Defense Innovation Board
Ten Commandments of Software

Executive Summary

The Department of Defense (DoD) must be able to develop and deploy software as fast or faster than its adversaries are able to change tactics, building on commercially available tools and technologies. Recognizing that "software" can range from off-the-shelf, non-customized software to highly-specialized, embedded software running on custom hardware, it is critical that the right tools and methods be applied for each type. In this context we offer the following ten "commandments" of software acquisition for the DoD:

1. Make computing, storage, and bandwidth abundant to DoD developers and users.
2. All software procurement programs should start small, be iterative, and build on success – or be terminated quickly.
3. The acquisition process for software must support the full, iterative life cycle of software.
4. Adopt a DevSecOps culture for software systems.
5. Automate testing of software to enable critical updates to be deployed in days to weeks, not months or years.
6. Every purpose-built DoD software system should include source code as a deliverable.
7. Every DoD system that includes software should have a local team of DoD software experts who are capable of modifying or extending the software through source code or API access.
8. Only run operating systems that are receiving (and utilizing) regular security updates for newly discovered security vulnerabilities.
9. Security should be a first-order consideration in design and deployment of software, and data should always be encrypted unless it is part of an active computation.
10. All data generated by DoD systems - in development and deployment - should be stored, mined, and made available for machine learning.

Motivation and Scope

The latest industry best practices for developing, fielding, and sustaining software applications and information technology (IT) systems are substantially outpacing the US government's industrial-era planning, programming, budgeting, and execution system (PPBES) methods. In the commercial software industry, there is no clear delineation between development, procurement, and sustainment; rather it is a continuous cycle that changes rapidly. New functionality is made available and deployed to users in months to weeks (and even days, for truly critical updates). Existing government appropriation structures make it difficult to implement this approach in the DoD.

Currently available commercial technology for rapidly deploying new advances in software, electronics, networking, and other areas means that our adversaries can rapidly develop new tactics that will be used against us. The only defense is to get inside our adversaries' observe, orient, decide, and act (OODA) loop, which requires the ability to rapidly develop and deploy software into operational environments. For software that is used as part of operations, whether it is run in the Pentagon or in the field, this will require new methods for (automated) testing and rapid deployment of updates, patches, and new functionality.

In this document, we provide ten "commandments" (principles) for DoD software that provide an approach to development that builds on the lessons learned in the software industry and enables rapid deployment of software into military operations. These principles are not universal and may not apply in all situations, but they provide a framework for improving the use of software in DoD operations going forward that we believe will provide substantial improvements compared to the current state of practice.

Software Types

Not all software is alike and different types of software require different approaches for procurement and sustainment. It is important to avoid a "one size fits all" approach to weapons systems. Acquisition practices for hardware are almost never right for software: they are too slow, too expensive, and too focused on enterprise-wide uniformity instead of local customization. Similarly, the process for obtaining software to manage travel is different than what is required to manage the software on an F-35. We suggest a taxonomy with four types of software requiring four different approaches:

- A: commercial ("off-the-shelf") software with no DoD-specific customization required;
- B: commercial software with DoD-specific customization needed;
- C: custom software running on commodity hardware (in data centers or in the field);
- D: custom software running on custom hardware (e.g., embedded software).

While many of the principles below apply to all DoD software, some are relevant only for specific types, as we indicate at the end of each description.

To amplify at the extremes of this continuum of software types, we note especially the tendency of large organizations to believe their needs are unique when it comes to software of Type A. Business processes, financial, human resources, accounting and other “enterprise” applications in DoD are generally not more complicated nor significantly larger in scale than those in the private sector. Commercial software, unmodified, can be deployed in nearly all circumstances. At the opposite end of the spectrum we recognize the highly coupled nature of real-time, mission-critical, embedded software with its customized hardware, denoted in Type D. Examples here include primary avionics or engine control, or target tracking in shipboard radar systems, where requirements such as safety, target discrimination and fundamental timing considerations demand that extensive formal analysis, test, validation and verification activities be carried out.

The DIB’s Ten Commandments of Software

Commandment #1. Make computing, storage, and bandwidth abundant to DoD developers and users, especially in operational systems. Effective use of software requires that sufficient resources are available for computing, storage, and communications. The DoD should adopt a strategy for rapidly transitioning DoD IT to current industry standards such as cloud computing, distributed databases, ubiquitous access to modernized wireless systems (leveraging commercial standards), abundant computing power and bandwidth that is made available as a platform, integration of mobile technologies, and the development of a DoD platform for downloading applications. Unit cost of IT infrastructure and services should be used as a metric in track improvements. An important metric of abundance must be the ability to actually deliver code, and DoD must be able to count the number of programmers within an organization and make sure that the balance of coders to managers is correct [All types]

Commandment #2. All software procurement programs should start small, be iterative, and build on success – or be terminated quickly. Good software development provides value to the customer quickly, based on working with users starting on day one and defining success based on customer value, not creation of code. Large software programs are doomed to fail because of the rigidity, process, competition protests, and bureaucracy that accompany them (typically starting with the Joint Capabilities Integration Development System (JCIDS) process). The separation of software development into research, development, test and evaluation (RDT&E), procurement, and operations & maintenance (O&M) appropriations (colors of money) – and the use of cost-based triggers within each acquisition category (ACAT) – causes delays and places artificial limitations on the program management office’s (PMO’s) ability to quickly meet the changing needs, resulting in increased lifetime cost of software and slower deployment. Modern (“agile”) approaches used in commercial software development will result in faster deployment *and* significant cost savings. [All types, especially B and C]

Commandment #3. The acquisition process for software must support the full, iterative life cycle of software. Software does not age well. It must be constantly maintained and updated, ideally in an automated fashion. The PPBES process is nominally a two (2) year timeline to request and receive funding, with initial planning occurring five (5) years prior to actual receipt, and funding must be requested by intent of use (RDT&E, procurement, and O&M). But this fiscal separation does not match the process of software development, where all creation of code is "development," whether it falls within the fiscal law definition or not. As an alternative, the DoD should make use of "level of effort" (or capacity) constructs to allow continuous development and testing. Assume that low criticality software that is routinely used will require 10% of the development cost to maintain (per year) and more critical software will likely require more resources. This funding must be planned for at the time of initial development, not as an annual allocation that could be interrupted. Enhanced software capability should never be considered "ahead of need." [All types]

Commandment #4. Adopt a DevSecOps culture for software systems. "DevOps" represents the integration of software development and software operations, along with the tools and culture that support rapid prototyping and deployment, early engagement with the end user, automation and monitoring of software, and psychological safety (e.g., blameless reviews). "DevSecOps" adds the integration of security at all stages of development and deployment, which is essential for DoD applications. These techniques should be adopted by the DoD, with appropriate tuning of approaches used by the community for mission-critical, national security applications. Open source software should be used when possible to speed development and deployment, and leverage the work of others. Waterfall development approaches (e.g., DoD-STD-2167A) should be banned and replaced with true, commercial agile processes. Thinking of software "procurement" and "sustainment" separately is also a problem: software is never "finished" but must be constantly updated to maintain capability, address ongoing security issues and potentially add or increase performance (see Commandment #3). [Type C; Type D when tied to iterative hardware development and deployment methodologies]

Commandment #5. Automate configuration, testing, and deployment of software to enable critical updates to be deployed in days to weeks, not months or years. While operational test and evaluation (OT&E) is useful, it must not be the pacing item for deployment of software, especially upgrades to existing software. Automated configuration management, unit testing, software/hardware-in-the-loop (SIL/HIL) testing, continuous integration, A/B testing, usage and issues tracking, and other modern tools of software development should be used to provide high confidence in software correctness and enable rapid, push deployment of patches, upgrades, and apps. Make use of modern software development tool sets that support these processes (and other types of development stack automation and software instrumentation) to enable code optimization and refactoring. [All types]

Commandment #6. Every purpose-built DoD software system should include source code as a deliverable. DoD should have the rights to and be able to modify (DoD-specific) code when new conditions and features arise. Providing source code will also allow the DoD to perform detailed (and automated) evaluation of software correctness, security, and performance, enabling more rapid deployment of both initial software releases and (most importantly) upgrades (patches and enhancements). [Types C, D]

Commandment #7. Every DoD system that includes software should have a local team of DoD software experts who are able to modify or extend the software through source code or API access. Modern weapons systems are software-driven and utilization of those systems in a rapidly changing environment will require that the system (software) be customizable by the user. In order to do this, all fielded DoD systems that include software must also have a local team (responsible to the operational unit) that has the skills and permission to modify and extend the software through either source code (Commandment #6) or application programming interface (API) access. Local experts should have “reachback” capabilities to larger team and the ability to pull new features into their code (and generate pull requests for features that they add which should go back into the main codebase [repository]). [Types B, C, sometimes D]

Commandment #8. Only run modern operating systems that are receiving (and utilizing) regular security updates for newly discovered security vulnerabilities. Outdated operating systems are a major vulnerability and the DoD should assume that any computer running such a system will eventually be compromised. Standard practice in industry is that security patches should be applied within 48 hours of release, though this is probably too big a window for defense systems. Treat software vulnerabilities like perimeter defense vulnerabilities: if there is a hole in your perimeter and people are getting in, you need to patch the hole quickly and effectively. [Types A, B, C]

Commandment #9. Security should be a first-order consideration in design and deployment of software, and data should always be encrypted unless it is part of an active computation. All data should be encrypted, whether in motion (across a network) or at rest (memory, disk, cloud, etc). A possible exception is real-time data that is part of an embedded control system and is being sent across an internal bus/network that is not accessible from outside that network. [Types A, B, C and D when possible]

Commandment #10. All data generated by DoD systems – in development and operations – should be stored, mined, and made available for machine learning. Create a new architecture to collect, share, and analyze data that can be mined for patterns that humans cannot perceive. Utilize data to enable better decision-making in all facets of the Department, providing significant advantages that adversaries cannot anticipate. Forge culture of data collection/analysis to meet the demands of a software-centric combat environment. Such data collection and analysis can be done without compromising security: in fact, a comprehensive understanding of the data the DoD collects can improve security. [All types]

Supporting Thoughts and Recommendations

In addition to the ten principles given above, we offer the following thoughts and recommendations for how the DoD can best take advantage of software as a force multiplier. While not directly related to software, they are enablers for adopting the principles required for rapid development and deployment of software.

Establish Computer Science as a DoD core competency. Do not rely solely on contractors as the only source of coding capability for DoD systems. Instead, the DoD should recruit, train, and retain internal capability for software development, including by service members, and maintain this as a separate career track (like DoD doctors, lawyers, and musicians). This should complement work done by civilians and contractors. Create new and expand existing programs to attract promising civilian and military science, technology, engineering and math (STEM) talent. Reach into new demographic pools of people who are interested in the work DoD does but otherwise would not be aware of DoD opportunities. Be able to count the number of programmers within an organization and make sure that the balance of developers to managers is correct

Use commercial process and software to adopt and implement standard business practices within the Services. Modern enterprise-scale software has been optimized to allow business to operate efficiently. The DoD should take advantage of these systems by adopting its internal (non-warfighter specific) business processes to match industry standards, which are implemented in cost-efficient, user-friendly software and software as a service [SaaS] tools. Rather than adopt a single approach across the entire DoD, the individual Services should be allowed to implement complementary approaches (with appropriate interoperability).

Move to a model of continuous hardware refresh in which computers are treated as a consumable with a 2-3 year lifetime. The current approach — in which technology refreshes take 8-10 years from planning to implementation — means that most of the time our systems are running on obsolete hardware that limits our ability to implement the algorithms required to provide the level of performance required to stay ahead of our adversaries. Moving to the cloud provides a solution to this issue for enterprise and other software systems that do not operate on local or specialized hardware. However for weapons systems, a continuous hardware refresh mentality would enable software upgrades, crypto updates, and connectivity upgrades to be rapidly deployed across a fleet, rather than maintaining legacy code for different variants that have hardware capabilities ranging from 2 to 12 years old (an almost impossibly large spread of capability in computing, storage, and communications). From a contracting perspective, this change would require DoD to provide a stable annual budget that paid for new hardware and software capability (see Commandment #3), but this would very likely save money over the longer term.

Defense Innovation Board
Metrics for Software Development

Software is increasingly critical to the mission of the Department of Defense (DoD), but DoD software is plagued by poor quality and slow delivery. The current state of practice within DoD is that software complexity is often estimated based on number of source lines of code (SLOC), and rate of progress is measured in terms of programmer productivity. While both of these quantities are easily measured, they are not necessarily predictive of cost, schedule, or performance. They are especially suspect as measurements of program success, defined broadly as delivering needed functionality and value to users. Measuring the health of software development activities within DoD programs using these obsolete metrics is irrelevant at best and, at worst, could be misleading. As an alternative, we believe the following measures are useful for DoD to track performance for software programs and drive improvement in cost, schedule, and performance.

#	Metric	**Target value (by software type)**[5]				**Typical DoD values for SW**
		COTS apps	**Custom -ized SW**	**COTS HW/OS**	**Real-time HW/SW**	
1	Time from program launch to deployment of simplest useful functionality	<1 mo	<3 mo	<6 mo	<1 yr	3-5 yrs
2	Time to field high priority fcn (spec → ops) or fix newly found security hole (find → ops)	N/A <1 wk	<1 mo <1 wk	<3 mo <1 wk	<3 mo <1 wk	1-5 yrs 1-18 m
3	Time from code committed to code in use	<1 wk	<1 hr	<1 da	<1 mo	1-18 m
4	Time req'd for full regression test (automat'd) and cybersecurity audit/penetration testing	N/A <1 mo	<1 da <1 mo	<1 da <1 mo	<1 wk <3 mo	2 yrs 2 yrs
5	Time required to restore service after outage	<1 hr	<6 hr	<1 day	N/A	?
6	Automated test coverage of specs/code	N/A	>90%	>90%	100%	?
7	Number of bugs caught in testing vs field use	N/A	>75%	>75%	>90%	?
8	Change failure rate (rollback deployed code)	<1%	<5%	<10%	<1%	?
9	% code avail to DoD for inspection/rebuild	N/A	100%	100%	100%	?
10	Number/percentage of functions implemented	80%	90%	70%	95%	100%
11	Usage and user satisfaction	TBD	TBD	TBD	TBD	?

[5] Target values are notional; different types of software (SW) as defined in DIB Ten Commandments.

Acronyms defined: Commercial off-the-shelf (COTS), apps is short for applications, specs is short for specifications, hardware/operating system (HW/OS), hardware/software (HW/SW)

12	Complexity metrics	#/type of specs # programmers structure of code #/skill level of teams #/type of platforms #/type deployments				Partial/ manual tracking
13	Development plan/environment metrics					
14	“Nunn-McCurdy” threshold (for any metric)	1.1X	1.25X	1.5X	1.5X each effort	1.25X Total $

Supporting Information

The information below provides additional details and rationale for the proposed metrics. The different types of software considered in the document are described here in greater depth, followed by comments on the proposed metrics, grouped into four categories: (a) deployment rate metrics, (b) response rate metrics, (c) code quality metrics, and (d) program management, assessment and estimation metrics.

Software Types (from DIB Ten Commandments)

Not all software is alike, and different types of software require different approaches for development, deployment, and life-cycle management. It is important to avoid a “one size fits all” approach to weapons systems. Acquisition practices for hardware are almost never right for software: they are too slow, too expensive, and too focused on enterprise-wide uniformity instead of local customization. Similarly, the process for obtaining software to manage travel is different than what is required to manage the software on an F-35. We suggest a taxonomy with four types of software requiring four different approaches:

- A: commercial (“off-the-shelf”) software with no DoD-specific customization required;
- B: commercial software with DoD-specific customization needed;
- C: custom software running on commodity hardware (in data centers or in the field);
- D: custom software running on custom hardware (e.g., embedded software).

Type A (COTS apps): The first class of software consists of applications that are available from commercial suppliers. Business processes, financial management, human resources, accounting and other “enterprise” applications in DoD are generally not more complicated nor significantly larger in scale than those in the private sector. Unmodified commercial software should be deployed in nearly all circumstances. Where DoD processes are not amenable to this approach, those processes should be modified, not the software.

Type B (Customized SW): The second class of software constitutes those applications that consist of commercially available software that is customized for DoD-specific usage. Customizations can include the use of configuration files, parameter values, or scripted functions that are tailored for DoD missions. These applications will generally require configuration by DoD personnel, contractors, or vendors.

Type C (COTS HW/OS): The third class of software applications is those that are highly specialized for DoD operations but can run on commercial hardware and standard operating systems (e.g., Linux or Windows). These applications will generally be able to take advantage of commercial processes for software development and deployment, including the use of open source code and tools. This class of software includes applications that are written by DoD personnel as well as those that are developed by contractors.

Type D (Custom SW/HW): This class of software focuses on applications involving real-time, mission-critical, embedded software whose design is highly coupled to its customized hardware. Examples include primary avionics or engine control, or target tracking in shipboard radar systems. Requirements such as safety, target discrimination, and fundamental timing considerations demand that extensive formal analysis, test, validation, and verification activities be carried out in virtual and "iron bird" environments before deployment to active systems. These considerations also warrant care in the way application programming interfaces (APIs) are potentially presented to third parties.

Types of Software Metrics

Deployment Rate Metrics

Overview: Consistent with previous Defense Innovation Board (DIB) commentary, and software industry best practices, an organizational mentality that prioritizes speed is the ultimate determinant of success in delivering value to end users. An approach to software development that privileges speed over other factors drives efficient decision-making processes; forces the use of increased automation of development and deployment processes; encourages the use of code that is machine-generated as well as code that is correct-by-construction; relies heavily on automated unit and system level testing; and enables the iterative, deliver-value-now mentality of a modern software environment. Thus we list these metrics first.

#	Metric	Target value (by software type)				Typical DoD values for SW
		COTS apps	Customized SW	COTS HW/OS	Real-time HW/SW	
1	Time from program launch to deployment of simplest useful functionality	<1 mo	<3 mo	<6 mo	<1 yr	3-5 yrs
2	Time to field high priority fcn (spec → ops) or fix newly found security hole (find → ops)	N/A <1 wk	<1 mo <1 wk	<3 mo <1 wk	<3 mo <1 wk	1-5 yrs 1-18 m
3	Time from code committed to code in use	<1 wk	<1 hr	<1 da	<1 mo	1-18 m

Background: These measures capture the rate at which new functions and changes to a software application can be put into operation (in the field):

1. The time from program launch to deployment of the "simplest useful functionality" is an important metric because it determines the first point at which the code can start doing useful work and also at which feedback can be gathered that supports refinement of the features. There is a tendency in DoD to deliver code only once it has met all of the specifications, but this can lead to significant delays in providing useful code to the user. We instead advocate

getting code in the hands of the user quickly, even if it only solves a subset of the full functionality. Something is better than nothing, and user feedback often reveals omissions in the specifications and can refine the initial requirements. As code becomes more customized, this interval of time might extend due to the need to run more complex tests to ensure that all configurations operate as expected, and that complex timing and other safety/mission-critical specifications are satisfied. It is important to note that this metric is not just about coding time. It also measures the time required to process and adjudicate the changes (including release approval), often the most time-consuming part of providing new or upgraded functionality.

2. Once the code is deployed, it is possible to measure the amount of time that it takes to make incremental changes that either implement new functions or fix issues that have been identified. The importance of the functionality or severity of the error will determine how quickly these changes should be made, but it should be possible to deploy high priority code updates much more quickly and in much smaller increments than typical DoD “block” upgrades. A similar measure to the time it takes to deploy code to the field is deployment frequency. Deployment frequency can be on-demand (multiple per day), once per hour, once per day, once per week, etc. Faster deployment frequency often correlates with smaller batch sizes.

3. The time from which code is committed to a repository until it is available for use in the field is referred to as “lead time,” and good performance on this metric is a necessary condition for rapid evolution of delivered software functionality. Shorter product delivery times demand faster feedback, which enables tighter coupling to user needs. For commercially available applications, the lead time will be based on vendor deployment processes and may be slower than what is needed for customized code, be it for commercial hardware/operating systems or custom hardware. However, we believe that in the selection of commercial software, emphasis should be given to the vendor’s iteration cycles and lead time performance. Embedded code will often require much more extensive testing before it is deployed, and therefore its lead time may be longer.

Response Rate Metrics

Overview: Our philosophy is that delivering a partial solution to the user quickly is almost always better than delivering a complete or perfect solution at the end of a contract, on the first attempt. Consistent with that, mistakes will occur. No software is bug-free, and so it is unrealistic and unnecessary to insist on that, except where certain safety matters are concerned.[6] Code that does most things right will still be useful while a patch is being identified and fielded. How gracefully software fails, how many errors are caught and resolved in testing, and how rapidly developers patch bugs are excellent measures of software development prowess.

[6] The Department and its suppliers (due to the requirements of the contracts to which they are bound) often resort to blanket pronouncements about safety and security, which often lead to applying the most extreme measures even when not needed; this risk-averse approach to treating everything as a grave risk to cyber security or safety has been labeled by the DIB as a “self-denial of service attack.” While cybersecurity is clearly critical for software systems, the Department needs to rely on product managers who use judgment to make subtle, nuanced, and risk-based judgments about trade-offs during the software development process.

#	Metric	Target value (by software type)				Typical DoD values for SW
		COTS apps	Customized SW	COTS HW/OS	Real-time HW/SW	
4	Time required for full regression test (automated) and cybersecurity audit/penetration testing[7]	N/A <1 mo	<1 da <1 mo	<1 da <1 mo	<1 wk <3 mo	2 yrs 2 yrs
5	Time required to restore service after outage	<1 hr	<6 hr	<1 day	N/A[8]	?

Background: These two metrics are intended for “generic” software programs with moderate complexity and criticality. Their purpose is to:

4. Measure the ability to conduct more complete functional tests of the full software suite (e.g., regression tests) in a timely fashion, to identify problems in deployed software that can be quickly corrected, and to restore service after an incident such as an unplanned outage or service impairment, occurs (also called “mean time to repair,” (MTR)).
5. Track the time required to resolve an interruption to service, including a bad deployment.

Code Quality Metrics

Overview: These metrics are intended to be used as a measure of the quality of the code and to focus on identifying errors in the code, either at the time of development (e.g., via unit tests) or in the field.

#	Metric	Target value (by software type)				Typical DoD values for SW
		COTS apps	Customized SW	COTS HW/OS	Real-time HW/SW	
6	Automated test coverage of specs/code	N/A	>90%	>90%	100%	?
7	Number of bugs caught in testing vs field use	N/A	>75%	>75%	>90%	?
8	Change failure rate (rollback deployed code)	<1%	<5%	<10%	<1%	?
9	% code avail to DoD for inspection/rebuild	N/A	100%	100%	100%	0%

[7] The two different response rate metrics for different types of software reflect the level of complexity of the software, the likely resources available to identify and fix problems, and the level of integration of the hardware and software.

[8] We note that for embedded systems, which must be running at all times and which are updated much less frequently, the notion of “restoring” service is not directly applicable.

Background:

6. Automated developmental tests provide a means of ensuring that updates to the code do not break previous functionality and that new functionality works as expected. Ideally, for each function that is implemented, a set of automated tests will be constructed that cover both the specification for what the performance should achieve as well as the code that is used to implement that function.

7. The percentage of specifications tested by the automated test suite provides rapid confidence that a software change has not caused some specification to fail, as well as confidence that the software does what it is supposed to do. Test coverage of the code is a common metric for software test quality and one that most software development environments can compute automatically (e.g., in a continuous integration (CI) workflow, each commit and/or pull request to a repository would run all the automated developmental tests and compute the percentage covered). For customized software and applications that run on commercial hardware and operating systems, 90% unit test coverage is a good target. Embedded code should strive for 100% coverage (i.e., no "dark" code) since it is often safety- or mission-critical.[9] The focus of these metrics is on developmental tests, as operational testing is important, but expensive, so it is far less expensive to find and fix defects through developmental testing.

8. Developmental tests do not cover every conceivable situation in which an application might be used, so errors will be discovered in the field. The percent of bugs caught in testing (via unit tests or regression tests) versus those caught in the field provide a measure of the both the quality of the code and the thoroughness of the testing environment. Bugs discovered late in the development cycle or after deployment can "cost" an order of magnitude more than early bugs (in terms of time to fix and impact to a program), and a software system that is mature finds many more bugs during testing and few in the field. Late bugs are particularly expensive when fixing those bugs can require hardware changes, and so code running on custom hardware should be tested more strenuously. Bugs should be prioritized by severity and the trends over time for serious bugs should be monitored and used to drive changes in the test environment and software development process.

9. When bugs do occur, it may be necessary to roll back the deployed code and return to an earlier version. Change fail percentage is the percentage of changes to production that fail, including software releases and infrastructure changes. This should include changes that result in degraded service or subsequently require remediation, such as those that lead to service impairment or outage, or require a hotfix, rollback, fix-forward, or patch. For COTS applications, this should occur rarely because the amount of testing done by the vendor, including test deployments to beta users, will typically be very high. There may be a higher change failure rate as the application becomes more customized—because it can be difficult to test for issues where there is a variety of hardware configurations operating in the field, for example—but for embedded code, the change failure rate should be small, due to the more safety-critical nature of that code leading to more emphasis on up-front testing.

Functionality metrics

Overview: These metrics are intended to capture how useful the software program is in terms of delivering value to the field. We envision that a software program will have a number of desired

[9] Safety- or mission-critical software often strives for more rigorous test coverage metrics, such as high branch coverage or in some cases high modified condition/decision coverage (MC/DC).

features that define its functionality. Software should be instrumented so that the use of those features is measured and, when appropriate, users of the software should be monitored or surveyed to determine their use of/satisfaction with the software.

#	Metric	Target value (by software type)				Typical DoD values for SW
		COTS apps	Custom -ized SW	COTS HW/OS	Real-time HW/SW	
10	Number/percentage of functions implemented	80%	90%	70%	95%	100%
11	Usage and user satisfaction	TBD	TBD	TBD	TBD	?

Background:

10. An ongoing software program will have some number of functions that it performs and a list of additional functions that are to be added over time. These new functions could be feature requires from users or desired features generated by the program office that are on the list for consideration to be implemented next. Keeping track of these features and the rate at which they are implemented provides a measure of the delivery of functionality to the user. This specific way in which functionality is measured will be dependent on the type of software being developed.

11. For software that is used by a person, the ultimate metric is whether the software is helping that person get useful work done. Keeping track of the usage of the software (and different parts of the software) can be done by instrumenting the code and keeping track of the data it generates. To determine whether or not the software is providing good value to the person who is using it, surveying the user may be the most direct mechanism (similar to rating software that you use on a computer or smart phone).

Program Management, Assessment, and Estimation Metrics

Overview: The final set of metrics are intended for management of software programs, including cost assessment and performance estimation. These metrics describe a list of "features" (performance metrics, contract terms, project plans, activity descriptions) that should be required as part of future software projects to provide better tools for monitoring and predicting time, cost, and quality. In its public deliberations regarding software acquisition and practices, the DIB has described how metrics of this type might be used to estimate the cost, schedule, and performance of software programs.

#	Metric	Target value (by software type)				Typical DoD values for SW
		COTS apps	Custom ized SW	COTS HW/OS	Real-time HW/SW	
12	Complexity metrics	#/type of specs # programmers structure of code #/skill level of teams #/type of platforms #/type deployments				Partial/ manual tracking
13	Development plan/environment metrics					

14	"Nunn-McCurdy" threshold (for any metric)	1.1X	1.25X	1.5X	1.5X each effort	1.25X Total $

Background:

12. Structure of specifications, code, and development and execution platforms**.**

To measure the complexity of a software program, and therefore assess the cost, schedule and performance of that program, a number of features must be measured that capture the underlying "structure" of the application. The use of the term "structure" is intentionally flexible, but generally includes properties such as size, type, and layering. Examples of features that can be captured that related to underlying complexity include:

- *Structure of specifications:* Modern specification environments (e.g., application life-cycle management [ALM] tools) provide structured ways of representing specifications, from program level requirements to derived specifications for sub-systems, or individual teams. The structure represented in these tools can be used as a measure of the difficulty of the application that is being designed.

- *Level and type of user engagement during application development:* How much time do developers spend with users, especially early in the program? How many developers are "on site" (in the same organization and/or geographic location as the end user)?

- *Structure of the code base (software architecture):* Modern software development environments allow structured partitioning of the code into functions, libraries/frameworks, and services. The structure of this partitioning (number of modules, number of layers, and amount of coupling between modules and layers) can provide a measure of the complexity of the underlying code.

- *The amount of reuse of existing code, including open source code:* In many situations there are well-maintained code bases that can be used to quickly create and scale applications without rewriting software from scratch. These libraries and code frameworks are particularly useful when using commodity hardware and operating systems, since the packages will often be maintained and expanded by others, leveraging external effort.

- *Structure of the development platform/environment:* This includes the software development environments that are being used, the types of programming methodologies (e.g., XP, agile, waterfall, spiral) that are employed, and the level of maturity of the programming organization (ISO, CMMI, SPICE).

- *Structure of the execution platform/environment:* The execution environment can have an impact on the ability to emulate the execution environment within the development environment, as well as the portability of applications between different execution environments. Possible platforms include various cloud computing environments as well as platform-as-a-service (PaaS) environments that support multiple cloud computing vendors.

13. Structure and type of development and operational environment.

To predict and monitor the level of effort required to implement and run a software application, measurement of the development and operation environments is critical. These measurements include the structure of those environments (e.g., waterfall versus spiral versus agile, use of continuous integration tools, integrated tools for issue tracking/resolution, code review mechanisms), the tempo of development and delivery, and use of the functionality provided by the application. Example of features that can be captured that relate to the structure and type of development and operation use:

- Number and skill level of programmers on the development team
- Number of development platforms used across the project
- Number of subcontractors or outside vendors used for application components
- Number and type of user operating environments (execution platforms) supported
- Rate at which major functions (included in specifications) are delivered and updated
- Rate at which the operational environment must be updated (e.g., hardware refresh rate)
- Rate at which the mission environment changes (driving changes to the code)
- Number (seats or sites) and skill level of the users of the software

14. Tracking software program progress

To properly manage the continuous development and deployment of software, DoD should be able to track the metrics above with minimal additional effort from the programmers because this information should be gathered and transmitted automatically through the development, deployment, and execution environments, using automated tools. Some examples of the types of metrics that are readily available include commit activity data (number and rate of commits), team size, number of commenters (on pull requests), number of pull request mergers, average and standard deviation of the months in which there was development activity, average and standard deviation of the number of commits per month.

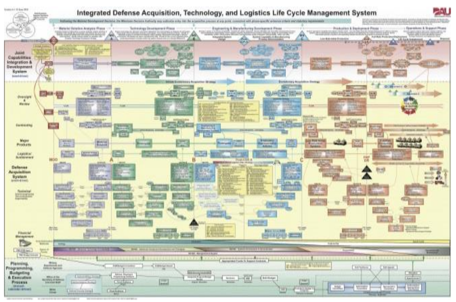
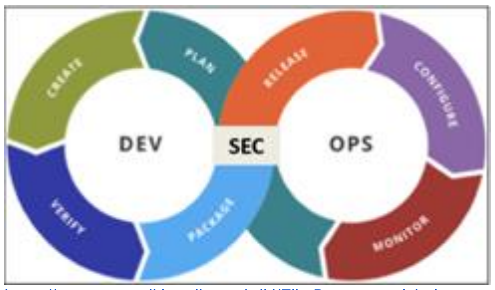
Thresholds should be established to determine when management attention is required, but also when a program is so far off its initial plan that it should be re-evaluated. Today's "Nunn-McCurdys" or "Critical Changes" refer to breaches in cost or schedule thresholds. The current 25% unit cost growth and 50% total program cost growth thresholds often will not make sense for continuously developed software programs.

An alternative to cost-based thresholds is to establish thresholds based on the metrics listed above, with different thresholds for different types of software. A notion of a "Nunn-McCurdy type breach" for software programs based on some of the above performance metrics recorded at lower levels of effort or on specific applications could serve as better means of identifying major issues earlier in a program. Commercially available software, with or without customization, should be the easiest type for which to establish accurate metrics, since it already exists and should be straightforward to purchase and deploy. Metrics for customized software running on either commercial or DoD-specific hardware is likely to be more difficult to predict, so a higher threshold can be used in those circumstances.

Defense Innovation Board Do's and Don'ts for Software

This document provides a summary of the Defense Innovation Board's (DIB's) observations on software practices in the DoD and a set of recommendations for a more modern set of acquisition and development principles. These recommendations build on the DIB's "Ten Commandments of Software." In addition, we indicate some of the specific statutory, regulatory, and policy obstacles to implementing modern software practices that need to be changed.

Executive Summary

Observed practice (Don'ts)	**Desired state (Do's)**	**Obstacles**
 Integrated Defense Acquisition, Technology, and Logistics Life Cycle Management System Defense Acquisition University, June 2010	 CREATE PLAN RELEASE CONFIGURE DEV SEC OPS VERIFY PACKAGE MONITOR https://commons.wikimedia.org/wiki/File:Devops-toolchain.svg (modifications licensed CC-BY-SA)	10 U.S.C. §2334 10 U.S.C. §2399 10 U.S.C. §2430 10 U.S.C §2433a 10 U.S.C. §2460 10 U.S.C. §2464 DODI 5000.02, par 5.c.(2) and 5.c.(3)(c)-(d)
Spend 2 years on excessively detailed requirements development	Require developers to meet with end users, then start small and iterate to quickly deliver useful code	DODI 5000.02, par 5.c.(2) CJCSI 3170.01I App A.1.b
Define success as 100% compliance with requirements	Accept 70% solutions[10] in a short time (months) and add functionality in rapid iterations (weeks)	10 U.S.C. §2399 OMB Cir A-11 pp 42-43
Require OT&E to certify compliance after development and before approval to deploy	Create automated test environments to enable continuous (and secure) integration and deployment to shift testing left	10 U.S.C. §139b/d 10 U.S.C. §2399 Cultural
Apply hardware life-cycle management processes to software	Take advantage of the fact that software is essentially free to duplicate, distribute, and modify	10 U.S.C. §2334 10 U.S.C. §2399 10 U.S.C. §2430 48 CFR 207.106 DODI 5000.02
Require customized software solutions to match DoD practices	For common functions, purchase existing software and change DoD processes to use existing apps	Culture

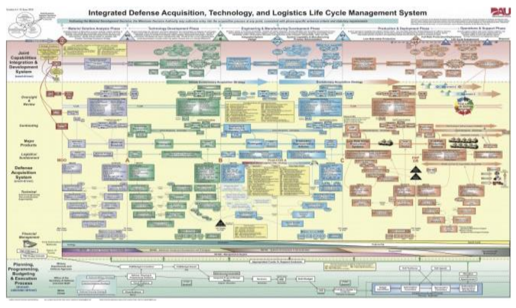
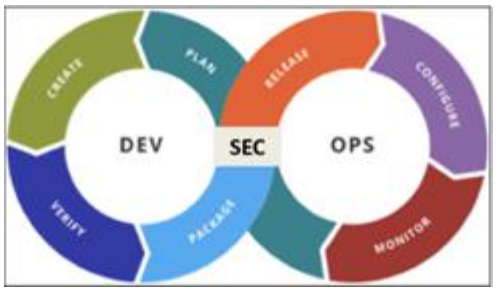
[10] 70% is notional. The point is to deliver the simplest, most useful functionality to the warfighter quickly.

Use legacy languages and operating systems that are hard to support and insecure	Use modern software languages and operating systems (with all patches up-to-date)	10 U.S.C. §2334 DoDI 5000.02, Enclosure 11 Culture
Evaluate cyber security after the systems have been completed, separately from OT&E	Use validated software development platforms that permit continuous integration & evaluation (DevSecOps)	DOT&E Memos Culture
Consider development and sustainment of software as entirely separate phases of acquisition	Treat software development as a continuous activity, adding functionality across its life cycle	10 U.S.C. §2399 10 U.S.C. §2430 10 U.S.C. §2460 10 U.S.C. §2464 DODI 5000.02, par 5.c.(2) and 5.c.(3)(c)-(d)
Depend almost entirely on outside vendors for all product development and sustainment	Require source code as a deliverable on all purpose-built DoD software contracts. Continuous development and integration, rather than sustainment, should be a part of all contracts. DoD personnel should be trained to extend the software through source code or API access[11]	Culture (no apparent statutory obstacle) FAR/DFARS technical data rights
Turn documents like this into a process and enforce compliance	Hire competent people with appropriate expertise in software to implement the desired state and give them the freedom to do so ("competence trumps process")	Culture

[11] As noted in the DIB's 10 Commandments of Software

Supporting Information

The information below, broken out by entry in the executive summary table (see table E.8 above), provides additional information and a rationale for each desired state.

Don't	**Do**
 Defense Acquisition University, June 2010	 CREATE PLAN RELEASE CONFIGURE DEV SEC OPS VERIFY PACKAGE MONITOR https://commons.wikimedia.org/wiki/File:Devops-toolchain.svg

The DoD 5000 process, depicted on the left in figure E.1, provides a detailed DoD process for setting requirements for complex systems and ensuring that delivered systems are compliant with those requirements. The DoD's "one size fits all" approach to acquisition has attempted to apply this model to software systems, where it is wholly inappropriate. Software is different than hardware. Modern software methods make use of a much more iterative process, often referred to as "DevOps," in which development and deployment (operations) are a continuous process, as depicted on the right. A key aspect of DevOps is continuous delivery of improved functionality through interaction with the end user.

Why this is hard to do, but also worth doing:[12]

- DoD 5000 is designed to give OSD, the Services, and Congress some level of visibility and oversight into the development, acquisition, and sustainment of large weapons systems. While this directive may be useful for weapons systems with multi-billion dollar unit costs, it does not make sense for most software systems.
- While having one consistent procurement process is desirable in many cases, the cost of using that same process on software is that software is delivered late to need, costs substantially more than the proposed estimates, and cannot easily be continuously updated and optimized.
- Moving to a software development approach will enable the DoD to move from a *specify, develop, acquire, sustain* mentality to a more modern (and more useful) *create, scale, optimize* (DevOps/DevSecOps) mentality. Enabling rapid iteration will create a system in which the United States can update software at least as fast as our adversaries can change tactics, allowing us to get inside their OODA loop.

[12] These comments and the similar ones that follow for other area were obtained by soliciting feedback on this document from people familiar with government acquisition processes and modern software development environments.

Acronyms defined: Office of the Secretary of Defense (OSD), OODA is short for the decision cycle of Observe, Orient, Decide, and Act.

Don’t	Do
Spend 2 years on excessively detailed requirements development	Require developers to meet with end users, then start small and iterate to quickly deliver useful code
Define success as 100% compliance to requirements	Accept 70% solutions in a short time (months) and add functionality in rapid iterations (weeks)

Developing major weapons systems is costly and time consuming, so it is important that the delivered system meets the needs of the user. The DoD attempts to meet these needs with a lengthy process in which a series of requirements are established, and a successful program is one that meets those requirements (ideally close to the program’s cost and schedule estimates). Software, however, is different. When done right, it is easy to quickly deploy new software that improves functionality and, when necessary, rapidly rollback deployed code. It is more useful to get something simple working quickly (time-constrained execution) and then exploit the ability to iterate rapidly in order to get the remaining desired functionality (which will often change in any case, either in response to user needs or adversarial tactics).

Why this is hard to do, but also why it is worth doing:

- Global deployment of software on systems which are not always network-connected (e.g., an aircraft carrier or submarine underway) introduces very real problems around version management, training, and wisely managing changes to mission-critical systems.
- In the world of non-military, consumer Internet applications, it is easy to glibly talk about continuous deployment and delivery. In these environments, it is easy to execute and the consequences for messing up (such as making something incredibly confusing or hard to find) are minor. The same is not always true for DoD systems—and DoD software projects rarely offer scalable and applicable solutions to address the need for continuous development.
- Creating an approach (and the supporting platforms) that enables the DoD to achieve continuous deployment is a non-trivial task and will have different challenges than the process for a consumer Internet application. The DoD must lay out strategies for mitigating these challenges. Fortunately, there are tools that can be build upon: many solutions have already been developed in consumer industries that require failsafe applications with security complexities.
- Continuous deployment depends on the entire ecosystem, not just the front-end software development.
- Make sure to focus on product design and *product* management, which prioritizes delivery of capability to meet the changing needs of users, rather than program/project management, which focus on execution against a pre-approved plan. This shift is key to user engagement, research, and design.

Don’t	Do
Require OT&E to certify compliance after development and before approval to deploy	Create automated test environments to enable continuous (and secure) integration and deployment to shift testing left
Evaluate cyber security after the system has been completed, separately from OT&E	Use validated software development platforms that permit continuous integration and evaluation

Why this is hard to do, but also worth doing:

- The DoD typically performs a cyber evaluation on software only after delivery of the initial product. Modern software approaches have not always explicitly addressed cyber security (though this is changing with “DevSecOps”). This omission has given DoD decision-makers an easy “out” for dismissing recommendations (or setting up roadblocks) for DevOps strategies like continuous deployment. Cyber security concerns must be addressed head on, and in a manner that demonstrates better security in realistic circumstances. Until then, change is unlikely.

- More dynamic approaches to address the cyber security concerns must be developed and implemented through some amount of logic and a fair bit of data. Case studies of red teaming also help: *Hack the Pentagon* should be able to provide some true examples that generate concern. It may be necessary to obtain access to some additional good data that goes beyond what corporations are willing to share publicly.

- To succeed, it will be important not to assume that it will be clear how these recommendations solve for all cyber security concerns. Recommendations should make explicit statements about what can be accomplished, taking away the reasons to say “no.”

Don’t	Do
Apply hardware life-cycle management processes to software	Take advantage of the fact that software is essentially free to duplicate, distribute, and modify
Consider development and sustainment of software as entirely separate phases of acquisition	Treat software development as a continuous activity, adding functionality across its life cycle

Why this is hard to do, but also worth doing:

- Program of record funding is specifically broken out into development and sustainment. These distinct categories of appropriations lead program managers and acquisition professionals to the conclusion that new functionality can only be added within development contracts and that money allocated for sustainment cannot be used to add new features. Vendor evaluation for development and sustainment contracts are different; vendors on sustainment contracts often do not have the same development competencies and frequently are not the people who built the original system. To create an environment that will support a DevOps/DevSecOps approach, DoD Commands and Services should jointly own the development and maintenance of software with contractors who provide more specialized capabilities. Contracts for software should focus on developing and deploying software (to operations) over the long term, rather than the typical, sequential approach - "acquiring" software followed by "sustaining" that software.

Don't	**Do**
Require customized software solutions to match DoD practices	For common functions, purchase existing software and change DoD processes to use existing apps

Business processes, financial, human resources, accounting and other "enterprise" applications in the DoD are generally not more complicated nor significantly larger in scale than those in the private sector. Commercial software, unmodified, should be deployed in nearly all circumstances. Where DoD processes are not amenable to this approach, those processes should be modified, not the software. Doing so allows the DoD to take advantage of the much larger commercial base for common functions (e.g., Concur has 25M active users for its travel software).

Don't	**Do**
Use legacy languages and operating systems that are hard to support and insecure	Use modern software languages and operating systems (with all patches up-to-date)

Modern programming languages and software development environments have been optimized to help eliminate bugs and security vulnerabilities that were often left to programmers to avoid (an almost impossible endeavor). Additionally, outdated operating systems are a major security vulnerability and the DoD should assume that any computer running such a system will eventually be compromised.[13] Standard practice in industry is to apply security patches within 48 hours of release, though even this is probably too big a window for defense systems. Treat software vulnerabilities like perimeter defense vulnerabilities: if there is a hole in your perimeter and people are getting in, you need to patch the hole quickly and effectively.

Why this is hard to do, but also worth doing:

- DoD looks at the cost of upgrading hardware as a major cost that is tied to "modernization." But hardware should be thought of as a consumable like any other, such as fuel and parts, that must be continually replaced for a weapon system to maintain operational capability. This change would require DoD to provide a stable annual budget that paid for new hardware and software capability.
- The advantage of using modern hardware and operating systems on DoD systems are manifold: better security, better functionality, reduced (unit) costs, and lower overall maintenance costs.

[13] See the DIB 10 Commandments of Software supporting thoughts and recommendations. "*Move to a model of continuous hardware refresh in which computers are treated as a consumable with a 2-3 year lifetime.*"

Don’t	Do
Turn documents like this into a process and enforce compliance	Hire competent people with appropriate expertise in software to implement the desire state and give them the freedom to do so (“competence trumps process”)

Why this is hard to do, but also why it is worth considering doing it:

- Good engineers want to build things, not just write and evaluate contracts. If their jobs are mainly contracting or monitoring, their software skills will quickly become outdated. This can be solved in the short term by a rotational program: do not allow programmers to stay in contracting for more than 4 years, so their technical capabilities are current.
- The government must team with commercial companies to ensure that it has access to the collection of talent required to develop modern software systems, as well as develop internal talent. The DoD should increase its use of contractors whose aim is not just to provide software, but to increase the software development capabilities and competency of the department. By making use of enlisted personnel, reservists, contractors, and other resources, it is possible to create and maintain highly effective teams who contribute to national security through software development.

Additional Obstacles

In addition to the specific obstacles listed above, we capture here a collection of statutes, regulations, processes and cultural norms that are impediments to implementing a modern set of software acquisition and development principles.

Statutes

The statutes below provide examples of impediments to the implementation of modern software development practices in DoD systems.

Acquisition strategy (10 U.S.C §2431a): 2431a(d) establishes the review process for major defense acquisition programs and is written around the framework of waterfall development for long timescale, hardware-centric programs. In particular, this statute establishes decision-gates at Milestone A (entry into technology maturation and risk reduction), Milestone B (entry into system development and demonstration), and entry into full-rate production. For many software programs this set of terms and approach does not make sense and is incompatible with the ability to deliver capability to the field in a rapid fashion.

Critical cost growth in major defense acquisition programs (10 U.S.C. §2433a [Nunn-McCurdy]): 2433 establishes the conditions under which Congress reviews a major program that has undergone critical cost growth and determines with it should continue. By the time a software program hits a Nunn-McCurdy breach it has already gone well past the point where the program should have been terminated and restarted using a different approach. All software procurement programs should start small, be iterative, and build on success – or be terminated quickly.

Independent cost estimation and cost analysis (10 U.S.C. §2334)

Working capital funds (10 U.S.C. §2208(r)):

- 2+ year lead times from plan to budget does not allow for continuous engineering
- Differentiating software development workload as Research, Development, Test and Engineering (RDT&E), Procurement, or Operations and Maintenance (O&M) is meaningless as there should be no final fielding or sustainment element to continuous engineering.
- System-defined program elements hinder the ability to deliver holistic capabilities and enable real-time resource, requirements, performance and schedule trades across systems without significant work.

Operational Test and Evaluation (10 U.S.C. §139b/d, 10 U.S.C. §2399): 139 establishes the position of the Director of Operational Test and Evaluation (DOT&E) and requires that person to carry out field tests, under realistic combat conditions, of weapon systems for the purpose of determining the effectiveness and suitability of those systems in combat by typical military users. 2399(a) states that a major defense acquisition program "may not proceed beyond low-rate initial production until initial operational test and evaluation of the program, subprogram, or element is completed." 2399(b)(4) further states that the program many not proceed "until the Director [of Operational Test and Evaluation] has submitted to the Secretary of Defense the report with respect to that program under paragraph (2) and the congressional defense committees have received that report." These are obstacles for DevSecOps implementation of software, where changes should be deployed to the field quickly as part of the (continuous) development process. They are an example of a "tailgate" process for OT&E that impedes our ability to deploy software quickly and drives a set of processes in which OT&E impedes rather than enhances the software development process. Instead of this process, Congress should allow independent OT&E of software to occur in parallel with deployment and also require that OT&E cycles for software match development cycles through the use of automated workflows and test harnesses wherever possible.

Additional issues:

- Testing and evaluation (T&E) must be integrated into the development life cycle to facilitate DevSecOps, and reduce operations and sustainment (O&S) costs. T&E should be present from requirements setting to O&S
- Programs need persistent and realistic environments that permit continuous, agile testing of all systems (embedded, networked, etc.) in a representative SoS environment
- Software environments should be part of the contract deliverables and accessible to T&E, including source code, build tools, test scripts, data

Definition of a major acquisition program (10 U.S.C. §2430): The designation of a program as a major acquisition program triggers a set of procedures that are designed for acquisition of hardware. This includes triggering of the DoD Instruction 5000.02, which is currently tuned for hardware systems. An alternative instruction, DoD Instruction 5000.75, is better tuned for software, but can only used for defense *business* systems; it is not valid for "weapons systems."

Depot level maintenance and repair; core logistics (10 U.S.C. §2460, 10 U.S.C. §2464): The definitions of maintenance, repair, and logistics are based on an acquisition model that is appropriate for hardware but not well aligned with the operation of modern software. For example, §2464 says that Services will "maintain and repair the weapon systems." But software is not maintained, it is *optimized* (with better performance and new functionality) on a *continuous* basis. §2460(b)(1) further states that depot level maintenance and repair "does not include the procurement of major modifications or upgrades of weapon systems that are designed to improve program performance."

Additional issues:

- DoD's challenge in shifting from applying a Hardware (HW) maintenance mindset to Software (SW) hinders DoD's ability to better leverage DoD's organic SW engineering infrastructure to deliver greater capability to the warfighter.
- DoD's acquisition process is not emphasizing an upfront focus on design for software sustainment and a seamless transition to organic software engineering sustainment to reduce the life-cycle cost of software and to speed delivery of capability over the life cycle. Such upfront emphasis is critical given the scope, complexity, and mix of the growing software sustainment demand, in the face of persistent affordability concerns.
- DoD's organic software engineering capabilities and infrastructure are critical to national security, but there is limited enterprise visibility of this infrastructure, its capabilities, workload, and resources to leverage it at the enterprise level to deliver greater capability more affordably to the warfighter.

Regulations

The regulations are the mechanism by which the DoD implements the statutes that govern its operations. They provide additional examples of impediments to the implementation of modern software development practices in DoD systems.

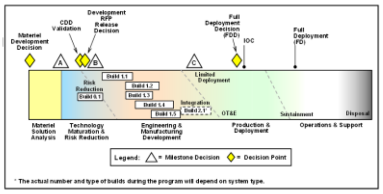
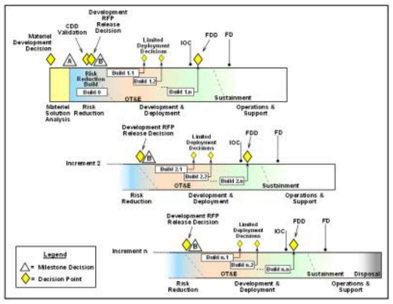
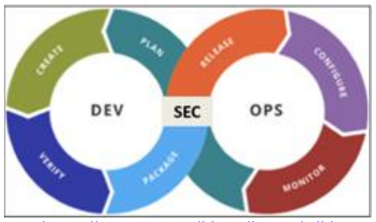
Cost estimating system requirements (48 CFR 252.215-7002) : These regulations set out the expectations for estimation of costs of a program against a set of system requirements. While perhaps appropriate for a hardware-oriented system, they do not take into account the type of continuous development cycle that is required to implement modern software.

Additional requirements for major systems (48 CFR 207.106): These regulations set out procedures for competition of contracts and are written in a manner that separates out the initial deployment of a system with the operation and sustainment of that system. This doesn't make sense for software.

Processes (Instructions)

The detailed processes used to implement the regulations are laid out in Department of Defense Instructions. We illustrate here some of the specific instructions that are obstacles to implementation of modern software development practices.

Major acquisition program development process (DODI 5000.02, par 5.c.(2) and 5.c.(3)(c)-(d)): These portions of the DoD Instructions apply to “Defense Unique Software Intensive” programs and “Incrementally Deployed Software Intensive” programs. While well-intentioned, they are still waterfall processes with years between the cycles of deployments (instead of weeks). These processes may be appropriate for some embedded systems, but are not the right approach for DoD-specific software running on commercial hardware and operating systems, as the diagrams below illustrate:

Definitely not this:	**Better, but still not right:**	**What we need:**
Specify, design, deploy, sustain  DODI 5000.02, Figure 4. Model 2: Defense Unique Software Intensive Program	 DODI 5000.02, Figure 5. Model 3: Incrementally Deployed Software Intensive Program	Implement, scale, optimize  CREATE PLAN RELEASE CONFIGURE DEV SEC OPS VERIFY PACKAGE MONITOR https://commons.wikimedia.org/wiki/File:Devops-toolchain.svg (modifications licensed CC-BY-SA)
Waterfall development	Waterfall development with overlapping builds	Continuous integration and deployment (DevSecOps)

Requirements for programs containing information technology (DoDI 5000.02, Enclosure 11): This enclosure attempts to define the requirements for ensuring information security. It is written under the assumption that the standard waterfall process is being used.

Preparation, Submission, and Execution of the Budget - Acceptance (OMB Cir A-11, II.10): This document is the primary document that instructs agencies how to prepare and submit budget requests for OMB review and approval. Section II.10 describes the conditions for acceptance of an acquired item by the government, and requires that the asset meets the requirements of the contract. The impact of this procedure is that it establishes a “100% compliance” mentality in order for the government to accept a software “asset.”

Culture

In this final section we catalog a list of culture items that do not necessarily require changes in statutes, regulations, or instructions, but rather a change in the way that DoD personnel interpret implement their processes. Changing the culture of DoD is a complex process, depending in large part on incentivizing the behaviors that will lead to the desired state.

Data and metrics

- Multiple, competing, and sometimes conflicting types of data and metrics used, or not used, for assessing software in DOD
- Inability to collect meaningful data about software development and performance in a low cost manner, at scale
- Inability to turn data into meaningful analysis and inability to implement decisions or changes to software activities (L/R/C)

Contracts

- Individual contracts are subject to review processes designed for large programs (of which they are likely enabling). This limits the agility of individual contract actions, even when modular contracting approaches are applied. In addition, the acquisition process is rigid and revolves around templates, boards, and checklists thus limiting the ability for innovation and streamlining execution.
- Contracts focus on technical requirements instead of contractual process requirements. The contract should address overall scope, PoP, and price. The technical execution requirements should be separate and managed by the product owner or other technical lead.
- Intellectual Property (IP) rights are often generically incorporated without considering the layers of technology often applied to a solution. A single solution might include open source, proprietary SW, and government custom code. The IP clauses should reflect all of the technology that is used.

Security Accreditation

- Although developing and operating software securely is a primary concern, the means to achieve and demonstrate security is overly complex and hampered by inconsistent and outdated/misapplied policy and implementation practices (e.g., overlaying historical DoD Information Assurance Certification and Accreditation Process (DIACAP) over risk management framework (RMF) controls for individual pieces of software versus system accreditation). The sense is that the certification and accreditation process is primarily a "check- the-box" documentary process, adds little value to the overall security of the system, and is likely to overlook flaws in the design, implementation, and the environment in which the software operates.
- The DoD needs to be able to calculate the true and component costs for implementing the RMF and certification and accreditation (C&A) in order to identify inefficiencies, duplicative capabilities, and redundant or overlapping security products and services that are being acquired or developed. Absent a set of metrics it is difficult to prioritize risk areas, investments, and evaluating risk reduction and return on investment.
- The DoD needs to ensure that each Joint Capability Area (JCA) flow-down its strategy, best practices, and implementation requirements/guidance for security and accreditation to allow the Component responsible for implementing the software to appropriately tailor RMF and plan the development, accreditation, and operation of the software.
- The DoD needs to provide automated tools and services needed to integrate continuous monitoring with the development life cycle, enable continuous assessment and accreditation, and delegate decision making at the lowest level possible. The DoD should embrace DevSecOps (not just DevOps) and provide policy supported processes, certified libraries, tools, and a toolchain reference implementation to produce "born secure" software

Testing and Evaluation

- The DoD lacks the realistic test environments needed to support test at the pace of modern software methods.
- The DoD lacks the modern software intellectual property (IP) regime needed to support test and evaluation at the pace of modern software methods
- The DoD lack the enterprise knowledge management/data analytics capability needed to support evaluation of test data at the pace of modern software methods

Workforce

- No defined requirements for software developers
- Antiquated policies (talent management, software development)
- Culture and knowledge (DoD, societal, defense contractors)

Appropriations/Funding

- 2+ year lead times from plan to budget does not allow for continuous engineering
- Differentiating software development workload as Research, Development, Test and Engineering (RDT&E), Procurement, or Operations and Maintenance (O&M) is meaningless as there should be no final fielding or sustainment element to continuous engineering.
- System defined program elements hinder the ability to deliver holistic capabilities and enable real-time resource, requirements, performance and schedule trades across systems without significant work.

Infrastructure

- Creating software: The DoD lacks availability of vetted, secure, reusable components, either as source code, or other digital artifacts (think hardened Docker containers or virtual machines (VMs) here). A repository of discoverable, well indexed, vetted, secure, and reusable components could go a long way. This also emphasizes the point that an awful lot of software now-a-days is software by construction with minimal “glue” code applied.
- Building/managing/testing software: There is a general lack of available tools to build software, especially automated tools (testing/scanning/fuzzing etc.) integrated into a secure pipeline supporting rapid agile development. There is also a significant need to have a common, government owned and managed code repository that all programs could/should/must use (e.g., government-furnished GitHub).
- Running/hosting software: The DoD needs to continually push the level of abstraction up as much as possible for programs. Traditionally programs, even cloud-based solutions, tend to start at Infrastructure as a Service (IaaS) and build their own rest of the stack. We need secure and available Platform as a Service (PaaS) and Function as a Service (FaaS) so that programs only need to focus on core business logic and not on securing a database or message bus over and over again.
- Operating/updating securely: Once developed and instantiated on a secure and available platform, we need to continually monitor, red team (automated?), and evolve the software. This requires proper instrumentation, logging, and monitoring of the platform, supporting libraries/components, and the core program code. A standard/common way to provide instrumentation and monitoring of the running services built into the infrastructure would be very helpful.

Requirements

- A byproduct of top-level requirement flow down is rigidity and over specificity at the derived requirements level that greatly hinders agile s/w design.
- Too often exquisite requirements are levied on a system that in turn drive extensive complex software requirements and design, affecting development, integration, and system test.
- Data sets are siloed within programs: a common “law of requirements” is that programs of record try to avoid dependencies with other programs of record. This is problematic for software-based capabilities because data is often siloed within single programs of record. We have network programs to “pass” data, but the promise of artificial intelligence (AI), including machine learning (ML), is that software algorithms can leverage pools of data from disparate sources (data lakes).
- By tying software to a program of record, it becomes harder to transfer that code across systems and data environments. As a result, DoD limits code reuse within and across Services.

Modernization and sustainment

- DoD’s challenge in shifting from applying a hardware maintenance mindset to software hinders DoD’s ability to better leverage DoD’s organic software engineering infrastructure to deliver greater capability to the warfighter.
- DoD’s acquisition process is not emphasizing an upfront focus on design for software sustainment and a seamless transition to organic software engineering sustainment to reduce the life-cycle cost of software and to speed delivery of capability over the life cycle. Such upfront emphasis is critical given the scope, complexity, and mix of the growing software sustainment demand, in the face of persistent affordability concerns.
- DoD’s organic software engineering capabilities and infrastructure are critical to national security, but there is limited enterprise visibility of this infrastructure, its capabilities, workload, and resources to leverage it at the enterprise level to deliver greater capability more affordably to the warfighter.

Acquisition Strategy

- Acquisition policy framework: Create a cohesive acquisition policy architecture within which effective, efficient software acquisition policy has a home.
- Acquisition management and governance: Flip the concept of an oversight model on its head.

DIB Guide: Detecting Agile BS

Agile is a buzzword of software development, and so all DoD software development projects are, almost by default, now declared to be “agile.” The purpose of this document is to provide guidance to DoD program executives and acquisition professionals on how to detect software projects that are really using agile development versus those that are simply waterfall or spiral development in agile clothing (“agile-scrum-fall”).

Principles, Values, and Tools

Experts and devotees profess certain key “values” to characterize the culture and approach of agile development. In its work, the DIB has developed its own guiding maxims that roughly map to these true agile values:

Agile value	DIB maxim
Individuals and interactions over processes and tools	“Competence trumps process”
Working software over comprehensive documentation	“Minimize time from program launch to deployment of simplest useful functionality”
Customer collaboration over contract negotiation	“Adopt a DevSecOps culture for software systems”
Responding to change over following a plan	“Software programs should start small, be iterative, and build on success – or be terminated quickly”

Key flags that a project is not really agile:

- Nobody on the software development team is talking with and observing the users of the software in action; we mean the *actual* users of the *actual* code.[14] (The PEO does not count as an actual user, nor does the commanding officer, unless she uses the code.)
- Continuous feedback from *users* to the development team (bug reports, users assessments) is not available. Talking once at the beginning of a program to verify requirements doesn’t count!
- Meeting requirements is treated as more important than getting something useful into the field as quickly as possible.
- Stakeholders (dev, test, ops, security, contracting, contractors, end-users, etc.)[15] are acting more-or-less autonomously (e.g., ‘it’s not my job.’)
- End users of the software are missing-in-action throughout development; at a minimum they should be present during Release Planning and User Acceptance Testing.

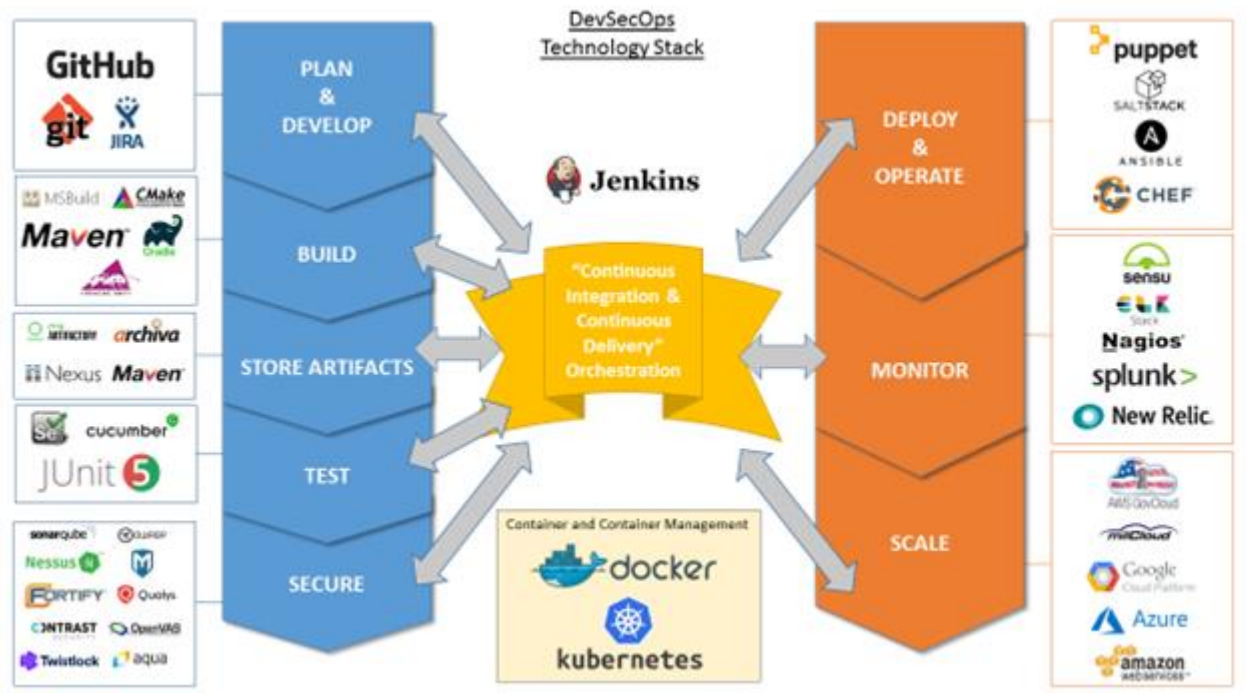
[14] Acceptable substitutes for talking to users: Observing users working, putting prototypes in front of them for feedback, and other aspects of user research that involve less talking.
[15] Dev is short for development, ops is short for operations

- DevSecOps culture is lacking if manual processes are tolerated when such processes can and should be automated (e.g., automated testing, continuous integration, continuous delivery).

Some current, common tools in use by teams using agile development (these will change as better tools become available):[16]

- Git, ClearCase, or Subversion - version control system for tracking changes to source code. Git is the *de facto* open source standard for modern software development.
- BitBucket or GitHub - Repository hosting sites. Also provide issues tracking, continuous integration "apps" and other productivity tools. Widely used by the open source community.
- Jenkins, Circle CI or Travis CI - continuous integration service used to build and test BitBucket and GitHub software projects
- Chef, Ansible, or Puppet - software for writing system configuration "recipes" and streamlining the task of configuring and maintaining a collection of servers
- Docker - computer program that performs operating-system-level virtualization, also known as "containerization"
- Kubernetes or Docker Swarm for Container orchestration
- Jira or Pivotal Tracker - issues reporting, tracking, and management

Graphical version:



Questions to Ask Programming Teams

[16] Tools listed/shown here are for illustration only: no endorsement implied.

- How do you test your code? (Wrong answers: “we have a testing organization,” “OT&E is responsible for testing”)
    - Advanced version: what tool suite are you using for unit tests, regression testing, functional tests, security scans, and deployment certification?
- How automated are your development, testing, security, and deployment pipelines?
    - Advanced version: what tool suite are you using for continuous integration (CI), continuous deployment (CD), regression testing, program documentation; is your infrastructure defined by code?
- Who are your users and how are you interacting with them?
    - Advanced version: what mechanisms are you using to get direct feedback from your users? What tool suite are you using for issue reporting and tracking? How do you allocate issues to programming teams? How to you inform users that their issues are being addressed and/or have been resolved?
- What is your (current and future) cycle time for releases to your users?
    - Advanced version: what software platforms to you support? Are you using containers? What configuration management tools do you use?

Questions for Program Management

- How many programmers are part of the organizations that owns the budget and milestones for the program? (Wrong answers: “we don’t know,” “zero,” “it depends on how you define a programmer”)
- What are your management metrics for development and operations; how are they used to inform priorities, detect problems; how often are they accessed and used by leadership?
- What have you learned in your past three sprint cycles and what did you do about it? (Wrong answers: “what’s a sprint cycle?,” “we are waiting to get approval from management”)
- Who are the users that you deliver value to each sprint cycle? Can we talk to them? (Wrong answers: “we don’t directly deploy our code to users”)

Questions for Customers and Users

- How do you communicate with the developers? Did they observe your relevant teams working and ask questions that indicated a deep understanding of your needs? When is the last time they sat with you and talked about features you would like to see implemented?
- How do you send in suggestions for new features or report issues or bugs in the code? What type of feedback do you get to your requests/reports? Are you ever asked to try prototypes of new software features and observed using them?
- What is the time it takes for a requested feature to show up in the application?

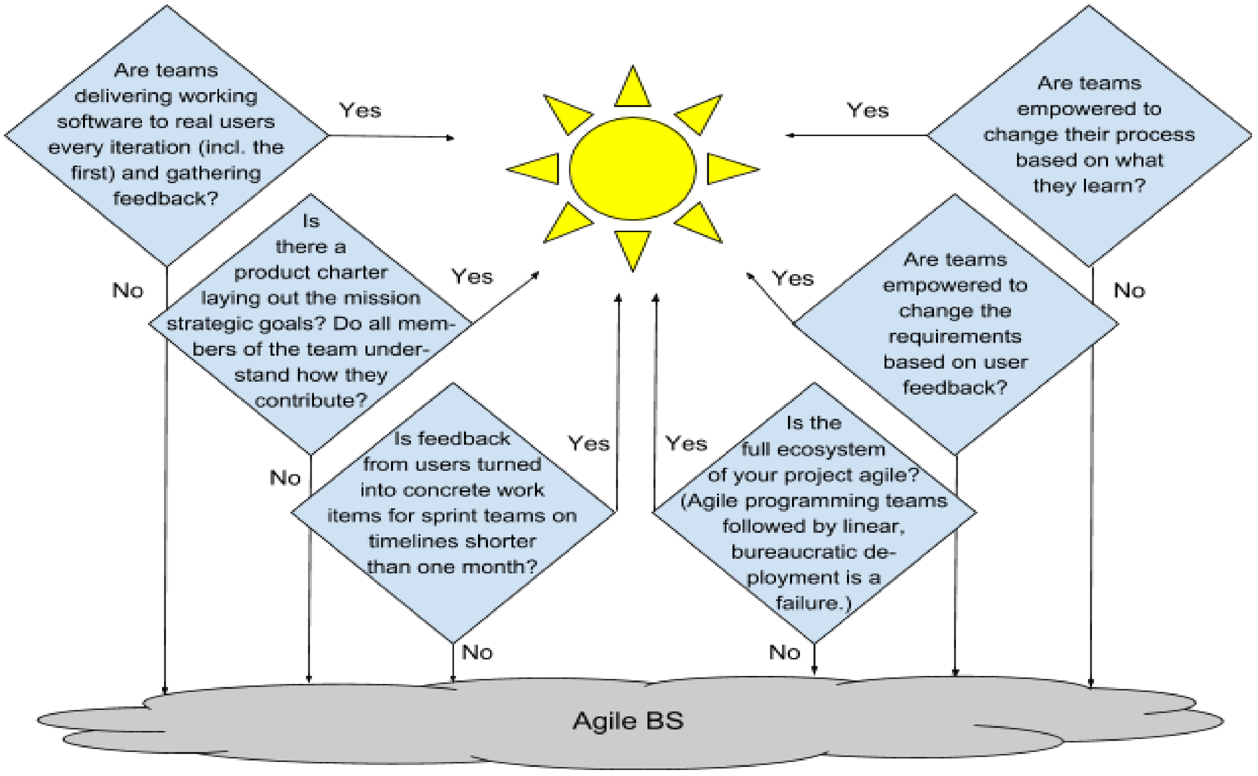
Questions for Program Leadership

- Are teams delivering working software to at least some subset of real users every iteration (including the first) and gathering feedback? (alt: every two weeks)

- Is there a product charter that lays out the mission and strategic goals? Do all members of the team understand both, and are they able to see how their work contributes to both?
- Is feedback from users turned into concrete work items for sprint teams on timelines shorter than one month?
- Are teams empowered to change the requirements based on user feedback?
- Are teams empowered to change their process based on what they learn?
- Is the full ecosystem of your project agile? (Agile programming teams followed by linear, bureaucratic deployment is a failure.)

For a team working on agile, the answer to all of these questions should be “yes.”

Graphical version:


Are teams delivering working software to real users every iteration (incl. the first) and gathering feedback?
Yes
No
Is there a product charter laying out the mission strategic goals? Do all members of the team understand how they contribute?
Yes
No
Is feedback from users turned into concrete work items for sprint teams on timelines shorter than one month?
Yes
No
Is the full ecosystem of your project agile? (Agile programming teams followed by linear, bureaucratic deployment is a failure.)
Yes
No
Are teams empowered to change the requirements based on user feedback?
Yes
Are teams empowered to change their process based on what they learn?
Yes
No
Agile BS

More information on some of the features of DoD software programs are included in the DIB’s “Ten Commandments of Software,” the DIB’s “Metrics for Software Development,” and the DIB’s “Do’s and Don’ts of Software.”

Is Your Development Environment Holding You Back? A DIB Guide for the Acquisition Community

A strong software development team is marked by some common attributes, including the use of practices, processes, and various tools.

An effective team starts with clear goals. The entire software team should have a clear sense of the project's goals and the value they seek to provide "the client." The goals should be translated into specific objectives, which may be measured in terms of agreed-upon key performance indicators (KPIs) or other frameworks. An effective development environment is one designed to deliver value toward those goals. (This KPI-driven paradigm should *not* be seen as an invitation to reprise an extended debate about requirements.)

Technical practices and processes that enable a development environment to deliver value toward those goals include:

- Organization of activities through discrete "user stories" that can be broken down into smaller components and continually prioritized by the product owner
- Relatively short "sprints" (often two weeks), each ending in a retrospective, that enable measurement and learning throughout the process
- Blameless postmortems that allow for maximum learning and speedy recovery from failures
- Automated testing, security, and deployment
- Testing (including user testing) and security should be shifted to the left and be part of the day-to-day operations within the development teams
- Continuous integration, in which developers integrate code into a shared repository several times a day, and check-ins are then verified by an automated build for early problem detection
- Continuous delivery or continuous deployment, in which the software is seamlessly deployed into staging and production environments
- Trunk-based development, in which team members work in small batches and develop off of trunk or master, rather than long-lived feature branches
- Version control for all production artifacts including open source and third party libraries
- Infrastructure as code: version control for all configuration, networking requirements, container orchestration files, continuous integration/continuous delivery (CI/CD) pipeline files
- Ability to execute A/B testing and canary deployments
- Ability to get rapid and continuous user feedback and to test new features with users throughout the development process

Effective teams will practice continuous delivery, in which teams deploy software in short cycles, ensuring that the software can be reliably released at any time. Continuous deployment can be measured by a team's ability to achieve the following outcomes:

- Teams can deploy on-demand to production or to end users throughout the software delivery life cycle.

- Fast feedback on the quality and deployability of the system is available to everyone on the team, and people make acting on this feedback their highest priority.

Specific measures that will help you gauge if your development environment is working as it should include development frequency; lead time for changes; time to restore service after outage; and change failure rate (rollback deployed code). These questions and data, borrowed from the 2017 State of DevOps Report from DORA, can help assess where your teams stand:

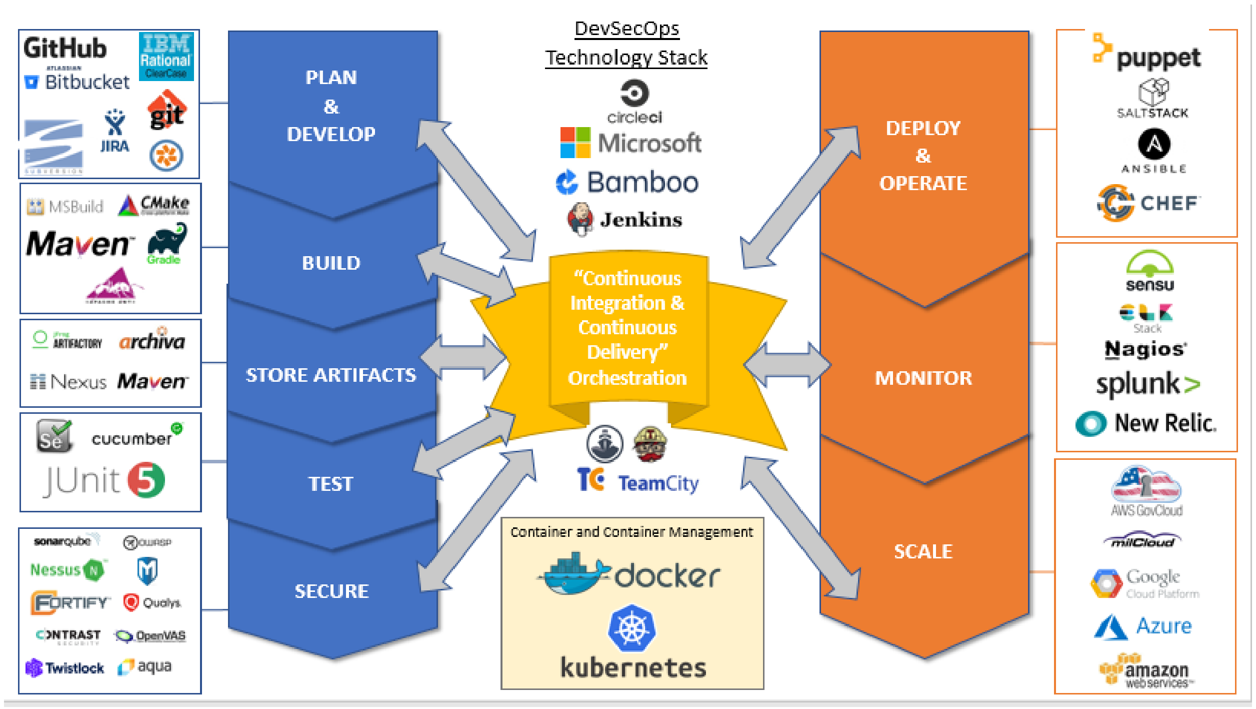
	High performance	**Medium performance**	**Low performance**
Deployment frequency How often does your organization deploy code?	On demand (multiple deploys per day)	Between once per week and once per month	Between once per week and once per month
Lead time for changes What is your lead time for changes (i.e., how long does it take to go from code-commit to code successfully running in production)?	Less than one hour	Between one week and one month	Between one week and one month*
Mean time to recover (MTTR) How long does it generally take to restore service when a service incident occurs (e.g., unplanned outage, service impairment)?	Less than one hour	Less than one day	Between one week and one day
Change failure rate What percentage of changes results either in degraded service or subsequently requires remediation (e.g., leads to service impairment, service outage, requires a hotfix, rollback, fix forward, patch)?	0-15%	0-15%	31-45%

* Low performers were lower on average (at a statistically significant level), but had the same median as the medium performers (2017 DevOps Report)

There is ***no exact set of tools*** that indicate that your development environment is working as it should, but the use of some tools will often indicate that the practices and processes above are in place. You commonly see effective software teams using:

- An issue tracker, like Jira or Pivotal Tracker
- Continuous integration and/or continuous integration/continuous delivery (CI/CD) tools, like Jenkins, Circle CI, or Travis CI
- Automated build tools, like Maven, Grable, Cmake, and Apache Ant
- Automated testing tools, like Selenium, Cucumber, J-Unit

- A centralized artifacts repository, like Nexus, Artifactory, or Maven
- Automated security tools for static and dynamic code analysis and container security, like Sonarqube, OWASP ZAP, Fortify, Nessus, Twistlock, Aqua, and more.
- Automation tools, like Chef, Ansible, or Puppet
- Automated code review tools, like Code Climate
- Automated monitoring tools, like Nagios, Splunk, New Relic, and ELK
- Container and container orchestration tools like Docker, Docker Swarm, Kubernetes, and more



Warning signs that you may have screwed up your development environment include:

- If teams cannot effectively track progress toward defined goals and objectives roughly every two weeks
- If teams cannot rapidly deploy various environments that mirror production to test their code such as in development, QA, and staging
- If teams cannot have real-time feedback regarding their code building, passing tests, and passing security scans
- If it takes months for end users to be able to see changes and provide feedback
- If teams cannot rapidly roll-back to previous versions or perform rolling-update to new versions without downtime
- If recovering from incidents results in significant drama or the assignment of blame
- If having code ready to deploy is a big event (it should happen routinely and without drama)
- If changes to the software frequently result in breaking it

If developers are not empowered to change the code or build new functionality based on user feedback, or to change their process based on what they learn.

Is Your Compute Environment Holding You Back?
A DIB Guide for the Acquisition Community

To enable software to provide a competitive advantage to the warfighter, DoD must adopt a strategy for rapidly transitioning DoD IT to current industry standards. This modernization agenda should include providing distributed databases and abundant computing power; making bandwidth available as a platform; integrating mobile technologies; and developing DoD platforms for downloading applications. This document outlines compute and infrastructure capabilities that should be available to DoD programmers (and contractors) who are developing software for national defense. The capabilities include:

1. **Scalable compute.** Access to computing resources should never be a limiting factor when developing code. Modern cloud environments provide mechanisms to provide any developer with a powerful computing environment that can easily scale with the needs of an individual programmer, a product development team, or an entire enterprise.

2. **Containerization.** Container technology provides sandbox environments in which to test new software without exposing the larger system to the new code. It "packages up" an application with all of the operating system services required for executing the application and allowing that application to run in a virtualized environment. Containers allow isolation of components (that communicate with each other through well-defined channels) and provide a way to "freeze" a software configuration of an application without freezing the underlying physical hardware and operating system.

3. **Continuous integration/continuous delivery (CI/CD) pipeline (DevSecOps platform).** A platform that provides the CI/CD pipeline is used for automated testing, security, and deployment. This includes license access for security tools and a centralized artifacts repository with tools, databases, and a base operating system (OS) with an existing authorization to operate (ATO).

4. **Infrastructure as code: automated configuration, updating, distribution, and recovery management.** Manual configuration management of operating systems and middleware platforms leads to inconsistencies in fielded systems and drives up the operating costs due to the labor hours required for systems administration. Modern software processes avoid this by implementing "infrastructure as code," which replaces manual processes for provisioning infrastructure with automated processes that use machine-readable definition files to manage and provision containers, virtual machines, networking, and other components. Adopting infrastructure as code and software distribution tools in a standardized way streamlines uniformity of deployment and testing of changes, which are both vital to realizing the benefits of agile development processes.

5. **Federated identity management and authentication backend with common log file management and analysis.** Common identity management across military, government, and contractors greatly simplifies the assignment of permissions for accessing information across multiple systems and allows rapid and accurate auditing of

code. The ability to audit access to information across multiple systems enables the detection of inappropriate access to information, and can be used to develop the patterns of life that are essential for proper threat analysis. Common identity management can ease the integration of multi-factor authentication across servers, desktops, and mobile devices. Along with public key infrastructure (PKI) integration, it allows verification of both the service being accessed by the user and the user accessing information from the service.

6. **Firewall configuration and network access control lists.** Having a common set of OS and application configurations allows network access control not just through network equipment, but at the server itself. Pruning unnecessary services and forcing information transfer only through intentional interfaces reduce the attack surface and make servers more resilient against penetration. Server-to-server communication can be encrypted to protect from network interception and authenticated so that software services can only communicate with authorized software elements.

7. **Client software.** Remote login through remote desktop access is common throughout DoD. This greatly increases the difficulty of integrating mobile platforms and of permitting embedded devices to access vital information, especially from the field. It also complicates uniform identity management and multi-factor verification, which is key to securing information. By moving to web client access mobile integration - and development - is greatly eased. It also becomes possible to leverage industry innovation, as this is where the commercial sector is heading for all interactions.

8. **Common information assurance (IA) profiles.** Information assurance (IA) for DoD systems is complex, difficult, and not yet well-architected. Test, certification, and IA are almost always linear “tailgate” processes instead of being integrated into a continuous delivery cycle. Common IA profiles integrated into the development environment and part of the development system architecture are less likely to have bugs than customized and add-on solutions.

Desired State with Examples

Effective use of software requires sufficient resources for computing, storage, and communications. Software development teams must be provided with abundant compute, storage, and bandwidth to enable rapid creation, scaling, and optimization of software products.

Modern cloud computing services provide such environments and are widely available for government use. In its visits to DoD programs, the DIB Software Acquisition and Practices (SWAP) team has observed many programs that are regenerating computing infrastructure on their own—often in a highly non-optimal way—and typically due to constraints (or perceived constraints) created by government statutes, regulations, and culture. This approach results in situations where compute capability does not scale with needs; operating systems cannot be upgraded without upgrading applications; applications cannot be upgraded without updating the operating systems; and any change requires a complete information assurance recertification.

Compute platforms are thus "frozen" at a point established early in the program life cycle, and development teams are unable to take advantage of new tools and new approaches as they become available. The DIB SWAP team has noted a general lack of good tools for profiling code, maintaining access and change logs, and providing uniform identity management, even though the DoD has system-wide credentials through Common Access Control (CAC) cards.

It would be highly beneficial to create common frameworks and/or a common set of platforms that provide developers with a streamlined or pre-approved Authority to Operate (ATO). Use of these pre-approved platforms should not be mandated, but they create cost and time incentives by enabling more consolidated platforms. DoD could make use of emerging government cloud computing platforms or achieve similar consolidation within a DoD-owned data center (hybrid cloud). DoD should move swiftly from a legacy data center approach to a cloud-based model, while taking into account the lessons learned and tools and services available from commercial industry, with assumed hardware and operating system updates every 3-5 years.

Warning signs

Some indicators that you may have screwed up your compute environment include:

- Your programmers are using tools that are less effective than what they used in school
- The headcount needed to support the system grows linearly with the number of servers or instances
- You need system managers deployed with hardware at field locations because it is impossible to configure new instances without high skill local support
- You have older than current versions of operating systems or vendor software because it is too hard to test or validate changes
- Unit costs for compute, network transport and storage are not declining, or are not measurable to be determined
- Logging in via remote desktop is the normal way to access an information service
- You depend on network firewalls to secure your compute resource from unauthorized access
- You depend on hardware encryptors to keep your data safe from interception
- You have to purge data on a regular basis to avoid running out of storage
- Compute tasks are taking the same or longer time to run than they did when the system was first fielded
- Equipment or software is in use that has been "end of lifed" by the vendor and no longer has mainstream support
- It takes significant work to find out who accessed a given set of files or resources over a reasonable period of time
- No one knows what part of the system is consuming the most resources or what code should be refactored for optimization
- Multifactor authentication is not being used
- You cannot execute a disaster recovery exercise where a current backup up of a system cannot be brought online on different hardware in less than a day

Getting It Right

These capabilities should be available to all DoD programmers and contractors developing software for national defense:

Scalable compute
- Modern compute architectures
- Environments that make transitions across cloud and local services easy
- Graphics Processing Unit (GPU)- and ML-optimized compute nodes available for specialized tasks
- Standardized storage elements and ability to expand volumes and distribute them based on performance needs
- Standardized network switching options with centralized image control
- Property management tagging—no equipment can be placed in a data center without being tagged for inventory and tracked for End of Life support from vendors
- Supply chain tracking for all compute elements

Containerization
- Software deployment against standard profile OS image
- Containers can be moved from physical to cloud-based infrastructure and vice versa
- Applications and services run in containers and expand or contract as needed
- OS updates separated from application container updates
- Centralized OS patch validation and testing
- Containers can be scaled massively horizontally
- Containers are stateless and can be restarted without impact
- Configuration management for deployment and audit

Continuous integration/continuous delivery (CI/CD) pipeline (DevSecOps platform)
- Select, certify, and package best of breed development tools and services
- Can be leveraged across DoD Services as a turnkey solution
- Develop standard suite of configurable and interoperable cybersecurity capabilities
- Provide onboarding and support for adoption of Agile and DevSecOps
- Develop best-practices, training, and support for pathfinding and related activities
- Build capability to deliver a Software Platform to the Defense Enterprise Cloud Environment
- Self-service portal to selectively configure and deliver software toolkit with pre-configured cybersecurity capabilities

Infrastructure as code: automated configuration, updating, distribution and recovery management
- Ability to test changes against dev environments
- Standardized profiling tools for performance measurement
- Centralized push of patches and updates with ability for rapid rollback
- Auditing and revision control framework to ensure proper code is deployed and running
- Ability to inject faults and test for failover in standardized ways

- Disaster recovery testing and failover evaluation
- Utilization tracking and performance management utilities to predict resource crunches
- Standardized OS patch and distribution repositories
- Validation tools to detect manual changes to OS or application containers with alerting and reporting

Federated identity management and authentication backend with common log file management and analysis

- Common identity management across all DoD and contractors
- Common multifactor backends for authentication of all users along with integration of LDAP/Radius/DNS or active directory services
- Integrated PKI services and tools for automated certificate installation and updating
- Common DRM modules that span domains between DoD/contractors and vendor facilities that can protect, audit and control documents, files, and key information. All encrypted at rest, even for plain text files.
- Useful for debugging and postmortem analysis
- Develop patterns of life to flag unusual activity by users or processes
- Automated escalation to defensive cyber teams

Firewall configuration and network access control lists

- Default configuration for containers is no access
- Profiles for minimal amounts of ports and services being open/run
- All network communications are encrypted and authenticated, even on the same server/container

Client software

- Web-based access the norm, from desktops/laptops as well as mobile devices
- Remote login used as a last resort - not as the default
- Security technical implementation guides (STIGs) for browsers and plugins, as well as common identity management at the browser interface (browsers authenticate to servers as well as servers authenticating to browsers)
- Minimal state kept on local hardware - purged at end of session

Common information assurance (IA) profiles

- Enforces data encrypted in flight and at rest
- Software versions across DoD with automated testing
- Application lockdowns at the system level so only authorized applications can run on configured systems
- “Makefile” to build configurations from scratch from base images in standardized approved configurations
- Use of audit tools to detect spillage and aid in remediation (assisted via DRM

SWAP Program Visits: Questions and Observations

Programs Reviewed

Reviewed 6 programs to date:

- Next Generation fighter jet
- Next Generation ground system
- Kessel Run—AOC Pathfinder
- Space tracking system
- Naval radar system
- Cross-service business system

What we hope to understand:

- Why is the software the way it is?
- How have you gone about developing and deploying it?
- What constraints/obligations have you been under and what would be your recommendations to change those?

Standard Questions

- What is the coding environment and what languages/SW tools do you use?
- What do the software and system architectures look like?
- What is the computational environment (processing, comms, storage)?
- How is software deployed and how often are updates delivered to the field?
- What determines the cycle time for updates?
- How does software development incorporate user feedback? What is the developer-user interface? How quickly are user issues addressed and fixed?
- How long does it take to compile the code from scratch?
- How much access does the DoD have to the source code?
- How is testing done? What tool suites are used? How much is automated? How long does it take to do a full regression test?
- How is cybersecurity testing done? How are programs/updates certified?
- What does the workforce look like (headcounts, skill sets)? How many programmers? How much software expertise is there in the program office?
- What is the structure of the contract with the government? How are changes, new features, and new ideas integrated into the development process?

Preliminary Observations

- Software is being delivered to the field 2-10X slower than it could be due to outdated requirements, test requirements, and lack of trust in SW
- Many systems are using legacy hardware and outdated architectures that make it much harder to exploit advances in computing and communications

- Program requirements were often formulated 5+ years ago (when the threat environment + available technologies were very different => wasted effort)
- New capabilities and features are added in multi-year (multi-decade?) development “blocks” instead of continuously and iteratively
- Most program offices don’t have enough expertise in modern SW methods
- Most SW teams are attempting to implement DevOps and “agile” approaches, but in most cases the capabilities are still nascent (and hence fragile)
- Transition to DevOps is often hindered by a gov’t support structure focused on technical performance in a waterfall setting (“waterfall with sprints”)
- Information assurance (IA) is complex, difficult, and not yet well architected
- Test, certification and IA are almost always linear “tailgate” processes instead of being integrated into a continuous delivery cycle.

What should be done differently in future programs?

- Spend time upfront getting the architecture right: modular, automated, secure
- Make use of platforms (hardware and software) that continuously evolve at the timescales of the commercial sector (3-5 years between HW/OS updates)
- Start small, be iterative, and build on success – or terminate quickly
- Construct budget to support the full, iterative life cycle of the software
- Adopt a DevOps culture: design, implement, test, deploy, evaluate, repeat
- Automate testing of software to enable critical updates to be deployed in days to weeks, not months or years (also requires changes in testing organization)
- Have a local team of DoD software experts who are capable of modifying or extending the software through source code or API access
- Separate development of mission level software from development of IA-accredited platforms

How to Justify Your Budget When Doing DevSecOps

As we transition software development from big spiral programs into DevSecOps, program managers will have to wrestle with using new practices of budget estimation and justification, while potentially being held to old standards that should no longer apply. In addition to all of the regular challenges of retaining a budget allocation (budget reviews, audits, potential reductions and realignment actions, all many times a year), defending a budget for a DevSecOps acquisition requires additional explanation and justification because those charged with oversight—whether inside the Department or in Congress—have come to expect specific information on a tempo that doesn't make sense for DevSecOps projects. Program managers leading DevSecOps projects therefore must not only do the hard work of leading agile teams toward successful outcomes, but also create the conditions that allow those teams to succeed by convincing cost assessors and performance evaluators to evaluate the work differently. Fortunately, commercial industry already has best practices for budget estimation and justification for DevSecOps and that DoD should follow industry approaches rather than create new ones

This DIB Guide is intended to help with this challenge. It seeks to provide guidelines and approaches to help program managers of DevSecOps projects[17] interact with those cost assessors and performance evaluators through the many layers of review and approval authorities while carrying out their vital oversight role. This guide should help with projects where the development processes is optimized for software rather than hardware and where most key stakeholders are aligned around the goal of providing needed capability to the warfighter without undue delay.

Questions that we attempt to answer in this concept paper:

1. What does a well-managed software program look like and how much should it cost?
2. What are the types of metrics that should be provided for assessing the cost of a proposed software program and the performance of an ongoing software program?
3. How can a program defend its budget if the requirements aren't fixed or are changing?
4. How do we estimate costs for "sustainment" when we are adding new features?
5. Why is ESLOC (effective source lines of code) a bad metric to use for cost assessment (besides the obvious answer that it is not very accurate)?

What does a well-managed DevSecOps program look like and how much should it cost?

The primary focus for DevSecOps programs is about regular and repeatable, sustainable delivery of innovative results on a time-box pattern, not on specifications and requirements without bounding time (Figure 1). The fixed-requirements spiral-development spending model has created program budgets that approach infinity. DevSecOps projects, on the other hand will be focused on different activities at different stages of maturity. In a DevSecOps project, management should be tracking services and measuring the results of working software as the product evolves, rather than inspecting end items when the effort is done, as would be expected

[17] Not all software is the same; we focus here only on software programs using or transitioning to DevSecOps.

in a legacy model. Software is never done and not all software is the same, but generally the work should look like a steady and sustainable continuum of useful capability delivery.

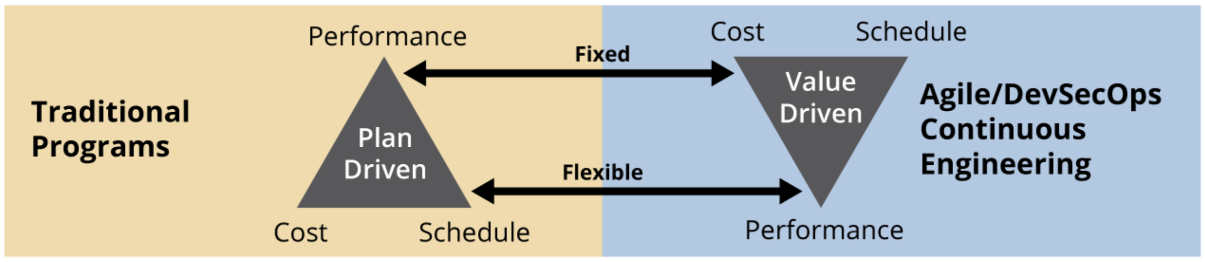


Figure 1. Value Driven Iron Triangle (Carnegie Mellon University, Software Engineering Institute).

- During the creation phase, program managers will most likely decide to adopt Agile based on criteria that fits their design challenge (e.g., software dependent). They would also be motivated to build their products on top of widely used software platforms that are appropriate for the technical domain at hand (e.g., embedded vs. web applications). During this phase team also establishes base capability and what they consider a minimum viable product (MVP).[18] This is where all programs start and many should end. Starting small and incrementing is not only the right way to do software, but it is also a great way to limit financial exposure. A key tenet of agile development is learning early and being ready to shift focus to increase the likelihood for success.
- During the scaling phase, the entire team (industry and government) commit and learn how to transition to appropriate agile activities that are optimizing for implementing DevSecOps for the project. This should focus the team on transitioning to a larger user base with improved mechanisms for automated testing (including penetration testing), red team attacks, and continuous user feedback. A key management practice in agile development is to keep software projects to a manageable size. If the project requires more scope, divide the effort into modular, easily connected chunks that can be managed using agile methods and weave the pieces together in implementation.
- Once into implementation, a well-managed program should have a regular release cadence (e.g., for IT projects every 2-3 weeks, while safety-critical products could run a bit longer, 3-4 weeks). Each of these releases delivers small increments of software that are as intuitive to use as possible and directly deployable to actual users. DevSecOps programs move from small successes into larger impacts.

With allowances made for different sizes of project, DevSecOps should share certain characteristics, including:

- An observer should easily find an engaged program office, as well as development teams that are small (5-11 people), and well connected to one another through structured meetings and events (a.k.a. "ceremonies").

[18] The MVP should not be overspecified since the main goal is getting the MVP into the hands of users for feedback.

- A set of agile teams work on cross-functional capabilities of the system and include a planning team and a system architecture team.
- The teams should have frequent interaction with subject matter experts and users from the field or empowered product owners. Active user engagement is a vital element of an Agile approach, but getting actual users (*not* just user representatives) to participate also needs to be a managed cost that the program needs to plan for.
- The project should have a development environment that supports transparency of the activities of the development teams to the customer. Maximal automation of reporting is the norm for commercial development and should be for DoD programs as well.
- The program should include engaged test and certification communities who are deeply involved in the early stages (i.e., who have "shifted left") and throughout the development process. Not just checkers at the end of that process. They would help design and validate the use of automation and computer-assisted testing/validation tools whenever possible as well.
- Capability should also be delivered in small pieces on a continuing basis—as frequently as every two weeks for many types of software (see the DIB's Guide to Agile BS).

The cost of a program always depends on the scale of the solution being pursued, but in an agile DevSecOps project, the cost should track to units of 5–11-person cross-functional team (team leader, developers, testers, product owners, etc.) with approximately 6–11 teams making up a project. If the problem is bigger than that, the overall project could be divided up into related groups of teams. A reliance on direct interaction between people is another central element of Agile and DevSecOps; the communication overhead means that this approach loses effectiveness with too many people in a team (typically 5–11 cross-functional members). Also, groups of teams have difficulty scaling interactions when the number of teams gets too large (less than twelve). A team-of-teams approach will allow scaling to fit the overall scope. Organizing the teams is also a valuable strategy where higher level development strategies and system architectures get worked out and the lower level teams are organized around cross-domain capabilities to be delivered. Cost incentives for utilizing enterprise software platform assets should be so attractive, and the quality of that environment so valuable, that no program manager would reasonably decide to have his/her contractor build their own.

Here are some general guidelines for project costs when pursuing a DevSecOps approach:

- Create: deliver initial useful capability *to the field* within 3-6 months (the use of commodity hardware and rapid delivery to deployment). If this cannot be achieved, it should be made clear that the project is at risk of not delivering and is subject to being canceled. Outcomes and indicators need to be examined for systematic issues and opportunities to correct problems. Initial investment should be limited in two ways: 1) in size to limit financial exposure and 2) in time to no more than 1 year.

- Scale: deliver increased functionality across an expanding user base at decreasing unit cost with increased speed. Investment should be based on the rate limiting factors of time and talent, not cost. Given a delivery cycle and the available talent, the program should project only spending to the staffing level within a cycle.

- Good agile management is not about money, it is about regular and repeated deliver. That is to say, it is about time boxing everything. Releases, staffing, budget, etc. Nick, strongly recommend that you rework this to reflect time boxing as the most important aspect of “defending your agile budget.

- Optimize: deliver increased functionality fixed or decreasing unit cost (for a roughly constant user base). Investment limit should be less than 3 project team sets[19].

What are the types of metrics that should be provided for assessing the cost of a proposed software program and the performance of an ongoing software program?

Assessing the cost of a proposed software program has always been difficult, but can be accomplished by starting one or more set of project teams at a modest budget (1-6 sets of teams) and then adjusting the scaling of additional teams (and therefore the budget) based on the value those teams provide to the end user. It may be necessary to identify the size of the initial team required to deliver the desired functions at a reasonable pace and then price the program as the number of teams scales up. The DIB recommends that program managers start small, iterate quickly, and terminate early. The supervisors of program managers (e.g., PEOs) should also reward aggressive early action to shift away from efforts that are not panning out into new initiatives that are likely to deliver higher value. Justifying a small budget and getting something delivered quickly is the best way to provide value (and the easiest way to get and stay funded).

The primary metric for an *ongoing* program should be user satisfaction and operational impact. This can be different for every program and heavily depends on the context. The challenge, and therefore the responsibility of the PM then is to define mission relevant metrics to determine achieved and delivered value. Examples could include, personnel hours saved, number of objects tracked or targeted, accuracy of the targeting solution, time to first viable targeting solution, number of sorties generated per time increment, number of ISR sensors integrated, etc. Other key metrics that are often advocated by agile programs (inside and outside of DoD) include:

- *deployment frequency* (Is the program getting increments of functionality out into operations?),
- *lead time* (how quickly can the program get code into operation?),
- *mean time to recover* (how quickly can the program roll back to a working version, if problems are found in operation?), and
- *change fail rate* (rate of failures in delivered code).

These four break down into two process metrics (release cadence and time from code-commit to release candidate, and two are quality metrics (change fail rate and time to roll back). In addition, each project should also have 3-5 key value metrics that are topical to the solution space being addressed. Metrics must be available both to the teams and the customer so they can see how their progress compares to the projected completion rate for delivering useful functionality. A key reason for Government access to those metrics is for supporting the real-time tracking of progress and prediction of new activities in the future. The biggest difference between a DevSecOps

[19] Average of 8 people per team with an average of 8 teams per project.

program and the classic spiral approach is that the cadence of information transparency between the developers and the customer is, at slowest, weekly, but if properly automated, should be instantly and continuously available.. Quality metrics and discovery timelines (such as defects identified early in development versus bugs identified in the field) can also be used to evaluate the maturity of a program. This kind of oversight enables fast and effective feedback before the teams end up in extremis, or set up unrealistic expectations.

Software projects should be thought of as a fixed cadence of useful capability delivery where the "backlog" of activities are managed to fit the "velocity" of development teams as they respond to evolving user needs. Data collected on developers inside of the software development infrastructure can be provided continuously, instead of packaged into deliverables that cannot be directly analyzed for concerns and risks.

The DIB's "Metrics for Software Development" provide a set of metrics for monitoring performance:

1. Time from program launch to deployment of simplest useful functionality.
2. Time to field high priority functions (spec → ops) or fix newly found security holes
3. Time from code committed to code in use
4. Time required for regression tests (automated) and cybersecurity audit/penetration tests
5. Time required to restore service after outage
6. Automated test coverage of specs/code
7. Number of bugs caught in testing vs field use
8. Change failure rate (rollback deployed code)
9. Percentage of code available to DOD for inspection/rebuild
10. Complexity metrics
11. Development plan/environment metrics

These data provide management flexibility since data about implementation of capability can be made *during* development—instead of at a major milestone review or after "final" delivery, when changing direction comes at a much higher cost and schedule impact. So data collection and delivery must be continuous as well. Another note, these metrics are recommendations and not intended to be prescriptive. Use what fits your program. Not all of these may be required.

An additional pair of overarching key metrics are headcount and expert talent available. If the project headcount is growing, but delays are increasing,, aggressive management attention is called for. The lack of expert talent also increases risks of failure.

How can a program defend its budget if the requirements are not fixed years in advance, or are constantly changing?
It is relatively easy to defend changing capability by making changes to the software of existing systems, as compared to starting up a new acquisition. Software must evolve with the evolving needs of the customers. This is often the most cost effective and rapid way to respond to new requirements and a changing threat landscape. A new approach to funding the natural activities of continuous engineering and DevSecOps requires a system that can prioritize new features and manage these activities as dependent and tightly aligned in time

(see Figure 1). A continuous deployment approach is needed for delivering on the evolving needs culled from user involvement combining R&D, O&M, Procurement, and Sustainment actions within weeks of each other, not years (see Figure 2). Great software development is an iterative process between developers and users that see the results of the interaction in new capability that is rapidly put in their hands for operational use.

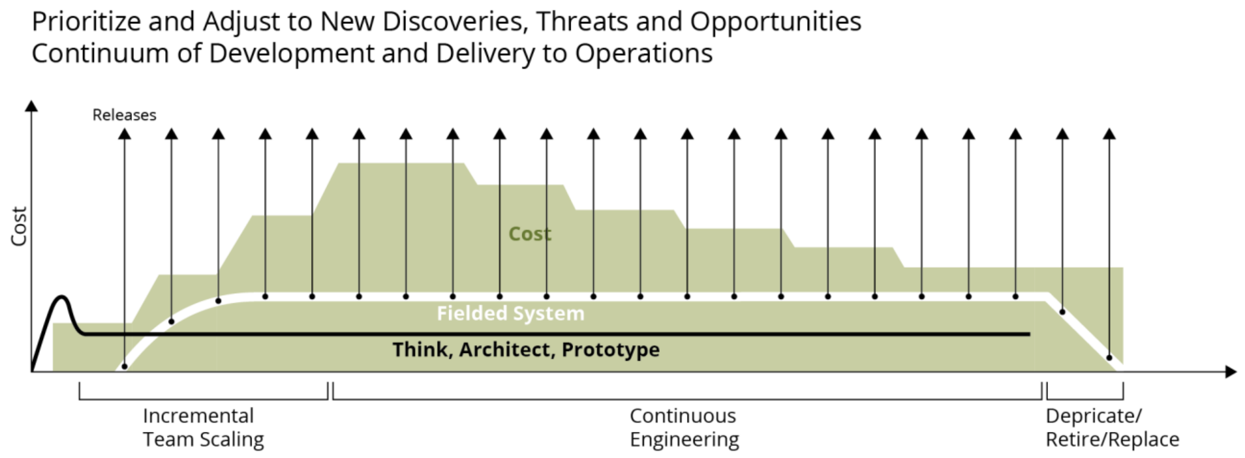

Prioritize and Adjust to New Discoveries, Threats and Opportunities
Continuum of Development and Delivery to Operations
Releases
Cost
Cost
Fielded System
Think, Architect, Prototype
Incremental Team Scaling
Continuous Engineering
Depricate/ Retire/Replace

Figure 2. Continuous Delivery of Modular Changes to Working Software (Carnegie Mellon University, Software Engineering Institute).

Elements to address include in budget justification and management materials:

- DevSecOps programs have to be at least as valuable and urgent to fund as a classic DoD spiral program in the hyper-competitive budget environment. Over time, DoD will realize that the DevSecOps approach is inherently more valuable. However, time is of the essence. It must be acknowledged that the current waterfall approach is no longer serving us well in the area of software. The mainstream software industry has already made the move to agile ten years ago and the methods are rigorously practices and proven valuable.
- The classic approach of doing cost estimates of designs based on fixed requirements has always been wrong, even when accounting for intended capability growth because the smart adversaries get a continuous vote on the threat environment. Accurate prediction of a rapidly changing technology environment and solution methods only exacerbate the unknowns of product development outcomes.
- DevSecOps programs have requirements, but start out at a higher level and use a disciplined approach to continuously change and deliver greater value.
- DIB's "Ten Commandments of Software" calls for the use of shared infrastructure and continuous delivery, which will reduce the cost of infrastructure and overhead, thus freeing up capital to advance unique military capability.
- Data available above the program manager's level has been insufficient for cost and program evaluation communities to assess software projects. However, the reporting of metrics that are a natural consequence of using DevSecOps approaches should be automated to provide transparency and rapid feedback.

The benefits of this approach are manifold. It allows for thoughtful rigor up front and early and the rapid abandonment of marginal or failure-prone approaches early in the design cycle before large investments are sunk. Details are allowed to evolve. More stable chunks of capability are defined at the "epic" level and a stable cadence of engineering and design pervades the life cycle. Under this operational concept, testing is performed early, during the architecture definition stage and continuously as new small deployments of functionality are delivered to the user. The identification of budget is redistributed as value is provided and validated for warfighting impact. A closer alignment of flexible requirements and budget allocation/ appropriation will be necessary in order to ensure that the national defense needs and financial constraints are continuously managed.

Continuous access to design and delivery metrics will illuminate developer effectiveness, user delight, and the pace of delivery for working code to include analytical data for in-stride oversight and user/programmatic involvement This will replace the standard practice of document-based deliverables and time-late data packages that take months to develop and are not current when provided.

The way that DoD has classically managed these activities is to break them up into different "colors of money" associated with hardware-centric phases (see Figure 3). This places an artificial burden on excellence in software. Rapid and continuous delivery of working code requires addressing these different types of requirements within shorter time-horizons than is natural for the existing federal budgeting process.

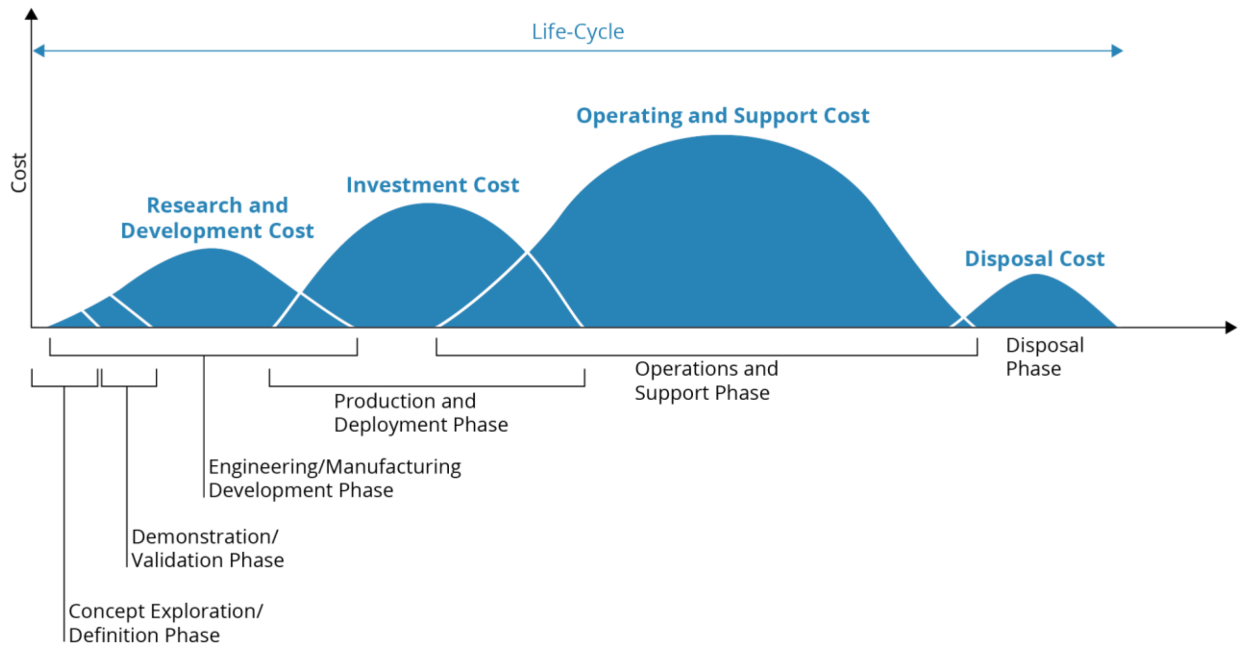

Life-Cycle
Cost
Operating and Support Cost
Investment Cost
Research and Development Cost
Disposal Cost
Disposal Phase
Operations and Support Phase
Production and Deployment Phase
Engineering/Manufacturing Development Phase
Demonstration/ Validation Phase
Concept Exploration/ Definition Phase

Figure 3. Notional DoD Weapon System Cost Profile (Defense Acquisition University).

In addition, the classic approach of developing detailed technical requirements far in advance of performing product design needs to be replaced. The new paradigm must begin with an architecture that will support the requirements and scale associated with needs for future compatibility (e.g., modularity security, or interoperability). Also, using an agile approach, a program can incorporate the best available technologies and methods throughout the entire life cycle and avoid a development cycle is longer than the useful life of the technology it is built on. Getting these things wrong is not recoverable. Establishing detailed requirements over a period of years before beginning, to be followed by long development efforts punctuated by major design reviews (i.e., Software Requirements Review, Preliminary Design Review, Critical Design Review, Test Readiness Review, Production Readiness Review) that require a span of years between events are inherently problematic for software projects for at least two reasons. First, these review events are designed around hardware development spirals that are time-late and provide little in the way of in-stride knowledge of software coding activities that can be used to aid in real-time decision making. Second, development teams are in frequent contact with users and adjusting requirements as they go, which up-ends the value of major design reviews that are out of cadence with the development teams. DevSecOps implementation methods such as feature demonstrations and cycle planning events provide much more frequent and valuable information on which program offices can engage to make sure the best value is being created.

Defending a budget has to be done in terms of providing value. Different programs value different things—increasing performance, reducing cost, minimizing the number of humans-in-the- loop—so there is no one size fits all measure. But in an agile environment, knowing what to measure to show value is possible because of the tight connection to the user/warfighter. Those users are able to see the value they need because they are able to evaluate and have an impact on the working software. This highlights the need to collect and share the measures that show improvement against a baseline in smaller increments.

How do we do cost for "sustainment" when we are adding new features?

The first step is to eliminate the concept of sustaining a fixed base of performance. Software can no longer be thought of as a fixed hardware product like a radar, a bomb, or a tank. That leads to orphaned deployments that need unique sustainment and a growth of spending that does not deliver new functionality (see Figure 4).

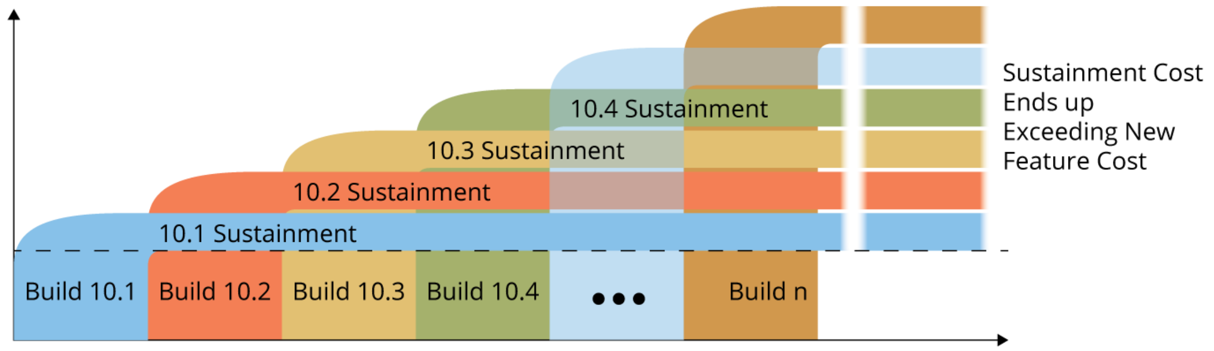


Figure 4. Layers of Sustainment to Manage Unique Deployments

Software can continue to evolve and be redeployed for comparatively little cost (see Figure 2). Users continue to need and demand greater performance and improved features, if for no other reason than to retain parity with warfighting threats. Also internal vulnerabilities and environmental updates must be continuously deployed to support ever improving cyber protections. The most secure software is the one that is most recently updated. Lastly, new capabilities for improved warfighting advantage are most often affordably delivered through changes to fielded products.

Software development is a very different way of delivering military capability. It should be considered more like a service of evolving performance. When new features are needed, they get put in the backlog, prioritized, and scheduled for a release cycle (see Figure 5). If the program is closer to providing satisfactory overall performance, then the program can dial down to the minimum level needed to satisfy the users and keep the environment and applications cyber-secure. It can be thought of as recursive decisions on how many (software) “squadrons” are required for our current mission set and then fund those teams at the needed staffing level to create, scale, or optimize the software (depending on the stage of continuous development). Because these patterns can be scaled up and down by need in a well-orchestrated way, new contracting models are available that might not have been used in the past. For example, fixed price contracts for a development program was strongly discouraged, but under this model, where schedule and team sizes are managed and capability is grown according to a rigorous plan (Figure 1), a wider array of business, contracting and remuneration models can be explored.

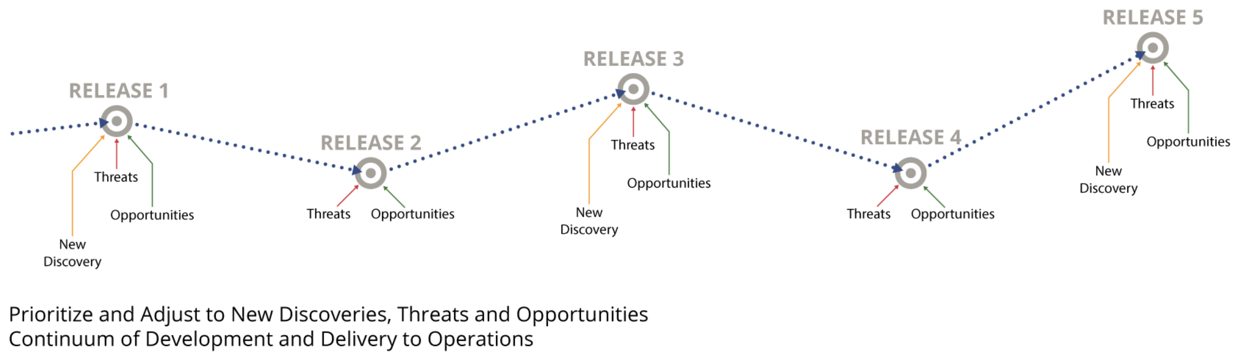


Figure 5. Release Cycle With New Opportunities, Discoveries and Response to Threats (Carnegie Mellon University, Software Engineering Institute).

Two financial protections built into acquisition laws and regulations need to be reexamined in the light of software being continuously engineered, vice sustained: Nunn-McCurdy and the Anti-Deficiency Act. The continuous engineering pipeline will continue to push out improved capability until the code base is retired. While Nunn-McCurdy is a valid constraint for large hardware acquisitions, it does not apply to software efforts. In a similar vein, software should also never trigger the Anti-Deficiency Act - just like keeping a ship full of fuel, or paying for air-traffic controllers; we know we are going to be doing these things for a long time. To build a ship that will need fuel for 40 years does not invoke the ADA. Therefore, starting a software project that will incrementally deliver new functionality for the foreseeable future should not do so either.

Why is ESLOC a bad metric to use for cost assessment?

The thing we really want to estimate and then measure is the effort required to develop, integrate, and test the warfighting capability that is delivered by software. SLOC might have been a used as a surrogate for estimating the effort required, but it has never been accurate. Not all software is the same, not all developers are the same, and not all development challenges use the same approaches to reduce problems into solutions. For example, in a project there may things like detailed algorithms that require deep expertise and detailed study to properly implement small amounts of code, running alongside large volumes of automatically generated code of relatively trivial complexity. Many different levels of effort are needed to create a line of code that will deliver military capability, and estimations of source code volume is an inherently problematic and error-filled approach to describing the capability thus produced. That's why DevSecOps efforts use measures of relative effort like story points to communicate across a particular set of teams how much effort it will take to turn a requirement into working software that meets an agreed upon definition of done within a set cadence of activity. Because these story points are particular to a specific team, they do not accurately transition to generally prescribable measures of cost.

Estimating by projecting the lines of code starts the effort from the end and works backwards. SLOC is an output metric (something to know when the job is done—akin to predicting what size clothing your child will wear as an adult). It does not capture the human scale of effort. Traditional models like COCOMO or SEER attempt to use a variety of parameters in their models to capture things like formality, volatility, team capabilities, maturity and others. However, these surrogates for effort have well documented error sources and have failed time and again to accurately capture the cost of executing a software program. There are also inherent assumptions built into these models that are obviated by performing agile development of capability models running on a software platform.

In the beginning stages of DoD's transformation to DevSecOps methods, the development and operations community will need to work closely with the cost community to derive new ways of predicting how fast capability can be achieved. For example, estimating how many teams worth of effort will be needed to invest in a given period of time to get the functionality needed. As they do this, it needs to be with the understanding that the methods are constantly changing and the estimation methods will have to evolve too. New parameters are needed, and more will be discovered and evolve over time.