Evolution of Integration, Build, Test, and Release Engineering Into DevOps and to DevSecOps

Vishnu Pendyala
San Jose State University, vishnu.pendyala@sjsu.edu

Chapter 1
Evolution of Integration, Build, Test, and Release Engineering Into DevOps and to DevSecOps

Vishnu Pendyala
Cisco Systems Inc., USA

ABSTRACT

Software engineering operations in large organizations are primarily comprised of integrating code from multiple branches, building, testing the build, and releasing it. Agile and related methodologies accelerated the software development activities. Realizing the importance of the development and operations teams working closely with each other, the set of practices that automated the engineering processes of software development evolved into DevOps, signifying the close collaboration of both development and operations teams. With the advent of cloud computing and the opening up of firewalls, the security aspects of software started moving into the applications leading to DevSecOps. This chapter traces the journey of the software engineering operations over the last two to three decades, highlighting the tools and techniques used in the process.

INTRODUCTION

Software Engineering teams have traditionally been responsible for branching strategies, code merges, nightly and production builds, validation of the builds, image generation and posting in addition to serving as consultants in Software Engineering practices to the product development teams. These functions continue to exist but have been transformed to adapt to the growing needs of the industry.

DOI: 10.4018/978-1-7998-1863-2.ch001

Globalization has come to stay. Teams operate in different time zones, often providing a seamless stream of development and operations activities round the clock. Software Configuration Management (SCM) tools such as Clearcase used for version control provided multi-site functionality to support code commits from all over the world – an excellent application of the distributed computing paradigm (Van Der Hoek, et al,1998). Software Engineering poses quite a few challenges when the code structure is complex, and the product dependencies are significant. Present day requirements of distributed teams and agile development add to these challenges.

Software Configuration Management (SCM) is key to effective product releases. The SCM tool employed to maneuver the Software Engineering processes of an organization should provide the necessary constructs to meet the requirements of the various releases. Interdependencies of the code and the volume of the code changes raise the complexity of the Software Engineering operations. With time, needs multiplied, operations scaled drastically, causing new tools, architectures, and patterns to be invented. From a handful of tools two decades ago, we now have a plethora of tools to manage Software Engineering operations. XebiaLabs recently came up with an entire periodic table of popular DevOps tools (Kaiser, 2018). The integration, build and release engineering discipline that existed originally has far transcended SCM related activities as its primary charter to a much broader DevSecOps role. This chapter traces through the journey of the Software Engineering discipline from the days of primarily performing builds, merges, releases, and tooling to the present day DevSecOps.

RELATED WORK

The DevOps area has predominantly been a domain of the industry than that of academia. Publishing articles is not as emphasized in the industry as it is in academia. This is one of the reasons for working on this book, so that insights into the tools, techniques, and processes employed in the industry, particularly, the large organizations are captured in the literature. Nevertheless, there is quite some literature already that captures the state-of-art in the DevOps and DevSecOps areas. The literature uncovered several interesting aspects of DevOps. This section captures a few of them. A framework for automated Round-Trip Engineering from development to operations and operations to development (Jiménez et al, 2018) is one of them. Round-Trip Engineering ensures that the Deployment and Configuration specifications are automatically ensured to be consistent with the system, thereby eliminating any technical debt on that count. This further confirms the need for tight integration of development and operations and automating the coupling as much as possible – one of the key points of this chapter.

Another channel of tight coupling between the development team and operations is through metrics. Metrics can provide an effective feedback mechanism in software organizations, which can be a substantial challenge in large organizations due to bureaucracy and cross-organizational environments (Cito et al., 2018). The authors identify feedback categories and phases and point to the tools that can help with the metrics generation. Culture plays an important role in DevOps (Sánchez-Gordón & Colomo-Palacios, 2018). Empathy is a critical component of the DevOps culture. Development teams and Operations teams must understand each other's perspectives and strive towards the overall productivity of engineers and the quality of the product. The authors survey the literature and summarize the trends about the DevOps culture. DevOps can be thought of like a Project Management methodology that fills in the lacunae in Agile methodology (Banica, et al, 2017).

Intertwined with culture is the skillset that the DevOps discipline demands. In the 26th European Conference on Information Systems, the authors (Wiedemann & Wiesche, 2018) categorize the skills needed to work in the DevOps area. The role of a Full-stack Engineer is gaining increasing relevance with the advent of DevOps. Full-stack engineering is particularly relevant in the Cloud Computing era (Li, Zhang, & Liu, 2017). Full-stack Engineers require broad skills covering all or most aspects of the software industry. Such skills are particularly important in fast-paced companies that produce several releases in a day. Describing such an environment where companies like Facebook release hundreds or even thousands of deployments into production daily, the authors (Savor et al, 2016) point out that it is possible to scale the teams and codebase several times without impacting the developer productivity.

Before the preceding work, excellent insights into the nature of software development at Facebook were provided by the authors of a different article (Feitelson et al, 2013). They point out that the differentiating characteristic of companies like Facebook is that the software they develop need not be "shipped" to customers as it runs on their servers. This enables rapid deployments of software updates in production. A different kind of domain is where software that is shipped is embedded. The complexity of embedded systems makes DevOps a formidable challenge in that domain (Lwakatare et al, 2016). Using multiple case studies, the authors explain why embedded systems are different when it comes to DevOps. The practice of DevOps in general, was surveyed and recommendations were made based on the survey (Erich et al, 2017). One such recommendation is to implement Continuous Delivery to the point of being able to release software updates on-demand.

From a software architecture perspective, microservices facilitate rapid deployability (Chen, 2018). Monolithic architectures, however modular they are designed to be, cannot scale-up to the level of microservices architecture when it comes to Continuous Deployment. Using microservices architecture, small teams can deploy their changes, without having to wait to merge changes from other teams. Because of the limited functionality in a microservice, deploying the software update is much faster as compared with monolithic architectures, which need to be deployed a whole. Changing to microservices architecture and adopting DevOps methodology requires substantial efforts. Designing a DevOps maturity model helps in the process (Bucena & Kirikova, 2017). The maturity model helps in identifying gaps in the current processes and goals for improvement.

DevOps brought-in a bunch of terms into the software engineering realm. Disentangling the terms and giving them a clear definition helps in better implementation of the DevOps practice. The authors of (Stahl, Martensson, & Bosch, 2017) survey the literature substantially to come up with definitions of the important terms used in the DevOps practice. One of the terms that is quite popular with DevOps is "Infrastructure-as-Code (IasC)" It is a tactic to speed-up the DevOps processes and is a good example of one of the many tactics that DevOps brought into the software engineering discipline to accelerate the pipelines (Artac, 2017). Software infrastructure typically comprises of several scripts and variable settings for setting up the infrastructure needed for the software to run. IasC treats these scripts and configuration files as source code as well, so that they can be versioned and treated as any other source code.

The evolution of DevOps is currently at the stage of encompassing security into DevOps and transitioning DevOps into DevSecOps. It has been observed that the increased automation of the processes that DevOps entails leads to improved product security (Rahman et al, 2016). The term, DevSecOps seems to have originated in 2012 in a blog post (Myrbakken et al, 2017) by a Gartner analyst. The key idea behind DevSecOps is to further break the barriers in the Software organization and make Security of the software product, everyone's business.

THE SOFTWARE ENGINEERING JOURNEY

Software Engineering organization in large companies traditionally comprises of some form of an Integration and Release Engineering team, a Platform Engineering team, a Tools team, and Program Management. The Platform Engineering team is typically responsible for porting the software across a wide variety of hardware and software platforms and maintaining the common code components of the software product. Porting involves making changes to the source code so that it works seamlessly across the platforms. Tools team makes the software to ensure developer productivity is high and processes run efficiently. Program management is responsible for managing software development projects. Integration engineering teams are responsible for builds, software configuration management, and sometimes, to some extent, quality assurance as well. The key component and highly visible role in Software Engineering organizations is still most often held by the team responsible for Builds, Release, and Integration engineering. The software development milestones have a huge dependency on the operations of this team. Let us start our journey by taking a closer look at this important function in its legacy form in the next subsection.

Integration Engineering

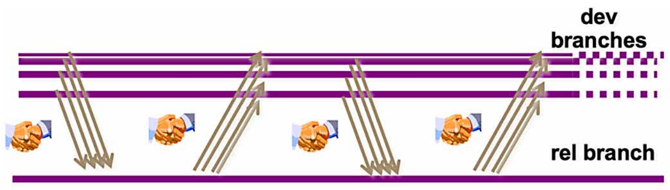
A typical large software organization has several products developed independently. Each of these products comprises of several features. Integration Engineering refers to the process of integrating these features and the individual changes that go into each of these products. Integration engineering is the interface between development and production. Interdependencies of the code and the volume of the code changes raise the complexity of builds and configuration management. The Integration Engineering team is responsible for branching strategies, code merges between product modules, nightly and production builds, validation of the builds, and image generation. Collecting metrics, creating dashboards, enabling access to the results of the builds and validation are the other activities that form the crux of Integration Engineering (Dyer, 1980).

A substantial portion of the source code is common to several products and product families. It would be chaos if the developers of each of these products check-into a single branch. Development is therefore segregated into more manageable '*development*' or '*dev*' branches. Developers check-in product-related changes into these *'dev'* branches which are periodically integrated into a '*release*' or '*rel*' branch. Each *'dev'* branch contains code changes contributed by the development team for a product or family of products. The '*rel'* branch incorporates the changes in all *'dev'* branches which merge to and from it periodically.

We, therefore, have the time-synchronized handshakes between the *'dev'* branches and the *'rel'* branch as shown in Figure 1. The merges to and from the *'rel'* branch are done against labels on the branches. Changes propagate to the *'rel'* branch and from the *'rel'* branch to the *'dev' br*anches with every merge. Because of the interdependencies of the code on different *'dev'* branches, this is accomplished through a physical merge, not by just updating the config_spec with the new label, if using Clearcase for software configuration or similar means if using other tools for the software configuration.

Handoffs to and from the release branch occur in Δt cycles where Δt is statically determined for each release based on the rate of code changes on all branches and their interdependencies. The time length of a cycle, Δt is inversely proportional to the rate of code changes on all branches in Δt, which handoff to the release branch and their interdependencies. We can mathematically model this relationship as,

Figure 1. Branch integration



$\Delta t \propto 1/[\,{}^{d}/_{dt}({}^{n}\int_{db=1} C)]\gamma_1\ \gamma_2\ \gamma_3 \ldots \gamma_n$ ....(1)

where db = development branch, C= code changes, ${}^{d}/_{dt}({}^{n}\int_{db=1} C)$ is the rate of code changes on all *'dev'* branches and $\gamma_1\ \gamma_2\ \gamma_3 \ldots \gamma_n$ are the correlation coefficients of the *'dev'* branches. The formula is only a conceptual representation of the relationships. In practice though, Δt is determined empirically, based on experience.

Each cycle comprises of 3 distinct phases on the *'dev'* branch: development, merge and build, which includes testing. The release engineering team, which manages the *'rel'* branch also generates an image after consuming a handoff. As was mentioned before development happens only on the *'dev'* branch – merges, builds, regression testing, and generating images are the only actions that happen on the *'rel'* branch, other than the handoffs. A handoff is typically a label, a snapshot of the source code, and information about the criteria this snapshot meets, like the test pass %s, etc. The label from a *'dev'* team is a sparse label of the files on the *'dev'* branch only, while the label from release engineering is a complete label on all files. After consuming the label from the *'rel'* team, changes in all *'dev'* branches will be visible in each of the individual development views.

In all the above activities, automation is essential. Software Engineering is a process and human-memory intensive. There are too many steps, dependencies and other factors that make it difficult to remember and do them manually, without the aid of scripts, checklists, and other aides. Manual processes have proven to be error-prone and time-consuming. Automation is essentially programming human expertise into scripts. When automation is not possible in entirety, it is a good idea to generate checklists, messages, and other aides. The very nature of Software Engineering makes it imperative that we automate as much as possible. Quality and productivity demand automation.

From Waterfall to Agile

The traditional software development paradigm is referred to as the Waterfall model because SDLC happens sequentially, in cascading stages. Requirements are collected upfront; development happens as one big project and the feedback loop between the development teams and operations is usually long. Over time, the software industry realized the perils of following the Waterfall model and the need for agility in the development (Sureshchandra & Shrinivasavadhani, 2008). Long feedback cycles result in a substantial risk. Teams operate in silos and bugs are discovered late in the cycle. Therefore, there is a need to break the one big project into more manageable smaller chunks. The branching model discussed in the section on Integration Engineering also needs to change to facilitate shorter release cadence. Code changes need to be integrated more rapidly than wait for Δt time cycle, which typically runs into days or weeks.

In the waterfall model, testing typically starts after all development is done. It is often too late and too expensive to fix bugs that late in the cycle. It is imperative to "fail fast" and recover from the failure fast as well. The cycle needs to be shortened even if it takes several cycles for completion of the project. Overheads need to be minimized and simplified to get into this iterative, agile mode of operations. Agility calls for flexible and highly collaborative environments and an entire rethink of the software development activity. For instance, companies have moved away from having many feature branches as described in the section on Integration Engineering to a single branch model that avoids merges and the heavy processes involved in managing numerous branches. In large organizations, thousands of developers could be working on a single branch. The source code instead uses 'feature toggles' for selectively exercising the code. Agile methodologies resulted in substantial improvements for companies. Some form of the Agile methodology has been successfully practiced by most large organizations.

One of the popular flavors of the Agile methodology is Scrum. Much like in the rugby football game by that name, where players flock together into a tightly packed team to grab the ball, in the scrum framework, teams collaborate closely with each other to develop the product. The idea of scrum is simple to understand, but difficult to practice. It originated in 1986, from a paper in the Harvard Business Review and is inspired by processes in the manufacturing firms like in the automotive and the photocopier industries. Scrum defines only three roles: Product Owner, Scrum Master, and the Team. The Product Owner is responsible for funding the project, setting the vision and release dates for the product. The scrum master makes sure that the team is productive and works to remove any blockers that the team may run into during the execution of the project. Scrum master, as the name indicates, is a key role, crucial for creating and sustaining a high-performance team. The team typically comprises of 5 to 9 members who do the real work of building the product. The team does not have a hierarchy, sub-teams or titles and functions seamlessly.

The work-cycle in scrum is called the sprint, which typically lasts for two weeks and comprises of many tasks to be accomplished in that cycle. A task is a fundamental unit of work in a sprint. The product is developed in increments. The end of a sprint marks the completion of a useable portion of a product, which can be released to the customers. This iterative development results in agile release cycles and shortened time to market. The simple operating environment results in low process overheads and quick decision making. Quality improves because of frequent testing and feedback from the field. Teams feel empowered and work-life balance is better achieved. Agile methodologies are big on automation, thus enhancing productivity. During a sprint, the team meets daily for a short duration, typically 15 minutes, standing and discuss these 3 key questions: (a) What did you do yesterday? (b) What will you do today? (c) Are there any blockers impeding the progress? Any blockers or issues are not resolved during the meeting – scrum meetings are not to be used for problem-solving.

If there are blockers discovered during the meeting that cannot be resolved by the scrum master, instead of extending the time, the scope is reduced – some of the tasks are downsized or eliminated. It is therefore imperative that the scrum master is an excellent problem solver and be able to unblock the team through collaboration, coaching, and leadership. In terms of documentation, the tasks that need to be implemented are described in form of "user stories" with the syntax, "As a <some user>, I want <some goal>, so that <some reason>." For instance, a user story in a sales analysis application could be, "As a Regional Director for the Asia Pacific, I want to be able to drill down to the sales numbers for a particular country with a few clicks so that I can change the sales strategy for that country if necessary." Documentation need not be exhaustive – working software is prioritized over comprehensive documentation.

Agile planning happens at different levels – task-level, done daily, feature level, done for a sprint and at a strategic level for the entire release. The development happens using timeboxed, lightweight iterations aligned with the sprint. The scrum framework prioritizes individuals over tools or processes, making sure that there are limits on the work in progress and feedback loops. One of the techniques often used is pair programming, where programmers work in pairs, one of them writing the code and the other reviewing it as it is being written. The pair keeps switching roles and collaborate closely. A sprint retrospective is held after every sprint, also for a short duration, where the entire team participates in reviewing what went well and what did not. The retrospective also follows a simple process. The team collectively decides what they should start doing for the next sprint, stop doing and what the team should continue doing going forward.

There are simple tools that help in the process of the timeboxed, iterative development. The tools include burndown charts which show the remaining work plotted against the days in the sprint and sprint backlog that is updated by the scrum master with the time required to complete the remaining tasks. Commercial software packages like Rally or Jira incorporate these tools. A key aspect of the framework is a sense of urgency that is shared by the entire team. The scrum methodology can be viewed as a shift in coding culture and requires buy-in from all stakeholders. It is a different way of doing software product development and can prove to be a major shift in the organization's culture. It must also be noted that Agile or Scrum frameworks are not a silver bullet and are not suited for every software product development. Often, large organizations use some components of the agile framework in conjunction with other methodologies as a middle-ground.

DevOps

Software Engineering Operations teams continue to strive to provide a consistent environment for global development. They engineer the products from the hands of the developers to the hands of the customers. Agile methodologies proved that collaboration and people must be top priority in software development. An extension to that idea is to break the barriers between development and operations teams further, resulting in the concept of DevOps. In some ways, DevOps can be thought of as extending the principles of agile software development. Silos are further broken down and development, quality assurance, and operations teams all act without any barriers.

One of the best practices of DevOps is Continuous Integration (CI), an idea proposed by Grady Booch, the inventor of the famed Unified Modeling Language, UML. The idea is to provide immediate feedback to the developer about their code changes and almost always have a working product that can be tested and possibly released. The code changes need to meet several criteria such as being buildable, pass sanity tests, go through static analysis checks successfully, reviewed and approved by peers/module owners, and so on. Most of the checks happen automatically. The code can be integrated into the product only if all the checks pass. Thus, all integration issues are addressed immediately, in a sharp contrast with what was described in the section on Integration Engineering. Continuous Integration, therefore, becomes the basis for all subsequent operations and automation.

Unlike huge changesets getting propagated across branches through the handshakes described in integration engineering, the changesets in Continuous Integration are small, much more manageable, and iterative. The automation around CI is crucial for developers to remain productive. Hence the need for tools – several of them – so many that a periodic table can be filled with them and even more. The pivotal tool is the CI engine, which does much more than the traditional 'cron' on Unix machines that typically spun off the builds in the waterfall model. There are currently many tools that function as a CI engine today. Jenkins, Travis, and Bamboo are a few of such CI engines. These CI engines take the code changes from the developers through a series of checks to validate the code diffs. The sequence of checks can be envisioned as a ‘pipeline,’ quite analogous to the line of pipes that transport liquids and gases to a production area. Just like the commercial liquid and gas pipelines are equipped with the required control devices, the CI engine pipelines have the necessary mechanisms to control the processes that take the code changes through the validations.

Along with continuous integration, there is a need for continuous testing as well, so that the developers get feedback on quality aspects, continuously. When the product is continuously tested, it is ready for deployment in production continuously as well, resulting in hundreds or even thousands of releases in a day. Continuous Integration, Continuous Testing, Continuous Deployment, and Continuous Delivery lead to continuous improvement. All these continuous processes can be implemented using the ‘pipelines’ that the CI engines provide. As can be envisioned, the pipelines can easily grow in complexity. The trend now is to ‘code’ the CI engine pipelines, so that they can be maintained better and there is change history. ‘Pipeline as Code’ often resides in the same repository as the source code.

Cloud computing has come to stay. Today, most of the computing, including that which happens in the pipeline, run in a private or public cloud. Cloud computing and virtualization enable spinning up a ‘virtual’ machine (VM) in no time. Multiple VMs, possibly running different operating systems can run on the same bare metal hardware providing isolation and optimal usage. Cloud computing provides access to the VMs seamlessly across the network, even if the bare metal machines are miles away and are owned by a 3rd party. A lightweight model of a VM is a container, which can run on a VM, providing an isolated environment for an application to run. The container packages any given application along with all its dependencies including configuration files and libraries so that the application is ready to run as soon as the container is brought up – quite convenient for testing and deploying as part of the pipeline. A container image is immutable so that it can be run and rerun many times.

The container image contains everything that an application needs to run and serves as an immutable snapshot of the application’s runtime environment. Multiple containers share the kernel running on physical hardware and provide isolated namespaces for the application to run. Therefore, a container includes its abstraction of memory, devices, network ports, processes, and filesystems, shielding the underlying kernel’s resources from direct access. The containers resources eventually use the resources provided by the underlying kernel but do not let the applications access them directly. Containers provide great portability suitable for instant deployment, particularly when using a microservices architecture. As a general guideline, all builds should be reproducible. Reproducibility is particularly important for production builds or builds which go out to customers. Containerization can help in reproducibility of builds since a container image can effectively store the configuration needed for a build to be reproduced.

Some of the functions that the DevOps teams perform are shown in Figure 2. As can be seen, the DevOps teams are responsible for most of the operations in software development, starting with setting up the repository to deploying and shipping the releases. Each one of these functions needs to be automated and automation requires tools. Hence the explosion of tools. For instance, the number of artifacts that are needed for the build and produced by it has grown so much that we now have tools like Artifactory and Nexus to handle them. Source code itself is versioned in tools like Git and Subversion. Huge files like the binary artifacts are not usually versioned with the source code, hence separate tools for them. For testing, we have tools like Selenium, JUnit, and TestNG. ElectricFlow and Julu help with deployment. Metrics and dashboards play an important role in monitoring and improving productivity. In the DevOps world, it is said that if it is measured, it is bound to improve. Tools like Kibana and Nagios help in creating dashboards that can show metrics.

Docker and Kubernetes are popularly used tools for containerization and their orchestration respectively. Configuration and provisioning tools include Chef, Puppet, and Ansible. Coverity and SonarQube are two of the tools that help in static analysis of the source code to detect any vulnerabilities and potential bugs, without actually running the code. Tools like Cobertura, JaCoCo, and Valgrind are used for measuring code coverage statistics. As we saw, collaboration plays a crucial role in software development and is one of the main driving forces for the DevOps movement. Multiple tools like Slack, HipChat, and Webex Teams are popularly used for instant messaging and collaboration. In addition to these open-source or commercially available tools, most large organizations have their internal tools to handle several software development operations. For instance, Cisco has its huge bug tracking system called CDETS and release posting tool called IRT.

Code bloating and code obsolescence is quite common over time. As highlighted in Figure 2, the DevOps team needs to work on reducing the code footprint and explore other ways to reduce the build times to reduce the wait-time for the developers to get feedback about their code changes. In some cases, particularly when the software is embedded, there are strict limits on how much memory the software can consume at runtime, requiring a check to be placed on the incremental size of the image built from the code changes. This is an example of a policy that needs to be put in place. As can be seen, software development is a disciplined activity, which needs to be regulated by several policies. Some of the other policies could be to allow commits only after sufficient approving reviews, mandate a double-commit to the master branch before committing to a release branch, and so on. The DevOps team is responsible for enforcing the policies. Instrumenting such mechanisms and the software development environment in general requires plenty of tooling on part of the DevOps teams. It is not hard to see that DevOps is, therefore, a substantial charter requiring strong technical and analytical skills.

DevOps to DevSecOps

Security is everyone's business, even in the software industry. Application security is critical, given their usage profile. That part has not changed, but the way security is achieved has gone through substantial changes due to paradigm shifts in the development processes. Traditionally, as shown in Figure 3a, boundaries were secured using firewalls. Companies and applications operated in silos. Development and Operations too operated in silos and were not well orchestrated. DevOps fixed the broken collaboration mechanisms and provided for continuous, seamless operations. Security continued to be ensured by protecting the organization's borders.

The scenario is depicted in Figure 3b. However, as cloud computing gained in adoption, borders weakened, and computing happened across borders. It was no longer enough to protect the

corporate borders using firewalls. Security had to be built into the application, resulting in the "Security as Code" paradigm and the birth of DevSecOps, as depicted in Figure 3c.

Cloud computing and DevOps brought in a series of "…as a Service" and "…as Code" paradigms, such as "Infrastructure as a Service," "Infrastructure as Code," and "Pipeline as Code." DevSecOps continued the trend with the "Security as Code" paradigm, taking a holistic view of security. Like DevOps, DevSecOps has to do

Figure 2. Typical responsibilities of the DevOps team

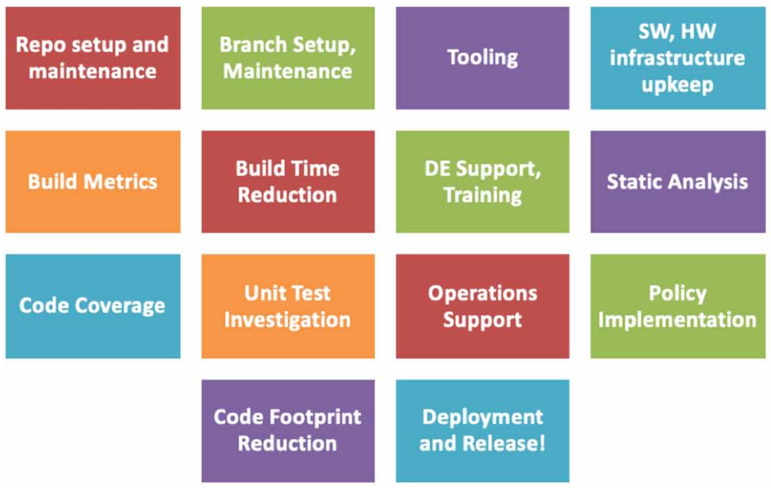
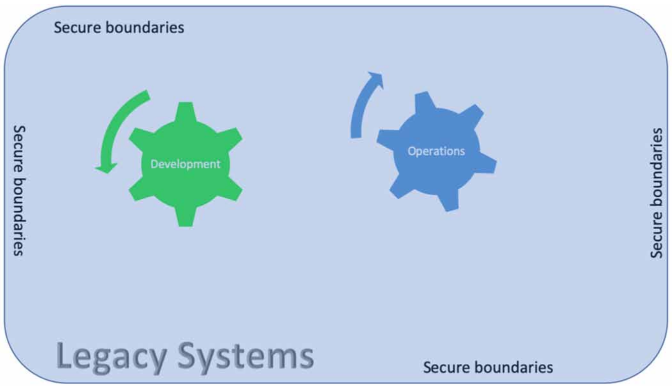


Figure 3a. Security in legacy software systems



a lot with the corporate mindset and is a culture shift. It can be viewed as a set of tools, techniques, and processes to build security into software. It requires buy-in from all stakeholders and is a community-driven effort. DevSecOps is still evolving through learning and exploration. With security moving into the application, security infrastructure needs to be 'cloud-aware' and security features need to be published via APIs. Security aspects are now built into the CI engine pipeline and automation tooling as much as possible. Security is part of the software building process as illustrated in Figure 4.

Development, security, and operations are the new building blocks of a software organization.

Figure 3b. Security with the advent of DevOps

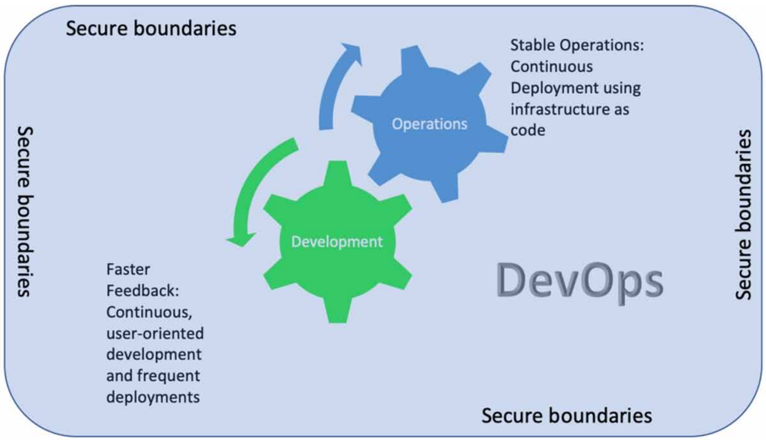
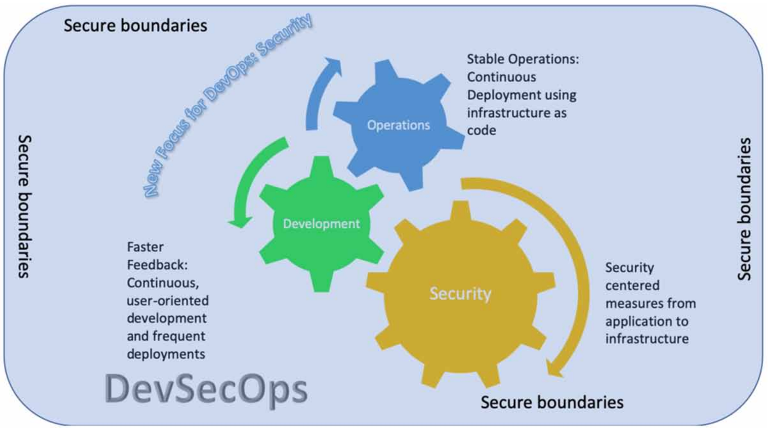
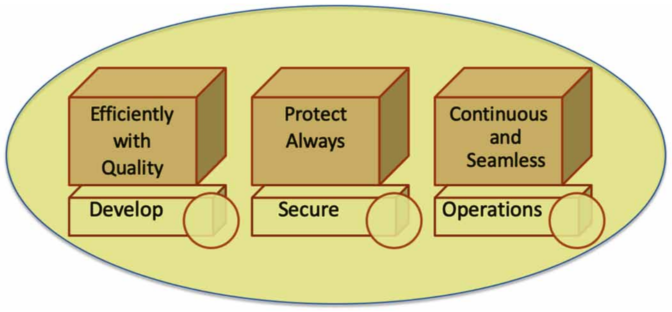

Secure boundaries
Operations
Stable Operations: Continuous Deployment using infrastructure as code
Secure boundaries
Development
Faster Feedback: Continuous, user-oriented development and frequent deployments
DevOps
Secure boundaries
Secure boundaries

Figure 3c. Security in DevSecOps


Secure boundaries
New Focus for DevOps: Security
Stable Operations: Continuous Deployment using infrastructure as code
Operations
Development
Security
Secure boundaries
Secure boundaries
Faster Feedback: Continuous, user-oriented development and frequent deployments
Security centered measures from application to infrastructure
DevSecOps
Secure boundaries

DevOps broke the silos between Development and Operations teams. DevSecOps extends the idea and broke the silos between the Security teams and the DevOps teams. DevSecOps orchestrates the workflows among the development, security, and operations teams to provide an integrated, seamless infrastructure for the development of the product. Security vulnerabilities in the code are continuously monitored and addressed paving way for "Continuous Security." Products are always security-ready, in addition to being deployable with every code commit. Product security is therefore tightly coupled with the pipeline controls. For instance, continuous testing now requires security aspects to be tested as well as part of the code commit validations in the pipeline. Security, which came into the picture in the later stages of software development, now needs to "shift left," to earlier stages of development as well, right from the beginning. There must now be at least a few agile user stories related to security in every sprint if agile methodologies are being used.

Figure 4. Building blocks of a software organization



Issues, Controversies, Problems

The DevSecOps area is still evolving and poses multiple challenges. It is a culture shift and driving change across organizations continues to be a challenge. Roadshows within the organization, identifying security champions to serve as brand ambassadors for DevSecOps, and promoting the benefits of DevSecOps by other means are some of the techniques that can be used to make the culture shift. Security certainly raises the complexity of the applications. Architecture changes to accommodate security aspects as applicable to on-premises, cloud, and container deployments must be considered right from the beginning. A security mindset must be inculcated among cross-functional teams.

Skilled manpower continues to be a challenge in the DevSecOps area. The author personally interviewed scores of candidates for open positions in his team and found that many engineers have restricted themselves to mere tool configuration and usage, without much experience at all in writing substantial scripts and implementing tools from scratch or understanding the underlying principles. It is also observed that some engineers continue to work in older waterfall methodologies and tools, without much exposure to the latest trends in the industry. Organizations, particularly the large, well-established ones must learn to quickly adopt newer technologies and train their personnel for the change. It is hard to drive change, but the risk of obsolescence should be enough motivation to move with the industry.

Another major challenge is the budget allotted for DevSecOps. The higher management may not always see the value or the complexity of the DevSecOps tasks, resulting in understaffed DevSecOps teams and inadequate tooling infrastructure. In such cases, it may help if the first-line managers and technical leads of DevSecOps teams meet with the higher management to impress upon the critical value that the DevSecOps methodologies provide and the complexities involved in them. It is also helpful to standardize the tool and process usage across large organizations, so that interoperability if needed, is better achieved. Legacy tools can pose challenges in terms of scaling and adapting to growing needs. It is imperative to quickly identify infrastructure that is not able to keep up and replace it with the industry-standard tooling.

FUTURE RESEARCH DIRECTIONS

The Software Engineering journey will of course not stop at DevSecOps and full-stack engineering. A hot area that is still evolving is implementing DevSecOps for Artificial Intelligence products and using Artificial Intelligence for DevSecOps. Machine Learning is the mortar of modernization and is becoming more and more ubiquitous. Machine Learning approaches can be used to detect security vulnerabilities and bugs in general. Analyzing the logs from the tools using AI techniques can help improve the quality of the tools – an area that can benefit from more research. There is also ample scope for building tools to integrate security aspects into the pipelines.

CONCLUSION

This chapter briefly examined the evolution of the Software Engineering domain into today's DevSecOps, presenting important tools, techniques, and observations, all along. Several aspects of Software Engineering have transformed drastically over the last three decades. For instance, the simple 'cron' in the Unix systems has now become a full-blown Continuous Integration engine acting as the backbone of the DevSecOps revolution. The chapter also identified a few challenges and solutions to address them. The domain continues to evolve further and holds plenty of promise for the future.

ACKNOWLEDGMENT

The author gratefully acknowledges the experience gained from various software organizations that was instrumental in writing this chapter.

REFERENCES

Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., & Tamburri, D. A. (2017, May). DevOps: introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (pp. 497-498). IEEE. 10.1109/ICSE-C.2017.162

Banica, L., Radulescu, M., Rosca, D., & Hagiu, A. (2017). Is DevOps another Project Management Methodology? *Informações Econômicas*, *21*(3), 39–51. doi:10.12948/issn14531305/21.3.2017.04

Bucena, I., & Kirikova, M. (2017). Simplifying the DevOps Adoption Process. BIR Workshops.

Chen, L. (2018, April). Microservices: architecting for continuous delivery and DevOps. In *2018 IEEE International Conference on Software Architecture (ICSA)* (pp. 39-397). IEEE. 10.1109/ICSA.2018.00013

Cito, J., Wettinger, J., Lwakatare, L. E., Borg, M., & Li, F. (2018, March). Feedback from Operations to Software Development—A DevOps Perspective on Runtime Metrics and Logs. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (pp. 184-195). Springer.

Dyer, M. (1980). The management of software engineering, Part IV: Software development practices. *IBM Systems Journal*, *19*(4), 451–465. doi:10.1147j.194.0451

Erich, F. M. A., Amrit, C., & Daneva, M. (2017). A qualitative study of DevOps usage in practice. *Journal of Software: Evolution and Process*, *29*(6), e1885.

Feitelson, D. G., Frachtenberg, E., & Beck, K. L. (2013). Development and deployment at Facebook. *IEEE Internet Computing*, *17*(4), 8–17. doi:10.1109/MIC.2013.25

Jiménez, M., Castaneda, L., Villegas, N. M., Tamura, G., Müller, H. A., & Wigglesworth, J. (2018, March). DevOps round-trip engineering: Traceability from dev to ops and back again. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (pp. 73-88). Springer. 10.29007/gq5x

Kaiser, A. K. (2018). Introduction to DevOps. In *Reinventing ITIL® in the Age of DevOps* (pp. 1–35). Berkeley, CA: Apress. doi:10.1007/978-1-4842-3976-6_1

Li, Z., Zhang, Y., & Liu, Y. (2017). Towards a full-stack DevOps environment (platform-as-a-service) for cloud-hosted applications. *Tsinghua Science and Technology*, *22*(01), 1–9. doi:10.1109/TST.2017.7830891

Lwakatare, L. E., Karvonen, T., Sauvola, T., Kuvaja, P., Olsson, H. H., Bosch, J., & Oivo, M. (2016, January). Towards DevOps in the embedded systems domain: Why is it so hard? In *2016 49th Hawaii International Conference on System Sciences (HICSS)* (pp. 5437-5446). IEEE.

Myrbakken, H., & Colomo-Palacios, R. (2017, October). DevSecOps: a multivocal literature review. In *International Conference on Software Process Improvement and Capability Determination* (pp. 17-29). Springer. 10.1007/978-3-319-67383-7_2

Rahman, A. A. U., & Williams, L. (2016, May). Software security in DevOps: synthesizing practitioners' perceptions and practices. In *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)* (pp. 70-76). IEEE. 10.1145/2896941.2896946

Sánchez-Gordón, M., & Colomo-Palacios, R. (2018, October). Characterizing DevOps Culture: A Systematic Literature Review. In *International Conference on Software Process Improvement and Capability Determination* (pp. 3-15). Springer. 10.1007/978-3-030-00623-5_1

Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., & Stumm, M. (2016, May). Continuous deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)* (pp. 21-30). IEEE. 10.1145/2889160.2889223

Stahl, D., Martensson, T., & Bosch, J. (2017, August). Continuous practices and DevOps: beyond the buzz, what does it all mean? In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 440-448). IEEE. 10.1109/SEAA.2017.8114695

Sureshchandra, K., & Shrinivasavadhani, J. (2008, August). Moving from waterfall to agile. In Agile 2008 conference (pp. 97-101). IEEE. doi:10.1109/Agile.2008.49

Van Der Hoek, A., Carzaniga, A., Heimbigner, D., & Wolf, A. L. (1998). *A reusable, distributed repository for configuration management policy programming*. Univ. Colorado, Boulder, Tech. Rep. CU-CS-864-98.

Wiedemann, A., & Wiesche, M. (2018). Are you ready for DevOps? Required skill set for DevOps teams. *Proceedings of the European Conference on Information Systems*.

ADDITIONAL READING

Allen, L., Fernandez, G., Kane, K., Leblang, D., Minard, D., & Posner, J. (1993). ClearCase MultiSite: Supporting geographically-distributed software development. In *Software Configuration Management* (pp. 194–214). Berlin, Heidelberg: Springer.

Bartusevics, A., & Novickis, L. (2015). Models for implementation of software configuration management. *Procedia Computer Science*, *43*, 3–10. doi:10.1016/j.procs.2014.12.002

Dyck, A., Penners, R., & Lichter, H. (2015, May). Towards definitions for release engineering and DevOps. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering* (pp. 3-3). IEEE. 10.1109/RELENG.2015.10

Mohan, V., & Othmane, L. B. (2016, August). SecDevOps: Is it a marketing buzzword?-mapping research on security in DevOps. In *2016 11th International Conference on Availability, Reliability and Security (ARES)* (pp. 542-547). IEEE.

Rahman, A. A. U., & Williams, L. (2016, May). Software security in DevOps: synthesizing practitioners' perceptions and practices. In *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)* (pp. 70-76). IEEE. 10.1145/2896941.2896946

Schwägerl, F., Buchmann, T., Uhrig, S., & Westfechtel, B. (2015, February). Towards the integration of model-driven engineering, software product line engineering, and software configuration management. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)* (pp. 1-14). IEEE.

Ur Rahman, A. A., & Williams, L. (2016, April). Security practices in DevOps. In *Proceedings of the Symposium and Bootcamp on the Science of Security* (pp. 109-111). ACM. 10.1145/2898375.2898383

Wiedemann, A., Forsgren, N., Wiesche, M., Gewald, H., & Krcmar, H. (2019). The DevOps Phenomenon. *Queue*, *17*(2), 40.

Williams, L. (2018, May). Continuously integrating security. In *Proceedings of the 1st International Workshop on Security Awareness from Design to Deployment* (pp. 1-2). ACM.

Yasar, H. (2017, August). Implementing Secure DevOps assessment for highly regulated environments. In *Proceedings of the 12th International Conference on Availability, Reliability and Security* (p. 70). ACM. 10.1145/3098954.3105819

KEY TERMS AND DEFINITIONS

Artificial Intelligence: An area of Computer Science that involves writing programs that can do things that would otherwise require human intelligence.

Everything as Code: A concept that everything that is needed to implement the software lifecycle can be treated as code, for example, pipeline as code.

Machine Learning: A branch of Artificial Intelligence which involves writing programs that can identify patterns, learn from data, and make predictions.

Pipeline as Code: Use a programming language to specify what needs to happen in the pipeline and version the file containing this 'pipeline program' along with the source code, so that it is much more maintainable.

Shift-Left: Assuming that the software lifecycle is drawn from left to right in chronological order, move certain aspects such as testing and security, which were previously done towards the end, to the earlier phases of the software development lifecycle.

Source Code Branch: An artifact in a version control system such as Git that allows parallel and independent development in the same files, unbeknownst to each other, until the branches merge.

Workflow: A series of processes through which software code changes need to go through from conception to product completion.