, nodes, or namespaces where application microservice containers are hosted

Overall, the control plane helps the administrator populate the data plane component with the configuration data that is generated from the policies resident in the control plane. The policies for function 3 above may include network routing policies, load balancing policies, policies for blue-green deployments, canary rollouts, timeout, retry, and circuit-breaking capabilities. These last three are collectively called by the special name of resiliency capabilities of the networking infrastructure services. Last but not the least are security-related policies (e.g., authentication and authorization policies, TLS establishment policies, etc.). These policy rules are parsed by a module that converts them into configuration parameters for use by executables in data plane proxies that enforce those policies.

### 2.2.2 Data Plane

The data plane component performs three different functions:

1. Secure networking functions
2. Policy enforcement functions
3. Observability functions

The primary component of the data plane that performs all three functions listed above is called the sidecar proxy. This L7 proxy runs in the same network namespace (which, in this platform, is the same pod) as the microservice for which it performs proxy functions. There is a proxy for every microservice to ensure that a request from a microservice does not bypass its associated proxy and that each proxy is run as a container in the same pod as the application microservice. Both containers have the same IP address and share the same IP Table rules. That makes the proxy take complete control over the pod and handle all traffic that passes through it [9,10].

The first category of functions (secure networking) includes all functions related to the actual routing or communication of messages between microservices. The functions that come under this category are service discovery, establishing a secure (TLS) session, establishing network paths and routing rules for each microservice and its associated requests, authenticating each request (from a service or user), and authorizing the request.

With the example of establishing a mutual TLS session, the proxy that initiates the communication session will interact with the module in the control plane of the service mesh to check whether it needs to encrypt traffic through the chain and establish mutual TLS with the backend or target pod. Enabling this functionality using mutual TLS requires every pod to have a certificate (i.e., a valid credential). Since a good-sized microservice application (consisting of many microservices) may require hundreds of pods (even without horizontal scaling of individual microservices through multiple instances), this may involve managing hundreds of short-lived certificates. This, in turn, requires each microservice to have a robust identity and the service mesh to have an access manager, a certificate store, and a certificate validation capability. In addition, mechanisms for identifying and authenticating the two communicating pods are

required for supporting authentication policies.

Other kinds of proxies include ingress proxies [11] that intercept the client calls into the first entry point of application (first microservice that is invoked) and egress proxies that handle a microservice's request to application modules residing outside of the platform cluster.

The second category of functions that the data plane performs is enforcement of the policies defined in the control plane through configuration parameters in the proxies (policy enforcement service). An example is the use of the information in the JWT token that is part of a microservice request to authenticate the calling service. Another example is the enforcement of access control policies for each request using either the code residing in the proxy itself or by connecting to an external authorization service.

The third category of functions that service proxies perform almost always in collaboration with application service containers are to gather telemetry data, which helps to monitor the health and state of the services, transfer logs associated with a service to the log aggregation module in the control plane, and append necessary data to application request headers to facilitate the tracing of all requests associated with a given application transaction. The application response is conveyed by proxies back to its associated calling service in the form of return codes, a description of response, or the retrieved data.

The service mesh is container orchestration platform-aware, interacts with the API server that provides a window into application services installed in various platform artifacts (e.g., pods, nodes, namespaces), monitors it for new microservices, and automatically injects sidecar containers into the pods containing these new microservices. Once the service mesh inserts the sidecar proxy containers, operations and security teams can enforce policies on the traffic and help secure and operate the application. These teams can also configure the monitoring of microservices applications without interfering with the functioning of the applications.

The provisioning of infrastructure, policy enforcement, and observability services can be automated using declarative code that is part of DevSecOps pipelines. While the development team should be overall aware of the security and management details of deployment of their code, the automation of the above mentioned services provides more time to them to concentrate their efforts on efficient development paradigms, such as code modularity and structuring.

# 3 DevSecOps – Organizational Preparedness, Key Primitives, and Implementation

DevSecOps incorporates security into the software engineering process early on. It integrates and automates security processes and tooling into all of the development workflow (or pipeline as later explained) in DevOps so that it is seamless and continuous. In other words, it can be looked upon as a combination of the three processes: Development + Security + Operations [12].

This section discusses the following aspects of DevSecOps:

- Organizational preparedness for DevSecOps
- DevSecOps Platform
- Fundamental building blocks or key primitives for DevSecOps

## 3.1 Organizational Preparedness for DevSecOps

DevSecOps is a software development, deployment, and life cycle management methodology that involves a shift from one large release for an entire application or platform to the continuous integration, continuous delivery, and continuous deployment (CI/CD) approach. This shift, in turn, requires changes in the structure of a company's IT department and its workflow. The most pronounced change involves organizing a DevSecOps group that consists of software developers, security specialists, and IT operations experts for each portion of the application (i.e., the microservice). This smaller team not only promotes efficiency and effectiveness in initial agile development and deployment but also in subsequent life cycle management activities, such as monitoring application behavior, developing patches, fixing bugs, or scaling the application. The composition of this cross-functional team with expertise in three areas forms a critical success factor for introducing DevSecOps in an organization.

## 3.2 DevSecOps Platform

DevSecOps is an agile, automated development and deployment process that uses primitives called CI/CD pipelines aided by automated tools to take the software from the build phase to deployment phase and finally to the runtime/operations phase. These pipelines are workflows that take the developer's source code through various stages, such as building, testing, packaging, delivery, and deployment supported by testing tools in various phases.

A DevSecOps platform denotes the set of resources on which various CI/CD pipelines (for each code type) run. At the minimum, this platform consists of the following components:

(a) Pipeline software
- CI software - pulls code from a code repository, invokes the build software, invokes test tools and stores back tested artifacts to Image registry
- CD software – pulls out artifacts, packages and based on computing, network and storage resource descriptions in IaC deploys the package

(b) SDLC software

- Build Tools (e.g., IDEs)
- Testing Tools (SAST, DAST, SCA)

(c) Repositories
- Source Code Repositories (e.g., GitHub)
- Container Image Repositories or Registries

(d) Observability or Monitoring Tools
- Logging and Log Aggregation Tools
- Tools that generate metrics
- Tracing Tools (sequence of application calls)
- Visualization Tools (combine data from above to generate dashboard/alerts)



In a DevSecOps platform, security assurance is provided during the build and deployment phases through built-in design features, such as zero trust, and testing using a comprehensive set of security testing tools, such as static application security tools (SAST), dynamic security testing tools (DAST), and software composition analysis (SCA) tools. In addition, security assurance is also provided during the runtime/operations phase by continuous behavior detection/prevention tools, some of which may even use sophisticated techniques such as artificial intelligence (AI) and machine learning (ML). Therefore, a DevSecOps platform not only runs during build and deployment phases but also during the runtime/operations phase.

In some DevSecOps platforms, the security tools (e.g., SAST, DAST, and SCAs) that perform application security analysis, such as identifying vulnerabilities and bugs through efficient scanning in the background can be tightly integrated with integrated development environments (IDEs) and other DevOps tools. This feature when present, makes these tools transparent to developers and avoids the necessity for them to call separate APIs for running these tools [13]. Depending on the IDE, the task performed, or the resources consumed by the tool, the tool may alternatively execute separate from the IDE.

### 3.2.1 Deliverables for DevSecOps Platform

The use of SAST, DAST, and SCA tools may not be limited to testing just the application code. DevSecOps may include the use of these tools for other code types such as IaC, as IaC defines the deployment architecture of the application and thus the key avenue to automatically assess and remediate security design gaps.

In summary, a DevSecOps platform delivers the following:

- Provides security assurance through the incorporation of adequate testing/checking within pipelines associated with all code types in the application environment. Security is not relegated to a separate task or phase.
- The DevSecOps platform also operates during runtime (in production), providing real-time assurance of security by assisting enforcement of zero trust principles and through continuous monitoring followed by alerts and correction mechanisms thus enabling the certification of continuous authority to operate (C-ATO).

### 3.3 DevSecOps – Key Primitives and Implementation Tasks

The key primitives and implementation tasks involved are:

- Concept of pipelines and the CI/CD pipeline
- Building blocks for the CI/CD pipeline
- Designing and executing the CI/CD pipeline
- Strategies for automation
- Requirements for security automation tools in the CI/CD pipeline

#### 3.3.1 Concept of Pipelines and the CI/CD Pipeline

DevSecOps, being a methodology or framework for agile application development, deployment, and operations – is made up of stages just like any other methodology [14]. The sequence and flow of information through the stages is called workflow, where some stages can be executed in parallel while others have to follow a sequence. Each stage may require the invocation of a unique job to execute the activities in that stage.

A unique concept that DevSecOps introduces in the process workflow is the concept of "pipelines" [15]. With pipelines, there is no need to individually write jobs for initiating/executing each stage of the process. Instead, there is only one job that starts from the initial stage, automatically triggers the activities/tasks pertaining to other stages (both sequential and parallel), and creates an error-free smart workflow.

The pipeline in DevSecOps is called the CI/CD pipeline based on the overall tasks it accomplishes and the two individual stages it contains. CD can denote either the continuous delivery or continuous deployment stage. Depending on this latter stage, CI/CD can involve the following tasks:

- Build, Test, Secure, and Deliver – the tested modified code is delivered to the staging area.
- Build, Test, Secure, Deliver, and Deploy – the code in the stage area is automatically deployed.

In the former, automation ends at the delivery stage, and the next task of deployment of the modified application in the hosting platform infrastructure is performed manually. In the latter, the deployment is also automated. Automation of any stage in the pipeline is enabled by tools that express the pipeline stage as code.

The workflow process for a CI/CD pipeline is depicted in Figure 1 below:

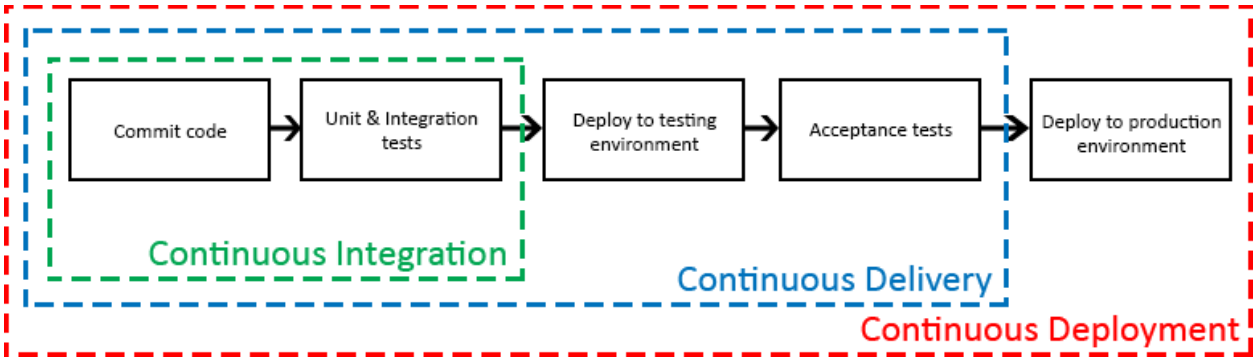


**Figure 1: CI/CD pipeline workflow [16]**

The unit and integration tests shown in the diagram use the SAST, DAST, and SCA tools described in Section 3.2. It should be noted that an organization has the option to continue the build process when a test fails. Depending on how the organization balances risk tolerance versus business needs, it may choose to fail-open (log and continue) or fail-closed (stop/break) when a specific test fails. In the fail-closed event, the developer gets the test outcome report, must fix the issues, and restart the CI process.

Continuous integration involves developers frequently merging code changes into a central repository where automated builds and tests run. Build is the process of converting the source code to executable code for the platform on which it is intended to run. In the CI/CD pipeline software, the developer's changes are validated by creating a build and running automated tests against the build. This process avoids the integration challenges that can happen when waiting for release day to merge changes into the release branch [17].

Continuous delivery is the stage after continuous integration where code changes are deployed to a testing and/or staging environment after the build stage. Continuous delivery to a production environment involves the designation of a release frequency – daily, weekly, fortnightly, or some other period – based on the nature of the software or the market in which the organization operates. This means that on top of automated testing, there is a scheduled release process, though the application can be deployed at any time by clicking a button. The deployment process in continuous delivery is characterized as manual, but tasks such as the migration of code to a production server, the establishment of networking parameters, and the specification of runtime configuration data may be performed by automated scripts.

Continuous deployment is similar to continuous delivery except that the releases happen automatically [17], and changes to code are available to customers immediately after they are made. The automatic release process may in many instances include A/B testing to facilitate slow rollout of new features so as to mitigate the impact of failures if there is a bug/error.

The distinction between continuous delivery and continuous deployment is shown in Figure 2 below.

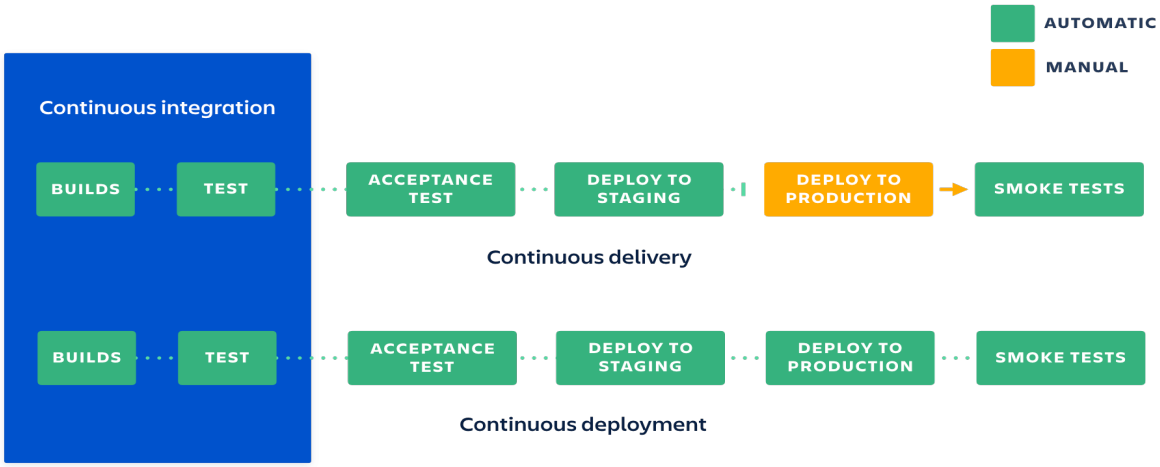


**Figure 2 - Distinction Between Continuous Delivery and Continuous Deployment [17]**

### 3.3.2 Building Blocks for CI/CD Pipelines

The primary software for defining CI/CD pipeline resources, building the pipelines, and executing those pipelines is the CI/CD pipeline software. There may be slight variations in the architecture of this class of software depending on the particular offering. The following is an overview of the landscape in which CI/CD tools (pipeline software) operate:

- Some CI/CD tools natively operate on the platform on which the application and the associated resources are hosted (i.e., container orchestration and resource management platform), while others need to be integrated into the application hosting platform through its API. Some advantages of using the CI/CD tools that are native to the application hosting platform are:
  - It makes it easier to deploy, maintain, and manage the CI/CD tool itself.
  - Every pipeline defined by the CI/CD tool becomes another platform-native resource and is managed the same way. In fact, all entities required for executing pipelines, such as Tasks and Pipelines (which then act as blueprints for other entities, such as Task Runs and Pipeline Runs, respectively), can be created as custom resource definitions (CRDs) built on top of resources native to the platform [18]. Software with this type of architecture may be used by other CI/CD pipeline software offerings to facilitate faster defining of pipelines.
- Some CI/CD tools integrate with code repositories to scan/inspect application code. These types of tools have association with code repositories for each application and for each environment. When changes in application modules, infrastructure, or configuration are made they are stored in these code repositories. The CI/CD pipeline software connected to

the code repositories through webhooks or some other means is activated on commits (push workflow model) or through pull requests from these repositories.

- Some CI/CD tools perform CD functions alone for the native platform (e.g., Jenkins X for Kubernetes platform) or for multiple technology stacks (e.g., Spinnaker for multi-cloud deployment). The difficulty with some in this class of tools is that they may lack native tools for completing the CI functions (e.g., tools to test code, build application images, or push them to registry).

### 3.3.3 Preparing and Executing the CI/CD pipeline

The purpose of creating the CI/CD pipeline is to enable frequent updates to source code, rebuilds, and the automatic deployment of updated modules into the production environment. The key tasks involved are [19]:

The preparatory tasks required are the following:
(1) Ensure that all individual components in the DevSecOps platform (Pipeline software, SDLC software, Code Repositories, Observability tools etc.) are available
(2) Ensure that these components are secure either through certification, validation or customized testing
(3) Integrate CI & CD tools with SDLC tools – Access tokens, Calling scripts, pipeline definitions
(4) Set up configuration details in IaC tool (with GitOps) based on deployment environment (i.e., application hosting platform in-premise or in the cloud)
(5) Integrate the runtime tools to the deployment environment
(6) Design the dashboard and define the events to monitor, the alerts to be generated and the application state variables (e.g., Memory utilization etc.) to monitor through connection to Tools such as log aggregators, metric generators and trace generators.

The execution tasks include [19]:

- Setting up the source code repository: Set up a repository (e.g., GitHub or GitLab) for storing application source code with proper version control.
- Build process: Configure and execute the build process for generating the executables (for those portions of the code that need to be updated) using an automated code build tool.
- Securing the process: Ensure that the build is free of static and dynamic vulnerabilities through unit testing with SAST and DAST tools. This and the above tasks are activated by the CI tool.
- Describing the deployment environment: This may involve describing (using the IaC) the physical/virtual resources to deploy the application either in the cloud or in the enterprise data center
- Creating the delivery pipeline: Create a pipeline that will automatically deploy the application. This and the previous task are enabled by the CD tool.

- Test the code and execute the pipeline: After proper testing, execute the CI tool whenever a new code appears in the repository. When the build process is successful, execute the CD tool to deploy the application into staging/production environment.
- Activate the run time tool and the dashboard to initiate run time monitoring

To reiterate, the three primary stages of the CI/CD process are build/test, ship/package, and deploy. The following features transform this into a pipeline:

- When an update is made to the source code for a service, the code changes pushed to the source code repository trigger the code building tool.
- The code development environment or a code building tool (such as an IDE), is often integrated with security testing tools (e.g., static vulnerability analysis tool) to facilitate the generation of secure compiled code artifacts, thus integrating security into the CI pipeline.
- The generation of compiled code artifacts in code building tools triggers the shipping/package tool, which may be integrated with its own set of tools (e.g., dynamic vulnerability analysis, dynamic penetration testing tools, software composition analysis tools for identifying vulnerabilities in the attached libraries) and also creates the configuration parameters relevant to the deployment environment.
- The output of the shipping/packaging tool is then automatically fed to the CD tool, which deploys the package into the desired environment (e.g., staging, production) [20].

The workflow of the CI/CD pipeline should not create the impression that there is no human element involved. The following teams/role players contribute to CI/CD pipeline [21]:

- Development team – Members of this team declare third-party off-the-shelf software (OSS) dependencies for their application, review recommendations from DevSecOps system around vulnerable dependencies, update them as suggested, and write adequate test-cases to ensure all functional verifications (to eliminate runtime bugs).
- The Chief Information Security Officer (CISO) – In consultation with the security team, the CISO defines the overall scope (depth and breadth) of the DevSecOps system so that it can be configured appropriately to meet the mission-critical needs of the applications.
- Security team – Members of this team create pipelines following best-practices, including tasks to perform all required security functions (e.g., SBOM generation, vulnerability scanning, code building, code signing, introduction of new testing tools, conducting audit etc.). Specifically, in some instances, members of the security team may be responsible for designing, building, and maintaining Policy as Code and the associated pipeline.
- Infrastructure team – Members of this team create, maintain, and upgrade the infrastructure.
- QA team – Members of this team develop integration test cases.
- Deployment team/release team – Members of this team create pipelines and packages for various environments (UAT/PreProd/Prod) and perform the configuration and provisioning appropriate for these environments.

Some of the many activities performed by these teams include the customization, update, and enhancement of the tools employed in the CI/CD pipeline (e.g., updating the static vulnerability analysis tool with the latest database of known vulnerabilities). Caution should be exercised during manual operations so that they do not block pipelines. The targets for mean time to production should be set up while also mitigating risks, such as insider threats, through the use of "merge (GitLab) or pull (GitHub) requests" and multiple approvers for those requests. This pipeline is designed, maintained, and executed by the post-release team who – in addition to monitoring functions – performs other processes, such as compliance management, backup processes, and asset tracking [22].



### 3.3.4 Strategies for Automation

Compared to other models of software development, which involve a linear progression from coding to release, DevOps uses a forward process with a delivery pipeline (i.e., build/secure, ship/package, and release) and a reverse process with a feedback loop (i.e., plan and monitor) that form a recursive workflow. The role of automation in these activities is to improve this workflow. Continuous integration emphasizes testing automation to ensure that the application is not broken whenever new commits are integrated into the main branch. Automation results in the following benefits:

- Generation of data regarding software static and runtime flows.
- Reduction of development and deployment times.
- Built-in security, privacy, and compliance for the architecture.

The following strategies are recommended [23] for automation so as to facilitate better utilization of organizational resources and derive the greatest benefit in terms of an efficient, secure application environment.

Choice of Activities to Automate: For example, the following are productive candidates for the automation of testing activities.

- Testing of modules whose functions are subject to regulatory compliance (e.g., PCI-DSS, HIPAA, Sarbanes-Oxley).
- Tasks that are repetitive with moderate to high frequency.
- Testing of modules that perform time-sequenced operations, such as message publishers and message subscribers.
- Testing of workflows (e.g., request tracing) involving transactions that span multiple services.
- Testing of services that are resource-intensive and likely to be performance bottlenecks.

After choosing the candidates for automation based on the above criteria, the usual risk analysis must be applied to choose a subset that provides an optimum return on investment and maximizes desirable security metrics (e.g., defense in depth). Some recommended strategies include:

- Using the cost-benefit ratio in hours saved per year to prioritize which processes to automate [23].
- Using key performance indicators (KPI) (e.g., mean time to identify faults or problems, rectify, or recover) as markers to refine the DevSecOps processes [23].
- Based on the application, applying different weights to infrastructure services (e.g., authorization and other policies enforcement, monitoring of system states to ensure secure runtime states, network resilience in terms of system availability, latency, mean time to recover from an outage etc.) to determine the allocation of resources to DevSecOps processes.

### 3.3.5 Requirements for Security Automation Tools in CI/CD Pipelines

The security automation tools for various functions (e.g., static vulnerability analysis, dynamic vulnerability analysis, software composition analysis) used in CI/CD pipelines need to have different interface and alerting/reporting requirements since they have to operate seamlessly depending on the pipeline stage (e.g., build, package, release) during which they are used. These requirements are:

- Security automation tools should work with integrated development environment (IDE) tools and help developers prioritize and remediate static vulnerabilities. These capabilities are needed to facilitate developer adoption and improve productivity.
- Security automation tools should be flexible to support specific workflows and provide scaling capabilities for security services.
- Tools that perform static vulnerability checks at the build phase ensure safe data flows, and those that perform dynamic vulnerability checks ensure safe application states during runtime.

It must be mentioned that security automation tools come with costs, and hence, the extent of the usage of these tools is based on risk factor analysis.

# 4 Implementing DevSecOps Primitives for the Reference Platform

Various CI/CD pipelines are involved in the reference platform (i.e., microservices-based application with service mesh that provides infrastructure services). Though the reference application is a microservices-based application, the DevSecOps primitives can be applied to monolithic applications as well as applications that are both on-premises and cloud-based (e.g., hybrid cloud, single public cloud, and multi-cloud).

In section 2.1, we referred to the five code types in our reference application environment. We also mentioned that separate CI/CD pipelines can be created for each of these five code types as well. The location of these five code types within the reference platform components will be discussed followed by separate sections that will describe the associated CI/CD pipelines.

1. Code types in the reference platform and associated CI/CD pipelines (Section 4.1)
2. CI/CD pipeline for application code and application services code (Section 4.2)
3. CI/CD pipeline for infrastructure as code (IaC) (Section 4.3)
4. CI/CD pipeline for policy as code (Section 4.4)
5. CI/CD pipeline for observability as code (Section 4.5)

Implementation issues for all CI/CD pipelines irrespective of code types will be addressed in the following sections:

- Securing the CI/CD pipelines (Section 4.6)
- Workflow models in the CI/CD pipelines (Section 4.7)
- Security testing in the CI/CD pipelines (Section 4.8)

This section will also consider the overall benefits of DevSecOps with a subsection on specific advantages in the context of the reference platform and the ability to leverage DevSecOps for continuous authorization to operate (C-ATO) in Sections 4.9 and 4.10, respectively.

## 4.1 Description of Code Types and Reference Platform Components

A brief description of the five types of codes stated above (i.e., application, application services, infrastructure, policy, and monitoring) is as follows:

- Application code and application services code – The former contains the data and application logic for a specific set of business transactions, while the latter contains code for all services, such as network connections, load balancing, and network resilience.
- Infrastructure as code (IaC) – The code for provisioning and configuring infrastructure resources which host application deployment in a repeatable and consistent manner [24]. This code is written in a declarative language and – when executed – provisions, de-provisions, and configures the infrastructure for the application that is being deployed. This type of code is like any other code found in an application's microservice except that it provides an infrastructure service (e.g., provisioning a server) rather than a transaction service (e.g., payment processing for an online retail application).

- Policy as code – This describes many policies, including security policies, as executable modules [25]. One example is the authorization policy, the code for which contains verbs or artifacts specific to the policy (e.g., allow, deny, etc.) and to the domain where it applies (e.g., REST API with verbs such as method [GET, PUT, etc.], path, etc.). This code can be written in a special-purpose policy language, such as Rego, or languages used in regular applications, such as Go. This code may have some overlap with the configuration code of IaC. However, for implementing policies associated with critical security services that are specific to the application domain, a separate policy as code that resides in the policy enforcement points (PEPs) of the reference platform is required.
- Observability as code – The ability to infer a system's internal state and provide actionable insights into when and, more importantly, why errors occur within a system. It is a full-stack observability that includes monitoring and analytics and that offers key insights into the overall performance of applications and the systems hosting them. In the context of the reference platform, observability as code is the portion of the code that creates agencies in proxies and creates functionality for gathering three types of data (i.e., logs, traces, and telemetry) from microservices applications [26]. This type of code also supplies or transfers data to the external tools (e.g., log aggregation tool that aggregates log data from individual microservices, provides analysis of tracing data for bottleneck services, generates metrics that reflect the application health from telemetry data, etc.). Brief descriptions of the three functions enabled by observability as code are:
  1. **Logging** captures detailed error messages, as well as debugs logs and stack traces for troubleshooting.
  2. **Tracing** follows application requests as they wind through multiple microservices to complete a transaction in order to identify an issue or performance bottleneck in a distributed or microservices-based ecosystem.
  3. **Monitoring, or metrics,** gathers telemetry data from applications and services.

Each of the code types has an associated CI/CD pipeline and is described in Sections 4.2 through 4.5. There may be overlaps among application service code, infrastructure as code, policy as code, and observability as code types.

The constituent components of the reference platform hosting the five code types are:

1. Business function component (consisting of several microservices modules with each of them often implemented as a container), which embody the application logic (e.g., interacting with data, performing transactions, etc.), thus forming the application code.
2. Infrastructure component (containing computer, networking, and storage resources) whose constituents can be provisioned using infrastructure as code.
3. Service mesh component (implemented through a combination of control plane modules and service proxies), which provides application services, enforces policies (e.g., authentication and authorization), and contains application services code and policy as code.

4. Monitoring component (modules involved in ascertaining the parameters that indicate the health of the application), which performs functions (e.g., log aggregation, the generation of metrics, the generation of displays for dashboard, etc.) and contains observability as code.

The distribution of policy and observability code types within the components of the service mesh are as follows:

- Proxies (ingress, sidecar, and egress): These house encoding policies related to session establishment, routing, authentication, and authorization functions.
- Control plane of the service mesh: This houses code for relaying telemetry information from services captured and sent by proxies to specialized monitoring tools, authentication certificate generation and maintenance, updating policies in the proxies, monitoring overall configuration in the service orchestration platform for generating new proxies, and deleting obsolete proxies associated with discontinued microservices.
- External modules: These house modules that perform specialized functions at the application and enterprise levels (e.g., such as the centralized authorization or entitlement server, the centralized logger, monitoring/alerting server status through dashboards, etc.) and build a comprehensive view of the application status. These modules are called by code from the proxies or the control plane.

### 4.2 CI/CD Pipeline for Application Code and Application Services Code

Application code and application services code reside in the container orchestration and resource management platform, and the CI/CD software that implements the workflows associated with it usually resides in the same platform. This pipeline should be protected using the steps outlined in Section 4.6, and the application code under the control of this pipeline should be subject to the security testing described in Section 4.8. Additionally, the orchestration platform on which the application resides should itself be protected using a runtime security tool (e.g., Falco) [27] that can read OS kernel logs, container logs, and platform logs in real-time and process them against a threat detection rules engine to alert users of malicious behavior (e.g., creation of a privileged container, reading of a sensitive file by an unauthorized user, etc.). They usually come with a set of default (predefined) rules over which custom rules can be added. Installing them on the platform spins up agents for each node in the cluster, which can monitor the containers running in the various pods of that node. The advantage of this type of tool is that it complements the existing platform's native security measures, such as access control models and pod security policies, that prevent violations of security by actually detecting them when they occur [27].

### 4.3. CI/CD Pipeline for Infrastructure as Code

The conventional approach to allocating infrastructure for applications consists of initially provisioning compute and networking resources with configuration parameters and ongoing tasks such as patch management (e.g., OS and libraries), establishing conformity to compliance regulations (e.g., data privacy), and making drift (where the current configuration no longer provides the intended operational state) correction.

Infrastructure as code (IaC) is a declarative style of code that encodes computer instructions that encapsulate the parameters necessary to deploy virtual infrastructure on a public cloud service or private data center via a service's management APIs [28]. In other words, infrastructures are defined in a declarative way and versioned using the same source code control tools (e.g., GitOps) used for the application code. Depending on the particular IaC tool, this language can either be a scripting language (e.g., JavaScript, Python, TypeScript, etc.) or a proprietary configuration language (e.g., HCL) that may or may not be compatible with standardized languages (e.g., JSON). The basic instructions consist of telling the system how to provision and manage infrastructure (whether that is an individual compute instance or a complete server, such as physical servers or virtual machines), containers, storage, network connections, connection topology, and load balancers. [29]. In some cases, the infrastructure may be short-lived or ephemeral, and the lifespan of the infrastructure (whether immutable or mutable) does not warrant continued configuration management. Provisioning could be tied to individual commits of application code using tools that can connect application code and infrastructure code in a way that is logical, expressive, and familiar to development and operations teams, where application code increasingly defines the infrastructure resource requirements for a cloud application [30].

IaC thus involves codifying all software deployment tasks (allocation of type of servers, such as bare metal, VMs or containers, resource content of servers) and the configuration of these servers and their networks. The software containing this code type is also called a resource manager or deployment manager. In other words, IaC software automates the management of the whole IT infrastructure life cycle (provisioning and de-provisioning of resources) and enables a programmable infrastructure. The integration of this software as part of the CI/CD pipeline not only results in agile deployment and maintenance but also in a robust application platform that is secure and meets performance needs.

### 4.3.1 Protection for IaC

When infrastructure is code as in IaC, it can include bugs and oversights that can potentially become vulnerabilities and, therefore, be exploited just as in application code. Thus, protecting the IaC is protecting the infrastructure definitions and eventually the deployment environment. Any piece of IaC has to be scanned for potential vulnerabilities before it enters the GitOps and is merged.

In addition, the assurance of a secure application platform can be obtained only if there is a methodical drift management process in place. This assurance can be obtained only if the architecture defined in IaC is what actually exists in the deployed environment since this equivalence could be altered by an inadvertent or intentional change made through a console or CLI – thus bypassing the IaC. Ensuring this equivalence has to be done immediately after deployment and periodically during runtime as any change to the architecture could result in the introduction of security design flaws and may require making changes to IaC.

### 4.3.2 Distinction Between Configuration and Infrastructure

Infrastructure is often confused with configuration [30], which maintains computer systems, software, dependencies, and settings in a desired, consistent state. For example, putting a newly purchased server onto a rack and connecting it to the switches so that it is connected to the existing networks (or launching a new virtual machine and assigning network interfaces to it) belongs to the definition of "infrastructure." In contrast, after the server is launched, installing an HTTPS server and configuring it belongs to configuration management.

## 4.4 CI/CD Pipeline for Policy as Code

Policy as code involves codifying all policies and running them as part of the CI/CD pipeline so that they become an integral part of the application runtime. Examples of policy categories include authorization policies, networking policies, and implementation artifact policies (e.g., container policies). Policy management capabilities in a typical "policy as code software" may come with a set of predefined policy categories and policies and also support the definition of new policy categories and associated policies by providing policy templates [31]. *The due diligence required for policy as code is that it should provide protection against all known threats that are relevant for the application environment including the infrastructure, and this can be ensured only if that code is periodically scanned and updated with appropriate changes to counter the threats relevant to the application class (e.g., web application) and the hosting infrastructure.* Some examples of policy categories and associated policies are given in Table 1 below.

**Table 1: Policy Categories and Example Policies**

| Policy Category | Example Policies |
|---|---|
| Networking policies and zero trust policies | • Blocking designated ports<br>• Designating ingress host names<br>• In general, all network access control policies |
| Implementation artifact policies (e.g., container policies) | • Hardening of servers, vulnerability scans for base images<br>• Ensuring that containers do not run as root<br>• Blocking privilege escalation for containers |
| Storage policies | • Set persistent volume sizes<br>• Set persistent volume reclaim policies |
| Access control policies | • Ensure that policies cover all data objects<br>• Ensure that policies cover all roles for administrative and application access<br>• Ensure that data protection policies cover data at rest, data in transit, and data in use |

| Policy Category | Example Policies |
|---|---|
| | • Ensure that policies of all types do not have conflicts |
| Supply chain policies | • Allow only approved container registries<br>• Allow only certified libraries |
| Audit and accountability policies | • Ensure that there are policies associated with audit and accountability functions |

The policies defined in the "policy as code software" may translate into the following in the application infrastructure runtime configuration parameters:

- Policy-enforcing executable (e.g., WASM in service proxies)
- Triggers for calling an external policy decision module (e.g., calling an external authorization server for an allow/deny decision based on the evaluation of access control policies relevant to the current access request)
- It may also impact the IaC to ensure that appropriate resources are provisioned in the deployment environment to enforce security, privacy, and compliance requirements.

### 4.5 CI/CD Pipeline for Observability as Code

Observability as code deploys a monitoring agent in each of the application's service components to collect the three types of data (described in Section 4.1), send them to specialized tools that correlate them, perform analysis, and display the analyzed consolidated data on dashboards to present an overall application-level picture [32]. An example of such consolidated data are log patterns, which provide a view of log data that is presented after the log data are filtered using some criterion (e.g., a service or an event). The data are grouped into clusters based on common patterns (e.g., based on timestamp or range of IP addresses) for easy interpretation. Unusual occurrences are identified, and those findings can then be used to steer and accelerate further investigation [33].

### 4.6 Securing the CI/CD Pipeline

There are some common implementation issues to be addressed for CI/CD pipelines irrespective of code type. Securing the processes involves the assignment of roles for operating the build tasks. Automation tools (e.g., Git Secrets) are available for this purpose. The following security tasks should be considered as a minimum for securing the CI/CD pipeline:

- Harden servers hosting code & artifact repositories
- Secure the credentials used for accessing repositories such as authorization tokens and for generating pull requests
- Controls on who can check-in and check-out in container Image registries since they are the storage for artifacts produced by CI pipeline and serve as bridges between CI and CD pipelines
- Logging all code and build update activities

- If build or test fails in CI pipeline - Send build reports to developers and stop further pipeline tasks. The code repositories should be configured to automatically block all pull requests from CD pipeline [34]
- If audit fails – Send build reports to security team and stop further pipeline tasks
- Ensuring that developers can only access the application code and not any of the five pipeline code types
- During the build and release process, signing the release artifact during each required CI/CD stage (preferably multi-party signing)
- During production release, verify that all required signatures (generated with multiple phase keys) are present to ensure that no one bypasses the pipeline.



### 4.7 Workflow Models in CI/CD Pipelines

The next common issue involves workflow models. All CI/CD pipelines can have two types of workflow models, which depend on the automated tools that are deployed as part of the pipeline:

1. Push-based model
2. Pull-based model

In the CI/CD tools that support the push-based model, changes made in one stage or phase of the pipeline trigger changes in the subsequent stages and phases. For example, through a series of encoded scripts, the new builds in the CI system trigger changes to the CD portion of the pipeline and thus change the deployment infrastructure (e.g., Kubernetes cluster). The security downside of using the CI system as the basis for change in deployments is the possibility of exposing credentials outside of the deployment environment in spite of best efforts to secure the CI scripts, which operate outside of the trusted domain of the deployment infrastructure. Since CD tools have the keys to production systems, push-based models are rendered insecure.

In a pull-based workflow model, an operator that pertains to the deployment environment (e.g., Kubernetes Operator, Flux, ArgoCD) pulls new images from inside of the environment as soon as the operator observes that a new image has been pushed to the registry. The new image is pulled from the registry, the deployment manifest is automatically updated, and the new image is deployed in the environment (e.g., cluster). Thus, the convergence of the actual deployment infrastructure state with the state declaratively described in the Git deployment repository is achieved. Additionally, the deployment environment credentials (e.g., cluster credentials) are not exposed outside of the production environment. Therefore, a pull-based model, which typically uses a GitOps repository for storing the source code and builds, is highly recommended.

#### 4.7.1 GitsOps Workflow Model for CI/CD – A Pull-based Model

The GitOps workflow model is an improvement on the CI/CD pipeline (for the delivery portion of the pipeline), which uses a pull-based workflow model instead of the push-based model supported by many CI/CD tools. In this model, the CI portion of the pipeline is unchanged since the CI engine (e.g., Jenkins, GitLab CI) is still used for creating builds for the changed code,

regression testing, and integrating/merging with the main source code in the relevant repositories, though it is not used to trigger continuous delivery (push updates directly) in the pipeline. Instead, a separate GitOps operator manages the deployment based on updates to the main (trunk) code.

An operator (e.g., Flux, ArgoCD) is an actor managed by an orchestration platform and can inherit the cluster's configuration, security, and availability. The use of this actor improves security because an agent that lives inside of the cluster listens for updates to all code and image repositories that it is allowed to access and pulls images and configuration updates into the cluster. The pull approach used by the agent has the following security features:

- ONLY carry out operations permitted by authorization policies defined in the orchestration platform; trust is shared with the cluster and not managed separately
- Bind natively to all orchestration platform objects, and know whether operations have completed or need to be retried

### 4.8 Security Testing – Common Requirement for CI/CD Pipelines for All Code Types

The last common issue is security testing. Whatever the code type is (e.g., application service, Iac, Pac or observability), the CI/CD pipelines of DevSecOps for microservices-based infrastructure with service mesh should include application security testing (AST) enabled by either automated tools or offered as a service. These tools analyze and test applications for security vulnerabilities. According to Gartner, there are four main AST technologies [35]:

1. Static AST (SAST) tools – Analyze an application's source, bytecode, or binary code for security vulnerabilities, typically at the programming and/or testing software life cycle (SLC) phases. Specifically, this technology involves techniques that look through the application in a commit and analyze its dependencies [36]. If any dependencies contain issues or known security vulnerabilities, a commit will be marked as insecure and will not be allowed to proceed to deployment. This can also include finding hardcoded passwords/secrets in code that should be removed.

2. Dynamic AST (DAST) tools – Analyze applications in their dynamic, running state during testing or operational phases. They simulate attacks against an application (typically web-enabled applications, services, and APIs), analyze the application's reactions, and determine whether it is vulnerable. In particular, DAST tools go one step further than SAST and spin up a copy of the production environment inside the CI job in order to scan the resulting containers and executables [36]. The dynamic aspect helps the system catch dependencies that are being loaded at launch time, such as those that would not be caught by SAST.
3. Interactive AST (IAST) tools – Combine elements of DAST with the instrumentation of the application under test. They are typically implemented as an agent within the test runtime environment (e.g., instrumenting the Java Virtual Machine [JVM] or .NET CLR) that observes operations or identifies and attacks vulnerabilities.
4. Software composition analysis (SCA) tools – Used to identify open-source and third-party components in use in an application, their known security vulnerabilities, and typically adversarial license restrictions.

### 4.8.1 Functional and Coverage Requirements for AST tools

In general, the overall metrics that testing tools (including the specific class of AST tools) should satisfy are [37]:

- Increase the quality of application releases by identifying security, privacy, and compliance gaps.
- Integrate with the tools that developers are already using.
- Be as few test tools as possible but provide the necessary coverage risk.
- Lower-level unit tests at the API and microservices level should have sufficient visibility to determine coverage.
- Include higher-level UI/UX and system tests.
- Have deep code analysis capabilities to detect runtime flaws.
- Increase the speed at which the releases can be done.
- Be cost-efficient.

The functional requirements for AST tools in particular include performing the following types of scans [35]:

- **Vulnerability scans:** Probe applications for security weaknesses that could expose them to attacks.
- **Container image scans:** Analyze the contents and build process of a container image in order to detect security issues, vulnerabilities, or deficient practices (e.g., hardcoded passwords/secrets)
- **Regulatory/compliance scans:** Assess adherence to specific compliance requirements.

The vulnerability scans are to be performed whenever the code in the source code repository is revised to ensure that the current revision does not contain any vulnerable dependencies [38].

The desirable features of AST tools and/or services, along with techniques for behavioral analysis, are [35]:

- Analyze source, byte, or binary code
- Observe the behavior of apps to identify coding, design, packaging, deployment, and runtime conditions that introduce security vulnerabilities.

Scanning *application code* for security vulnerabilities and misconfiguration as part of CI/CD pipeline tasks should involve the following artifacts:

- Container images should be scanned for vulnerabilities.

- After the container is built from a base-image (that is scanned as stated above), the container's file system should be scanned for both vulnerabilities and misconfigurations.
- Git repositories (containing application source code) should be scanned for both vulnerabilities and misconfigurations.

Container images include OS packages (e.g., Alpine, UBI, RHEL, CentOS, etc.) and language-specific packages (e.g., Bundler, Composer, npm, yarn, etc.).

Scanning *infrastructure as code* for security vulnerabilities reduces the operations workload by preventing those vulnerabilities from making it to production, although it cannot replace checking for runtime security since the risk of drift will always exist. However, the reasons for all post-deployment (runtime) changes to the architecture (due to drift) must be analyzed and addressed by pushing appropriate updates to the IaC so that it becomes part of the pipeline and does not reoccur in subsequent deployments. This approach facilitates the use of runtime checks for remediating security design flaws.

The infrastructure-as-code files can be found in the following:

- The container orchestration platform itself to facilitate deployments (e.g., Kubernetes YAML infrastructure-as-code files).
- The dedicated infrastructure-as-code files found as part of CI/CD pipeline software (e.g., HashiCorp Terraform infrastructure-as-code files, AWS CloudFormation infrastructure-as-code files).

*Application services code*, *policy-as-code*, and *observability-as-code files* can be found in the data plane and control plane components of the dedicated application services component (e.g., service mesh) and should be scanned for both security vulnerabilities (e.g., information leakage in authorization policies) and misconfiguration.

### 4.9 Benefits of DevSecOps Primitives to Application Security in the Service Mesh

The benefits of DevSecOps include:

- Better communication and collaboration between various IT teams especially between developers, operations and security teams and other stakeholders. Results in better productivity [39]
- Streamlined software development, delivery and deployment process – less downtime, faster time to market and lower infrastructure & operational costs and efficiency due to automation [39]
- Reduction of attack surfaces by implementing zero trust which also restricts lateral movement and thus prevents attack escalation. This is further facilitated by continuous monitoring with modern behavior prevention capabilities.

- Security Benefits: Better security through validation of every request monitoring, alerts and feedback mechanisms because of Observability as code. These are described in more detailed in the following paragraphs. Specific capabilities include:
  (a) Runtime - Killing a malicious container
  (b) Feedback - to the right repository due to an errant program to update code and re-trigger the pipelines
  (c) Monitor new and terminated services and adjust associated services (e.g., service proxy)
  (d) Enable Security assertions: - Non-bypassable – by proxies executing in same space, secure sessions, robust authentication & authorization and secure state transitions
- Enable Continuous Authorization to operate (C-ATO) described in detail at the end of this section

Validation of each request and the feedback mechanisms mentioned above are further described below:

- Validation of every request: Every request from a user or client application (service) is authenticated and authorized (using mechanisms such as OPA or any external authorization engine or admission controllers [40] that are integral parts of the platform). While authorization engines provide application domain-specific policy enforcement, admission controllers provide platform-specific policy relating to end-point objects of a specific platform (e.g., pods, deployment, namespaces). Specifically, admission controllers mutate and validate. Mutating admission controllers parse each request and make changes to the request (mutate) before forwarding it down the chain. An example is setting default values for specifications that are not set by a user in the request so as to ensure that workloads running on the cluster are uniform and follow a particular standard defined by the cluster administrator. Another example is adding a specific resource limit for the pod (if the resource limits are not set for that pod) and then forwarding it down the chain (mutate the request by adding this field if it is not present in the request). By doing so, all of the pods in the cluster will always have a resource limit set according to a specification unless explicitly stated. Validating admission controllers reject requests that do not follow a particular specification. For example, none of the pod requests can have security context set to run as root user [40].
- Feedback mechanisms:
  - Some remediation for issues discovered at runtime may have to be handled or fixed at the source code. There should be a process for automatically opening an issue against the right code repository to fix the problem and re-trigger the DevSecOps pipeline.
  - Provide feedback loops to the application-hosting platform (e.g., a notification to kill a pod that contains a malicious container).
  - Provide proactive dynamic security by monitoring application configuration (e.g., monitoring new pods/containers introduced into the application and generating and injecting proxies to take care of their secure communication needs).

- Enable several security assertions regarding the application: non-bypassable (i.e., policies always enforced under all usage scenarios), trusted and untrusted portions of the overall application code, absence of credential and privilege leaks, trusted communication paths, and secure state transitions.
- Enable assertions regarding performance parameters (e.g., network resilience parameters, such as continued operation under failures, redundancy, and recoverability features).
- Overall, faster incorporation of feedback, results in quicker software improvements

### 4.10 Leveraging DevSecOps for Continuous Authorization to Operate (C-ATO)

In the reference platform, the runtime status or execution state of the entire application system is due to a combination of executions of infrastructure code (e.g., networking routes for inter-service communication, resources provisioning code), policy code (e.g., code that specifies authentication and authorization policies), and session management code (e.g., code that establishes an mTLS session, code that generates JWT tokens) as revealed by the execution of observability as code. The observability as code of the service mesh relays the output from the execution of infrastructure, policy, and session management codes during runtime to various monitoring tools that generate applicable metrics and log aggregation tools and tracing tools, which in turn relay their output to a centralized dashboard. The analytics that are integral to the output of these tools enable system administrators to obtain a comprehensive global view of the runtime status of the entire application system. It is the runtime performance of a DevSecOps platform enabled through continuous monitoring with zero trust design features that provides all of the necessary security assurance for cloud-native applications.

The activities in the DevSecOps pipelines that enable continuous ATO are:

- Checking for compliant code: The following codes can be checked for compliance with a Risk Management Framework

(a) IaC – Generate network routes, Resource provisioning
(b) Policy as Code – Encodes AuthN and AuthZ policies
(c) Session Management Code – mTLS session, JWT tokens
(d) Observability code

Specific risk assessment features include the capability to generate actionable tasks, specify code-level guidance, and test plans for verifying compliance [38]. In addition, risk assessment tools can provide complete traceability for all of the artifacts displayed in the dashboard, as well as the reporting capabilities needed for continuous ATO.

- Dashboard that displays runtime status: Provides alerts and feedback necessary to fix security and performance bottleneck issues (that impact availability) by the process that triggers new pipelines. In addition, dashboard generation tools have features that enable system administrators to analyze macro-level features, as well as dynamically change the composition of the artifacts to be displayed based on the evolving system and consumer needs of the environment in which the application operates.

# 5 Summary and Conclusion

This document provides comprehensive guidance for the implementation of DevSecOps primitives for a reference platform hosting a cloud-native application. It includes an overview of the reference platform and describes the basic DevSecOps primitives (i.e., CI/CD pipelines), its building blocks, the design and execution of the pipelines, and the role of automation in the efficient execution of workflows in CI/CD pipelines.

The architecture of the reference platform – in addition to the application code and the code for providing application services – consists of functional elements for infrastructure, runtime policies, and continuous monitoring of the health of the application, which can be deployed through declarative codes with separate CI/CD pipelines types. The runtime behaviors of these codes, the benefits of the implementation for high-assurance security, and the use of the artifacts within the pipelines for providing a continuous authority to operate (C-ATO) using risk management tools and dashboard metrics are also described.