Foreword

This document is intended to act as a best practice guide for Department of Defense (DoD) programs that are interested in practicing Development, Security, and Operations (DevSecOps). The expected result of reading this document is for all DoD entities, including those selected as official 873 and 874 Agile Pilot programs, to have a sense of direction on where to begin their DevSecOps journey. This document is not intended to be an engineering guideline document nor a “cookie-cutter” solution to successfully implement DevSecOps.

Although the technological implementation of DevSecOps heavily relies on specific tools that provide crucial services in an automated pipeline, this document emphasizes the organizational cultural change necessary for stakeholders to use DevSecOps as a platform to provide the iterative, automated, and secure delivery of software.
# Table of Contents

*Foreword*  
1

*Glossary*  
3

1. **Introduction**  
   1.1 Purpose  
   1.2 Organization and Culture Change Prerequisites  
6

2. **Cultural Shift**  
   2.1 Organization Buy-In  
   2.2 Communication and Collaboration  
   2.3 “Shift-Left” Mentality to Create Checks and Balances  
   2.4 Beginning Roadmap  
      2.4.1 DevSecOps Assessment  
7

3. **Automation**  
   3.1 Deployment Pipeline  
   3.2 Continuous Integration (CI)  
      3.2.1 Testing  
   3.3 Continuous Delivery vs. Continuous Deployment (CD)  
   3.4 Monitoring  
      3.4.1 Pipeline Metrics  
      3.4.2 Infrastructure Metrics  
      3.4.3 Application Metrics  
10

4. **Infrastructure as Code**  
   4.1 Environments  
      4.1.1 Architectural Parity Between Environments  
   4.2 Containers and Container Orchestration  
      4.2.1 Pets vs. Cattle  
      4.2.2 Dockerfile Composition  
      4.2.3 Private/Internal Registry  
      4.2.4 Orchestration  
      4.2.5 Security  
   4.3 IaC Warnings  
17

5. **DevSecOps Anti-Patterns**  
   5.1 Over Emphasis of Pipeline Tools  
   5.2 Bringing on New Roles Without Implementing Organizational Changes  
   5.3 Using a Deployment Pipeline Without Traceability  
   5.4 Vendor Lock  
26

*Works Cited*  
29
This glossary is excerpted from DZone Inc.’s 48 DevOps Terms You Should Know (A DevOps Glossary) ¹

Glossary

- **Agile Software Development** – a software development methodology and philosophy focused on user feedback, software quality, and the ability to respond quickly to changes and new product requirements

- **Artifact Repository** – centralized storage for all binaries used during the build automation process

- **Commit** – a record of a change made to an artifact stored in a version control system with an accompanying description of what was modified

- **Container** – resource isolation at the OS—rather than machine—level, usually—in UNIX-based systems—in user space (Isolated elements vary by the containerization strategy used and often include file system, disk quota, CPU and memory, I/O rate, root privileges, and network access.)

- **Container Orchestration** – a strategy to manage the lifecycles of containers across various environments

- **Continuous Delivery** – a software engineering approach in which continuous integration, automated testing, and automated deployment capabilities allow software to be developed and deployed rapidly, reliably, and repeatedly with minimal human intervention (The trigger or decision to deliver to production is still manual.)

- **Continuous Deployment** – a software development practice in which every code change goes through the entire pipeline and is put into production automatically, resulting in many production deployments every day (It does everything that Continuous Delivery does, but the process is fully automated, and there’s no human intervention at all.)

- **Continuous Integration** – a software development process where a branch of source code is rebuilt every time code is committed to the source control system (The process is often extended to include the deployment, installation, and testing of applications in staging and production environments.)

- **Continuous Testing** – the process of executing unattended automated tests as part of the software delivery pipeline across all environments to obtain immediate feedback on the quality of a code build

- **Deployment** – the grouping of every activity that makes a program available for use and moving that program to the target environment

- **Deployment Pipeline** – an automated manifestation of a process used for moving software from version control into the hands of end users

¹ https://medium.com/@DZoneInc/48-devops-terms-you-should-know-a-devops-glossary-37e60fd23752
• **DevOps** – an IT organizational methodology where all teams in the organization, especially development and operations teams, collaborate on the development and deployment of software to increase software production agility and achieve business goals

• **Fail Fast** – a strategy in which something is tried, it fails, feedback is delivered quickly, it is adapted accordingly, and is tried again

• **Infrastructure as Code (IaC)** – the management and provisioning of the virtual infrastructure through version-controlled configuration files

• **Infrastructure as a Service (IaaS)** – a self-service computing, networking, and storage utility provided on demand over a network

• **Integration Testing** – testing that occurs after unit testing but before validation testing, where individual software components are combined and tested as a single group to ensure they work as a whole

• **Issue Tracking** – a process that allows project stakeholders to track the flow of defects and new features from identification to resolution

• **Mean Time Between Failures (MTBF)** – the measure of the reliability of a system or component, calculated by averaging the time between system failures

• **Mean Time to Recovery (MTTR)** – the average time it takes a system or component to recover from a failure and return to production status

• **Microservices** – an architectural style that structures an application as a collection of services that are highly maintainable and testable, loosely coupled, independently deployable, organized around business capabilities, and owned by a small team

• **Microservice Architecture** – the practice of developing software as an interconnected system of several independent, modular services that communicate with each other

• **Production** – the final stage in a deployment pipeline where the software is used by the intended audience

• **Rollback** – an automatic or manual operation that restores a database or program to a previously defined state

• **Staging Environment** – an environment used to test a newer version of software before it’s moved to live production (Staging is meant to replicate as much of the live production environment as possible, providing the best chance to catch bugs before the release of software.)

• **Unit Testing** – a testing strategy in which the smallest unit of testable code is isolated from the rest of the software and tested to determine if it functions properly

• **Version Control System (VCS)** – a system used for storing, tracking, and managing changes to software (This system commonly operates through a process of creating
branches [i.e., copies for safely creating new features] from the stable master version of the software and merging stable feature branches back into the master version.

- **Virtual Machine (VM)** – a software emulation of a physical computing resource that can be modified independent of the hardware attributes
1. Introduction

Given the mandate across the Department of Defense (DoD) to begin delivering software “at the speed of relevance,” many programs have begun to successfully practice Agile software development to (1) modernize software development and delivery practices within the United States Government, and (2) ultimately deliver secure and functional software into the hands of the warfighter at a faster pace than our adversaries. The backbone that makes this efficient delivery of software possible is the blend of Continuous Integration (CI) and Continuous Delivery/Deployment (CD).

The software development practice of integrating code changes together as early and as frequently as possible—in combination with instantiating the packaged codebase in an environment that identically mimics production—drastically reduces the strain felt by IT Operations teams during software delivery. Development, Security, and Operations (DevSecOps), which consists of a collection of principles, practices, and technologies, allows for the delivery of integrated, tested, and secure software at the speed of relevance.

DevSecOps has become an industry standard because of its ability to improve software quality and security; it also promotes the organizational cultural change required for its successful practice. When it comes to software, the common notion of “well this is the way we’ve always done it,” is based on traditional and outdated practices that are no longer supported by industry and the DoD.

1.1 Purpose

Throughout the DoD, there are limited proven examples of successful implementations of DevSecOps, including by both the 873 and 874 pilot programs. The aim of this document is to provide DoD programs that are initiating their DevSecOps journey with a standardized base of foundational knowledge that can be referenced and applied across the DoD. This approach gives all programs a similar advantage to not only be successful but to also take advantage of learning about similar challenges they encounter.

1.2 Organization and Culture Change Prerequisites

Before an automated pipeline is implemented or DevSecOps can be successfully practiced, a foundational knowledge and competency in Agile development must be demonstrated. DevSecOps relies heavily on specific Agile principles, such as iterative development, continuous customer feedback, and cross-stakeholder communication and collaboration. Failure to make the cultural change to Agile before implementing an automated DevSecOps pipeline causes a “tool-centric” focus, emphasizing effort and importance of the tools that make up the pipeline instead of the culture of the employees who are using the pipeline as a tool for successful and efficient software delivery.
2. Cultural Shift

Automated pipelines and tools are crucial to the delivery of software, but the people that use and interact with the technology are the core asset of software. Without a culture where everyone is on the “same page” and striving towards the same goal as a singular unit, any project will ultimately fail. This section introduces the necessary cultural shift required to implement DevSecOps and why it is more important, and often more challenging, than implementing the tools and automation.

2.1 Organization Buy-In

Many reference documents and articles produced on DevSecOps tend to focus on automation, pipeline service tooling, Continuous Integration (CI)/Continuous Delivery & Deployment (CD), etc. However, even though the technological components of DevSecOps are crucial to building, testing, and delivering software, the stakeholders involved—from project inception to delivery—are the key to successfully implementing and practicing of DevSecOps.

Buy-in from all stakeholders, including leadership, middle-management, engineering, operations, development, and testing teams, is critical to the success of DevSecOps adoption. Stakeholders across the organization must change their way of thinking from “I” to “we” and understand that the failure to successfully deliver, maintain, and sustain software and its underlying infrastructure is the failure of the entire organization, not one specific team or individual.

In 2015, Ian Head, former research director at Gartner Inc., estimated that, “by 2018, 90 percent of infrastructure & operations (I&O) organizations attempting to use DevOps without specifically addressing their cultural foundations will fail” [2]. Before beginning a DevSecOps journey, it is imperative to understand that a successful implementation of DevSecOps cannot be measured by a completely automated pipeline or the interaction between development and operations teams alone; all stakeholders in the organization must be committed to changing the way they view their job responsibilities and, most importantly, interact with each other.

2.2 Communication and Collaboration

The cultural shift that comes along with DevSecOps also introduces new ways for stakeholders to do their jobs, specifically

- communicating with others to break down organizational silos
- managing artifacts
- measuring project progress
- responding to critical incidents

Ensuring the complete transparency across team members in real time is not only necessary for shifting toward a proper DevSecOps culture, but it is also an easy integration task that can immediately take place while establishing new services throughout the pipeline or modifying existing services. All project resources, such as wiki pages, version control systems (VCSs), Continuous Integration (CI) servers, and projects within issue tracking systems, should be connected and viewable by everyone working on the project. This does not mean that all
stakeholders should have commit privileges to the VCS, for example, but everyone should be able to view all version-controlled artifacts at any given time. Such transparency allows the project’s stated progress to be highly visible since all project artifacts are viewable by all team members.

The key to enabling transparency and visibility into project-related communications is using a project-stakeholder-wide chat service that easily integrates with other services that make up the DevSecOps pipeline. Taking advantage of a concept called ChatOps—using chat clients, chatbots, and real-time communication tools to facilitate how software development and operation tasks are communicated and executed. ChatOps provides a backbone of communication through a commonly accessed messaging service. ChatOps has three main categories of use in a DevSecOps environment:

1. pipeline service integration hooks with chat services such as HipChat, Slack, and Mattermost
2. ChatBots such as Hubot that automate tasks like initiating an integration build directly from a chat window
3. notification systems for incident management

ChatOps enables all team members—from project managers to developers—to chat in real time (avoiding the delayed response often caused by using email as a primary form of communication) and view notifications from all services related to any project activity or artifact.

2.3 “Shift-Left” Mentality to Create Checks and Balances
As previously discussed, the biggest roadblock to successfully practicing DevSecOps is the cultural shift and organizational buy-in. Early involvement of specialists in the project lifecycle is critical to project success. Specialists include, but are not limited to, operations testing and security teams. During project initiation and planning activities, development, testing, security, and operations teams should be represented to give input and to voice any concerns. Granted, security should be treated as a top priority, meaning that security requirements should be valued with equal, if not greater, importance than functional requirements. However, it’s important to understand that the concept of “shift-left” does not apply to only security teams but also to all technical specialists who act as stakeholders.

Shifting the involvement of technical specialists to the earliest possible phases of the project may initially cause disagreements among developer, operations, testing, and security teams, but ultimately these uncomfortable conversations must take place early to avoid costs downstream because of insufficient opportunities to communicate/collaborate. The later in the delivery cycle an issue is found, the costlier it is to fix. So, finding issues sooner decreases costs and improves quality in the long run. It is best to think of shifting-left as creating a system of “checks and balances” with increased touchpoints between developer, operations, testing, and security teams. All four teams will eventually agree on the requirements that satisfy functional, non-functional, and security needs, and the teams will participate in a common forum of discussion (due to their early involvement) until this agreement is achieved.
Shifting expertise left also helps to break down “organizational silos,” which inherently reinforce waterfall software development. Silos are created when groups within an organization work as disconnected entities. Reinforcing early, continuous communication helps to instill the idea that all stakeholders’ responsibilities are of equal value, and ultimately everyone is striving towards the same common goal of efficiently delivering secure software in an iterative, automated fashion.

2.4 Beginning Roadmap

2.4.1 DevSecOps Assessment

As an organization starting from scratch or as one already practicing some hybrid form of DevSecOps, it’s important to have the correct mindset when beginning the journey to successfully implementing an automated deployment pipeline and practicing DevSecOps. The cultural change and the deployment pipeline take a considerable amount of time and effort to make possible. A common approach to this tremendous undertaking is to begin by offering Agile and DevSecOps training courses across the organization so that all stakeholders become familiar with DevSecOps concepts and gain a brief understanding of the direction everyone will be headed. Next, and most importantly, is to perform a comprehensive assessment of the organization’s entire Software Development Life Cycle (SDLC). Steps of an assessment often include the following:

1. Conduct interviews with stakeholders to record their processes and current practices involved in producing the system(s).
   - Identify non-standardized manual processes and gaps that can be replaced with automation.
     - Identify portions of processes that simply cannot be automated (e.g., comparison between testing in a simulation vs. testing on actual hardware for embedded systems).
   - Identify organizational/procedural bottlenecks and policies that inhibit a DevSecOps cultural shift (e.g., lack of stakeholder-wide communication channels).
   - Identify the need for rearchitecting applications that were not initially designed to operate as microservices.

2. Produce a report with a gap analysis and recommendations proposing organizational, technical, and procedural improvements.

   - VSM is a Lean manufacturing technique that analyzes the flow of work, supporting systems, and people involved in delivering a capability to a user.
   - DevSecOps practices benefit from VSM planning up front by mapping the deployment pipeline.

Conducting an assessment is the recommended and proven method of gaining an understanding of DevSecOps roadblocks. An assessment is also a platform for establishing goals for making appropriate cultural and policy (if necessary) changes as well as incremental steps toward producing a DevSecOps pipeline.
3. Automation
Although the other principles of DevSecOps (i.e., collaboration, IaC, and monitoring) are important, the automation of manual processes is what gives DevSecOps its resiliency, quick recovery upon failure, and immediate transparency across stakeholders. The main goal of automation in a DevSecOps ecosystem is to mitigate risk when delivering value from one environment to another (e.g., from development to test/staging and from test/staging to production).

3.1 Deployment Pipeline
The deployment pipeline is the software factory that automates the delivery of value. In addition to tools, a pipeline also includes automated procedures that enable collaboration, management risk/security, resilience, builds, third-party packages, and delivery systems that meet stakeholder needs across business units. The most common misconception regarding deployment pipelines is that there is a “one size fits all” or “cookie cutter” solution that can simply be dropped into place, and it will work for every project that uses it. The primary underlying services\(^2\) that make up a pipeline remain in place once implemented and integrated. However, the configuration for each project travelling through the pipeline is changed based on project requirements, specifically during the CI/CD phase. The automated processes that connect the services in a pipeline are the glue that make CI and CD possible.

Engineering the pipeline is a challenge and a considerable portion of project effort. This engineering process in itself should be done in a DevSecOps fashion. This means that all services within a pipeline should be produced and deployed using automation and IaC. The pipeline itself is a living entity that provides multiple services that, over time, becomes vulnerable, requires maintenance, and should support continuous availability through scalability and reliability. Since this process requires such a tremendous effort, the appropriate mindset of its participants should be in place when architecting and engineering a pipeline. Stakeholders should understand and accept the following realities:

- Completely automated deployment pipelines take a long time to successfully implement. This work requires not only knowledge and mastery of each individual service that makes up the pipeline but also the appropriate knowledge and effort to automate creating and instantiating those pipeline services via IaC.
- The pipeline won’t be implemented all at once. Delivered value of the pipeline occurs over time, in exactly the same manner as an application going through a pipeline that is already in place.

3.2 Continuous Integration (CI)
Conversations about CI tend to start with the tool chosen for the CI server, with popular choices being Jenkins, Bamboo, GitLab, and TeamCity. However, the automation of the builds initiated

\(^2\) Services that make up a pipeline include VCS, CI server, private container image registry, private artifactory, monitoring system, wiki, issue tracking system, static code analysis tool, and communications system.
by the CI server are only as efficient as the discipline of the developers whose code it’s integrating. Successful CI starts with developer discipline and the modern software engineering best practice of committing and pushing code to a VCS early and often. Early and frequent integration into the master branch of a VCS drastically reduces the risk of integration errors when build automation actually takes place.

Martin Fowler, Chief Scientist at the global software consulting firm ThoughtWorks, defined eleven best practices that should be addressed when crafting a deployment pipeline’s CI process: [3].

1. **Maintain a single-source repository.**
2. **Automate the build.**
3. **Make the build self-testing.**
4. **Everyone commits to the baseline (master branch) every day.**
5. **Fix broken builds immediately.**
6. **Every commit to the baseline should be built.**
7. **Keep the build fast.**
8. **Test in a clone of the production environment.**
9. **Make it easy to get the latest deliverables.**
10. **Everyone can see the results of the latest build.**
11. **Automate deployment.**

### 3.2.1 Testing

In a perfect DevSecOps pipeline, there are three major forms of automated testing that should occur:

1. **Automated Security Testing**
2. **Automated Functional and Unit Testing**
3. **Automated Application Performance and Integration Testing**

The first two forms, automated security testing and automated functional and unit testing, take place at the CI (automated build) phase of the DevSecOps pipeline. The last form of automated testing takes place in a production-like (staging/testing) environment after an initial deployment and ultimately only occurs if quality criteria for the first two forms are met.

#### 3.2.1.1 Automated Security Testing

To take full advantage of the power of automated security testing, it’s important that the quality criteria for testing pass/failure is determined as early in the software development lifecycle as possible. These specific criteria vary on a project-by-project basis and require collaboration among the PO/end-user, development, security, testing, and operations teams as well as the designated project AO (authorizing officer). To be as effective as possible, all of
these stakeholders need to be involved in the discussion of quality criteria for automated testing.

Security automation takes place at multiple phases throughout the DevSecOps pipeline. Since the software pushed through the pipeline and the pipeline itself are both living entities that mature over time, security must be treated as a state. For example, a software application is only as secure as the most recent security tests/assessments say it is.

With every new code commit, the attack vector of a given software application has now increased and must be reassessed appropriately. This concept applies to the application code itself, third party dependencies/libraries, and the dependencies used by the underlying infrastructure. The security of all three categories must be assessed in an automated fashion as part of the DevSecOps pipeline using automated static code analysis, automated penetration testing, and automated container vulnerability scanning. It is important to implement automated security testing as the first form of test automation.

The first form of automated security testing that should be implemented is static code analysis. Automated static code analysis takes place at the continuous integration (CI) phase of the DevSecOps pipeline. CI is readily available via third-party plugins (e.g., SonarQube, HP Fortify, SWAMP, SpotBugs, and SCALe) for standard CI services like Jenkins and Bamboo. Static code analysis can be used for web applications as well as embedded systems software. Successful implementation as early (left) in the SDLC as possible allows for vulnerabilities to be mitigated as they occur in the initial development of a project’s MVP. However, implementation of automated static code analysis cannot be considered “successful” once it is integrated into the CI process alone. It is imperative that the correct information security personnel are delegated the responsibility of viewing the reports generated by static code analysis tools. For example, an information security professional whose primary skillset is network/perimeter security should not be tasked with analyzing a static code analysis report. Information security teams may be limited to a specific set of personnel, but an information security professional with a professional background in software engineering/development is preferred. Tasking the evaluation of static code analysis reports to individuals who have “taken coding classes but have never practiced the discipline as a profession,” creates additional risk and could complicate the iterative feedback conversations between security and development teams.

The second form of automated security testing, the implementation of automated penetration of web applications, takes place at the CI phase of the pipeline. Tools such as OWASP ZAP and Gauntlt, which provide a suite of penetration testing tools (e.g., sqlmap, nmap, sslyze and Garmr) should be strategically placed throughout the CI build plan to provide continuous automated reports that can be assessed by information security teams.

If using containers, the third form of automated security testing, automated scanning of Docker images, is necessary. This form of automated security testing is more important than scanning of the application code itself. Automated vulnerability scanning of Docker images requires security and operations (IT) teams to work in unison throughout the composition and sustainment of custom Dockerfiles. To prevent an exploitable infrastructure from being deployed into production, vigilant and constant security maintenance of IaC scripts is required.
3.2.1.2 Automated Functional, Unit, and Performance Testing

Once an initial form of automated security testing is integrated into a DevSecOps pipeline, automated functional testing should be implemented. In addition to writing the automation scripts themselves, the most important aspect of this process, and one that is often overlooked, is ensuring that an appropriate logging mechanism for test results is in place. A logging solution such as ELK (Elasticsearch, Logstash, and Kibana) or Splunk is crucial to pinpointing a specific place test automation failure. For web applications, a web browser automation tool like Selenium is necessary to simulate the functionality a user performs in a browser.

In combination with test automation frameworks such as JUnit, NUnit, pyTest, PHPTest, TestNG, and Mocha, automated functional/unit testing can be integrated into the CI process, and predetermined quality criteria can used as pass/fail thresholds for CI builds.

The amount of testing that can be automated at this phase varies from project to project. The forms of automated testing that occur ultimately depend on the type of software pushed through the pipeline. For example, web applications have a tremendous amount of front end/GUI automated testing, while embedded systems software do not. Automated integration testing, however, takes place regardless of the type of software built and deployed.

Finally, automated application performance testing is the form of automated testing that takes place outside of the CI phase of the DevSecOps pipeline. Application performance testing takes place after an initial deployment to a staging/test/production environment that exposes issues in non-functional requirements that could not be replicated in a development environment. Availability issues discovered by application performance testing are in themselves security issues (as previously discussed) due to a fundamental violation of the CIA triad (availability). By taking advantage of a tool like Dynatrace, operations teams can address architectural and systems engineering concerns from an availability perspective as early as possible, ensuring that scalability and reliability are built into the architecture and continuously refined as the MVP evolves.

3.3 Continuous Delivery vs. Continuous Deployment (CD)

To select which form of artifact transition is appropriate to meet business needs, it is necessary to understand the fundamental difference between continuous delivery and continuous deployment.

- **Continuous Delivery** is a software engineering practice that allows frequent releases of new software to staging or various test environments through the use of automated testing. Continuous delivery relies on a form of manual deployment to production.

- **Continuous Deployment** is the automated process of deploying changes directly to production by verifying intended features and validations to minimize risk.

In the case for most organizations across the DoD, continuous deployment is not an option due to end-user training and continuous authority to operate (cATO). Delivery to production may not be possible in either the same way as delivery to staging or in an automated fashion. These unavoidable manual gaps should be addressed during an organization’s initial DevSecOps assessment.
The following are recommendations for successful continuous delivery that have not been previously addressed:

- **Include (if relevant) the database as part of the deployment pipeline.** Managing changes to a database throughout development can be challenging, but using tools like Flyway and a database build automation tool like dbmaestro allow the automatic migration and deployment of an updated database if changes are manually made to the schema, or changed by altering object models using a model view controller (MVC).

- **Deploy the same way every time.** Where applicable, the deployment method should be replicated when transitioning artifacts from one environment to another.

One of the main misconceptions about continuous delivery is the “manual” deployment to production. Although the decision or trigger to deploy to production is indeed manual, the actual deployment mechanism should be automated. This process should be the standard for all software delivered, except for embedded systems, where software requires manual, physical installation.

### 3.4 Monitoring

In addition to automating the manual processes that are required to promote value to different environments, it’s important to also automate the monitoring of the overall success of the pipeline and DevSecOps culture. Apart from troubleshooting errors, these collected metrics and centralized logs are essential for security and auditing purposes. Additionally, the collection and visibility of all metrics helps to promote a culture of complete transparency.

Monitoring the metrics of the pipeline is important to optimize performance over time, but infrastructure and application monitoring is equally important to ensuring the health of delivered/deployed value. This means that essentially anything that can be monitored should be monitored. The metrics collected throughout the entire use of the pipeline—from where areas of code in source control interact to the health of deployed clusters—should be used to determine a standard that can be used to detect anomalies over time.

The metrics that should be collected can be broken down into three categories:

1. **Pipeline Metrics**
2. **Infrastructure Metrics**
3. **Application Metrics**

#### 3.4.1 Pipeline Metrics

The Defense Innovation Board’s (DIB’s) *Metrics for Software Development* study [4] recommends four different categories of software metrics that should be collected throughout a DevSecOps pipeline:

1. **Deployment Metrics**
   a. Time from program launch to deployment of simplest useful functionality
   b. Time to field high priority function or fix newly found security hole
   c. Time from code committed to code in use

2. **Response Rate Metrics**
a. Time required for fully automated regression test and cyber security audit/penetration testing
b. Time required to restore service after outage (aka Mean Time to Recovery or MTTR)

3. **Code Quality Metrics**
   a. Automated test coverage of code
   b. Number of bugs caught in testing vs. field use
   c. Change failure rate (rollback deployed code)

4. **Program Management, Assessment and Estimation Metrics**
   a. Complexity Metrics
   b. Development plan/environmental metrics
   c. “Nunn-McCurdy” threshold (for any metric)

In addition to these metrics, the following should also be collected and evaluated:
- Deployment frequency
- Deployment time
- Number of failed deployments
- Cost of failed builds
- Automated test/pass percentage
- Mean time to detection
- Issue volume (# of tickets) and resolution time
- Attack vector details (i.e., IP, stack trace, time, and rate of attack)

### 3.4.2 Infrastructure Metrics
Collecting and monitoring infrastructure metrics is not a new trend for operations teams. Taking advantage of combinations of tools such as Nagios, DataDog, and Prometheus allows the quick formulation of an expected infrastructure performance baseline as well as a platform for automatic warnings and alerts to be generated upon infrastructure failure or whenever defined resource thresholds are approached.

It is important to understand why using more than one infrastructure monitoring tool is necessary. For example, if Nagios is implemented and generates an alert that a host’s disk space is 90% full, it is working as intended. But the mere knowledge that disk consumption has reached 90% is not enough to immediately address the problem. The desired information is “why has the disk reach 90%?” This is why using separate monitoring tools in unison is recommended.

When implementing a logging strategy, use the following practices:
- Forward logs to third-party logging software as a service (SaaS) solutions to ensure that developers are not relying on server access to troubleshoot errors.
- Ensure that developer access to production servers is blocked. Developers should be granted access to the instantiated logging application to view errors. This is not only a layer of defense against insider threat, but it also helps promote a culture that reinforces the use of IaC. Manual changes, even for troubleshooting systems in
production, should not be made. All changes to fix errors should be made to the version-controlled IaC, which is responsible for creating and provisioning the infrastructure.

- Automate the creation of issues upon receiving an automatically generated alert. If infrastructure resource thresholds are defined correctly, automatically generated critical alerts MUST be addressed, resulting in a corresponding issue created in an issue-tracking system. This process should be automated to allow the responder to focus on addressing the concern of the alert immediately rather than spending time creating an issue with a time estimation.

3.4.3 Application Metrics
To improve the overall experience of the end user over time, it is absolutely necessary to monitor the performance of an application deployed to production. Any type of problem that may cause a negative user experience—from slow response time to an inadequate amount of application instances—should be tracked and monitored. Having a complete understanding of how users interact with the system and collecting the metrics generated through this interaction helps provide a roadmap for those aspects of the system that need to be improved and why.
4. Infrastructure as Code
A scripted and version-controlled infrastructure is one of the backbones of CI and CD. Any changes made to IaC scripts can be easily applied to any environment, removing the need to log into a specific host to manually make changes.

4.1 Environments
Ensuring parity (i.e., the state of being identical) between the four commonly used environments (i.e., development, integration/test, staging, and production) is essential to minimize the chances of a catastrophe when delivering a product from one environment to another. Due to the fact that environments should be viewed as independent, volatile entities, their creation and maintenance should be automated.

The true way to establish environment parity across all environments is to script the creation of infrastructure using IaC. In addition to establishing environment parity, IaC inherently brings along with it a culture of quality. By scripting all infrastructure, environments now have the following attributes:

- **Consistent** – Environments are consistent in terms of their configuration and deployment approach. Consistency reduces the number of variables when issues are found and saves time with integration and deployment.

- **Automated** – Environments can be deployed automatically and, when using containers, almost instantly.

- **Repeatable** – The same IaC can be used infinitely many times to create or recreate the same environment.

- **Versionable** – Detailed traceability is allowed across changes over time.

- **Reviewable** – The quality of scripts can be included in code reviews.

- **Diffable** – Changes against scripts can be compared.

- **Testable** – Unit, functional, and integration testing can be applied to the infrastructure.

- **Human-Readable** – Many IaC scripting frameworks are declarative, and the code itself provides documentation.

- **Verifiable** – Environments can be versioned into known working states.

Each attribute is important in its own regard, but since IaC must be treated with the same integrity as application code, developer and operations teams can maintain configuration control of the IaC code and, by extension, track differences in configuration by looking at the history of changes to the code. IaC helps support the concept of “immutable infrastructure” — an infrastructure that cannot be changed after it is created and provisioned. Any modifications to infrastructure created via IaC should be made to the scripts responsible for creation and provisioning. The old infrastructure is deprovisioned once the updated infrastructure is deployed.
Due to the combination of expertise required to craft IaC scripts that are acceptable for production use, developer, operations, testing, and security teams should be involved in the IaC scripting process.

In addition to aligning with delivery requirements, all four of the mentioned technical disciplines (development, security, test, and operations) must be in constant communication about necessary changes to third-party dependency versions that may affect the project. An example is an announcement that a third-party Maven dependency crucial to a Java application is subject to a critical vulnerability. Developer, operations, and security teams must immediately communicate about how changing the version of this specific dependency will affect the overall build status of the project due to dependency version compatibility issues, interaction with the specific version of the web server used, etc. The types of issues that involve all stakeholders require that the communication and collaboration foundation of a DevSecOps culture is already in place.

4.1.1 Architectural Parity Between Environments
It’s common for different IaC frameworks to be used in combination. For example, to provision a local development environment, Vagrant (to create the virtual infrastructure by interacting with a hypervisor such as VirtualBox) can be used in combination with other tools such as Chef, SaltStack, or Ansible (to provision the created infrastructure with the software necessary to provide a specific service).

With the adopted use of containers across industry and the DoD, it is crucial to understand that the architectural integrity between environments is maintained. This means that if containers are selected as the desired infrastructure, they will be deployed in production, and they should
also be used for all other environments (specifically development). It is highly discouraged to create IaC for virtual machines (VMs) in a development environment that, in theory, may match the exact requirements of containers used in production.

4.2 Containers and Container Orchestration

More information on the basics of containers can be found on the Docker website.\(^3\)

4.2.1 Pets vs. Cattle

To understand the immutability aspect of containers, it’s first necessary to understand the “Pets vs. Cattle” concept. Traditionally, IT operations teams have treated their infrastructure as “pets,” meaning that they are treated as vital systems that can never have down time. These hosts are usually manually configured, maintained, and upgraded. With CI and CD being pillars that support DevSecOps, manually provisioned, maintained, and upgraded systems no longer have a place in the infrastructure ecosystem.

Operations teams have now moved to the “cattle” model, which treats infrastructure as a scalable and disposable entity. Updates and fixes required for a given piece of infrastructure are now made in the associated IaC provisioning script that is responsible for its creation. Containers (and, in extension, microservices) are treated as cattle. If a container requires additional resources, it can automatically call to fulfill the need of necessary demand. Additionally, if an update is needed, a change can be made to the original Docker image the container is using, and a new container is created with the configuration change, replacing the old one. The immutability of containers reinforces configuration changes made at the IaC level, which are then pushed through the entire pipeline, and deployed to the appropriate environment, eliminating manual changes to infrastructure.

4.2.2 Dockerfile Composition

Every container is created from an underlying image, which is “an executable package that includes everything needed to run an application—the code, a runtime, libraries, environment variables, and configuration files” [5]. A Dockerfile is the IaC script responsible for installing all of the specified software and subsequently creating a Docker image. Before discussing best practices for authoring, it’s important to understand the “layer” concept Docker images are built on.

Layers are essentially intermediary images that are created throughout the execution of the commands specified in the Dockerfile while building the Docker image. An efficient feature that Docker uses is layer caching, which will only download updated layers that it detects when downloading a new version of an image. An example of a Dockerfile with eleven layers is shown in Figure 23.

\[^{3}\] [https://docs.docker.com/get-started/]
When writing Dockerfiles, the goal is to use the least number of layers possible, which translates into condensing any installation scripts into the least number of Dockerfile commands (i.e., RUN, COPY, and ADD).

Looking at the example in Figure 3, it is clear that using the “&&” syntax to execute multiple commands only upon the success of its predecessor helps pack the installation of necessary packages into one layer instead of seventeen separate layers.
Although outside of the scope of writing Dockerfiles, it is necessary to understand where the resources being installed in a container image are coming from. As shown in Figure 34, there are twelve different packages installed using the command line package manager, apt-get. Packages used in IaC scripts should never be downloaded directly from public repositories. All versions of packages should be vetted by information security teams. Once they are determined to be risk free, they should be stored in a private artifactory such as JFrog or Nexus.

4.2.3 Private/Internal Registry
All container images used should be stored in a private or internal container registry. Images downloaded from DockerHub should not be used in production and should first undergo an image vulnerability scan using a tool like Anchore, Twistlock, or Clair. If vulnerabilities in an image are found, a new Dockerfile should be written to create a custom image that mitigates, at minimum, any critical vulnerabilities.\(^4\)

4.2.4 Orchestration
For introductory and tutorial purposes, it’s acceptable to create containers one at a time using the “docker-run” command. However, even for implementing a local development environment, starting and managing each container individually is cumbersome and inefficient. To solve this, one can take advantage of a container orchestration tool that allows for the easy creation, maintenance, and destruction of 1-n containers. Orchestration solutions most importantly provide a built-in load balancing and rollback capability, auto-scaling, service clustering, and updates. The following are the four commonly used container orchestration tools:

- Docker-Compose
- Docker-Swarm
- Kubernetes
- RedHat OpenShift

To successfully implement a container orchestration solution, an established DevSecOps pipeline that supports the automated build and testing of container images is necessary. Production quality orchestration tools rely on image version tags to automatically rollback when a deployment failure takes place.

Once a container orchestration tool is selected, there are a few important topics that should be addressed by all stakeholders, some of which may have already been accomplished throughout the instantiation of the DevSecOps pipeline:

- **Map out a detailed path from container development to production.** This mapping should describe how an updated container image moves throughout the pipeline from code commit to a staging environment and eventually to production. With every update to the code repository, a new version of a container image should be automatically

\(^4\) The mitigation of medium-low severity remaining vulnerabilities will be determined by an organization’s information security policy regarding containers.
built, tested, pushed to an internal registry, deployed to a staging environment, and then (after continuous non-functional testing), deployed to production.

- **Establish proper cluster monitoring.** Kubernetes and OpenShift have internal resources that allow cluster metrics to be collected. These metrics-gathering tools are the primary criteria for the horizontal autoscaling-capability. These features should be used in combination with other monitoring tools, such as Prometheus and Grafana, to provide stakeholders with user-friendly graphs and visualizations of cluster-health and collected metrics.

- **Vigorously test non-functional requirements in staging.** One of the most valuable benefits of using an orchestration tool is the autoscaling capability. The reliability and availability of the cluster should be tested in the staging environment, under conditions that will exceed expected production resource consumption. This testing ensures that the thresholds defined in the Kubernetes Horizontal Pod Autoscaler are never surpassed, causing an accidental self-inflicted DoS (denial of service) due to cluster misconfiguration. Autoscaling threshold criteria should be adjusted appropriately as cluster resource metrics are collected over time.

### 4.2.4.1 Cloud-Managed Orchestration

The power of complete administration when running a native container orchestration tool, such as Kubernetes, comes with a few drawbacks. Properly installing a Kubernetes cluster is a complex undertaking and often requires a Kubernetes expert to assist with the cluster configuration. Cloud providers, such as Amazon Web Services (AWS), offer fully managed container-orchestration tools that take away much of the pain and expertise required to instantiate a Kubernetes cluster.

AWS offers two container orchestration tools: AWS Elastic Container Service (ECS) and AWS Elastic Kubernetes Service (EKS). ECS is the native container-orchestration tool for AWS. It provides the user with two underlying infrastructure options to host their containers: EC2 and Fargate. Taking advantage of the Fargate feature eliminates the need to choose underlying EC2 virtual machine images by automatically providing the underlying infrastructure where the containers run. ECS provides features that handle autoscaling; load balancing; blue/green deployments; monitoring; and logging by integrating with AWS CodeDeploy, AWS CloudWatch, and AWS CloudTrail.

EKS takes the managed-orchestration concept a step further from an ease of use perspective by eliminating the need to configure the Kubernetes Control Pane. EKS deploys the Kubernetes management infrastructure across numerous AWS reliability zones to ensure that there is not a single point of failure within the cluster.

When choosing a container orchestration tool, it is important to select the platform that complements the business/mission goals. If working in a standard AWS or AWS GovCloud, one of these managed-orchestration tools should be considered due to its ease of use, lack of underlying expertise required for initial install/configuration, and the option to integrate with other services on AWS.
4.2.5 Security
When addressing container security, there are a few different contexts that must be covered.

4.2.5.1 Integrity of Libraries and Packages Baked into a Container Image (Integrity of Third-Party Dependencies Used in the Application)
With built-in input sanitization offered by modern web frameworks, the most discovered exploitable vulnerabilities are due to a vulnerability found in a third-party package that is required by the application itself. The integrity of all artifacts used to build a container image must be maintained, whether required by the service or for development of the application provided by that service.

A concrete example of this can be shown using a JBoss web server that deploys a .war artifact built by the build automation tool Maven. Depending on the specific version of JBoss being used, the Dockerfile used to build the image requires a specific version of Java. For this example, the JBoss version is EAP 7.2, which is using OpenJDK 1.8. The “pom.xml” file is the next IaC artifact used by Maven and is responsible for managing third-party libraries used throughout the project. It is a best practice to configure both Docker and Maven to pull these artifacts from an internal, private artifactory service such as JFrog or Nexus.

Dependencies or libraries required for the project should first be analyzed and vetted by a collaborative effort between developer, information security, and operations teams. Mitigating the vulnerabilities found in the software (while building the infrastructure through IaC and the application source code itself) is crucial to providing a secure infrastructure and code to be deployed through the pipeline.

Once container images are successfully built and appropriately tagged using pre-vetted libraries and dependencies, the image must be pushed to a private registry service like Harbor or Amazon Web Services (AWS) Elastic Container Registry (ECR). Maintaining the integrity of images stored in the private registry through proper access control is crucial; without securing the registry, anyone can push any image, with any name. This means that a malicious image could be created and pushed to the registry, and the attacker could wait for someone to pull and deploy the image to their stack. At that point, the attacker has a “foothold” in the network.

4.2.5.2 Automated Vulnerability Scans of Images in the Private Registry
An automated vulnerability scan should be conducted as part of the CI process every time an image in the private registry is updated by tools like Anchore, Clair, or Twistlock. The application of Anchore using Jenkins is described below and is illustrated in Figure 45, which is excerpted from the official Anchore Container Image Scanner Plugin documentation [7]:

• A Jenkins job will build a container image, and pushes it to a registry that is pre-configured in the Anchore engine service.

• The Anchore build step will interacts with the Anchore engine by “adding” the image (which instructs the Anchore engine to pull the image from the registry), and then performing a policy evaluation check on the image. The build step can optionally be configured to fail the build if the policy evaluation results in a 'STOP' action.
• The plugin will store the resulting policy evaluation results with the job, for later inspection/review.

![Diagram of Anchore Jenkins Plugin Interacting with the Anchore Engine](https://wiki.jenkins.io/download/attachments/101482571/image_1.png?version=1&modificationDate=1527095831000&api=v2)

**Figure 4: Diagram of Anchore Jenkins Plugin Interacting with the Anchore Engine**

### 4.2.5.3 Runtime Security

Even though, in theory, once a container image is securely built and deployed as a container into a cluster, the run-time security of the living container is still a concern. This should be addressed using a run-time container security tool like Falco. The following explanation excerpted from Falco’s official GitHub page [8] explains that:

Falco is a behavioral activity monitor designed to detect anomalous activity in your applications. Powered by Sysdig’s system call capture infrastructure, Falco lets you continuously monitor and detect container, application, host, and network activity—all in one place—from one source of data, with one set of rules.

Falco can detect and alert on any behavior that involves making Linux system calls. Falco alerts can be triggered by the use of specific system calls, their arguments, and by properties of the calling process. For example, Falco can easily detect incidents including but not limited to:

- A shell is running inside a container.
- A container is running in privileged mode, or is mounting a sensitive path, such as /proc, from the host.
- A server process is spawning a child process of an unexpected type.
- Unexpected read of a sensitive file, such as /etc/shadow.
- A non-device file is written to /dev.
- A standard system binary, such as ls, is making an outbound network connection.

Adding a container runtime security tool reinforces the “defense in depth” information security concept. Falco can also be configured to integrate alerts with messaging services like Slack to

---

[8]: https://wiki.jenkins.io/download/attachments/101482571/image_1.png?version=1&modificationDate=1527095831000&api=v2
provide an extra layer of security transparency throughout an organization’s ChatOps ecosystem.

4.2.5.4 Security (Availability) Through Scalability and Reliability
With availability being the third point of the CIA (confidentiality, integrity and availability) triad, the availability of services provided by containers is imperative. Although a security concern, the task of mitigating the security risks due to a lack of reliability and scalability is handled by development/architecture and operations teams. The inherent auto-scaling capabilities that are built into container orchestration tools like Kubernetes and OpenShift ensure the resiliency of availability.

4.3 IaC Warnings
It’s important to point out a few caveats when implementing or introducing IaC:

- **Learning Curve.** The time required for technical staff to become competent in different IaC frameworks, best use cases, and interactions should be considered. Rarely is one IaC tool used alone; team members should become familiar with at least two to three IaC frameworks. A designated learning period should be built into project cost/time estimates and offered to as many team members as possible.

- **New Employee Contributions.** Although IaC allows for the expedited onboarding of new employees, giving them access to IaC scripts in version control also creates additional risk. New employees should be appropriately guided and mentored when first exposed to a project’s IaC content to avoid accidental changes to a project’s environments.

- **IaC Script Evolution.** The continuous improvement of IaC scripts takes place after an initial working version has been committed to a VCS. Over time, IaC scripts are tweaked to account for infrastructure vulnerability mitigation and application performance. As environments change over time, so will their accompanying IaC scripts.

- **Containers: Not a “Save-All” Solution.** Simply converting an existing monolithic application into container format is not a one-stop solution. Applications that were not developed with a Microservice architecture in mind must be re-architected to take full advantage of the scalability that is inherent in container orchestration tools.
5. DevSecOps Anti-Patterns

5.1 Over Emphasis of Pipeline Tools
Throughout the DoD, many conversations about DevSecOps begin with and are completely centered around the types of tools used to provide services within the pipeline (e.g., issue tracker, VCS, CI server, static/dynamic code analysis). These discussions reflect a fundamental misunderstanding of DevSecOps. Emphasis on the organizational cultural shift of stakeholder-wide collaboration from project inception is something that must happen before pipeline/automation tools are selected and implemented. The pipeline that produces software is a tangible portion of a DevSecOps culture, but the team members interacting with the pipeline must buy into the state of mind that comes along with the organizational cultural shift.

What is the solution to this misperception in the DoD? It involves continuous collaboration and open communication.

- Heavy continuous collaboration between all project stakeholders, including the following:
  - Design/architecture decisions
  - Environment/network configuration
  - Deployment planning
  - Code reviews
  - Infrastructure as code/automation

- Constantly available open communication channels, including the following:
  - Include development, security, test, and operations teams together in all project meetings.
  - Ensure that chat, email, and wiki services are available to all team members.
  - Have development, security, test, and operations teams must report together as one project team.

5.2 Bringing on New Roles Without Implementing Organizational Changes
A common bad practice is to rebrand an existing team as the “DevSecOps” group or hire specialists as DevOps engineers. Before explaining why this is an anti-pattern, we first define the skillset required for a DevOps engineer. DevOps engineers, according to industry, should have the following qualifications:

- Comfort with collaboration, open communication, and reaching across discipline silos
- A strong focus on business outcomes
- The ability to use a wide variety of open source technologies and tools, such as open source vulnerability assessment and mitigation
- Software development/engineering experience, such as test-driven development
- Systems and IT operations experience
• Comfort with frequent, incremental code testing and deployment (i.e., Agile competency)
  ○ A strong grasp of automation tools, such as IaC, CI/CD, test automation, etc.
• Experience with cloud platforms and underlying services
• Containers/container orchestration (e.g., Container image vulnerability assessment and mitigation)
• Data management skills
• A “security first” mindset toward everything

These skills, which essentially cover the entire SDLC, should not be the responsibility of one employee or a designated group; they should be shared across development, operations, and security teams.

According to Jez Humble, co-author of Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, “just calling someone a DevOps engineer can create a third silo in addition to dev and ops [...] clearly a poor (and ironic) way to try and solve these problems” [9].

A single person serving a combined DevSecOps role does not scale. What should be done instead? A more effective approach is to cultivate a collaborative peer model, as depicted in Figure 5.

![Figure 5: Example of the DevSecOps Collaborative Peer Model](image-url)
5.3 Using a Deployment Pipeline Without Traceability

Establishing an automated deployment pipeline is an integral component of DevSecOps, but the mere automation of build, test, and deployment is not enough to finalize the entire pipeline. It’s essential that there is traceability of value (code commit to production), from start to finish, as software makes its way through the DevSecOps pipeline.

A ChatOps notification should be created from every interaction with a pipeline service, including the following:

- Commit/push
- CI build
- Deployment (pass/fail)
- Tests (pass/fail)
- Wiki page edit/creation

5.4 Vendor Lock

As previously discussed, there is no “cookie cutter” solution for a DevSecOps pipeline due to the nature of the build and deployment process varying from project to project. However, the integral services that make up the pipeline are always the same and can be swapped out for different products that provide the same functionality.

It’s most common and recommended to create a hybrid pipeline consisting of open source and COTS (commercial off the shelf) software, based on either existing comfortability/licensing with specific tools or the current expertise that’s available within the organization.

Certain vendors offer an entire suite of tools that provide most of the integral services that are needed to make up a DevSecOps pipeline. Although this may seem appealing—especially for organizations with a limited number of personnel—from a licensing, installation, and ease of use perspective, it should be avoided. During pipeline creation and customization for each specific project, convenience features that are offered by specific services are used to make it easier to automate build, test, and deployment. Becoming too reliant on specific services and features provided by a chain of vendor tools makes replacing a single service in the pipeline cumbersome, and can potentially break functionality across the rest of the pipeline until integration is successful.
Works Cited


