Systems Engineering and DevSecOps: Reviewing the Principles

Dr. Richard Turner, rturner@sei.cmu.edu

As software engineering adopts a more continuous delivery mode for embedded and complex systems, systems engineering must both adapt and influence DevSecOps and related practices. In this article, I revisit the principles of agile, lean and DevSecOps, and provide a commentary on possible model clashes or disconnects that could be increase risks to system development, deployment, and evolution. (Much of the material in this article was previously presented a Software Engineering Institute Blog Post.)

Are there fundamental issues?

I believe there are. The interaction of these two disciplines is not well understood, and experience from early application suggests model clashes between them. The following table identifies some of the fundamental differences between systems engineering as generally practiced and systems engineering for evolving software engineering environments. Mitigation of the clashes could enhance the success rate of DevSecOps adoption and support adjustments to both disciplines. However, mitigation requires identifying the specifics and understanding the context and sources of the clashes.

Systems Engineering as Generally Practiced	DevSecOps-Based Systems Engineering
Large-batch processing (products, documents, events)	Small batch processing (products, documents, events)
Single-pass lifecycle (all requirements done before the design is initiated; all design done before implemented)	Incremental, iterative multi-pass lifecycle (small batches of products and their artifacts built/tested iteratively, delivered incrementally)
Single-point design	Set-based design
Solution intent fixed early (all requirements defined in detail early)	Most of solution intent variable early (only near-term requirements in detail; others are higher level with details based on learning)
Fixed point, large-batch integration (components all “done” before integration occurs)	Cadence-based, small-batch integration used as frequently as feasible; integrate as available to prevent rework (for software, may be daily)
Centralized, command-and-control leadership	Mix of centralized and decentralized leadership; “servant leadership”
Detailed, allocated baseline early; high overhead change management practices in play for the rest of development	Allocated baseline level of abstraction allows learning-based change throughout development; no high-overhead change processes
Hardware and software treated separately, integrated late	Hardware and software treated together, integrated early and frequently
Large-batch model-based engineering used to improve the detail of requirements and design prior to implementation; often abandoned after design	Model-based engineering moves between large- and small-batch modeling activities; models and simulations flow with implementation and support the full lifecycle, development through sustainment
Projective (to be) requirements and design documentation dominates early discussion and activities	Projective documentation takes second place to working prototypes and demos; used to guide, not specify; documentation is as-built, not to-be.
systems engineering function separate from hardware and software development functions	systems engineering function integrated into capability-focused teams that include all disciplines needed (HW, SW, UX, reliability, etc.)
Component-based work breakdown structure	Capability-based work breakdown structure
systems engineering primarily as artifact transformation (e.g., Requirements->Architecture->Design)	systems engineering as a service (facilitation of artifact transformation; focus on communication, coordination, conflict resolution, collaboration)
System architecture decisions neutral to development approach	System architecture decisions strongly support loosely coupled components/subsystems, especially for software capabilities
Assumption that early work is correct and that late failure is a surprise	Assumption that early work is inherently flawed, and learning from early failure feeds the evolution of knowledge about the system
System and software architecture frozen early	Intentionally extendable and iteratively evolving architecture throughout development and sustainment
User participation only early and late	User participation continuous throughout lifecycle

Due to the breadth of domains covered by both disciplines, I have gone back to the basic principles of each to better understand the model clashes. Systems engineering principles are generally less focused on activities than the lean, agile, and DevSecOps principles. I therefore present them first and then discuss the DevSecOps principles in terms of their interaction with the systems engineering principles and activities.

Systems Engineering Principles and Activities

The Systems Engineering Body of Knowledge (SEBoK) defines systems engineering as

> *"…a **transdisciplinary** approach and a means to characterize and manage the development of successful systems, where a successful system satisfies the needs of its customers, users, and other stakeholders. Systems engineering focuses on holistically and concurrently understanding stakeholder needs; exploring opportunities; documenting requirements; and synthesizing, verifying, validating, and evolving solutions while considering the complete problem, from system concept exploration through system disposal."*

Systems engineering principles have generally not been as visible as those for DevSecOps. Earlier lists have recently been revisited by the NASA Systems Engineering Research Consortium to address some of the differences identified in Table 1, but the adoption of these refined principles by practitioners is unknown. The principles are somewhat generic because they must apply across so many domains. Here are the 14 NASA principles (I've highlighted some of the key concepts for this article).

NASA Systems Engineering Research Consortium Systems Engineering Principles	
Principle 1: Systems engineering **integrates the system and the disciplines** considering the budget and schedule constraints.	Principle 2: Complex systems build complex systems.
Principle 3: A focus of systems engineering during the development phase is a **progressively deeper understanding** of the interactions, sensitivities, and behaviors of the system, stakeholder needs, and its operational environment.	Principle 4: Systems engineering has a critical role through the entire system lifecycle.
Principle 5: Systems engineering is based on a middle-range set of theories.	Principle 6: Systems engineering maps and **manages the discipline interactions** within the organization.
Principle 7: Decision quality **depends on the system knowledge** present in the decision-making process.	Principle 8: Both policy and law must be properly understood to not overly constrain or under constrain the system implementation.
Principle 9: Systems engineering decisions are made under **uncertainty**, accounting for **risk**.	Principle 10: **Verification** is a demonstrated understanding of all the system functions and interactions in the operational environment.
Principle 11: **Validation** is a demonstrated understanding of the system's value to the system stakeholders.	Principle 12: Systems engineering solutions are constrained based on the **decision timeframe** for the system need.
Principle 13: **Stakeholder expectations change** with advancement in technology and understanding of system application.	Principle 14: The real physical system is the only perfect representation of the system.

Comparison of Systems Engineering to Lean-Agile Principles

DevSecOps success relies heavily on the application of fundamental Lean and Agile principles. The following sections present short descriptions of Lean-Agile and DevSecOps principles along with a short description of key related systems engineering activities. Given that there are numerous versions of Agile and Lean principles, I have used the collective principles as articulated in the SAFe Scaled Agile Framework as being most comparable to systems engineering:

Principle 1: Take an Economic View. Decisions are made by comparing clearly stated or unconsciously considered values. In systems development, specifically addressing values allows decisions to be made in an economic framework. Value should be a factor in prioritization and sequencing of work.

Understanding and intentionally capturing value in requirements and design components as they are seen by the multiple stakeholders enables better impact analyses and prioritization in development and

sustainment. Using a common value-determination process, including the gamut of stakeholders, can provide visibility into decisions, support decisions at deeper and deeper layers of implementation, and support temporal, internal and external influences that impact aspects of value. Appendix C of *The Incremental Commitment Spiral Model (ICSM): Principles and Practices for Successful Systems and Software* provides a discussion of values-based systems engineering.

Principle 2: Apply Systems Thinking. Systems thinking broadens the focus of development to encompass the full value stream in acquisition, development, and operational organizations. It considers more factors than those related to requirements or how the product system operates; it enables understanding of the socio-technical system that encompasses the product and its context.

Nearly all systems engineering incorporates systems thinking by definition. Understanding the full scope of the effort (including the DevSecOps activities and requirements) and the associated value streams and networks are critical to the holistic nature of systems thinking.

Principle 3: Assume Variability; Preserve Options. Locking in a single, detailed description of a system that will take years to develop can become a barrier as soon as a change in one or more naturally evolving factors--threats, political landscapes, economics, technology, or markets--invalidates an assumption or specification. Acquirers and developers must acknowledge that variability and uncertainty are facts of life, and that investing in and maintaining options with decisions made at the last responsible moment is a good way to manage change.

While there are specific systems engineering tasks that look at risk management, safety, and security-failure modes, there is less activity associated with understanding how environmental changes impact the actual development, once approved. Identifying useful options and managing the impact of changes require ongoing resources and intentional activities.

Principle 4: Build Incrementally with Fast, Integrated Learning Cycles. This principle provides rapid feedback on estimates, assumptions, and feasibility quickly enough to eliminate much of the high cost of rework. Coupled with small batch size, it provides a high degree of stability in work planning and enhanced agility to take advantage of opportunities resulting from uncertainty and variability. It eliminates much of the overhead of maintaining large, monolithic and generally inaccurate master schedules and focuses on delivering value quickly.

This principle is a key area of concern. Systems engineering generally drives software development and sustainment to the bottom of the traditional V model. Adaptation to the continuous, incremental, and iterative nature of DevSecOps forces an earlier and sustained focus on the software-related systems engineering activities. The cultural challenge for systems engineering is moving from relatively rare interactions to a continuous involvement in the software development and evolution.

Principle 5: Base Milestone Completion on the Objective Evaluation of Working Systems. Milestones are traditionally treated as gates, with passage based on a set of static technical artifacts with little evidence of their completeness or accuracy. Demonstration of status is more useful and provides more learning opportunities.

Technical reviews (particularly those in support of milestone gates and progress measurement) are often predicated on boilerplate documentation, overly formalized plans, incomplete or inadequately vetted requirements, or design specifications that include guesses made to remove "to be determined" items rather than acknowledging further analysis is required at the milestone. The scope is also often very broad, driven by the complex scheduling of critical resources.

Principle 6: Visualize and Limit Work in Progress (WIP), Reduce Batch Sizes, and Manage Queue Lengths. Visualizing and limiting work in progress regulates the number of tasks that are being worked on at any one time. It also keeps the human resources from being overwhelmed by the context switching between tasks. Managing batch size and queue lengths supports the focus on WIP with the principle of "stop starting and start finishing," since the user gets value only with completed work, and work waiting in a queue is a waste.

Systems engineering is often understaffed, and the continuous nature of the DevSecOps environment puts a strain on available systems engineering resources. Understanding how much work is being

expected and its production rate supports maximizing the flow and increasing the value of many systems engineering activities. Staffing practices are a significant factor for systems engineering in applying this principle.

Principle 7: Apply Cadence and Synchronize with Cross-Domain Planning. While predictive or "push" scheduling usually ignores uncertainty, management and users need reasonable estimates. Setting cadences and synchronizing across the various teams and activities is the Lean answer to bounding uncertainty and are essential to:

- provide a predictable cycle of results and feedback opportunities;
- align metrics;
- convert unpredictable events into predictable ones;
- provide opportunities to understand, resolve, and integrate the work of multiple teams, and at the same time, manage multiple stakeholder perspectives.

Aligning different cadences between systems engineering and software engineering activities is a challenge; adjustments should not reduce the value of either discipline.

Principle 8: Unlock the Intrinsic Motivation of Knowledge Workers. To ensure motivation and engagement among team members, create an environment marked by autonomy, mutual respect, and mission understanding.

Most systems engineering technical activities are likely unaffected by this principle. However, effectively managing the systems engineering workforce entails consideration of whether the systems engineering personnel are sufficiently engaged by software engineering and other disciplines to maintain interest, as well as situational awareness. This principle is particularly important in large complex programs, such as weapons systems, highly regulated systems, and systems of systems, where the work may be spread across a large number of organizations or companies.

Principle 9: Decentralize Decision Making. Decentralized decision making is a key component for achieving the shortest sustainable value-delivery time. Decisions that require sequential acceptance by multiple levels of authority can destroy cadence, delay progress, and often lead to decisions based on outdated information. Strategic decisions are more effective if centralized, but all others should be delegated to the level closest to the information involved.

Most systems engineering activities support rather than make decisions. Regardless of the decision maker, recommendations made by the systems engineering workforce should be accomplished by those closest to the problem. It is critical that those making a recommendation have sufficient access to information and the scope of visibility to understand the systemic consequences of those recommendations. Analysis paralysis is contagious and should not be allowed to become a factor (See variability and options above.).

Comparison of Systems Engineering to DevSecOps Principles

DevSecOps principles are built on the Lean, Agile, and DevOps principles. DevSecOps broadens these principles and applies them to integrate development, security, and operations activities into a continuous integration/continuous deployment (CI/CD) pipeline. The SEI Guide to Implementing DevSecOps for a System of Systems in Highly Regulated Environments defines these principles as follows:

Principle 1: Collaboration. Full stakeholder engagement in every aspect of the software development lifecycle facilitates full awareness and input on all decisions and outcomes. Developers, operators, engineers, end users, customers, and other stakeholders are active participants in decision making and work progress.

Having ongoing access to systems engineering expertise is key in maintaining DevSecOps activities. In the same way, having software engineers involved in the technical systems engineering activities reduces the opportunities for significant conflict and associated rework. Collaboration with project and program management can also be improved with collaboration among systems and software engineers.

Principle 2: Infrastructure as Code (IaC). IaC are software artifacts that specify the hardware/software components needed to run correctly, as well as the details of how each should be accessed, configured, and installed. Infrastructure components can be actual, virtualized, or a mix of both.

While IaC is not specifically a systems engineering activity, the use of IaC provides for more complete documentation of the execution environment maintained in the same repository as the code, and supports the configuration management issues that often plague software and system components. It also eases the transition of the code to an altered or completely new environment by providing a clear description of what was expected and identifying what software components may need to be changed.

Principle 3: Continuous Integration. Continuous integration is often and automatically unifying individual components of a system into a single entity. Unification occurs on a regular basis. The components, once unified, are meant to function together as a whole. The components may have dependencies on one another to function properly.

When coupled with IaC, continuous integration is the implementation of short learning cycles/increments that allows systems engineering continuous visibility into the state of the code and assures that code being developed by teams or teams of teams will not run into unexpected integration problems late in the development cycle. Rather than developing multiple components or capabilities in separate insular silos, continuous integration enables rapid access to integration issues before they cause significant rework. (See also the Environment Parity principle.)

Principle 4: Continuous Delivery. Continuous Delivery refers to the automated transfer of software to a staging environment that has parity with the production environment. Once delivered, the operations organization may conduct further testing, but must decide whether and when to manually deploy the software into production. An example of this would be unclassified software that runs on classified data produced by another system and that may be independently changing; operations may want independent testing using live data before deployment. It also allows the operations team to decide if a set of updates are of enough value to deploy.

Principle 5: Continuous Deployment. Continuous deployments need no operations team activity and transfers operational software directly into a production environment. It relies solely on the rigorous static testing of source code and dynamic testing of deployable artifacts within the CI/CD pipeline.

Both of the continuous modes pass the fully integrated and tested software, including complete documentation and deployment information, to the operational organization. A continuous mode of transition to the user provides a more rapid resolution for evolving cybersecurity vulnerabilities. While both modes limit delay in the delivery of capability, each provides for different circumstances. When the testing is completed in a duplicated operational environment, the concept of continuous deployment makes sense. If there is not absolute congruity between the testing environment and the operational environment--perhaps because of security- or infrastructure needs--continuous delivery allows the organization to adjust the cadence of deployment to their need without impacting the velocity of the software development.

Continuous delivery/deployment provides systems engineering with a sequence of complete, fully documented software. The drawbacks include the level of trust required and in the rapid baseline evolution.

Principle 6: Environment Parity. When two or more system environments are as identical as possible, they are said to be in parity. In DevSecOps, parity is pursued between development, staging, and production environments. IaC and deployable artifacts are critical to achieving parity.

Like IaC, maintaining environment parity supports the continuous integration and accelerates certain kinds of testing. An example of maintaining environment parity is including security testing from the initial development all the way through deployment. If the environment is constantly changing, there is greater risk of significantly delaying the identification of a defect due to an environmental anomaly.

Principle 7: Automation. A pipeline is the technical implementation of DevSecOps principles that assists all stakeholders in every aspect of software development including building, testing, delivery, and monitoring. For engineers, the main use of a pipeline is to continuously and iteratively build, integrate,

test, and deliver or deploy code through automation. For the purposes of software development, a pipeline is used for code development and for project management.

Automation has a significant impact on systems engineering by providing significant visibility in the status of the software and providing for verification and validation (V&V) activities throughout the lifecycle. It ensures that testing at every level is always performed, and that no package can be signed off until it has been integrated and tested. Automation also enables earlier and consistent inclusion of V&V across systems and components.

Principle 8: Monitoring. Continuous monitoring of performance metrics simultaneously drives pipeline improvement and the quality of the software under development. Security is also monitored for both the software being developed and for the pipeline automation.

So now what?

Now that we have compared the principles, it appears that the principles align fairly nicely, but the foci of the practices are very different. It is clear that the details of agile, lean and DevSecOps are fairly narrow, very specific and are designed to be highly automated. Systems engineering takes a broader perspective in the sense of incorporating the broader, systems view. These should be mutually supportive. Unfortunately, the context, values, and incentives of many practices run counter to other practices In and between both disciplines. This is not insurmountable, but there needs to be collaboration on mitigations and solutions. Hopefully, there is growing understanding by both disciplines of the needs and goals of the other, and the general alignment of principles will provide room for innovation and improving outcomes.

Acknowledgements

Copyright 2021 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

DM21-0296

Additional Resources

The SEI Technical Report Guide to Implementing DevSecOps for a System of Systems in Highly Regulated Environments by Jose Morales, Richard Turner, Suzanne Miller, Peter Capell, Patrick Place, and David James Shepard.

The SEI Technical Note Agile Software Teams: How They Engage with Systems Engineering on DoD Acquisition Programs by Eileen Wrubel, Suzanne Miller, Mary Ann Lapham, and Tim Chick.

The SEI Webinar DevSecOps Implementation in the DoD: Barriers and Enablers with Hasan Yasar, Eileen Wrubel and Jeff Boleng.

The SEI presentation video Continuous Iterative Development and Deployment Practices With Hasan Yasar and Eileen Wrubel

The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software, a 2013 book by Barry Boehm, Jo Ann Lane, Supannika Koolmanojwong, and me. Appendix C of the book discusses a value-based theory of systems engineering; an earlier version of that material can be found here.

The SEI Blog eight-part series Challenges to Implementing DevOps in Highly Regulated Environments by Jose Morales.