SYM-AM-20-032

PROCEEDINGS
OF THE
SEVENTEENTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

Acquisition Research: Creating Synergy for Informed Change

May 13–14, 2020

Published: March 30, 2020

Approved for public release; distribution is unlimited.

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

Disclaimer: The views represented in this report are those of the author and do not reflect the official policy position of the Navy, the Department of Defense, or the federal government.

The research presented in this report was supported by the Acquisition Research Program of the Graduate School of Defense Management at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

Factors Limiting the Speed of Software Acquisition

David Tate – joined the research staff of the Institute for Defense Analyses' (IDA) Cost Analysis and Research Division in 2000. Prior to that, he was an Assistant Professor of industrial engineering at the University of Pittsburgh and the Senior Operations Research Analyst (Telecom) for Decision-Science Applications, Inc. At IDA, he has worked on a wide variety of resource analysis and quantitative modeling projects related to national security. These include an independent cost estimate of Future Combat Systems development costs, investigation of apparent inequities in Veterans' Disability Benefit adjudications, and modeling and optimization of resource-constrained acquisition portfolios. Dr. Tate holds bachelor's degrees in philosophy and mathematical sciences from the Johns Hopkins University and MS and PhD degrees in operations research from Cornell University.

John Bailey – has worked for both the former Computer Science Division and the Cost Analysis and Research Division of IDA since the 1980s. Prior to that, he conducted software programming research for General Electric and co-founded Software Metrics, Inc. While completing his PhD in software reuse methodology, he helped Rational establish its Ada development environment, which was later acquired by IBM. Dr. Bailey is now applying computer science best practices to a 70-acre cattle farm in northern Virginia.

Abstract

The factors limiting the speed of software acquisition and enhancement fall into these principal categories, in rough order of importance:

1. Required functionality – what you need the software to do (and not do)
2. Architecture – the organizing structure of the software and its operating environment
3. Technology maturity – to what extent the intended design uses novel solutions
4. Resources – the people, skills, funds, data, and infrastructure needed to do the work
5. Testing strategy – acquiring the information to fix defects early in development
6. Contract structure – the alignment of contractor incentives with DoD satisfaction
7. Change management – the processes for trading-off performance, schedule, cost, and sustainability

This paper considers this taxonomy and examines how each category affects the pace of development.

Introduction

Improving the agility of defense acquisition is a high priority goal for both the Office of the Secretary of Defense (OSD) and the military departments. Improving the speed at which the Department of Defense (DoD) can develop, deploy, and update software-enabled capabilities would enable more general acquisition agility, given modern defense systems' critical dependence on software. This point was emphasized in both a 2018 Defense Science Board report and in the 2019 Defense Innovation Board (DIB) Software Acquisition and Practices (SWAP) study.

There is now widespread consensus that software development is the pacing activity not only in traditional information systems acquisition, but in all major defense systems acquisition. As early as 20 years ago, the Naval Postgraduate School was already teaching students that "software is now acknowledged as sitting on the critical path of most major weapon systems and is widely regarded as the highest-risk element in an acquisition" (Nissen, 2019). Other recent research has confirmed this insight (Tate, 2016). If we want our acquisition enterprise to deliver capabilities more quickly and upgrade them more frequently, we need to speed up defense software development.

Given this universal interest in accelerating software acquisition and sustainment, it is important to understand the fundamental factors that limit how quickly software can be developed, deployed, and upgraded. These factors fall into seven principal categories; in rough order of importance, they are:

1. Required functionality – what you need the software to do (and not do)
2. Architecture – the organizing structure of the software and its operating environment
3. Technology maturity – to what extent the intended design uses novel solutions
4. Resources – the people, skills, funds, data, and infrastructure needed to do the work
5. Testing strategy – acquiring the information to fix defects early in development
6. Contract structure – the alignment of contractor incentives with DoD satisfaction
7. Change management – the processes for trading-off performance, schedule, cost, and sustainability

This paper considers each category in this taxonomy in turn and examines how each affects the pace of both initial development and future upgrades.

Required Functionality

The time to develop and field a software-enabled capability is primarily determined by what the software needs to do and in what context—the *content* of the software. This seems obvious and has been common knowledge with regard to the cost of systems for decades, but it is still common for schedules to be imposed on software development efforts independent of their content.

Although content is often thought of in terms of requirements, it is important to note that, historically, not all mandatory performance attributes have been managed in the same way. In particular, “negative requirements” that describe outcomes the system should avoid are often handled outside the primary requirements management process once development begins. They are thus at risk of being neglected as the design evolves or of being waived or relaxed when programs face time pressure. Negative requirements include cybersecurity specifications, safety standards, reliability/availability/maintainability (RAM) thresholds, interoperability requirements, and other aspects of system suitability.

The minimum time to deploy the first version of a new software-enabled system is driven by the content of the *minimum viable product* (MVP), which the DIB defines as “the first point at which the code can start doing useful work and also at which feedback can be gathered [from users] that supports refinement of features” (Tate, 2016). What counts as “useful work” is a stakeholder decision that involves trade-offs between lead time and capability—discussed further in the “Change Management” section. Note that a system that is not secure or cannot work with real data cannot be the MVP—it is not yet useful for any actual operational mission. Minimum standards for the negative requirements must be included in the definition of the MVP.

For any software project, there is a theoretically ideal staffing profile that would achieve the most efficient compromise between maximizing work rate and incurring unnecessary rework. This profile would allow the team to take maximum advantage of parallel efforts on separable tasks while avoiding concurrent effort on coupled tasks. Trying to go faster than this ideal schedule often leads to significant redesign, rework, and delay (if not cancellation). Going slower than the ideal schedule (e.g., due to resource constraints) results in unnecessary delay and inefficient use of labor resources.

The length of the ideal schedule depends on how tightly coupled the software subsystems (modules) are. Software designs that do not explicitly enforce independence of subtasks can lead to systems that are not modular enough to support full parallelization of development efforts, resulting in longer development schedules for future increments. To implement a fixed set of content, agile/lean methods (where feasible) are (at best) modestly faster than traditional waterfall development. The time savings realized by agile development come mostly from not implementing functions that turn out not to be critical—the savings are due to work avoided, not higher productivity.

Architecture

As noted above, software and hardware architectures for platforms and systems strongly influence how rapidly and effectively new capabilities can be added. Therefore, the selection of an architecture for a new defense system has important consequences throughout the acquisition process. For major defense systems, achieving an MVP that is structured to support decades of ongoing agile improvements can require years of initial architecture, design, and development effort.

Every software-enabled system requires some architecture and design prior to the beginning of coding. The amount of effort needed for an architecture depends on a number of factors:

- How important is it for the system to be upgradeable in the future?
- How fast/cheap/frequent/extensive do those upgrades need to be?
- What is the intended life span of the platform or system and its upgrades?
- How portable/reusable does the software need to be?
- What other systems will the new system need to interoperate with?
- What are the cybersecurity needs of the system?
- How much is the cyber threat environment expected to change over time?
- How close to the cutting edge of current technical capabilities is the system intended to operate?

The answers to these questions help determine the appropriate software architecture for the new system, which (combined with hardware architecture) in turn determines the required cost and schedule to implement the MVP. Systems that will be easy and cheap to modify over a long life cycle require more architecture effort (and thus more content in the MVP and a longer initial development increment). In particular, the more you want to use agile/lean/DevSecOps methods for ongoing parallel system improvement of large systems or to compete upgrades among multiple vendors, the more you need a modular architecture.

The architecture is part of the MVP content. An MVP with an architecture that does not support long-term operational goals is really just a prototype. It is generally not possible to convert a prototype into a full-up operational system through successive upgrades; the result is unsuitable and unsustainable. In particular, systems with significant cybersecurity concerns must address those requirements in the architecture and design before coding begins. Adding security later does not work. For some mission needs, a quick-and-dirty architecture that supports current (but not future) requirements might be appropriate, providing some needed capability quickly at the expense of future sustainment and upgrades. This choice should be a conscious decision at design time, understood by stakeholders.

Similarly, when adding new capabilities to a legacy system whose architecture was not originally designed to support ongoing upgrades, stakeholders need to decide whether it is more important to implement specific new capabilities quickly or to make the system more rapidly upgradeable in the future. In some cases, it may be preferable to accept cost and delay today to re-architect and rewrite the legacy code to enable a longer and more agile future upgrade path.

Technology Maturity

Whenever system designs involve technologies that are less than fully mature, the software and hardware design processes contain an element of experimentation and exploration. This can lead to a redesign that changes the nature of the software required or one that shifts functions originally envisioned as being performed by hardware into software. This adds both new development and regression testing burdens to the software project and lies on the critical path for the program. As a result, technologies with a current Technology Readiness Level (TRL) less than 6 (as defined in Title 10 U.S. Code, § 2366b(a)(3)(D)) can contribute to software delays.

At the same time, critical software technologies tend to be overlooked in Independent Technology Readiness Assessments. Past examples where this has led to major program delays include sensor fusion algorithms, ad hoc network management, and automated traction control for off-road vehicles.[1]

The DoD has recently stated in multiple venues that machine learning, artificial intelligence, and autonomous systems will be key enablers of future U.S. military capabilities. These are immature technologies that impose novel burdens throughout the acquisition life cycle, including learning how to specify requirements that are testable, how to validate and maintain training data sets for machine learning, and how to assess the effectiveness and suitability of machine cognition and human-machine teaming concepts (Tate & Sparrow, 2018).

It is important to remember that, for TRLs greater than 4, technology maturity is always measured relative to a specific set of requirements and intended operational environments.[2] A technology can be mature (i.e., TRL 6 or higher) for certain operational uses or environments, yet immature for others, as demonstrated by the traction control example. This is just as true of software technologies as it is for lasers, engines, or sensors. Failure to recognize that a given software technology is critical to success or that it has not yet been successfully demonstrated in the intended operational role and environment can lead to costly delays in development down the road. (For an annotated list of TRL definitions, see *Technology Readiness Levels and their Definitions*, Deskbook 5000.2-R, Appendix 6 [DoD, n.d.].)

Resources

Software development is about enough skilled labor, appropriate infrastructure, and (increasingly) adequate data. As a result, the most important potential resource issues that software projects can face are a shortage of appropriately skilled personnel (through either insufficient funding to purchase their labor or lack of access to the right talent) or insufficient

[1] Traction control algorithms used in commercial vehicles assume that the vehicle is driving on a hard surface. They do not perform well in off-road situations where the loss of traction is due to surface failure (e.g., mud or sand) rather than low friction with the road.

[2] TRL 4 requires “Component and/or breadboard validation in a laboratory environment,” which is independent of requirements or intended operational environments. TRL 5 requires “Component and/or breadboard validation in a relevant environment,” which explicitly compares the environments where the technology has already been demonstrated against some intended environment.

access to platforms or data sets needed to implement the desired capabilities. These challenges are time-phased—enough of each of these things is needed at the right times to support the ideal schedule of the project. Valid input data is often needed early in development. For machine learning, obtaining a sufficient quantity of accurately labeled training data can be a bottleneck in capability development.

An adequate supply of appropriately skilled developers is not a given. The supply of cleared software professionals is not keeping up with the demand for national security software (Tate, 2017). As a result, programs often have trouble hiring and keeping the quality and quantity of software talent for which they had planned. Staffing shortfalls cause delays, and project staff turnover and loss of institutional knowledge cause additional delays. Mergers and acquisitions, common in defense industries, sometimes exacerbate this problem. Either not having enough people or not having the best people as part of an experienced team can cause program delays.

Similarly, funding instability or mismatches between planned spending profiles and actual workloads can lead to inefficient use of labor resources and subsequent schedule delays. This is particularly true for projects with significant experimentation and discovery during the development of the MVP, as the duration and cost of the experimentation cannot be predicted with accuracy. Such projects should ideally be identified in advance and provided with significant levels of contingency funding, but this is politically difficult in many cases.

For many software-enabled defense systems, the direct costs of software development are a relatively small fraction of total program costs. However, as noted above, software delays tend to be on the critical path, resulting in delays that add cost at a roughly constant burn rate. For major programs, this represents not merely a delay in achieving the desired capabilities but also a large opportunity cost, wasting funds that could have been used elsewhere.

It is important to note that the majority of software development effort (and cost)—often as much as 80% of total effort—occurs after the system has initially been fielded. Software deployment generates an ongoing future workload that lasts as long as the system is in use. The resources needed to perform future upgrades and integration are generally the same resources needed for initial development. This means that future enhancements compete with new systems for people, as well as for dollars.

In addition to people, many advanced software capabilities also require specialized infrastructure and data. These can include supercomputing resources, labeled training data sets, modeling and simulation support, and test ranges with specialized instrumentation. Shortfalls in any of these areas can also lead to significant fielding delays.

Testing Strategy

Although it is theoretically possible to do “too much testing” in a software-intensive development program, this is nearly unknown in practice. Finding and fixing defects are the most expensive identifiable software cost drivers, and high defect levels are the primary reason for software schedule and cost overruns (Jones & Bonsignour, 2012). Software development success is strongly correlated with a commitment to continuous, rigorous testing from day one, identifying defects early—when they are relatively easy to isolate and correct.

While it is hard to do too much testing, it is easy to do testing wrong. In order to realize the benefits of continuous testing and early defect elimination, the test strategy for the program must ensure that the right information is collected at the right times and fed back into the development process efficiently. Unaddressed defects have cascading effects—they interfere with development of additional capabilities, mask other defects, and complicate the required fixes. Past research has found that the average cost to remedy a defect grows by an order of magnitude between the design phase and the implementation phase and grows by another order of magnitude between implementation and post-deployment maintenance (Kan, 2003). Bear in mind here that "defects" can include flaws in requirements, documentation, adherence to architectural specifications and standards, or other issues that are not necessarily "bugs" in the code.

The most common (and costly) testing error by far is to not do it early enough. For example, the Warfighter Information Network—Tactical (WIN-T) underwent an initial operational test and evaluation (IOT&E) in May 2012, with planned full-rate production (FRP) scheduled for September 2012. IOT&E revealed numerous problems that should have been caught much earlier in testing, including poor network stability, poor performance, and poor reliability. FRP was deferred until after a follow-on operational test and evaluation (FOT&E) could be conducted. That FOT&E revealed continuing problems, leading to another schedule slip. The system finally entered FRP in June 2015.

When testing is done right, test activities make the project cheaper and faster than it otherwise would have been. Delays "due to testing" are almost always actually delays due to not testing—that is, extra time required to fix defects that were discovered late in development (when testing is expensive) or after fielding (when testing is very expensive) because of insufficient early testing. Finding ways to incentivize contractors (and program offices) to perform adequate testing is challenging.

For new systems or for modification of legacy systems that are intended to transition to agile development for future increments, it is important to develop the associated agile test strategies and automated testing infrastructure during MVP development. This should involve all stakeholders and be done in parallel with (and be informed by) MVP design and implementation.

Contract Structure

Typically, a small percentage of the overall profit on a major defense system comes from software development. At the same time, the future upgrade and maintenance contracts for a software-intensive system can provide an open-ended source of future revenue. This is analogous to losing money on each laser printer sold but making big profits on the toner cartridges. This gives contractors a strong incentive to erect barriers to sustainment competition that all too often also function as barriers to efficient upgrades and modernization.

The DoD would prefer that all software be modular, reusable, and open to enhancement by third-party vendors. It is very difficult to write a contract that incentivizes (or requires) the prime contractor to make that happen. Modular design makes it easy for competitors to break the prime's monopoly on sustainment; reusable code makes it possible for competitors to benefit from the prime's work and potentially to gain access to their proprietary technologies. For these reasons, contractors prefer to restrict government data rights as much as they can. As Van Atta, Kneece, Lippitz, and Patterson (2017) note, "There is a vast legacy of defense systems, amounting to billions of dollars in sustainment costs, for which the necessary IP data and rights for organic depot or competitive sustainment were not acquired."

The services are sometimes complicit here—for example, by waiving statutory requirements for modular open systems architecture (MOSA) on the grounds that requiring MOSA would delay initial fielding and cost too much. Even if this were true, it is a false economy—the delays and costs of fielding subsequent upgraded capabilities often outweigh the original savings. As an example, the AWACS Block 40/45 upgrade program, intended to migrate the hardware and software architectures and applications on the E-3 AWACS aircraft from legacy proprietary systems to new open architecture hardware and software, has now been struggling for nearly 20 years to match the existing operational capabilities of the legacy Block 30/35 aircraft.

Admittedly, it is hard to write requirements for modularity, openness, and data rights that are verifiable at the time of system delivery and that actually make it likely that future upgrades will be easier and cheaper. This is similar to the problem of writing software quality requirements that are enforceable. Although quality requirements can sometimes be enforced through user participation in agile development teams, during development it is hard for users and other stakeholders to tell whether the choices being made will actually facilitate future upgrades as desired. Furthermore, current law prohibits making access to intellectual property rights a condition of contract award (although access to data rights can be an evaluation factor).

The DoD often struggles with aligning contractor incentives with service interests, so that the contractor profits more only when the service gets more of what it really needs (including agility). There is no free lunch—firms will require more compensation up front for initial system development if they cannot count on monopoly profits during sustainment. If the services are serious about wanting to speed up deployment of future capabilities, they must accept this and plan for it by developing contracting strategies that will allow them to realize the benefits of open system architectures.

Change Management

There is a widening disconnect in the defense acquisition world between the people who control system requirements and the people who actually develop and field systems. This is true of all requirements but especially true of software requirements (and requirements that end up being implemented in software). In theory, the process looks something like this:

1. Characterize the future fight and define mission needs.
2. Identify capability gaps and alternative ways to mitigate them.
3. Analyze the alternatives and select a preferred alternative.
4. Set threshold requirements.
5. Develop and field a system that meets those requirements.

In practice, there are problems with how this is realized. Translating future warfighting concepts into specific mission needs is hard, and translating mission needs into threshold performance requirements for individual systems is even harder. Thresholds are often set at aspirational levels, rather than at genuine threshold levels below which there would be no military utility. Changes during development concerning what to implement in hardware versus what to implement in software can lead to changes in system architecture that break other parts of the development or hinder future maintainability. Seemingly minor changes in the intended use of machine learning subsystems can require retraining from scratch or even development of entirely new training data sets.

As a result, some programs are unexecutable. Faced with unexecutable programs, program managers must make trades. The realities of annual funding make it easy to accept future costs and operational shortfalls (e.g., due to poor reliability or architecture violations) in order to save money and minimize delays in initial deployment. Similarly, deferring important testing is a common way to save time and money in the short run.

Worse yet, program managers generally do not have the authority to relax key requirements. They must appeal to a variety of senior stakeholders and achieve a consensus that relaxing the requirement is the best way forward for the program. This need for consensus also adds friction and delay to the process. Furthermore, once a program has funding authority, the services seldom step back and reconsider whether the system being developed is still the preferred alternative, given what they now know about costs and capabilities.

Agile software development methods depend on empowering a coalition of developers and users to decide what functionality is most useful on an iterative and ongoing basis. As noted above, the services have historically been unwilling to devolve that level of decision authority to development teams or to ensure embedded stakeholder support (including actual users) in program offices. This is especially true for systems that are not “pure” software systems—it is much easier to devolve authority for user interface design and database transactions than it is to devolve authority to change the operational specifications of weapon systems. Nevertheless, to realize the benefits of agile development, programs will need to have significantly more authority to prioritize and trade requirements, to enforce architectures, and to deny external change requests than is typically delegated to them.

Summary

Software development takes time. As with any other kind of project, how much time it takes depends on the content of the project, how it is managed, and the preexisting conditions:

- How much software is needed?
- How complicated or novel is the software to be developed?
- What is the software expected to do (today and tomorrow)?
- What resources are available for the work?
- What are the contractor’s incentives?
- What change management authorities does the development team have?
- Is this an initial development, or an upgrade to an existing system?
- If it is an initial development,
  - How easy to upgrade does it need to be?
  - What is the minimum viable product?
  - Have testable requirements been clearly specified?
  - Does the test strategy support early discovery and correction of defects?
- If it is an upgrade,
  - How well does the legacy architecture support insertion of new capability?
  - What data rights does the government own?
  - What institutional knowledge from the original development still exists?
  - Are agile development processes and tools in place?

For programs to field new software-enabled capabilities quickly, someone must have spent the time and money in the past to create an environment that supports rapid capability insertion. Ideally, this environment would include modular (and preferably open) software architecture, adequate data rights, platforms with excess space and power available, an industrial base that can provide enough people with the right skills, curated input and training data, developmental test infrastructure (including modeling and simulation resources where appropriate), and localized change management authority within the developer/stakeholder team.

Putting these enabling environmental features into place will often require accepting delay, up-front expense, and reduced capability in the initial increments of those platforms. Unless service leadership accepts this reality and empowers new system developers to preserve these features even when faced with cost overruns, schedule delays, and demands for greater capability up front, they will not be implemented. Absent this kind of empowerment, software capability insertion will continue to be as slow, expensive, and unreliable as it is today.

Acknowledgments

This work is an extension of research initially performed in support of the 2019 Defense Innovation Board (DIB) Software Acquisition and Policy (SWAP) Study.

References

Defense Innovation Board. (2019). *Software is never done: Refactoring the acquisition code for competitive advantage*. Retrieved from https://media.defense.gov/2019/Mar/26/2002105909/-1/-1/0/SWAP.REPORT_MAIN.BODY.3.21.19.PDF

Defense Science Board. (2018). *Design and acquisition of software for defense systems*. Retrieved from https://apps.dtic.mil/dtic/tr/fulltext/u2/1048883.pdf

DoD. (n.d.). *Technology Readiness Levels and their definitions* (Deskbook 5000.2-R, Appendix 6).

Jones, C., & Bonsignour, O. (2012). *The economics of software quality*. Boston, MA: Addison-Wesley.

Kan, S. H. (2003). *Metrics and models in software quality engineering*. Boston, MA: Addison-Wesley.

Major defense acquisition programs: Certification required before Milestone B approval,10 U.S.C. § 2366b(a)(3)(D).

Nissen, M. E. (1997). *JSOW Alpha Contracting case study*. Retrieved from https://fas.org/man/dod-101/sys/smart/docs/jsowcase.htm

Tate, D. M. (2016). *Acquisition cycle time: Defining the problem* (IDA Document NS D-5762). Alexandria, VA: Institute for Defense Analyses. Retrieved from https://www.ida.org/research-and-publications/publications/all/a/ac/acquisition-cycle-time-defining-the-problem-revised

Tate, D. M. (2017). *Software productivity trends and issues* (IDA Document NS D-8367). Alexandria, VA: Institute for Defense Analyses. Retrieved from https://www.ida.org/research-and-publications/publications/all/s/so/software-productivity-trends-and-issues-conference-paper

Tate, D. M., & Sparrow, D. A. (2018). *Acquisition challenges of autonomous systems* (IDA Document NS D-8982). Alexandria, VA: Institute for Defense Analyses. Retrieved from https://www.ida.org/research-and-publications/publications/all/a/ac/acquisition-challenges-of-autonomous-systems-conference-paper

Van Atta, R., Kneece, R., Lippitz, M., & Patterson, C. (2017). *Department of Defense access to intellectual property for weapon systems sustainment* (IDA Paper P-8266). Alexandria, VA: Institute for Defense Analyses.

Acquisition Research Program
Graduate School of Defense Management
Naval Postgraduate School
555 Dyer Road, Ingersoll Hall
Monterey, CA 93943

www.acquisitionresearch.net