Defense Innovation Board Metrics for Software Development

Version 0.9, last modified 9 Jul 2018

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, can 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)[i]				Typical DoD values for SW
		COTS[ii] apps	**Custom -ized SW[iii]**	**COTS HW/OS[iv]**	**Real-time HW/SW[v]**	
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)[vi]	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[vii]	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 available to DoD for inspection/rebuild	N/A	100%	100%	100%	0%
10	Complexity metrics	#/type of specs structure of code #/type of platforms			# programmers #/skill level of teams #/type deployments	Partial/ manual tracking
11	Development plan/environment metrics					
12	"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.[viii] 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.

#	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[ix]	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[x]	?

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	Custom-ized 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%

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.[xi] 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.

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.

<table>
<tr><th rowspan="2">#</th><th rowspan="2">Metric</th><th colspan="4">Target value (by software type)</th><th rowspan="2">Typical DoD values for SW</th></tr>
<tr><th>COTS apps</th><th>Custom-ized SW</th><th>COTS HW/OS</th><th>Real-time HW/SW</th></tr>
<tr><td>10</td><td>Complexity metrics</td><td colspan="4" rowspan="2">#/type of specs # programmers
structure of code #/skill level of teams
#/type of platforms #/type deployments</td><td rowspan="2">Partial/ manual tracking</td></tr>
<tr><td>11</td><td>Development plan/environment metrics</td></tr>
<tr><td>12</td><td>"Nunn-McCurdy" threshold (for any metric)</td><td>1.1X</td><td>1.25X</td><td>1.5X</td><td>1.5X each effort</td><td>1.25X Total $</td></tr>
</table>

Background:

10. 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 lifecycle 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.

11. 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

12. 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. 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 efforts 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.

---

[i] Target values are notional; different types of software (SW) as defined in DIB Ten Commandments.
[ii] *Commercial-Off-the-Shelf* (COTS)
[iii] *Software* (SW)
[iv] *Hardware/Operating System* (HW/OS)
[v] *Hardware/Software* (HW/SW)
[vi] "*Fcn*" is short for "function"; "*spec*" is short for "specifications", "*ops*" is short for "operations"
[vii] "*Req'd*" is short for "required"; "*automat'd*" is short for "automated"
[viii] 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.
[ix] 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.
[x] 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.
[xi] 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).