Withdrawn Draft

Warning Notice

The attached draft document has been withdrawn and is provided solely for historical purposes. It has been superseded by the document identified below.

Withdrawal Date	March 8, 2022
Original Release Date	September 29, 2021

Superseding Document

Status	Final
Series/Number	NIST Special Publication 800-204C
Title	Implementation of DevSecOps for a Microservices-based Application with Service Mesh
Publication Date	March 2022
DOI	https://doi.org/10.6028/NIST.SP.800-204C
CSRC URL	https://csrc.nist.gov/publications/detail/sp/800-204c/final
Additional Information	

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

Draft NIST Special Publication 800-204C

Implementation of DevSecOps for a Microservices-based Application with Service Mesh

Ramaswamy Chandramouli

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-204C-draft

National Institute of Standards and Technology
U.S. Department of Commerce

Draft NIST Special Publication 800-204C

Implementation of DevSecOps for a Microservices-based Application with Service Mesh

Ramaswamy Chandramouli
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-204C-draft

September 2021

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
James K. Olthoff, Performing the Non-Exclusive Functions and Duties of the Under Secretary of Commerce for Standards and Technoogy & Director, National Institute of Standards and Technology

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 *et seq.*, Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

National Institute of Standards and Technology Special Publication 800-204C
Natl. Inst. Stand. Technol. Spec. Publ. 800-204C, 40 pages (September 2021)
CODEN: NSPUE2

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-204C-draft

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. All NIST Computer Security Division publications, other than the ones noted above, are available at http://csrc.nist.gov/publications.

Public comment period: *September 29, 2021* through *November 1, 2021*

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: sp800-204c-comments@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems.

Abstract

Cloud-native applications have evolved into a standardized architecture consisting of multiple loosely coupled components called microservices (implemented as containers), supported by code for providing application services called service mesh. Both of these components are hosted on a container orchestration and resource management platform, which is called a reference platform in this document. Due to security, business competitiveness, and its inherent structure (loosely coupled application components), this class of applications needs a different application development, deployment, and runtime paradigm. DevSecOps (consisting of three acronyms for Development, Security, and Operations, respectively) has been found to be a facilitating paradigm for these applications with primitives such as Continuous Integration, Continuous Delivery, and Continuous Deployment (CI/CD) pipelines. These pipelines are workflows for taking the developer's source code through various stages, such as building, testing, packaging, deployment, and operations supported by automated tools with feedback mechanisms. In addition to the application code, and the code for application services (service mesh), the architecture has functional elements for infrastructure (computing, networking, and storage resources), runtime policies (authentication, authorization etc.) and continuous monitoring of the health of the application (Observability), which can be deployed through declarative codes. Thus, separate CI/CD pipelines can be created for all of the five code types.

The objective of this document is to provide guidance for the implementation of DevSecOps primitives for a reference platform hosting a cloud-native application with the functional elements described above. The benefits of this approach for high security assurance and for enabling continuous authority to operate (C-ATO) are also discussed.

Keywords

container orchestration and resource management platform; DevSecOps; CI/CD pipelines; infrastructure as code; policy as code; observability as code; GitOps; workflow models; static AST; dynamic AST; interactive AST; SCA.

Acknowledgments

The author would like to express their first thanks to Mr. David Ferraiolo of NIST for initiating this effort to provide a targeted guidance for the implementation of DevSecOps primitives for the development, deployment, and monitoring of services in microservices-based applications with service mesh infrastructure. Sincere thanks to Mr. Nicolas Chaillan, CSO US Air Force, for a detailed and insightful review and feedback. Thanks are also due to Zack Butcher of Tetrate, Inc. for suggesting the title for this document. The author also expresses thanks to Isabel Van Wyk of NIST for her detailed editorial review.

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or

b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:

   i. under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or
   ii. without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: sp800-204c-comments@nist.gov

Executive Summary

Cloud-native applications have evolved into a standardized architecture consisting of the following components:

- Multiple loosely coupled components called microservices (implemented as containers)
- An application services infrastructure called Service Mesh (providing services such as secure communication, authentication, and authorization for users, services, and devices)

Due to security, business competitiveness, and its inherent structure (loosely coupled application components), this class of applications needs a different application, deployment, and runtime monitoring paradigm – collectively called the software life cycle paradigm. DevSecOps (consisting of three acronyms for Development, Security, and Operations, respectively) is one of the facilitating paradigms for the development, deployment, and operation of these applications with primitives such as Continuous Integration, Continuous Delivery, and Continuous Deployment (CI/CD) pipelines.

CI/CD pipelines are workflows for taking the developer's source code through various stages, such as building, functional testing, security scanning for vulnerabilities, packaging, and deployment supported by automated tools with feedback mechanisms. In addition to the application code and the code for providing application services, this architecture may be made up of functional elements for infrastructure, runtime policies (such as zero trust policy), and continuous monitoring of the health of the application, the last three of which can be deployed through declarative codes. Thus, separate CI/CD pipelines can be created for all five code types. The runtime behavior of each of these code types is also described to highlight the roles that they play in the overall execution of the application.

Though cloud-native applications have a common architectural stack, the platform on which the components of the stack run may vary. The platform is an abstraction layer over a physical (bare metal) or virtualized (e.g., virtual machines, containers) infrastructure. In this document, the chosen platform is a container orchestration and resource management platform (e.g., Kubernetes). 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*.

The objective of this document is to provide guidance for the implementation of DevSecOps primitives for the reference platform. The benefits of this implementation for high security assurance and the use of the artifacts within the pipelines for providing continuous authority to operate (C-ATO) using risk management tools and dashboard metrics are also described.

Table of Contents

1 Introduction

Cloud-native applications are made up of multiple loosely coupled components (called microservices 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, such as security. The incorporation of zero trust principles into this class of application provides techniques where access to all protected resources is enforced through identity-based protection, as well as 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, and 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. Thus, DevSecOps has the necessary primitives and other building blocks to meet the design goals of cloud-native applications.

1.1 Scope

DevSecOps primitives can, in theory, 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 [1]:

- 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.).

The reference platform, along with DevSecOps implementation components, can therefore be looked upon as having the following functional elements or code types:

1. The application code, which contains the application logic for transactions and database access
2. The service mesh code, which provides application services such as network routes, network resiliency services (e.g., load balancing, retries etc.), and security services
3. Infrastructure (consisting of computing, networking, and storage resources) as code
4. Policy as code for generating 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 for (a) triggering software related to logging, tracing (communication pathways involved in executing application request), and monitoring for recording all transactions and for (b) keeping track of application states during runtime.

This document covers the implementation of pipelines or workflows associated with all five code types listed above. Out of them, the infrastructure as code, policy as code, and observability as code belong to a special class called declarative code. A declarative code is written in a special purpose language called declarative language that declares the requirements, 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

DoD Enterprise DevSecOps Initiative: The Department of Defense (DOD) Enterprise DevSecOps initiative spans many open-source projects with many state-of-practice ingredients. The artifacts developed under this initiative are used by 12 U.S. federal agencies, five nations, and a substantial number of DoD contractors and vendors. The open-source projects under this initiative include but are not limited to [2]:

- The Iron Bank, a repository of DOD-vetted hardened container images
- Platform One, the DevSecOps platform that the department created for software deployments from jets to bombers to space to clouds; it is based on the concept of continuous authority to operate (cATO), which updated the DOD's authorization process to accommodate the speed and frequency of modern continuous software deployments

The DoD Enterprise DevSecOps initiative has the following state-of-practice ingredients:

1. **Abstracted**: To avoid drifts, be agnostic to the environment (e.g., cloud, on-premise, classified, disconnected), and prevent lock-ins with cloud or platform layers, CNCF-compliant Kubernetes and OCI-compliant containers are leveraged – open-source stacks with U.S. observation of code and continuous scanning.
2. **GitOps/Infrastructure as Code (IaC)**: No drift, everything is code (including configuration, networking etc.), and the entire stack is automatically instantiated.
3. **Continuous Integration/Continuous Delivery pipeline (CI/CD)**: Fully containerized and using Infrastructure as Code (IaC)
4. **Hardened Containers**: Hardened "Lego blocks" to bring options to development teams (one size fits all lead to shadow IT)
5. **Software Testing**: Mandated high-test coverage
6. **Baked-in Security**: Mandated static/dynamic code analysis, container security, bill of material (supply chain risk), etc.
7. **Continuous Monitoring**:
    - **Centralized logging and telemetry**
    - Automated alerting
    - **Zero trust**, leveraging service mesh as sidecar (part of SCSS) down to the container level
    - **Behavior detection** (automated prevention leveraging AI/ML capabilities)
    - CVE signatures scanning
8. **Chaos engineering**: Dynamically kills/restarts container with moving target defense

1.3 Target Audience

Since DevSecOps primitives span development (build and test for security, package, delivery, deployment, continuous monitoring) and 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 background information for the contents of this document:

- Special Publication (SP) 800-204, *Security Strategies for Microservices-based Application Systems* [3], discusses the characteristics of microservices-based applications and the overall security requirements and strategies for addressing those requirements.

- SP 800-204A, *Building Secure Microservices-based Applications Using Service-Mesh Architecture* [4], 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) that uses service proxies that operate independent of the application code.
- SP 800-204B [5], *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 such as attribute-based access control (ABAC) that can be used to express a wide set of policies and is scalable in terms of user base, objects (resources), and deployment environment.

1.5 Organization of this document

The organization of the rest of the document is 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 four 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 (CAT).”
- Chapter 5 provides a 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 implemented as containers, a container orchestration and resource management platform is used for deployment, operations, and upgrading services. Kubernetes is one such open-source container orchestration and resource management platform that is widely in use.

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 as a means 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 higher 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 requests within the application), traffic control, caching, secure ingress, service-to-service (east/west traffic), and egress communication.

Taking the example of secure communications between Kubernetes containers, these are not natively secure since there is no easy way to enforce TLS between pods as this would require in maintaining hundreds of TLS certificates. 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, 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 do 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 data plane and the control plane.

2.2.1 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 layer 7 (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 [6,7].

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 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 [8] 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).

The third category of functions that service proxies perform sometimes 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.

2.2.2 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 [7] (components in the Istio service mesh that perform these functions are given in parentheses):

1. Service discovery and configuration of the Envoy sidecar proxies (Pilot)

2. Automated key and certificate management (Citadel)
3. API for policy definition and the gathering of telemetry data (Mixer)
4. Configuration ingestion for service mesh components (Galley)
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 (Sidecar Injector)

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.

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 generation, and observability services can be automated using declarative code that is part of DevSecOps pipelines. The development team can concentrate their efforts on efficient development paradigms, such as code modularity and structuring, without worrying about the security and management details of their implementation.

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 [9].

This section discusses the following aspects of DevSecOps:

- Organizational preparedness for DevSecOps
- Seamless evolution from DevOps to DevSecOps (for organizations that already have a DevOps team in place)
- 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 behavior, developing patches, fixing bugs, or scaling the application. This cross-functional team with expertise in three areas forms a critical success factor for introducing DevSecOps in an organization.

3.2 Seamless Evolution from DevOps to DevSecOps

DevOps 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. These pipelines are nothing but workflows that take the developer's source code through various stages, such as building, testing, packaging, and deployment supported by automated tools with feedback mechanisms. DevOps is primarily a development/deployment pipeline where security assurance is provided through the use of some basic static application security tools (SAST), dynamic security testing tools (DAST), and software composition analysis (SCA) tools. Thus, a DevOps platform only runs during the build and deployment phases of the software life cycle.

In a DevSecOps platform, security assurance is provided through built-in design features, such as zero trust, during the build and deployment phases in addition to the use of a comprehensive set of security testing tools, such as DAST and SCA tools. Security assurance is also provided during the runtime/operations phase by continuous behavior detection/prevention tools that 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 runtime/operations phase. DevSecOps pipelines also run in a production environment as opposed to DevOps, which runs in a separate environment for testing/staging since workflows associated with the latter terminate in the deployment phase.

Another distinguishing characteristic of a DevSecOps platform is that the security tools (e.g., SAST, DAST, and SCAs) are so tightly integrated with IDEs and other DevOps tools that they perform application security analysis, such as identifying vulnerabilities and bugs through efficient scanning in the background. This is often invisible to developers without making them call separate APIs for running these tools [10].

In summary, DevSecOps has the following distinguishing features as compared to DevOps:

- Security testing and the robust incorporation of security controls are integral to all pipelines instead of being relegated to a separate task or phase.
- The DevSecOps platform operates during runtime (in production), providing real-time assurance of security through correction mechanisms and 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 automation tools in the CI/CD pipeline

3.3.1 Concept of Pipelines and the CI/CD pipeline

DevSecOps is a methodology or framework for agile application development, deployment, and operations for cloud-native applications [11] and is made up of stages just like any other methodology. The sequence and flow of information through the stages is called workflow, where some stages can be executed in parallel while some 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” [12]. With pipelines, there is no need to individually write jobs for each stage of the process. Instead, there is only one job that starts from the initial stage, automatically triggers the activities 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. CI stands for the continuous integration stage. 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:

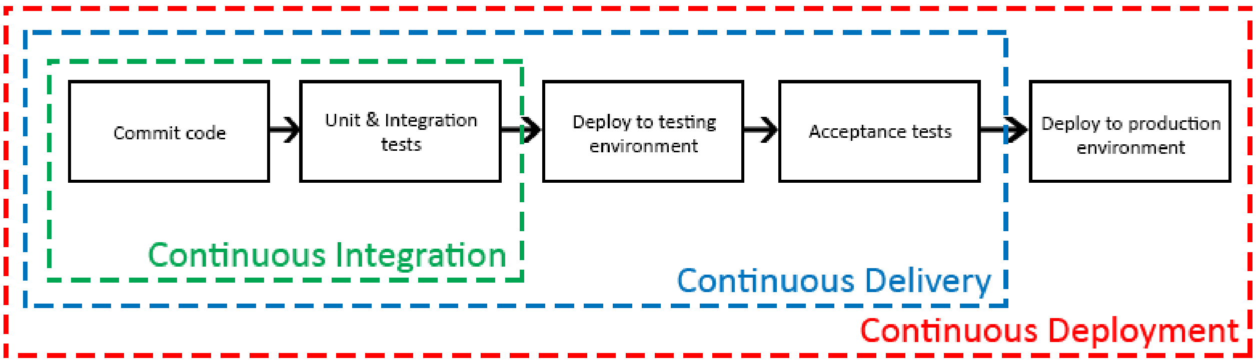

Figure 1: CI/CD pipeline workflow [13]

~~Figure 3.1. CI/CD pipeline workflow [13]~~

The unit and integration tests shown in the diagram use SAST, DAST, and SCA tools described in Section 3.2. These tests and associated tools are common to DevOps and DevSecOps.

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 [14].

Continuous delivery is the next 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 releases happen automatically [14], and changes to code are available to customers immediately after they are made. The distinction between continuous delivery and continuous deployment is shown in Figure 2 below.

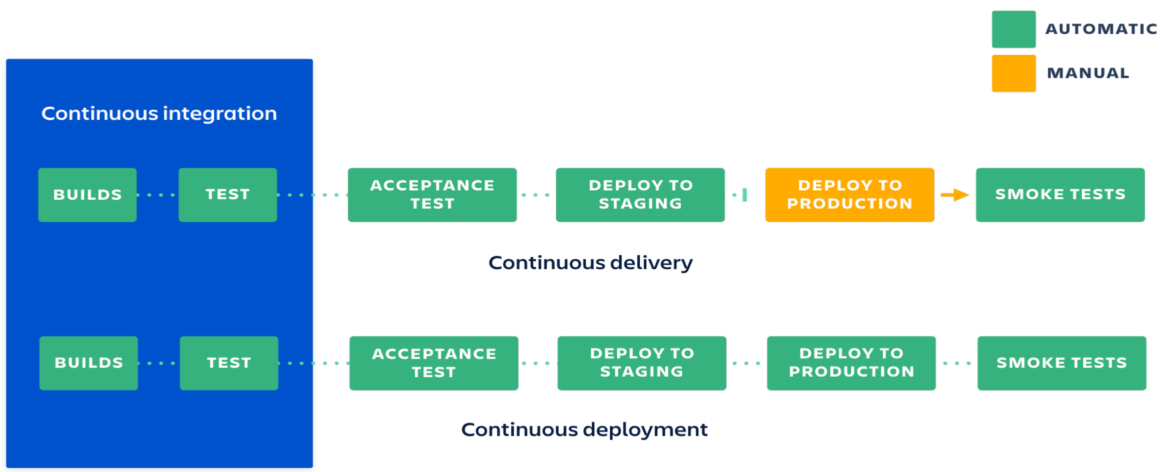

Figure 2 - Distinction between Continous Delivery and Continous Deployment [14]

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 needed are hosted (i.e., container orchestration and resource management platform, such as Kubernetes), while others need to be integrated into the application hosting platform through its API. An example of a former class of tool is Tekton, and examples for the latter class include Jenkins, Travis, Bamboo, and CircleCI. Some advantages of using the CI/CD tools that are native to the application hosting platform are:
  (a) It makes it easier to deploy, maintain, and manage the CI/CD tool itself.
  (b) 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 TaskRuns and PipelineRuns, respectively), can be created as

CustomResourceDefinitions (CRDs) built on top of resources native to the platform [15]. 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 tool offerings (e.g., GitLab, Jenkins X) integrate with a Git Repository – one for each application and for each environment (staging/production). All changes in application modules, infrastructure, or configuration are made and stored in these Git repositories. The CI/CD pipeline software is connected to the Git repositories through Git webhooks and gets activated on commits (push workflow model) or pull requests in 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 Designing 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 [16]:

- 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.
- Creating the deployment environment: Create the environment to deploy the application using an automated deployment tool.
- Creating the delivery pipeline: Create a pipeline that will automatically build and deploy the application using an automated pipeline tool.
- Test the code and execute the pipeline: Test the code in the pipeline using an automated testing tool, and execute the pipeline to complete the deployment of updated code.

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 building tool, which is usually 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 or 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 then deploys the package into the desired environment (e.g., staging, production etc.) [17].

The workflow of the CI/CD pipeline should not create the impression that there is no human element involved. The following teams provide input into the CI/CD pipeline [18]:

- Development team incorporates new features into the application
- Build or integration team creates new CI pipelines and introduces new CI tools and testing tools
- Security team conducts audits and patching
- Infrastructure team creates, maintains, and upgrades the infrastructure
- QA team develops integration test cases
- Deployment team/release team creates pipelines and packages for various environments (UAT/PreProd/Prod) and performs 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 etc.). Caution should be exercised during manual operations so that they do not block pipelines. Targets for mean time to production should be set up while also mitigating risks, such as insider threats, through the use of "merge requests" and multiple approvers in merge requests.

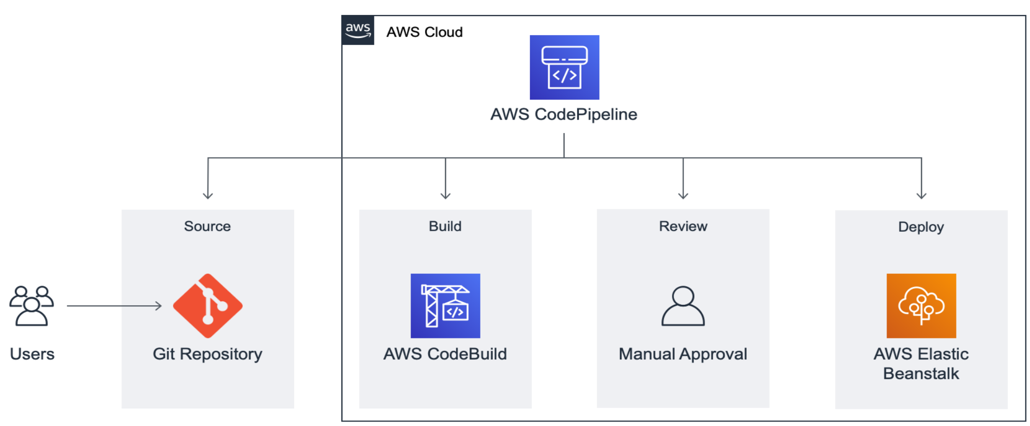

Figure 2 - Examples of Tools involved in CI/CD Pipeline Tasks

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 [19].

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
- Better utilization of the team's expertise by assigning routine tasks to the tools

The following strategies are recommended [22] for automation so as to facilitate better utilization of organizational resources and derive the greatest benefit in terms of an efficient, secure application environment.

<u>Choice of Activities to Automate</u>: 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 constrict operations

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 [22]
- Using key performance indicators (KPI) (e.g., mean time to identify faults or problems, rectify, or recover) as markers to refine the DevSecOps processes [22]
- 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, etc.) to determine the allocation of resources to DevSecOps processes

3.3.5 Requirements for Automation Tools in CI/CD Pipelines

The main utility of a performance testing tool is to identify the root cause of an issue, such as application response time or a bottleneck operation. The desirable features of this class of tool are:

- Easy integration into developers' CI/CD pipelines, which enables this class of tools to be an integral part of the DevSecOps process rather than one managed by a central platform engineering or site reliability engineering team that could become hamper deployments
- An accessible interface that can be used by all participants involved in the software release workflow and have features that analysts/engineers can use to perform insightful analyses into specific datasets
- Coverage for a wide variety of back-end infrastructure components in multiple clouds as well as on premises; this abstraction feature is now natively present in container orchestration and resource management platforms (e.g., Kubernetes) or through stacks (e.g., vSphere or OpenStack)

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) at 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.

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 (hybrid cloud, single public cloud, and multi-cloud).

The application architecture consists of other functional elements (in addition to elements for housing executable application code and providing application services) such as for infrastructure resources, runtime policies, and continuous monitoring of the health of the application, which can all be deployed through declarative codes. Hence, separate CI/CD pipelines can be created for all of these code types as well. These five code types will first be considered in the context of the reference platform 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 Code Types in Reference Platform and Associated CI/CD Pipelines

The constituent components of the code types in the reference platform are:

1. Microservices-based application component (implemented as a series of containers), which contains the application logic (e.g., interacting with data, performing transactions etc.) and 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 DevSecOps methodology for developing, testing, delivering, and deploying these five types of codes (i.e., application, application services, infrastructure, policy, and monitoring) requires the corresponding pipelines:

- CI/CD pipelines for 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, network resilience etc.
- CI/CD pipeline for infrastructure as code (IaC) – The process of writing the code for provisioning and configuring the steps of infrastructure resources to automate application deployment in a repeatable and consistent manner [23]. 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).
- CI/CD pipeline for policy as code – Describes many policies, including security policies, as executable modules [24]. 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 WASM, 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.
- CI/CD pipeline for 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, as well as 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 [25]. 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.

The policy and observability code types span the following components of the service mesh.

- 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 reside 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) [32] 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 [32].

4.3. CI/CD Pipeline for Infrastructure as Code

Infrastructure as code (IaC) 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 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.

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 correction (where the current configuration no longer provides the intended operational state).

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 [33]. Depending on the particular IaC tool, this language can either be a scripting language (e.g., Go, 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 unit of these instructions is called "configuration" and tells 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. [34]. 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 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 [35].

4.3.1 Comparison of Configuration and Infrastructure

Infrastructure is often confused with configuration [34], 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 HTTP server and configuring it belongs to configuration management. In physical data centers, specific teams purchase servers, install servers, and connect networking cables with the underlying infrastructure in mind.

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 [36].

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 • Designating ingress host names
Implementation artifact policies (e.g., container policies)	• Ensuring that containers do not run as root • Blocking privilege escalation for containers
Storage policies	• Set persistent volume sizes • Set persistent volume reclaim policies
Access control policies	• Ensure that policies cover all data objects • Ensure that there are no policy conflicts
Supply chain policies	• Allow only approved container registries • Allow only certified libraries

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)

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 [20]. 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 [21].

4.6 Securing the CI/CD Pipeline

There are some common implementation issues to be addressed for CI/CD pipelines irrespective of code type. The first – security – involves securing the artifacts, including hardening the servers and nodes used for generating the build and establishing the authenticity of the build components. 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 performed when securing the CI pipeline [26]:

- Authentication and validation in code and build repositories
- Logging all activities associated with code and build updates
- Sending build reports to developers and stopping everything if a build fails
- Sending build reports to security and stopping everything if the audit or validation fails

Securing the CD pipeline involves the following:

- Multi-party signing with TUF and in-toto to sign each CD pipeline phase
- Signing the final artifacts with multiple phase keys 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. A pull-based model, which typically uses a GitOps repository for storing the source code and builds, is therefore highly recommended.

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.

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 (eg. 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, 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 [27]:

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 [28]. 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.

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 [28]. The dynamic aspect helps the system catch dependencies that are being loaded at launch time, such as those 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 – are 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 [29]:

- Increase the quality of application releases
- 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
- 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 are that perform the following types of scans [30]:

- **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
- **Regulatory/compliance scans:** Assess adherence to specific compliance requirements

The vulnerability scans are to be performed whenever the code in GitHub is revised to ensure that the current revision does not contain any vulnerable dependencies [31].

The desirable features of AST tools and/or services, along with techniques for behavioral analysis, are [27]:

- 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.
- Filesystems should be scanned for both vulnerabilities and misconfigurations.
- Git repositories 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 runtime security since there will always be the risk of drift. The infrastructure-as-code files can be found in the following:

- In the container orchestration platform itself to facilitate deployments (e.g., Kubernetes YAML infrastructure-as-code files)
- In 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)

Both *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 [37]:

- Streamlined software delivery and deployment processes
- Reduction of the number of silos between IT groups
- Reduction of attack surfaces by implementing zero trust down to the container level
- Introduction of moving target defenses by threating containers as cattle versus pet and going back to an immutable state
- Reduction of lateral movement capabilities by leveraging zero trust and continuous monitoring with modern behavior prevention capabilities
- Increased communications between team members and other stakeholders
- Faster incorporation of feedback, resulting in quicker software improvements
- Automation of repetitive, manual tasks, leading to increased efficiency
- Less downtime and faster time to market
- Infrastructure for continuous software development and delivery (as well as admission controller and the use of OPA to add guard rails to the runtime)

As already stated, the primary goal of DevSecOps is to have a platform that ensures runtime security of microservices-based applications. Runtime security is assured through the following:

(a) <u>Validation of every request</u>: Every request from a user or client application (service) is authenticated and authorized (using mechanisms such as Open Policy Agent [OPA] or any external authorization engine), and admission controllers 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 [38].

(b) Feedback mechanisms:

- 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 (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)

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 within the DevSecOps pipelines within the service mesh context that enable continuous ATO are:

- Checking for compliant code: Checking whether the updated infrastructure, policy, session management, and observability codes are compliant with the chosen risk management framework (e.g., NIST Risk Management Framework) for the enterprise. This will involve tools with risk assessment features, such as the capability to generate actionable tasks, specify code-level guidance, and test plans for verifying compliance [31].

- Generating a dashboard that displays the runtime status: Using the runtime output of observability code for generating the dashboard that displays the runtime status of the overall application

Risk assessment tools should provide complete traceability for all of the artifacts displayed in the dashboard, as well as the reporting capabilities needed for continuous ATO.

Dashboard generation tools should 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 a 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, the last three of 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.

References

[1] Garrison J, Nova K (2017) *Cloud Native Infrastructure* (oreilly.com). Available at https://www.oreilly.com/library/view/cloud-native-infrastructure/9781491984291/

[2] Chaillan NM (2020) *DoD Enterprise DevSecOps Initiative and Platform One*. Available at https://software.af.mil/dsop/documents

[3] Chandramouli R (2019) Security Strategies for Microservices-based Application Systems. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-204. https://doi.org/10.6028/NIST.SP.800-204

[4] Chandramouli R, Butcher Z (2020) Building Secure Microservices-based Applications Using Service-Mesh Architecture. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-204A. https://doi.org/10.6028/NIST.SP.800-204A

[5] Chandramouli R, Butcher Z, Aradhna C (2021) (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-204B. https://doi.org/10.6028/NIST.SP.800-204B

[6] Agarwal G (2020) *How to Manage Microservices on Kubernetes With Istio.* Available at https://medium.com/better-programming/how-to-manage-microservices-on-kubernetes-with-istio-c25e97a60a59

[7] Kubernetes Advocate (2021) *Managing Microservices With Istio Service Mesh* in Kubernetes. Available at https://medium.com/avmconsulting-blog/managing-microservices-with-istio-service-mesh-in-kubernetes-36e1fda81757

[8] Ramakani A (2020) *Kong API Gateway – From Zero to Production.* Available at https://medium.com/swlh/kong-api-gateway-zero-to-production-5b8431495ee

[9] Srinath K (2020) *DevSecOps – Baking Security into Development Process*. Available at https://medium.com/faun/devsecops-baking-security-into-development-process-9579418ad9a7

[10] Rubinstein D (2020) *AppSec vs. DevSecOps, and what that means for developers*. Available at https://sdtimes.com/security/appsec-vs-devsecops-and-what-that-means-for-developers/

[11] Red Hat (2021) *What is DevSecOps?* Available at https://www.redhat.com/en/topics/devops/what-is-devsecops

[12] Red Hat (2021) *What is CI/CD?* Available at

https://www.redhat.com/en/topics/devops/what-is-ci-cd#:~:text=CI%2FCD%20is%20a%20method,continuous%20delivery%2C%20and%20continuous%20deployment

[13] Ali M (2018) *Continuous Release Practices are evolving, Here is our story.* Available at https://medium.com/the-telegraph-engineering/continuous-release-practices-are-evolving-here-is-our-story-2a4d164e9cac

[14] Pittet S (2021) *Continuous integration vs. continuous delivery vs. continuous deployment*. Available at https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[15] Wuestkamp K (2020) *K8s–native Jenkins-X and Tekton Pipelines*. Available at https://itnext.io/k8s-native-jenkins-x-and-tekton-pipelines-e2b5a61a1d22

[16] aws.amazon.com (2021) *Create Continuous Delivery Pipeline*. Available at https://aws.amazon.com/getting-started/hands-on/create-continuous-delivery-pipeline/?trk=gs_card

[17] aws.amazon.com (2021) *Setting up a CI/CD pipeline by integrating Jenkins with AWS CodeBuild and AWS CodeDeploy*. https://aws.amazon.com/blogs/devops/setting-up-a-ci-cd-pipeline-by-integrating-jenkins-with-aws-codebuild-and-aws-codedeploy/

[18] mylocaldevstack.pl (2019) *The future of DevOps – Assembly Lines*. Available at https://medium.com/@mylocaldevstack/the-future-of-devops-assembly-lines-40227546d750

[19] Socher R (2020) *Best Terraform Tutorial Guides: An Overview*. Available at https://faun.pub/best-terraform-tutorial-guides-an-overview-65a6fcee0a24

[20] Pariseau B (2020) Mendix dumps cluttered DevOps monitoring tools for Datadog. Available at https://searchitoperations.techtarget.com/news/252497697/Mendix-dumps-cluttered-DevOps-monitoring-tools-for-Datadog?track=NL-1841&ad=938089&asrc=EM_NLN_151848158&utm_medium=EM&utm_source=NLN&utm_campaign=20210315_Kong+Mesh%27s+built-in+Open+Policy+Agent+simplifies+IT+security+management

[21] Boutet R (2018) *Log Patterns: Automatically cluster your logs for faster investigation* (datadoghq.com). Available at https://www.datadoghq.com/blog/log-patterns/

[22] Soni A (2020) *GitOps: The Next Big Thing for DevOps and Automation!* Available at https://medium.com/searce/gitops-the-next-big-thing-for-devops-and-automation-2a9597e51559

[23] Fallon A (2020) *Understand the role of infrastructure as code in DevOps (*TechTarget.com). Available at

https://searchitoperations.techtarget.com/feature/Understand-the-role-of-infrastructure-as-code-in-DevOps?utm_campaign=20210809_The+next+DevSecOps+challenge%3A+People&utm_medium=EM&utm_source=NLN&track=NL-1841&ad=939963&asrc=EM_NLN_174809933

[24] Ahmed M (2020) Introducing Policy As Code: The Open Policy Agent (OPA). Available at https://www.magalix.com/blog/introducing-policy-as-code-the-open-policy-agent-opa

[25] Singh P (2021) Tackle Kubernetes observability with the right metrics. Available at https://searchitoperations.techtarget.com/tip/Tackle-Kubernetes-observability-with-the-right-metrics?track=NL-1841&ad=938191&asrc=EM_NLN_153034984&utm_medium=EM&utm_source=NLN&utm_campaign=20210322_DevSecOps+leaves+Excel+in+the+dust

[26] Devopscurry (2021) *Securing your CI/CD pipelines with DevSecOps in 2021*. Available at https://medium.com/devopscurry/securing-your-ci-cd-pipelines-with-devsecops-in-2021-1a6a6e34f2e7

[27] Gartner (2021) Gardner D, Horvath M, Zumerle D *Magic Quadrant for Application Security Testing*. Available at https://www.gartner.com/doc/reprints?id=1-262TXQZV&ct=210518&st=sb

[28] Lewkowicz J (2021) *A guide to automated testing providers*. Available at https://sdtimes.com/test/a-guide-to-automated-testing-providers/

[29] Lewkowicz J (2021) *Automated testing is a must in CI/CE pipelines*. Available at https://sdtimes.com/test/automated-testing-is-a-must-in-ci-cd-pipelines/

[30] Angel R (2020) *How to be a CI/CD Engineer.* Available at https://circleci.com/blog/how-to-be-a-cicd-engineer/

[31] Alexey K (2020) *Ultimate guide to CI/CD security and DevSecOps.* Available at https://circleci.com/blog/security-best-practices-for-ci-cd/

[32] Agarwal G (2020) *Kubernetes Security With Falco*. Available at https://betterprogramming.pub/kubernetes-security-with-falco-2eb060d3ae7d

[33] Marko K (2021) *Terraform cheat sheet: Notable commands, HCL and more*. Available at https://searchitoperations.techtarget.com/tip/Terraform-cheat-sheet-Notable-commands-HCL-and-more?utm_campaign=20210726_Infrastructure+as+code+still+a+big+security+buzz&utm_medium=EM&utm_source=NLN&track=NL-1841&ad=939808&asrc=EM_NLN_172629823

[34] Guo T (2021) *On DevOps 8: Infrastructure as Code: Introduction, Best Practices and How to choose the Right tool.* Available at https://medium.com/4th-coffee/on-devops-8-infrastructure-as-code-introduction-best-practices-and-choosing-the-right-tool-2c8f46d1f34

[35] Pulumi.com (2020) *Delivering Cloud Native Infrastructure as Code*. Available at https://cdn2.hubspot.net/hubfs/4429525/Content/Pulumi-Delivering-CNI-as-Code.pdf

[36] Chong T (2021) *Product In-Depth: Instantly Secure your Cloud Infrastructure with Policy-as-Code*. Available at https://www.magalix.com/blog/instantly-secure-your-cloud-infrastructure-with-policy-as-code?utm_medium=email&_hsmi=123795689&_hsenc=p2ANqtz-HbHGaLCOrCV_WXjrKE-JgWUWUX9Tg1s9ot5HSox2Ts1rgAhwquoZkBKqZYo-OcHJmlUHG8eFhC5Tfu7MW_Dn3i-g3Ng&utm_content=123795689&utm_source=hs_email

[37] Sheldon R (2021) *Top 30 DevOps interview questions and answers for 2021*. Available at https://whatis.techtarget.com/feature/Top-30-DevOps-interview-questions-and-answers?utm_campaign=20210712_New+Kubernetes+use+case%253A+Hacking&utm_medium=EM&utm_source=NLN&track=NL-1841&ad=939635&asrc=EM_NLN_170130891

[38] Prasad A (2020) *Kubernetes Admission Controllers: Request Interceptors.* Available at https://medium.com/cloudlego/kubernetes-admission-controllers-request-interceptors-47a9b12c5303