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.

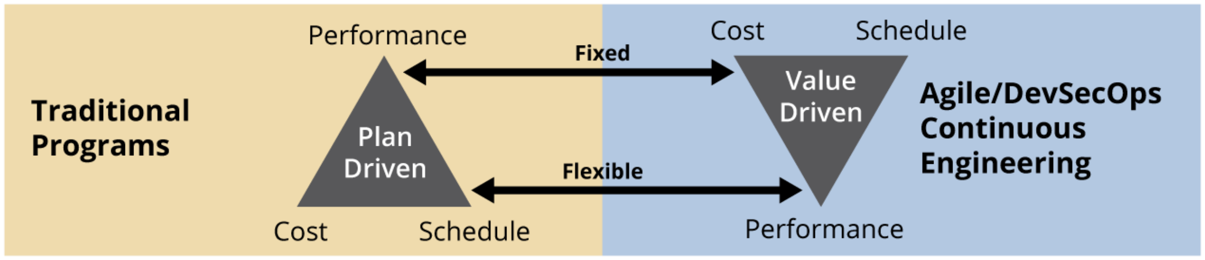

Figure 1. Value Driven Iron Triangle (Carnegie Mellon University, Software Engineering Institute).

- 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.

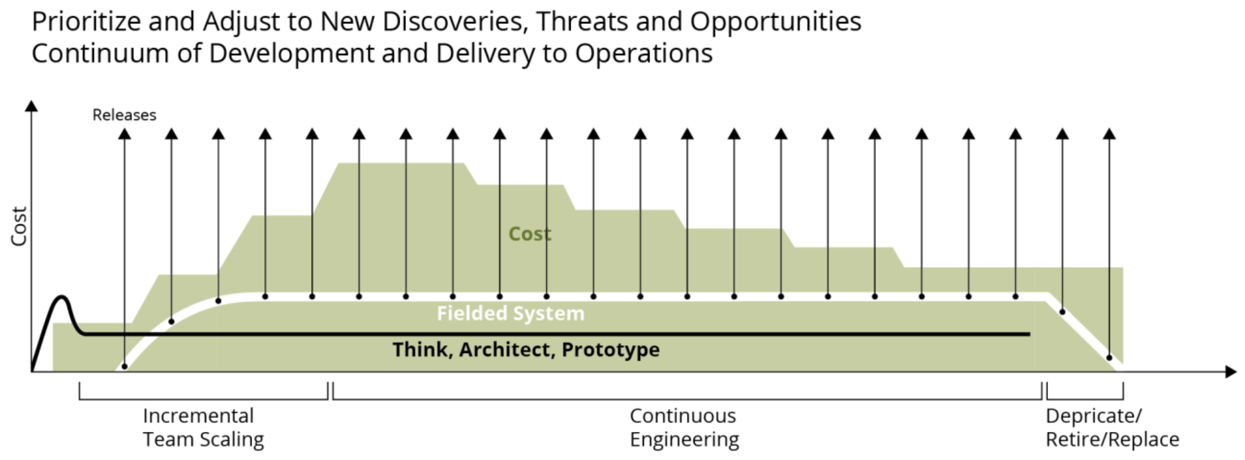
Prioritize and Adjust to New Discoveries, Threats and Opportunities
Continuum of Development and Delivery to Operations
Releases
Cost
Cost
Fielded System
Think, Architect, Prototype
Incremental Team Scaling
Continuous Engineering
Depricate/ Retire/Replace

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.

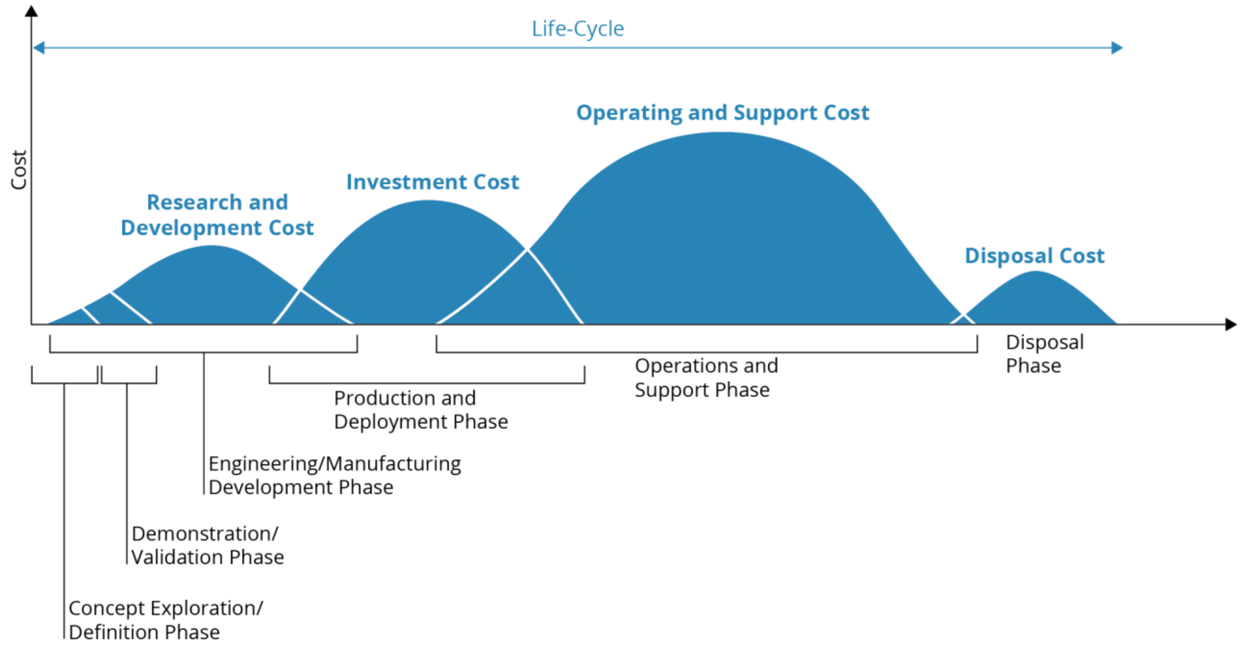
Life-Cycle
Cost
Operating and Support Cost
Investment Cost
Research and Development Cost
Disposal Cost
Disposal Phase
Operations and Support Phase
Production and Deployment Phase
Engineering/Manufacturing Development Phase
Demonstration/ Validation Phase
Concept Exploration/ Definition Phase

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).

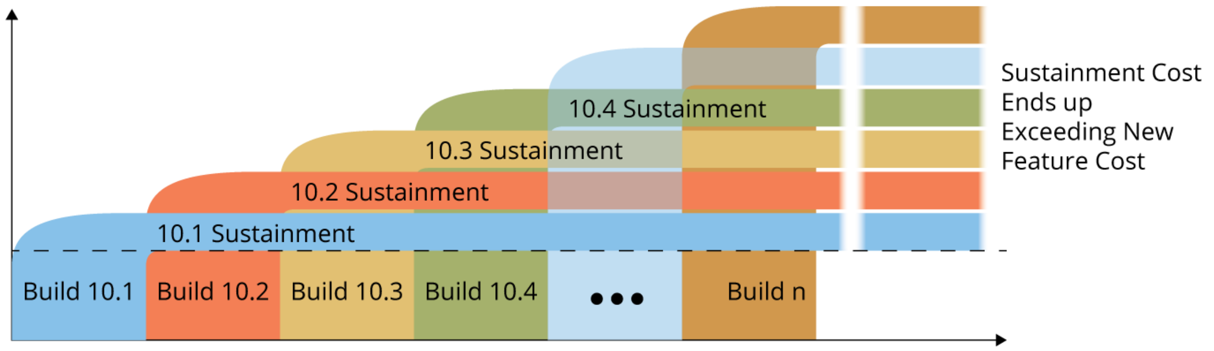

Figure 4. Layers of Sustainment to Manage Unique Deployments

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.

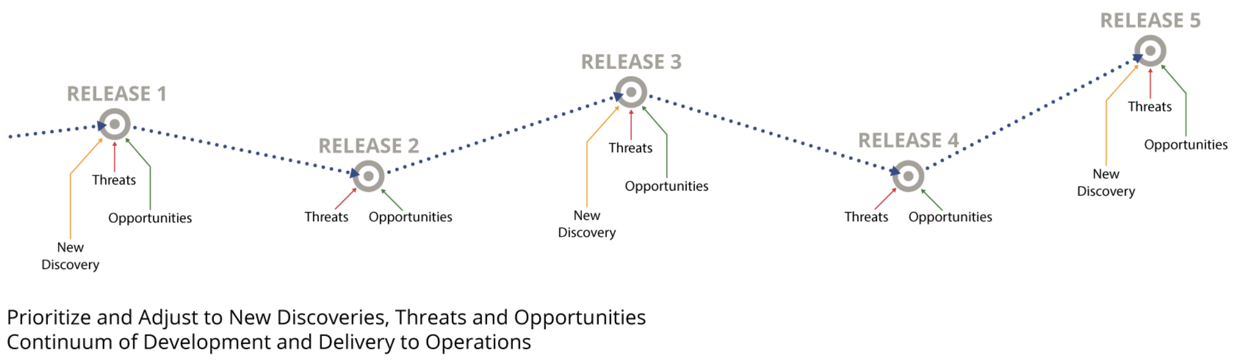

Figure 5. Release Cycle With New Opportunities, Discoveries and Response to Threats (Carnegie Mellon University, Software Engineering Institute).

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.