Carnegie Mellon University
Software Engineering Institute

Report to the Congressional Defense Committees on National Defense Authorization Act (NDAA) for Fiscal Year 2022 Section 835 Independent Study on Technical Debt in Software-Intensive Systems

Ipek Ozkaya
Forrest Shull
Julie Cohen
Brigid O'Hearn

November 2023

TECHNICAL REPORT
CMU/SEI-2023-TR-003
DOI: 10.1184/R1/24043392

Software Solutions Division

https://www.sei.cmu.edu

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

This report was prepared for the SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100

DM23-1063

Table of Contents

List of Tables

Abstract

A team from Carnegie Mellon University's Software Engineering Institute (SEI) conducted an independent study to satisfy the requirements of the Fiscal Year 2022 National Defense Authorization Act (NDAA) Section 835, Independent Study on Technical Debt in Software-Intensive Systems.

This report describes the conduct of the study, summarizes the technical trends observed, and presents the resulting recommendations. The study methodology includes a literature review, a review of SEI reports developed for program stakeholders, deep dives on program data from SEI engagements with Department of Defense (DoD) programs, and interviews conducted using the 10 study elements specified in Section 835(b).

The study concludes that programs are aware of the importance of managing technical debt. Furthermore, a number of DoD programs have established practices to actively manage technical debt. During this study, the DoD published several guidance documents that begin to include technical debt and technical debt management as an essential practice for successful software development. Study recommendations include that the DoD must continue to update policy/guidance and empower programs to incorporate technical debt practices as part of their software development activities while enabling research in improved tool support and data collection.

1 Background

The Fiscal Year 2022 National Defense Authorization Act (NDAA) Section 835, Independent Study on Technical Debt in Software-Intensive Systems requires the secretary of defense to "enter into an agreement with a federally funded research and development center to study technical debt in software-intensive systems" [NDAA 2021]. Satisfying this requirement, Carnegie Mellon University's Software Engineering Institute (SEI), a recognized leader in the practice of managing technical debt, was asked to lead this work with a start date of May 1, 2022.

Per the Section 835(b) study elements [NDAA 2021], this study is designed to

> *include analyses and recommendations, including actionable and specific guidance and any recommendations for statutory or regulatory modifications, on the following:*
>
> *(1) Qualitative and quantitative measures which can be used to identify a desired future state for software-intensive systems.*
>
> *(2) Qualitative and quantitative measures that can be used to assess technical debt.*
>
> *(3) Policies for data access to identify and assess technical debt and best practices for software-intensive systems to make such data appropriately available for use.*
>
> *(4) Forms of technical debt which are suitable for objective or subjective analysis.*
>
> *(5) Current practices of Department of Defense software-intensive systems to track and use data related to technical debt.*
>
> *(6) Appropriate individuals or organizations that should be responsible for the identification and assessment of technical debt, including the organization responsible for independent assessments.*
>
> *(7) Scenarios, frequency, or program phases during which technical debt should be assessed.*
>
> *(8) Best practices to identify, assess, and monitor the accumulating costs technical debt.*
>
> *(9) Criteria to support decisions by appropriate officials on whether to incur, carry, or reduce technical debt.*
>
> *(10) Practices for the Department of Defense to incrementally adopt to initiate practices for managing or reducing technical debt.*

Section 835(d) requires the Secretary to "submit to the congressional defense committees a report on the study required […] along with any additional information and views as desired in publicly releasable and unclassified forms" [NDAA 2021].

This report serves as the required independent study due no later than 18 months after entering into the agreement. The following sections briefly describe how we conducted the study, summarize the technical trends observed, and present the resulting recommendations. Appendix A further summarizes the specific technical content applicable to each of the 10 study elements specified in Section 835(b).

Definition of Technical Debt. While software professionals sometimes define *technical debt* differently, we scoped this work according to the definition in NDAA Section 835, which is "an element of design or implementation that is expedient in the short term, but that would result in a technical context that can make a future change costlier or impossible" [NDAA 2021].

- This definition conforms to the one used by the SEI based on a substantial body of work with both industry and the Department of Defense (DoD) [Avgeriou 2016, Kruchten 2019, Ozkaya 2022]. In addition, this definition is broadly accepted as also noted by a recent study on the future of software engineering emphasizing that successful software delivery must include technical debt management [Avgeriou 2023].
- This definition also conforms to the definition in Department of Defense Instruction (DoDI) 5000.87, *Operation of the Software Acquisition Pathway*, "Consists of design or implementation constructs that are expedient in the short term but that set up a technical context that can make a future change costlier or impossible. Technical debt may result from having code issues related to architecture, structure, duplication, test coverage, comments and documentation, potential bugs, complexity, coding practices, and style which may accrue at the level of overall system design or system architecture, even in systems with great code quality" [DoD 2020b]. A similar definition is also provided in the DoD's recently published *Software Engineering for Continuous Delivery of Warfighting Capability* [DoD 2023c]. These definitions further conform to the notion that delayed upgrades, technology refresh, and sustainment items also become technical debt.

Related to technical debt, this report also refers to technical debt items. A *technical debt item* is a single issue that connects affected development artifacts with consequences for the quality, value, and cost of the system triggered by one or more causes related to business, change in context, development process, and people and teams [Kruchten 2019].

2 Study Methodology

The analyses and recommendations for this technical debt study draw from a series of activities led by the SEI and executed according to a roadmap agreed to with the Office of the Under Secretary of Defense for Acquisition & Sustainment (OUSD(A&S)):

- **Literature review**. The study team completed a literature review that summarizes the state of the practice. A particular focus area is the gap between automation (i.e., static analysis tools) and its ability to alert users to comprehensive symptoms of technical debt and the tools needed to guide developers and decision makers in the tradeoff analysis needed to decide whether to resolve or continue to keep the technical debt. (See Appendix C.)
- **Interviews.** Within the scope of this study, the SEI led 16 engagements, which included interviewing stakeholders from the U.S. Federal Government and industry, to gain a broad view of the state of the practice. The study elements specified in Section 835(b) were covered in the interviews. Most interviews were held with one to five stakeholders representing various roles. Of the interviews,
  - Eleven were held with DoD organizations.
  - Four were held with industry.
  - One was held with a program under another U.S. Federal Government agency, the National Aeronautics and Space Administration Jet Propulsion Laboratory (NASA JPL).
- **Deep dives on program data**. SEI subject matter experts engaged with DoD programs outside of this study to examine their practices, data, and decision making related to technical debt in more depth. This work helped program teams move beyond buzzwords and develop a more fine-grained understanding of actual practices and how programs treat different types of issues.
- **Report for program stakeholders**. The SEI's ongoing work with a large, safety-critical program provided further detailed insights and informed the recommendations in this report. The SEI developed a report describing the state of the practice, issues to be aware of at the program level, and examples of technical debt's cybersecurity impact, which was delivered to the program. The report formed the basis for a discussion with the program that elicited feedback about the feasibility and importance of these issues in the program context. The goal of the report was to highlight, using examples, the various ways technical debt can manifest itself and be detected. An excerpt of this program's report is included in Appendix B, which is intended to provide examples of technical debt and how it can be recorded. These examples can be used as guidance for technical debt identification techniques and serve as templates for recording similar kinds of technical debt as technical debt items.

3 Data Sources

The data and experiences we used to draw observations, distill findings, and formulate recommendations come from several sources. We chose these sources to

- provide coverage across the DoD by including Army, Navy, Air Force, and Joint programs
- cover important domains of interest for the DoD, including software in embedded weapons systems, command and control (C2) software, and defense business systems (DBS)
- provide points of comparison for DoD practice against other federal agencies and industry[1]

The SEI organized the interview data into three categories based on the maturity of managing technical debt that the interviewees described as the practices they used:

- Interviews categorized into the *aware* stage reflected those with an awareness of technical debt in the systems and the need to manage it but did not have active technical debt management practices.
- Interviews categorized into the *establishing practices* stage reflected those who had created technical debt management practices and had some small-scale experiences with technical debt.
- Interviews categorized into the *actively managing* stage were those where technical debt explicitly appeared in the artifacts and software development practices, while evidence of the entire program or organization following technical debt management may not have been present.

This data does not represent a DoD-wide assessment of the state of technical debt management practices, but it does represent an assessment of the different stages of adopting technical debt management practices. Of the organizations we interviewed, five were explicitly following an Agile process as described in the Scaled Agile Framework (SAFe®), and seven were explicitly following Agile software development processes following Scrums, backlog management, and expressing requirements as functional and enabler stories (i.e., supporting the activities needed to provide future functionality). The remaining four had aspects of Agile software development and followed gates driven by in-house processes. We did not find any correlation between the organization's current technical debt management stage and the software development process that it followed.

[1] We chose NASA JPL as an additional federal agency due to its familiarity with embedded systems. We drew industry experiences from both the traditional defense industrial base (DIB) and leading-edge software companies in Silicon Valley who develop ultra-large-scale software codebases. NASA JPL and industry interviews included safety-critical systems and business enterprise systems, which allowed for comparison to DoD's embedded weapon systems, C2 systems, and DBS.

Table 1 summarizes the 16 interviews based on the service or agency, the type of system, and the stage of managing technical debt that they represent.

Table 1: Summary of the Technical Debt Interviews

ID	Service/Agency	Type of System	Stage of Managing Technical Debt
1	AF	Defense Business System	Establishing Practices
2	AF	Defense Business System	Establishing Practices
3	AF	Defense Business System	Establishing Practices
4	NASA JPL	Embedded Safety-Critical System	Aware
5	Army	Embedded Weapon System	Aware
6	AF	Embedded Weapon System	Actively Managing
7	Navy	Command and Control System	Actively Managing
8	Navy	Command and Control System	Aware
9	Joint	Command and Control System	Actively Managing
10	AF	Defense Business System	Aware
11	Joint	Embedded Weapon System	Actively Managing
12	AF	Embedded Weapon System	Aware
13	Industry	Business Enterprise System	Actively Managing
14	Industry	Business Enterprise System	Actively Managing
15	Industry	Embedded Safety-Critical System	Actively Managing
16	Industry	Embedded Safety-Critical System	Actively Managing

4 Findings and Examples

We organized this section to align findings with the study methodology. Data gathering and analysis has focused on the study elements specified in Section 835(b) but are presented in the aggregate in this section for clarity. Table 2 (page 19) and Appendix A (page 21) further organize the findings to align with the study elements.

It is important to note that technical debt is distinct from (although related to) other software engineering concepts, such as vulnerabilities and defects. Managing technical debt is more nuanced than managing vulnerabilities and defects. For example, delayed maintenance and technology upgrades often result in costly technical debt due to accumulating system-wide changes that are postponed. While technical debt does have associated negative effects, intentionally incurring and actively managing some technical debt can enable beneficial tradeoffs, such as being able to field a critical capability more quickly.

Technical debt is context specific. While teams or programs can learn from technical debt examples from other organizations, each instance of technical debt and its management will not similarly manifest itself for each program. Characteristics of each program and project significantly influence tradeoffs and consequently whether an issue may be considered as technical debt or not. However, examples are still useful since they can help teams and organizations take steps to understand their own technical debt in different ways.

4.1 Findings from Literature Review

The purpose of the literature review was to identify practices and techniques of managing technical debt and identifying open challenges and gaps in practice. The literature review highlighted different practices and aspects of managing technical debt, including the following:

- having developers record technical debt in issue trackers
- connecting technical debt identified in code comments to tasks in issue trackers
- using machine learning (ML) algorithms to crawl code comments and identify the ones that are relevant to technical debt (i.e., self-admitted technical debt)
- managing technical debt during Agile software development
- dealing with different aspects of technical debt, ranging from requirements to quality to security to design

Although not represented clearly in the academic literature, tool developers and industry strongly advocate incorporating code analysis tools into existing software development and DevSecOps processes. This practice avoids accumulating defects and implementation mistakes, which might otherwise give the false impression of technical debt. Technical debt management is not an activity that can be solely managed as part of test activities. The variety of practices covered in the academic literature to manage technical debt and the consensus that tools alone cannot detect technical debt items provide further evidence that technical debt management needs to be treated as a set of practices, not as a single task.

4.2 Findings from Interviews

We organized the findings from interviews based on the following common themes we observed.

1. **DoD programs are aware of technical debt as a concept and its presence in their systems.** The DoD programs we interviewed were familiar with the term *technical debt* and used it appropriately. Some programs managed and/or tracked technical debt separately from other issues.
   a. Three DoD programs we interviewed managed technical debt as part of their Agile enabler stories. These enabler stories accurately captured the cost-benefit tradeoffs involved in deciding when to carry and when to resolve technical debt issues.
   b. All programs we interviewed were aware of technical debt's impacts, even if they were not actively managing it currently. All programs recognized the value in managing technical debt but did not always feel that their processes were mature enough to do that yet.
   c. There was clear evidence that even if a program was paused (i.e., not actively sustaining or adding new capabilities), technical debt continued to accumulate. This technical debt occurs because of changes in the environment, such as falling behind on versions of commercial off-the-shelf (COTS) tools or security patches, and mission space that the software must adapt to.

   In addition to awareness, all the programs we interviewed welcomed the practice of managing technical debt if it fit seamlessly into their existing processes rather than creating out-of-cycle independent reviews that required additional time and resources.

2. **DoD programs do not employ consistent technical debt management practices.** Today there are wide disparities in the ways that DoD programs manage technical debt, including good practices that should be captured for other programs. Although our study did not use a comprehensive sample, and that sample did not intend to represent the DoD as a whole, we characterized a few different levels of adoption within the 12 U.S. Federal Government programs we interviewed:
   a. Three programs are actively managing their technical debt with dedicated, explicit technical debt practices.
   b. One program is managing its technical debt as part of its ongoing practices for dealing with software defects and vulnerabilities (i.e., technical debt is not called out separately).
   c. Three programs are in the process of establishing technical debt practices.
   d. Five programs are aware of technical debt and have preliminary efforts, but not at the program level.

   The two defense industrial base (DIB) organizations and two industry teams we interviewed all have practices for managing technical debt. As large organizations, they have teams with dedicated technical debt practices as well as teams that are managing technical debt as part of their ongoing practices. One is investigating approaches to scale technical debt management practices organization wide. All of the DIB and industry interviewees emphasized that

their experience may not reflect organization-wide practices, also hinting at variations in how technical debt is managed.

3. **Managing technical debt is often deprioritized.** In the absence of the continuous management of technical debt, other program priorities may take over, even when not paying down technical debt will adversely impact the technical baseline and overall program schedule and budget.
   a. All interviewed programs mentioned experiencing some degree of pressure to prioritize new capabilities over paying down technical debt. Paying down technical debt can speed delivery of future capabilities, but in the short run, users and other stakeholders may not notice it.
   b. Programs have experienced negative impacts from too much technical debt as a result of deprioritization. Excessive technical debt can slow the release of future capabilities. In one case, a program had to pause the delivery of new capabilities to deal with excessive technical debt. This pause had a severe impact on the program's ability to deliver capabilities since what was intended as a short pause stretched to multiple years where no new capabilities were delivered.
   c. Continuous prioritization and reprioritization of technical debt is natural, given other program demands. One DoD program we interviewed has succeeded in getting ahead of technical debt prioritization challenges. It created an explicit technical debt management entry in its overall release planning and allocated 20 percent of each release cycle to technical debt reduction before delivery. This strategy of using a planning goal to allocate effort to technical debt in an ongoing way was useful for ensuring that technical debt did not accumulate in ways that would harm the program's ability to deliver capabilities regularly. This approach should be considered by other programs looking to institute technical debt management.
4. **Use of metrics and reporting technical debt is not common practice.** Use of metrics and reporting can help demonstrate that a program is taking technical debt seriously. Metrics and reporting practices should enable iterative changes, which can be accomplished by incorporating technical debt reporting into Agile software development practices, often using scaling frameworks such as SAFe®. Programs should not report detailed metrics that can be taken out of context and that do not provide insight; instead, they should provide process metrics that demonstrate that they are taking metrics and reporting seriously. Candidate high-level metrics include (1) reporting the percentage of time spent on some recurring cycle that is planned to be spent paying down technical debt, (2) reporting the time actually spent, (3) reporting the growth in technical debt items on the backlog, and (4) reporting deployment frequency with a mapping to technical debt items that hinder delivery. While some interviewees acknowledged reporting the percentage of time on recurring cycles, time spent on growth, and debt items, none of them mapped deployment frequency to technical debt items. Some programs reported tracking technical debt based on effort spent on technical debt resolution or the percentage of time spent per iteration.

5. **Programs use tools to help prevent unintentional quality issues from turning into technical debt.** Some programs use modern software development and analysis tools effectively to catch symptoms early and prevent technical debt from being introduced. They also use procedures to help them address technical debt over time despite other pressures. The interviewed programs all appreciated the importance of recording technical debt, although some had challenges in creating development environments that could support managing technical debt using modern software engineering tools. Three programs used existing recording practices and related enabler stories to manage technical debt. Two categories of tools were commonly utilized by our interviewees:
   a. static code analysis tools for catching programming mistakes and defects
   b. iteration planning and issue tracking tools for managing tasks, stories, and (where applicable) related technical debt items

6. **Categories of funding can hamper appropriate management of technical debt.** Programs sometimes delay or defer technical debt changes based on their access to and perceptions about the different DoD appropriation categories for the funding (i.e., "colors of money") needed versus the funding available. The technical work requires technical debt to be addressed on an ongoing basis as part of the software development lifecycle. There also needs to be awareness of the technical debt that is being handed over to sustainment organizations.

   For example, if addressing technical debt is perceived as an operations and maintenance (O&M) function and a program has only research, development, test, and evaluation (RDT&E) funding, the program may be told not to regularly prioritize technical debt despite the fact that this approach allows technical debt to accumulate. Our interviewees shared examples representing both successful and unsuccessful practices.
   a. One of the programs interviewed was successful in creating a continuous technical debt management practice by addressing technical debt during their Agile iterations, discussing it at each release planning activity, and explicitly communicating that it was not an O&M-only concern. This spread awareness that technical debt should be managed across all "colors of money."
   b. One of the interviewed programs raised challenges around addressing the skewed perception that resolving technical debt items sometimes appears as if taxpayers need to pay for the implementation twice (i.e., work and rework). For example, in embedded systems with decades of life expectancy, a technical debt item may be partially resolved and accepted through standard processes. However, the complete resolution of a technical debt item that was previously deemed to be "good enough" may require additional updates to refactor or improve efficiency in a later iteration. Understanding whether the investment in fixing this technical debt item requires the experience of the team can enable the team to judge whether any recurring problems and delays caused by the item are worth the cost of repairing.

   In this study, we have also seen that access to appropriate development environments, software issue management, and analysis tooling are important success factors for technical debt management. If concerns over "colors of money" prevent programs from procuring tools and a shared infrastructure where these tools can be run frequently/continuously, it can also

hamper or prevent good technical debt practices. Programs must have modern software engineering tools and infrastructures as one strategy against accumulating unintentional technical debt and to properly manage it. If DoD policy and guidance calls out managing and resolving technical debt as critical and explicitly includes recommendations around how to utilize different "colors of money" to fund it flexibly, then challenges around ownership (e.g., development versus sustainment) and double dipping (i.e., work and rework) can be reduced.

7. **The Software Acquisition Pathway (SWP) explicitly highlights technical debt management, which creates a seamless entry point for DoD programs already on the SWP to start technical debt management practices.** The SWP policy today requires that "programs […] actively manage technical debt" during the execution phase [DoD 2020b, 3.3.b(2)].
   a. This policy statement may make more programs aware of the issue and the need to manage it. Currently, more than 55 programs have adopted the SWP and can be expected to be familiar with this requirement.
   b. Not all acquisition programs with software have explicitly adopted the SWP. However, the DoD has been using the SWP as an incubator for modern software practices (i.e., programs on other pathways look to the SWP as a source of good technical practices) [GAO 2022].
   c. Of the programs we interviewed, two that were already actively managing their technical debt were also on the SWP.
8. **The emergence of AI-augmented software development tools and their relationship to technical debt is still not fully understood.** After we started this study, AI-augmented tools evolved rapidly with increasing adoption. These tools have both (1) risks due to inconsistencies in their recommendations and (2) risk of introducing unknown security issues. However, they also have the potential, when used intentionally, to control the risks. None of the DoD programs we interviewed mentioned the role of such emerging tools, but industry interviewees did. Since AI-generated code may contain subtle issues while able to generate large amounts of code very quickly, respondents at a recent technical workshop raised the issue that the rush to deploy such tools today may be creating a growing wave of future technical debt for industry [Shull 2023].
9. **Industry has established best practices in understanding and managing technical debt, which can inspire the DoD.** One best practice observed from industry that large DoD programs can consider adopting is regularly surveying developers to identify common technical debt accumulation areas. Google[2] is one such industry organization. Google has taken an empirical approach to understanding how technical debt manifests itself in its teams and has been publishing an engineering satisfaction survey since 2018 to understand how unnecessary complexity and technical debt may have hindered engineers [Jaspan 2023]. The survey results have helped teams focused on developer productivity at Google understand the *somewhat common* areas of technical debt, which include dead and abandoned code, code quality issues, code degradation, unnecessary dependencies, and delayed migration [Jaspan 2023].

[2] Google's parent company is Alphabet.

While not reflecting the empirical rigor of a multiyear survey conducted and analyzed with regular cadence, similar categories of examples emerged from our study as well:

- Some programs had to pay additional funds to a COTS product supplier to maintain security updates after a project was at its end of life. This happens most often with operating systems, such as Windows XP.
- Some programs struggled with issues resulting from the undisciplined use of open source software (OSS). Sometimes this was a situation where the OSS version used by the program was not up to date, and making the update would require potentially significant amounts of rework to the rest of the code. Other times, there were instances where out-of-date OSS created security issues. In one example, the program identified the use of OSS during active software development that had not been updated for 84 months, which caused it to be 44 versions behind, resulting in additional complexity and design issues.

4.3 Findings from Deep Dives on Program Data

The deep dives with subject matter experts on DoD program data outside of this study provided relevant takeaways worth including in this study since they reveal nuances in managing technical debt:

- **Educating stakeholders is key.** A safety-critical program is exploring ways to incorporate technical debt management practices into its software development practices. The goal is to reinforce the importance of eliminating quality issues as they occur while making informed decisions about the technical debt that it takes on and its cost of resolution. In addition to educating contractors appropriately, program priorities include asking for the right analysis and relevant artifacts from them. Providing education in general concepts while using program-specific technical examples also enables increasing the competence of stakeholders in their ability to identify their context-specific technical debt.
- **Analyzing data collected by tools allows for identifying technical debt correctly.** A business enterprise system program analyzes the outputs of static analysis tools on its codebases to identify trends and develop strategies for eliminating systemic issues and fine-tuning the analysis tools to eliminate noise. This practice is invaluable for improving the return on investment from using static analysis tools and ensuring that the identified issues are neither overblown as technical debt nor neglected when symptoms start accumulating. Getting the balance right requires some work.

5 Recommendations

The following recommendations apply to the findings from Section 4 and Appendix A of this study.

A. The DoD needs to share best practices to empower programs to incorporate technical debt management into software development lifecycle activities as one of the core software engineering practices.

The effective management of technical debt is critical for modern software practice, especially with respect to sustaining an appropriate cadence for deploying capabilities. In both our interviews and the substantial evidence found in the related literature, we confirmed that technical debt management is an explicit focus for developers in industry and a normal part of day-to-day software engineering work.

As our interviews and literature search revealed, there is no one-size-fits-all set of metrics or measurement approach. Programs with different scopes and scales have different needs. However, regardless of the program context, a key aspect of technical debt management is bringing visibility to instances of technical debt and making tradeoffs explicit for the long-term mitigation of it. Therefore, the DoD should look for opportunities to make it easy for programs to incorporate technical debt management practices into the software development process. Even if these practices do not span the entire lifecycle of a system, they would still provide benefit. Resource challenges are not easy to resolve, and mandating more practices and metrics to report are not likely to result in positive change. The DoD can embrace a phased approach by relying on practices that are already in place in programs.

Recommended practices, which could be documented in a DoD Technical Debt Guidebook, can be rolled out in the three stages described below.

Stage 1: Bring visibility to existing technical debt. Some programs are aware that the concept of technical debt exists and are experiencing the effects of their own technical debt. However, they have not started to adopt the practices required to address technical debt. These programs need a starting point for tackling these challenges that will not overwhelm them. While it may be relatively easy to put tools in place that will scan software code, that may result in so many technical debt issues that a team may become overwhelmed and find it impossible to get started. In contrast, the following techniques can help bring visibility to existing technical debt more gradually:

- Configure existing issue tracking and management tools to include a technical debt category so that these instances can be tracked and handled separately.
- During design and architecture reviews, explicitly capture technical debt, including remediation strategies. These reviews typically surface issues that are not clear-cut defects or vulnerabilities, but would require substantial rework to improve the code.
- During development, empower developers to manually document as technical debt any issues that are difficult to resolve and that require further tradeoff and root cause analysis.

- As part of regular release reviews, capture technical debt items, including remediation strategies. These technical debt items may include overarching concerns (e.g., end-of-life of software, hardware, operating systems) that will require substantial rework.

Recurring examples of technical debt that surface through applying these practices will be related to overall technical risks, and they can be paired with risk management practices to ensure appropriate priorities are assigned and resources are secured.

Stage 2: Establish goals. The three DoD programs that were actively managing their technical debt were successful because they clearly identified and related their technical debt items to Agile enabler stories, reviewed technical debt enabler stories regularly during sprints and other reviews, and prioritized these stories alongside other capability priorities.

This approach allowed the teams to apply measures, such as

- percentage of resources allocated to be spent on managing quality and technical debt per delivery increment (e.g., sprint, iteration, gate, release)
- percentage of technical debt items in the backlog, which enables the program to visualize the technical debt that is carried

It is essential to allocate time (e.g., in Agile capacity planning) to address technical debt during each delivery increment.

Stage 3: Establish tooling and measurement environments. Once an understanding of the level of existing technical debt starts to emerge, programs can assess their existing tooling to manage technical debt and incorporate other tools as needed. Broadly, good practice should encompass the following:

- use of *automation and tool support* through modern software engineering tools (e.g., configuration management, continuous integration, code analyzers, development tools, issue trackers) to ensure quality and prevent unintentional technical debt from creeping in
- the expectation that software developers are encouraged to *manually* record instances of technical debt items as they occur so that they can be paid down in the future
- the establishment of heuristics at the program level, such as establishing an overall threshold for the percentage of open technical debt items in the backlog (For example, technical debt items should make up no more than 10-15 percent of overall backlog items. If the program exceeds the technical debt threshold, a technical debt reduction sprint should be planned.)
- the use of tools and procedures that allow both tool-discovered design issues and developer-reported technical debt instances to be tracked, prioritized against other work, and regularly addressed alongside new capability development

As tooling decisions are made, special attention should be given to assess where these emerging tools may fit. Improved capabilities in code completion and code review tools can effectively prevent developers from introducing unintentional coding errors. These coding errors, when accumulated in magnitudes, can create brittle codebases and result in technical debt. When tools are used to generate multiple lines of code or portions of codebases, it is important to ensure that experts

are also in the loop assessing conformance against the cross-cutting runtime and sustainment goals using architecture and quality assurance practices.

This stage is also when system-specific quantifiable metrics can be more effectively addressed. These metrics must be considered along with both (1) other program metrics (e.g., backlog items, defect rates) and (2) system metrics (e.g., metrics for software quality, system complexity, vulnerability, and security assessment).

Candidate programmatic metrics include reporting

- the *planned time* spent paying down technical debt
- the *actual time* spent addressing technical debt
- the *total* number of technical debt items in the backlog (noting increases/decreases)
- *deployment frequency* with a mapping to technical debt instances that hinder delivery

System metrics should not be expressed in terms of technical debt; they should be expressed in terms of system concerns. Recurring areas of concern can create technical debt. One example of a technical debt strategy to combat recurring areas of concern might be to scan the system quarterly for dead code and allocate time for its removal. Other system concerns can vary based on the domain of the system, the organization, and other context-specific characteristics.

As already mentioned, these best practices could be compiled into a DoD Technical Debt Guidebook, which might include excerpts of this report along with examples from DoD programs that have successfully implemented technical debt management practices. The Guidebook should incorporate references to other DoD resources advocating modern software engineering practices. A recently published reference example is the *DoD Risk, Issue, and Opportunity (RIO) Management Guide for Defense Acquisition Programs.* It emphasizes, similar to this report, the iterative nature of technical debt management and its inclusion in sprint planning [DoD 2023b]. Envisioned DoD Technical Debt Guidebook references should make it clear that technical debt management is an iterative and continuous process and that programmatic metrics should

- allow establishing baselines, continuous evaluation, prioritization, and triaging
- allow establishing control over the existing technical debt
- be utilized effectively for intentional trade-off decisions and value creation

B. The DoD should continue to update existing policy and guidance to include technical debt management practices.

A useful first step toward technical debt management in the DoD is the SWP, which requires that SWP programs manage technical debt [DoD 2020b]. It would be helpful to update the policy/guidance to provide important information about how technical debt management can be instantiated in DoD programs. The policy/guidance update should be based on lessons learned from real programs with mature practices, including some from this study, to demonstrate that these practices fit well within the DoD context.

At a minimum, the policy/guidance update should include the following:

- Programs should employ both automated (e.g., static code analysis scans) and manual (e.g., opportunities for developers to add technical debt items to the backlog and tag them as technical debt when intentionally taking on debt or identify technical debt in design reviews) mechanisms for identifying technical debt.
- Programs should track technical debt items on the backlog separate from other types of items, such as vulnerabilities and defects.
- Programs should allocate appropriate effort during iteration capacity planning for resolving technical debt items, and they must ensure that this effort is protected from the pressure to focus on new capabilities.
- Program roadmaps should include the effort for managing technical debt to ensure that it is planned and that effort is allocated to it over time.
- Software should at least pass a code quality scan as well as unit tests before allowing check-ins when in a continuous integration and continuous delivery/continuous deployment (CI/CD) environment to avoid having unintentional quality issues creep in that may result in technical debt.

In addition to describing practices, the policy/guidance update should also include common areas where DoD programs are most likely to accumulate technical debt. Our interviews highlighted examples of these common areas, including management of open source software versions, evolution of missed security patching, technology obsolesce, and postponed large-scale refactoring.

We understand that the SWP is an incubator for good software practices in the DoD, and the policy and guidance demonstrate expectations about mature practices that should be used for software. However, A&S should update each acquisition pathway's policy to recommend using SWP practices, which would ensure technical debt management practices for software are required on other pathways and ensure that the lessons learned as part of following the guidance are disseminated at scale. One such policy that should be updated by OUSD Research and Engineering (R&E) is DoDI 5000.88, *Engineering of Defense Systems* and its associated guidebook, which provide guidance for adaptive acquisition pathways [DoD 2020a]. The Software Engineering section of that policy should be expanded to include technical debt. Additionally, the *Engineering of Defense System Guidebook*, which very briefly mentions technical debt, should be expanded to include the bullets listed above [DoD 2022].

It should be noted that the DoD published several guidance documents during this study that begin to include technical debt and technical debt management as an essential practice for successful software development [DoD 2023a, 2023b, 2023c]. An additional positive step is that the *Systems Engineering Plan (SEP) Outline Version 4.1* now includes a requirement to address technical debt management in the Software Engineering section [DoD 2023d]. The common thread in of all these documents is increasing the competence of the DoD in its execution of modern software engineering practices. All of these guidance documents make it clear that continuous technical debt management is an essential part of successful delivery. Future revisions to these guidance documents should incorporate or reference the DoD Technical Debt Guidebook, if the DoD pursues it as proposed in Recommendation A, to provide more concrete guidance.

C. The DoD should make available and encourage appropriate training to help programs understand technical debt management.

Training can help institutionalize important technical debt practices by making the issue visible to more stakeholders and ensuring that these stakeholders are armed with the practices and strategies needed to manage technical debt effectively. The goal should be to ensure that these practices are part of the commonly expected baseline of software management in the DoD.

The Defense Acquisition University (DAU) may be able to develop role-based technical debt training for various roles (e.g., executives, program managers, chief of software, acquisition professionals, development teams). Providing targeted training for these roles will enable them to use consistent vocabulary, concepts, and practices. Some of this training can also be made available to contractors to ensure that everyone involved in a program uses the same vocabulary.

Depending on the targeted role, training content should include, but not be limited to

- explaining what technical debt is and reviewing representative examples
- differentiating between (1) causes of technical debt and (2) actual technical debt that needs to be monitored within systems
- the relationship between technical debt items and enablers, vulnerabilities, defects, and new capabilities
- selecting best-fit tools and customizing their detection and reporting capabilities to a program's needs
- understanding the role of qualitative (e.g., developers' perception of existing technical debt) and quantitative (e.g., mean time to resolution) measures
- establishing data analysis pipelines from issue trackers and scan results
- recognizing technical debt during design reviews
- conducting tradeoff analysis, which feeds into prioritizing which debt to resolve and which to carry

Additionally, the DAU should review existing software courses to ensure that technical debt and technical debt management are properly included. Some of the courses to update might include

- LOG 270 Introduction to DoD Software Life Cycle [DAU 2023a]
- ACQ 1700 Agile for DoD Acquisition Team Members [DAU 2023b]

With recent DoD training initiatives to increase the competencies of the digital workforce, such as Digital University and new DAU credentialing in areas like DevSecOps, perhaps there are other training opportunities that might provide viable ways to increase workforce knowledge in technical debt and technical debt management. While this recommendation has areas that focus on the DAU, other training organizations outside of the DAU should also be considered based on a Service's or program's needs and priorities. Excerpts from this report could also be compiled into a roadshow briefing that could be given to numerous groups across the DoD to provide high-level awareness on technical debt.

D. OUSD(A&S) should require continuous collection of technical-debt-related data and metrics.

The interviewed programs that are successfully managing technical debt use metrics that are similar to those used for defect and vulnerability management, such as mean time to resolution, duration open, rate of recurrence, and density. Using metrics like these, successful programs map technical debt items discovered to the number of issues identified, prioritized, and addressed over a given delivery tempo. Also, each technical debt item is sized according to its scope and allocated to a sprint or iteration based on its scope and system context.

This approach mirrors experiences from industry, as indicated by both our interviews and literature review. For example, Google explored 117 metrics, including technical-system-quality-related metrics, as indicators of common areas of technical debt identified in its quarterly surveys (e.g., dependencies, code quality, migration, code degradation). Google's analysis showed that no single metric predicts reported categories of technical debt. Additional evidence from our industry interviews and other research align with the conclusion that no single generalizable metric can be used to understand leading indicators of technical debt.

Teams must select specific metrics for their specific context. Our study results also show that the design implications of technical debt are different in different contexts. For example, coupling and cohesion are two widely used system-modularity-related design metrics, where loosely coupled software is expected to be easier to modify. However, in systems where high performance is desired, compromises from modularity have to be made. Robust technical debt management practices that do not solely rely on metrics will allow these tradeoffs and their implications to be expressed clearly as well. Given these observations, we recommend that programs use programmatic metrics for quantifying technical debt while using technical metrics to provide insights for system-level quality and design issues by contextualizing them based on program tempo, high-priority architectural concerns, areas of change, and refactoring costs. Other data, such as existing design concerns, rework and refactoring costs, and data about technical debt items, should also be collected and regularly analyzed.

E. The Office of the Under Secretary of Defense (Comptroller) should update the Financial Management Regulation (FMR) DoD 7000.14 to clarify that both RDT&E and O&M funds can be used to resolve technical debt issues.

Our interviews and published research make it clear that technical debt management must be an ongoing process that is supported by both qualitative and quantitative practices as outlined in this report. This implies that the responsibility for addressing technical debt cannot be exclusively deferred to O&M teams. Dedicating only O&M funds to technical debt management delays the resolution of issues, which likely increases risk and introduces additional challenges due to handoffs between development and O&M teams.

The Office of the Under Secretary of Defense (Comptroller) should update the applicable software, software maintenance, and software support sections within the FMR. Those applicable sections might include Volume 2A, Chapter 1: General Information and Volume 4, Chapter 27: Internal Use Software. Some specific areas of focus might be within Volume 2A, Chapter 1, Section 010212: Budgeting for Information Technology and Automated Information Systems with

additions on technical debt and technical debt management in the RDT&E appropriations and O&M appropriations subsections.

F. OUSD (R&E) should ensure that more programs have access to modern development, analysis, and CI/CD tools and practices.

Several programs we interviewed had already embraced DevSecOps approaches to incorporating code quality and security analysis tools (e.g., CheckMarx, Fortify, SonarQube, CAST) into their environments. These tools and others that assist with development activities (e.g., integrated development, automated code review, automated unit and integration testing) are essential to ensure timely quality development and avoid unintentional technical debt. They also enable the timely detection of implementation errors. However, these tools must be configured to ensure that false positives are minimized, and high-priority issues are, in fact, detected appropriately.

OUSD (R&E) should make these tools and practices available to expand these technical debt management best practices to more programs. This recommendation aligns with GAO-23-105867, which recommends "The Secretary of Defense should ensure that the Under Secretary of Defense for Research and Engineering, with the input of the military departments, **establishes an overarching plan**—which identifies associated resources—to **enable the adoption of modern engineering tools, across all programs**. This should **include** (1) mission engineering, (2) systems engineering, and (3) **software engineering**. (Recommendation 3)." GAO further states that "these officials explained that it is **difficult for program offices to justify investing in and adopting these tools** because of the high potential costs and uncertainty of benefits" [GAO 2023].

G. OUSD(R&E) should invest in / utilize technical debt research areas.

Based on the literature review and interviews, we recommend research investments in the following areas of technical debt. Some advances may come from the larger software engineering research community and commercial companies, and others may require DoD Science and Technology (S&T) investments to close the gap and adapt approaches for the defense domain. Either way, advances in these areas should be encouraged and tracked to help advance the state of the practice. These recommended research focus areas also align with a future software engineering study focused on better incorporating technical debt management as a focus area [Avgeriou 2023].

- ***Improved tooling:*** Tooling to support developers is evolving fast, especially with tools powered by artificial intelligence (AI), ML, and foundation models that assisting with programming tasks getting increasing attention. The software engineering community does not yet know the implications of these emerging tools, and research can empower their targeted development to help avoid unintentional technical debt and to better track intentional technical debt.
- ***Infrastructure:*** Understanding trends and systems' existing context is critical for improved technical debt management. Research into establishing infrastructures for collecting and assessing data without overloading developers would have significant benefits on both improving the management of technical debt as well as improving system quality in the long run with an empirical basis. Such automated data collection throughout software development

processes can also help with software acquisition decisions (e.g., cost of ownership, prioritization, portfolio management).

- ***Technical debt as a value-creation activity:*** Technical debt is a positive and value-creation-focused design concept. Research into when technical debt can be taken on; how it can be communicated; how it accelerates and improves development; and how it can be resolved without burdening the overall cost of ownership, system quality, and business goals is essential and significantly lagging. The DoD, as the owner of many legacy systems, can exemplify continuous system evolution and sustainment by enabling research in technical debt as a value-creation activity.
- ***Technical debt workforce competency:*** Technical debt management is a design tradeoff and incentive management activity. Research into how to increase maturity in the technical debt competency of the workforce while simplifying technical debt management through incentive management can serve both program managers as well as technical stakeholders.
- ***System-level technical debt metrics:*** There is no single quantitative, system-level technical debt metric. Programs need to baseline their technical debt based on their context and track it using programmatic measures similar to those used for defects and vulnerabilities. Research in proposing how to use system-level metrics, or recommending the use of existing ones, in relationship to technical debt would be a valuable contribution.

Table 2 specifically aligns the recommendations with the study elements from Section 835(b) posed by Congress. Additional findings on the study elements are included in Appendix A.

Table 2. Recommendations Against FY22 NDAA Section 835(b) Study Elements

FY22 NDAA Section 835(b) Questions [NDAA 2021]	**Recommendations**						
	A (Best Practices)	**B (Policy & Guidance)**	**C (Training)**	**D (Role of Metrics)**	**E (FMR)**	**F (Tools)**	**G (Research)**
(1) Qualitative and quantitative measures which can be used to identify a desired future state for software-intensive systems	✓			✓		✓	✓
(2) Qualitative and quantitative measures that can be used to assess technical debt	✓			✓		✓	✓
(3) Policies for data access to identify and assess technical debt and best practices for software-intensive systems to make such data appropriately available for use	✓	✓		✓	✓		
(4) Forms of technical debt which are suitable for objective or subjective analysis	✓		✓			✓	✓

FY22 NDAA Section 835(b) Questions [NDAA 2021]	Recommendations						
	A (Best Practices)	B (Policy & Guidance)	C (Training)	D (Role of Metrics)	E (FMR)	F (Tools)	G (Research)
(5) Current practices of Department of Defense software-intensive systems to track and use data related to technical debt	✓			✓	✓	✓	
(6) Appropriate individuals or organizations that should be responsible for the identification and assessment of technical debt, including the organization responsible for independent assessments	✓	✓	✓		✓		
(7) Scenarios, frequency, or program phases during which technical debt should be assessed	✓	✓	✓		✓		
(8) Best practices to identify, assess, and monitor the accumulating costs technical debt	✓	✓				✓	
(9) Criteria to support decisions by appropriate officials on whether to incur, carry, or reduce technical debt	✓	✓	✓				✓
(10) Practices for the Department of Defense to incrementally adopt to initiate practices for managing or reducing technical debt	✓		✓			✓	

Appendix A: Findings Against FY22 NDAA Section 835(b) Study Elements

In this appendix, we summarize the findings and specific analysis topics requested by the study elements from Section 835(b) [NDAA 2021]. The findings and specific analysis from the interviews, which also covered the study elements, are provided in Section 4.2. We provide the recommendations in Section 5 along with a mapping of the recommendations to Section 835(b) questions in Table 2.

(1) Qualitative and quantitative measures which can be used to identify a desired future state for software-intensive systems.

- The best indicator of the desired future state is related to successful deployment frequency with demonstrated capability against the plan.
- Programs do not pay down technical debt for its own sake; they do so to ensure that systems stay adaptable and able to effectively field new capabilities as needed.
- Technical debt instances need to be prioritized/aligned with a regular cadence to ensure that technical debt that is taken on to accelerate program priorities is recognized and paid down in a timely way.

(2) Qualitative and quantitative measures that can be used to assess technical debt.

- Since technical debt is always accumulating, it is important to set a goal to allocate a regular percentage of effort to paying down technical debt and fencing other priorities as part of the overall program budget, and not tying it to O&M spending.
- Understanding whether the number of technical debt instances on the backlog is increasing or decreasing is necessary to know whether that percentage needs to be ratcheted up or down.
- In terms of qualitative measures, categorizing the type of technical debt seen in the program can help the program identify areas of concern that require more attention. These categories are best identified based on recurring empirical analysis across programs, similar to those reported in a 2023 article by Ciera Jaspan and Collin Green [Jaspan 2023].
- There is no single quantitative system-level technical debt metric. Programs need to baseline their technical debt based on their context and track it using programmatic measures similar to those used for defects and vulnerabilities.

(3) Policies for data access to identify and assess technical debt and best practices for software-intensive systems to make such data appropriately available for use.

- Programs reported using technical-debt-related data internally (e.g., enabler stories, technical debt in sprint management percentages) to good effect but not reporting it to senior stakeholders—either because the senior stakeholders did not care (i.e., were not sure

what to do with the information) or because they would take it out of context. This study information suggests that communicating technical debt with an empirical basis through recommended practices needs to be encouraged.

- The DoD needs to empower reporting when technical debt instances are accumulating (i.e., not being addressed quickly enough). This empowerment needs to be done before addressing technical debt begins to interfere with a program's ability to deliver capability updates, when programs can prioritize technical debt appropriately.

(4) Forms of technical debt which are suitable for objective or subjective analysis.

Technical debt can be identified from various software development artifacts such as code, architecture, and issue trackers. Regardless of where they have been identified, all forms of technical debt are suitable for objective or subjective analysis.

- It is important for programs to have processes in place for logging and tracking both tool-discovered and developer-reported technical debt instances.
- Training or other assistance should be provided to help establish good practices in programs, in particular related to the appropriate use of tools. Some programs we interviewed reported that they use software development tools out of the box without calibrating them; hence, in many cases, they get noisy data.
- Each program needs to understand that they should set priorities and the critical context.
  - Some programs stressed that architectural forms of technical debt were important, perhaps more so than code-level issues, which is also consistent with industry best practices.
  - Some programs found technical debt critical to manage due to its mapping to cybersecurity risks as well as technology enablers.

It is important to note that in this report, we focus on technical debt following the definition provided in NDAA Section 835, which focuses on implementation artifacts. Literature has identified many taxonomies, such as requirements debt, social debt, and documentation debt. While the spirit of applying the concept of debt to different tasks is understandable, those tasks are not within the scope of technical debt.

(5) Current practices of Department of Defense software-intensive systems to track and use data related to technical debt.

Best practices include the following:

- Use the same tracking system (e.g., Jira) for technical debt as for other elements in the backlog so that the whole set of issues can be periodically reviewed, reprioritized, and planned to be addressed on some timeline.
- Do not allow code to be checked back into the repository without passing a clean scan and relevant test so that unchecked technical debt is not allowed to accumulate. (However, note that this approach may not be feasible for legacy systems that have years of accumulated technical debt to pay down.)
- Make technical debt an explicit part of software development planning rather than postponing it to sustainment.

- Make sure to focus on both programmatics (e.g., tracking in project management) and engineering (e.g., architecture reviews, design tradeoff analyses) to ensure that programmatic and engineering aspects of technical debt analysis/reduction are addressed.
- Make sure that program-specific metrics and reporting practices enable iterative changes. This can be accomplished by incorporating technical debt reporting into Agile software development practices, often using scaling frameworks such as SAFe®. Programs should not report detailed metrics that can be taken out of context and that will not provide insight. However, they could provide process metrics that show they are taking metrics and reporting seriously. Candidate high-level metrics include (1) reporting the percentage of time spent on some recurring cycle that is planned to be spent paying down technical debt, (2) reporting the time actually spent, (3) reporting the growth in technical debt items on the backlog, and (4) reporting deployment frequency with a mapping to technical debt instances that hinder delivery.

(6) Appropriate individuals or organizations that should be responsible for the identification and assessment of technical debt, including the organization responsible for independent assessments.

- Assessing technical debt should be a regular process step (e.g., ensuring that developers check code back in only after it passes a clean scan). The organization responsible for technical debt management is both the team that sets up the software infrastructure/software factory and the development team. Individuals with infrastructure/software factory expertise should be responsible for ensuring that the appropriate tools, analysis automation, and data collection analysis capabilities are integrated into the development and deployment pipelines. The development team should be responsible for ensuring technical debt discussions occur and that the detection, documentation, prioritization, and resolution of technical debt items are common practices.
- We did not hear much support for assessments by independent/outside teams. Part of the issue was the concern that different organizations have different definitions of what constitutes technical debt, which may not be appropriate in context. However, one program did conduct an independent assessment that resulted in their active technical debt management.
- Roles and responsibilities should be assigned to developers, architects, project managers, and program managers.

(7) Scenarios, frequency, or program phases during which technical debt should be assessed.

- Technical debt must be managed continuously as part of the RDT&E cycle. Similarly, any technical debt carried over to sustainment must be factored into O&M resources.
- Scenarios of technical debt management that we identified through our interviews include the following:
  - a program organically managing its own technical debt aligned with its iteration and release tempo

- a program that sets expectations with contractors for reporting technical debt and guides its management through prioritization
- a program that leaves all technical debt management to the contractor (not recommended)
- a program that initiates a retrospective assessment (independent or in house) of technical debt and initiates technical debt management as a consequence

(8) Best practices to identify, assess, and monitor the accumulating costs technical debt.

- Empower a culture change to have developers disclose and prioritize technical debt.
- Monitor the delivery cadence.
- Record technical debt items.
- Map technical debt to rework costs, and use this data for prioritization.
- Make technical debt management part of the overall software development process.
- Map technical debt to technology enablers.
- Review technical debt as part of both iteration planning and risk management.
- Incorporate scanning tools into DevSecOps pipelines to avoid unintentional code quality issues that result in or partially contribute to technical debt.
- Conduct architecture reviews to identify high-cost technical debt and technical debt reduction sprints.
- Consider the approach that one program took when it mapped its technical debt to different categories. This approach enabled more concrete resolution discussions. The categories are
  - problem reports
  - static code analysis findings
  - vulnerability findings
  - regression test automation
  - out-of-date test procedures
  - design-related issues
  - memory/throughput challenges
  - safety assurance process
  - pipeline tool improvements
- Identify design flaws, their rework cost, and the consequences of not resolving them as technical debt items.
- Recognize that it will never be possible to address 100 percent of technical debt; therefore, establish a prioritization schema.

(9) Criteria to support decisions by appropriate officials on whether to incur, carry, or reduce technical debt.

Factors that effectively inform this decision can include the following:

- how taking on or reducing technical debt hinders or accelerates technology enablers
- how taking on or reducing technical debt hinders or accelerates new capability
- program lifecycle phase
- risks and system structure and behavior (quality attribute) priorities hindered or enabled by technical debt
- cost of addressing technical debt
- cost of retaining technical debt

(10) Practices for the Department of Defense to incrementally adopt to initiate practices for managing or reducing technical debt.

- Empower technical debt to be a budget item in program planning.
- Make training available to avoid misconceptions and to encourage standardization and the adoption of known best practices.
- Make the use and availability of modern software engineering tools with technical debt management capabilities nonnegotiable.
- Transition a phased approach for establishing technical debt practices as described in Recommendation A on page 12 of this report.

Appendix B: Technical Debt Item Examples

The SEI's ongoing work with a large, safety-critical program provided further detailed insights and informed the recommendations in this report. The SEI developed and delivered to the program a report describing the state of the practice, issues to be aware of at the program level, and examples of technical debt's cybersecurity impact., The report formed the basis for a discussion with the program that elicited feedback about the feasibility and importance of these issues in the program context. The goal of the report was to highlight, using examples, the various ways technical debt can manifest and be detected.

This appendix provides an excerpt from the program's report, including examples of technical debt and how it can be recorded. These examples can serve as valuable guidelines for identifying technical debt and can be used as templates for documenting similar technical debt items.

An organization needs to actively monitor four categories of technical debt to ensure that existing DevSecOps, software quality, and security management practices are well aligned to also support technical debt management. We organize these categories based on the artifact they are detected from.

1. *Detect technical debt from code*, where code-level conformance and structural analysis indicate maintainability and concerns related to the structure of the system and the codebase.
2. *Detect technical debt from symptoms* that signal architecture issues.
3. *Detect technical debt from architecture* during design reviews and analysis of decisions.
4. *Detect technical debt from development and deployment infrastructure*, which are not typically part of the delivered system but may impact its delivery, security, and quality.

To reason about technical debt, estimate its magnitude, and offer information on which to base decisions, you must anchor technical debt to explicit technical debt items that identify parts of the system: code, design, test cases, or other artifacts. A technical debt item is a single issue that connects affected development artifacts with consequences for the quality, value, and cost of the system triggered by one or more causes related to business, change in context, development process, and people and teams.

We next demonstrate each of the four categories of technical debt detection with examples. These examples of criteria, techniques, and technical debt item descriptions are from actual systems and developer discussions, drawing on the concepts of secure design [Arce 2014], and abstracted for a general audience. In order to exemplify the relationship between technical debt and cybersecurity in some of the examples, we refer to the Common Weakness Enumeration (CWE™). CWE is a categorized, publicly accessible list of software and hardware weakness types (https://cwe.mitre.org). The CWE was recently expanded to include quality characteristics such as maintainability that impact security [CISQ 2019].

™ CWE is a trademark of The MITRE Corporation.

B.1 Detect Technical Debt from Code

Technical debt takes different forms in different types of development artifacts. The source code embodies many design and programming decisions. The code can be subjected to review, inspection, and analysis with static checkers to find issues of finer granularity: while such analysis can detect some types of technical debt such as code clones and unnecessary complexity, almost all other violations detected will be symptoms that require further analysis [CISQ 2021, OMG 2018].

Static analysis checkers that are part of DevSecOps tool chains assist with detecting growing complexity, business logic nonconformances, and some basic classes of design issues such as very large classes and single points of failure. When not actively managed, all of these issues start accumulating unintended future rework, resulting in technical debt. Furthermore, typical examples of technical debt, such as greater complexity, increase opportunities for vulnerabilities.

Static analysis is not the only approach to examine code for technical debt and its symptoms. Examining the code at a high level with a focus on architecture is another approach to surface code conformance issues that results in technical debt. To understand the impact of change driven by technical debt, developers need to identify the modules of a system that are the focus of a change and follow the dependencies to the modules that will be affected by the change. Relevant characteristics for analyzing individual elements and their dependencies include complexity of individual software elements, interfaces of software elements, interrelationships among the software elements, system-wide properties, and interrelationships between software elements and stakeholder concerns.

Here is an example of a technical debt item that signals accumulating system complexity and uncovers needed design analysis and rearchitecting using static code analysis, which alerts for CWEs. In this example shown in Table 3, the static code analysis that the team regularly runs reveals many small, avoidable coding issues related to reliability, security, performance efficiency, and maintainability that were never addressed due to schedule pressure and lack of coding guidelines. Together they have caused the modifiability of the codebase to degrade.

Table 3: Example of Recognizing Technical Debt with Static Code Analysis

Name	Accumulated CWEs from violating maintainability quality rules resulted in technical debt.
Summary	Automated static source code analysis revealed an increasing number of issues with the following weaknesses and security implications of maintenance and evolution: CWE-561 Dead Code, CWE-1047 Modules with Circular Dependencies (120 issues), CWE-1074 Class with Excessively Deep Inheritance (37 issues). Due to the severe number of these issues, system modifiability has degraded significantly.
Consequences	We have already received two vulnerability reports in the dead code area; more are likely to emerge. There are increasing numbers of defects at the area of the codebase with the deep inheritance hierarchy. Modules with circular dependencies also take longer to incorporate new capabilities, increasing maintenance and evolution costs. In general, these areas of the codebase are difficult to maintain, which affects security by making it more difficult or time-consuming to find and fix vulnerabilities.
Remediation approach	Dead code: Remove the dead code. • Address during local refactoring within an iteration. Circular dependencies and excessive inheritance: These will require rearchitecting. • Designers need to understand how the architecture and evolution of the software influence security considerations under many circumstances. Address this in the next architecture review. The addition of continuous integration processes creates a requirement for architecture modularity and flexibility to support security, as changes to systems are pushed automatically and at ever shorter periodicity. • Understanding and restructuring module dependencies to eliminate circular dependencies and excessive inheritance will require planning across iteration boundaries.
Reporter / assignee	The dead code and inheritance hierarchy issues were automatically reported as a result of the static code analysis scan: As the software development lead, I am reporting this as a composite technical debt item. I have also created two related issues in the backlog and linked to this issue: 1. Remove dead code (assigned to the developer team for the next iteration). 2. Remove circular dependencies and deep inheritance (assigned to the architect to resolve as part of the architecture refactoring effort).

It is important to recognize that there is no one-size-fits-all tool that automatically uncovers single instances of such technical debt items. Running a static analysis tool for the first time can yield thousands of issues. Recording all individual issues that tools identify as separate technical debt items or composing them as one major technical debt item is unwieldy and an incorrect approach. Furthermore, such an approach often leads to these issues lingering in the backlog as they are perceived as false positive noise, and developers might disable the rules for detecting them during future checks. Following the process to understand system quality goals provides a focus for the development team to create a manageable number of issues. They record the relevant results as technical debt items so they can start managing them. As this example highlights, identification of such violations will point to areas of further analysis to look at clusters of technical debt. Areas where large clusters of technical debt issues accumulate are good candidates for rework and architectural changes.

Going forward, the organization can address how to ensure that the team does not inject new debt into the source code so no one has to deal with these many issues again. The causes can be identified, and process improvement practices put in place to address them. Creating coding guidelines and providing training for developers improves their savviness at recognizing when they

potentially inject technical debt in the code. Running static analysis in a continuous integration environment promotes clean code where developers get immediate feedback on the issues during a commit and are required to fix them before acceptance.

B.2 Detect Technical Debt from Symptoms

Technical debt symptoms are not always simple to recognize. Automated tools, such as tools that check for code quality or secure coding violations, can uncover some symptoms that signal technical debt. As seen in the previous section, one step in the right direction is to use agreed-upon CWEs associated with maintainability checks as a basis for identifying technical debt related to security issues [CISQ 2019]. This approach also helps with concrete quantification.

Other symptoms such as major faults or delivery delays in the system can also signal technical debt. Establishing continuous monitoring for such symptoms and reacting promptly will prevent technical debt from accumulating in the first place. For example, symptoms of technical debt can be exposed using metrics that indicate recurring defects and vulnerabilities, increasing number of defects and vulnerabilities in one particular area of the system, or defects that have not been possible to resolve, reducing delivery tempo. These should stimulate further analysis.

Repeated security breaches traced to security-related bugs, such as a crash or exploit enabled by an out-of-bounds number, are additional examples of technical debt items that can be detected by their symptoms. Table 4 summarizes such an example of a technical debt issue that increases vulnerabilities.

Table 4: Example of Recognizing Technical Debt from Observable Symptoms

Name	Screen spacing creates numerous unexpected crashes across the codebase due to API incompatibility.
Summary	The source code uses a very large negative letter-spacing in an attempt to move the text offscreen. The system handles up to -186 em fine, but crashes on anything larger. A similar issue was fixed with a patch, but there were several other similar reports. Time permitting, I'm inclined to want to know the root cause of this. My sense is that if we patch it here, it will pop up somewhere else later.
Consequences	We already had 28 reports from seven clients. And it definitely leaves the software vulnerable. Finding the root cause can be time-consuming given that existing patches did not resolve the issue.
Remediation approach	We already patched this twice. The responsible thing to do is to first find the root cause and create a fix at the source. My previous experience tells me that the external Web client and our software again has an API incompatibility, but further analysis is needed. The course of action is to verify where the root of this is and see if we can fix it on our side. If the external Web client team needs to fix it, we would need to negotiate.
Reporter / assignee	DevSecOpsTeam / External WebClientTeam

While patches provide immediate relief, tracing interconnections in the design revealed a dependency on an external library maintained by another group, as the developer suspected. The dependencies to external software elements were not analyzed and designed for security issues, which resulted in multiple crashes across the system with the same root cause. Repeated crashes are

symptoms pointing to the technical debt in this example. They should trigger further architecture analysis and identification of the external dependency which, if not fixed, will widen the system's security risk exposure. The additional rework is caused here by creating multiple patches, which increased system complexity without resolving the security issue.

A tendency sometimes exists to immediately categorize all such symptomatic defects and vulnerabilities as technical debt. This approach results in an artificial increase in the number of issues while hindering the opportunity to do deep analysis and find the root cause. In a highly dynamic DevSecOps environment where organizations are under attack, the symptoms of vulnerabilities associated with an attacker's behavior need to be communicated between operations and development teams to trace operational weaknesses to root cause vulnerabilities in the source code. This further refines the goal of using a static analysis tool to address the vulnerability associated with attackers' behavior, rather than executing static analysis tools out of context and trying to deal with the myriad results [Izurieta 2019]. The same mindset needs to be embraced when using such tools to detect symptoms to identify and mitigate technical debt.

B.3 Detect Technical Debt from Architecture

The key difference between detecting technical debt using code analysis and detecting it at the architecture level is that the code is more concrete, tangible, and visible. Code can be explored using software tools, but that provides information at a lower level of granularity, sometimes giving the impression that fixing local issues will eliminate technical debt. Code analysis does not reveal systematic architecture issues which may point to broader types of technical debt. Architecture analysis can reveal such technical debt that is more encompassing and pervasive. It involves choices about the structure or the architecture of the system: choice of platform, middleware, technologies for communication, user interface, or data persistency. It is typically more difficult to detect and assess architectural decisions resulting in debt with tools, and the cost associated with repaying the debt is larger and intertwined in a complex network of structural dependencies.

Architecture analysis allows a team to assess whether design decisions will meet the quality attribute requirements early in development. Malicious external attacks that expose the vulnerabilities of a system at runtime are lagging indicators of the failure to meet a security quality attribute requirement. As operations staff employ countermeasures, development staff trace the cause to the source code vulnerability to aid in patching the system in a first response. Tracing further to the root cause when there is a design or architecture issue and remediating the technical debt can prevent the issue or related issues from resurfacing and benefit the business/program by positioning the system to make it easier to analyze, maintain, and evolve over its life span.

Lightweight architecture analysis techniques surface risks in design decisions that can lead to technical debt. A number of analysis techniques have proven useful for examining the architecture as it is being designed and used throughout the software development life cycle including thought experiments, reflective questions, checklists, scenario-based analysis and walkthroughs, analytics models, prototypes, and simulations. Developers often use existing frameworks and components to provide some of the structure and behavior of the system. The choices made to use these frameworks and components are design decisions that affect the quality and security of the system. In

the example shown in Table 5, a design decision made early in the development effort has resulted in a security breach.

Table 5: Example of Recognizing Technical Debt Requiring Architecture Rework to Enhance Security

Name	Missing Authentication for Critical Function (CWE: 306) requires significant architectural rework.
Summary	The authentication for functionality for user identity management had been assumed out of scope in the first release. This resulted in the recent security breach and compromised the data in the system. No critical information was compromised; however, we cannot continue to operate before adding authentication.
Consequences	Given the number of features that depend on this, we are looking at significant rearchitecting. The consequences will depend on the associated functionality, but we will have to reassess read/write accesses to our sensitive data and recreate administrative and other privileged functionality.
Remediation approach	Divide the software into anonymous, normal, privileged, and administrative areas. Identify which of these areas require a proven user identity and use a centralized authentication capability. Identify all potential communication channels, or other means of interaction with the software, to ensure that all channels are appropriately protected. Our developers sometimes perform authentication at the primary channel but open up a secondary channel that is assumed to be private. For example, a login mechanism may be listening on one network port, but after successful authentication, it may open up a second port where it waits for the connection but avoids authentication because it assumes that only the authenticated party will connect to the port. In general, if the software or protocol allows a single session or user state to persist across multiple connections or channels, authentication and appropriate credential management need to be used throughout.
Reporter / assignee	Reported by a Dev engineer during system integration test. Remediation assigned to multiple team members including the DevSecOps team and lead architect.

CWE-306, Missing Authentication for Critical Functionality, is a vulnerability that enables attackers to gain the privilege level of the exposed functionality. The technical impact of the weakness can be used to determine the cost to the development team of carrying the technical debt and the risk exposure to the business. Manual analysis is needed to understand the underlying design issue and the cost of remediating the debt by improving the design.

Trade-offs made among system qualities to meet the organization's mission or business goals may lead to such technical debt. For example, since authentication consumes system resources and results in timing lags that can degrade performance, the decision may be made to omit reauthentication given the context (e.g., authentication occurs in the control panel software, but not in the vehicle it is operating). As hardware performance improves over time and software changes enlarge the attack surface, this decision should be revisited. Whether it is easy or difficult to reinsert authentication depends on whether architecture decisions made early on will support this kind of evolution. Recording this as a technical debt issue proactively gives the team an opportunity to revisit the decision as hardware and software assumptions evolve and resolve it in a timely fashion. Even better, if the technical debt item is acknowledged and recorded at the time the decision is made—that is, when the decision to skip authentication was agreed upon—architects and designers could consider other choices that would simplify reintroducing authentication at a later time.

B.4 Detect Technical Debt from Development and Deployment Infrastructure

Technical debt also occurs in the development and deployment infrastructure. This section describes two examples of technical debt, one related to the suboptimal design and coding of test infrastructure (Table 6) and another to misalignment between the infrastructure and the code itself (Table 7).

Infrastructure has become a key software development artifact. Analyzing for technical debt in the infrastructure that serves the completed code to a running system in operation encompasses issues in build, test, and deployment code. Current DevSecOps trends are increasing automation capabilities and tool support, and these trends have exposed deficiencies in the production process used by development organizations. Infrastructure-related technical debt impedes a team's ability to evolve a system or fix known issues. These problems often influence an organization's ability to achieve business goals, particularly if they slow velocity or hinder the ability to release in small, rapid increments. Analysis techniques for code and design can be applied to build scripts, test suites, and deployment scripts to detect the presence of technical debt.

Consider the following first example of test suites. Test suites are, in effect, code. Suboptimal design and coding of tests also leads to the same weaknesses as with the product code that have security implications related to maintenance and evolution. In this example the development team would like to reuse new Test Helper modules for a legacy test framework. The development team is migrating integration tests to the new test framework. There are two parallel sets of Test Helper modules to maintain during migration. Duplication is a source of technical debt and requires changes in two places. Often, changes are not synced, resulting in unintended drift between frameworks. The remediation approach allows the legacy test framework to reuse the new test framework's Test Helper modules, which are cleaner (better documentation, linted, obvious errors fixed). The technical debt item exemplified in Table 6 shows the team's analysis to get insight into the maintainability of the test framework.

Table 6: Example of Recognizing Technical Debt in the Test Infrastructure

Name	Maintaining two parallel Test Helper modules results in inconsistencies.
Summary	While the DevTeam has been migrating its integration tests to the new test framework, there have been two parallel Test Helper modules to maintain, one for the new framework and another for the legacy framework. The redundancy is resulting in inconsistencies and unneeded work.
Consequences	This test code is a source of technical debt and requires team members to make changes in two places. Often, they forget, which leads to unintended drift between the two frameworks. Scaling this infrastructure to dozens of teams will magnify the challenges as we roll out the testing framework.
Remediation approach	Reuse the new test framework's Test Helper modules. The goal isn't 100% code reuse between the old and new test framework, but 80–90%. The test methods from the legacy module that remain are here for three reasons: • When ported to the new test framework, the test methods were refactored into different modules and will require updating legacy tests to load new modules. • Navigating the page in the old test framework is hacky and has been cleaned up in the new test framework so they won't ever share implementations. • Subtle refactoring changes make the new implementation fail certain tests. This test failure should be followed up by using the old implementation and then refactoring once all tests have been migrated.
Reporter / assignee	DevTeam / QATeam

The misalignment of the build, test, deployment, and delivery strategies and accompanying tools is another area where technical debt appears in the development and deployment infrastructure. Technical debt can appear in the misalignment between the infrastructure and the code in the following ways:

- *Testing.* As software evolves rapidly, new tests may be missing, may test an older interpretation of the requirements, or may interact with other tests in unknown ways.
- *Infrastructure of the operational system.* Deferred binding generates a responsibility for the development team to make architecture decisions to accommodate the change during deployment, delivery, and runtime and a responsibility for the staff of the operational system to make the change.

In the second example in Table 7, the security implications of a change request impact not only the code, but also its alignment with test, deployment, and delivery. These issues are documented as a technical debt description and included in the backlog.

Table 7: Example of Recognizing Technical Debt Within Infrastructure Misalignment

Name	Database misalignment with continuous delivery pipeline impacts security during upgrade.
Summary	A database engine upgrade reveals that the security implications of the upgrade are not well understood and controlled. Secondary and tertiary dependencies are not well documented or understood. These dependencies are presently precluding us from completing the upgrade because we are constantly running into issues.
Consequences	Designers need to understand how change influences security considerations under these secondary and tertiary dependencies. The need for security considerations will appear during continuous delivery in • testing, since all possible variations of states will need to be verified to guarantee that they uphold the security posture of the system (among, of course, other tested behavior) • deployment, when permissions, access control, and other security-related activities and decisions need to take place • delivery and runtime, in the form of configuration changes, enabling and disabling of features, and sometimes dynamic loading of objects The addition of continuous integration processes creates a requirement for security flexibility, as changes to systems are pushed automatically and at ever shorter periodicity.
Remediation approach	Analyze for the database and infrastructure dependencies and rework the design for secure updates.
Reporter / assignee	Reported by the Ops engineer doing the upgrade. Remediation assigned to multiple team members including the DevSecOps team and lead architect.

The organization needs a deliberate strategy for managing technical debt not only for development, but also for testing and production. An agile or flexible architecture complements continuous integration processes and allows the team to explore technical options rapidly with minimal ripple effect. The architecture can be understood in terms of design decisions that influence the time and cost to implement, test, and deploy changes and operate the software without introducing bugs and vulnerabilities.

When there are distributed teams, coordination issues can create misaligned assumptions about design decisions which can cause technical debt. Distributed teams face coordination challenges as the architecture is apportioned to them for implementation, and then again when they hand off their implementations to an integrated testing environment. Tests and infrastructure should be designed and aligned for their purpose, implemented following sound coding practices, and executed in alignment with the functionality and attributes they are meant to support.

Appendix C: Literature Reviews of Technical Debt Management Practices

Technical debt is a mature research field, which is evidenced by the increasing number of publications and systematic literature reviews since the first research paper was published in 2010 [Brown 2010]. A recent tertiary study reviewing secondary studies in managing technical debt reported 532 unique research studies [Junior 2022]. For example, consistent with the findings of this study, Besker and Martini [Besker 2018] found 42 other research papers and concluded that there is agreement in the reviewed literature that architecture technical debt is of primary importance and is related to the challenge of complexity, maintenance, and evolvability.

Other studies have similar findings [Lenarduzzi 2020]; they also emphasize that there is lack of empirical evidence on how to assess the repayment of technical debt and how to validate a set of tools to do so.

Table 8 summarizes recent systematic literature reviews and the outcomes they report.[3] The findings of these studies are consistent with our observations and recommendations based on our interviews. Studies focusing on tool reviews also have findings consistent with our study. For example, in the recent study by Lefever and colleagues, *On the Lack of Consensus Among Technical Debt Detection Tools*, the authors discuss several commonly used industry tools and conclude that the tools report very different results for even simple measures such as size, complexity, file cycles, and package cycles. Furthermore, most tools give little new insight other than "big files are bad," which further validates that consistent tooling and automation is a significant gap [Lefever 2021].

Table 8: Literature Review

Paper Title	Number of Studies Reviewed	Key Observation
Consolidating a Common Perspective on Technical Debt and its Management Through a Tertiary Study [Junior 2022]	19 (tertiary)	While some confusion around understanding what technical debt may constitute still exists, most identify technical debt as tradeoffs between design decisions. The findings of the paper, which focused on 19 secondary studies covering 532 papers, are consistent with the findings of our interviews.

[3] For a more detailed analysis of literature and historical analysis of technical debt, refer to *Technical Debt Management: The Road Ahead for Successful Software Delivery* [Avgeriou 2023].

Paper Title	Number of Studies Reviewed	Key Observation
A Systematic Literature Review on Technical Debt Prioritization: Strategies, Processes, Factors, and Tools [Lenarduzzi 2020]	38	Research has found that most investigated technical debt is identified in code and architectural artifacts. Despite the research in prioritizing technical debt, the attributes considered are limited. The findings of this study are also consistent with our interview results, which found that more data-driven research is needed to take advantage of existing software cost-benefit-analysis-based techniques for technical debt prioritization for its resolution.
A Tertiary Study on Technical Debt: Types, Management Strategies, Research Trends, and Base Information for Practitioners [Rios 2018]	13 (tertiary)	Similar to other studies, this study also identified technical debt in code, architecture, and design artifacts to be critical. Its findings also include that while a number of indicators of the presence of technical debt are known, little is known about how to guide the use of these indicators to achieve greater effectiveness/efficiency in technical debt item identification and resolution activities. This, hints at research with a more data-driven and empirical basis, just as our recommendations included.
Managing Architectural Technical Debt: A Unified Model and Systematic Literature Review [Besker 2018]	42	This study emphasizes that there is a side agreement in the reviewed literature that architecture technical debt is of primary importance.
Analyzing the Concept of Technical Debt in the Context of Agile Software Development: A Systematic Literature Review [Behutive 2017]	38	In this study, most of the literature reviewed discussed technical debt in the context of Agile software development. It reports that consequences of poor software development results in technical debt. The studied literature commonly lists the following as the causes and consequences of the most costly debt: - Causes: push quick delivery, lack of attention to architecture and design, lack of understanding of technology, inadequate test coverage - Consequences: reduced productivity, quality degradation, increased maintenance costs - Refactoring is the most popular practice used to repay technical debt.
Identification and Analysis of the Elements Required to Manage Technical Debt by Means of a Systematic Mapping Study [Fernández-Sánchez 2017]	63	This study found the following: - Technical debt is context dependent: It includes issues such as the history of product development, prospects, or time to market. - Time-to-market push is the most referenced cause of technical debt. - Establishing communication among stakeholders can help significantly in effectively managing technical debt.

Paper Title	Number of Studies Reviewed	Key Observation
Identification and Management of Technical Debt: A Systematic Mapping Study [Alves 2016]	100	The study reported that the empirical evaluation of the indicators of technical debt is lagging and does not provide strong correlation. Similar results were reported in 2023 based on data reported by Google engineers [Jaspan 2023].
A Systematic Mapping Study on Technical Debt and its Management [Li 2015]	94	As one of the earlier reviews on the subject, this study reported the need for more empirical evidence on management process, while also reiterating the negative effects of technical debt on maintainability.
The Financial Aspect of Managing Technical Debt: A Systematic Literature Review [Ampatzoglou 2015]	69	The confusion around the “color of money” that we observed is also reflected in this study, which concludes that there is a lack of a clear mapping between financial and software engineering concepts. This is an important area of open research.

Abbreviations and Acronyms

A&S
Acquisition and Sustainment

AF
Air Force

AI
Artificial Intelligence

API
Application Programming Interface

C2
Command and Control

CI/CD
Continuous Integration and Continuous Delivery/Continuous Deployment

COTS
Commercial off-the-Shelf

CWE™
Common Weakness Enumeration

DAU
Defense Acquisition University

DBS
Defense Business Systems

DevSecOps
Development, Security, and Operations

DIB
Defense Industrial Base

DoD
Department of Defense

DoDI
Department of Defense Instruction

FFRDC
Federally Funded Research and Development Center

FMR
Federal Management Regulation

GAO
Government Accountability Office

JPL
Jet Propulsion Laboratory

ML
Machine Learning

NASA
National Aeronautics and Space Administration

NDAA
National Defense Authorization Act

O&M
Operations and Maintenance

OSS
Open Source Software

OUSD
Office of the Under Secretary of Defense

R&E
Research and Engineering

RDT&E
Research, Development, Test, and Evaluation

RIO
Risk, Issue, and Opportunity

S&T
Science and Technology

SAFe®
Scaled Agile Framework

SEI
Software Engineering Institute

SWP
Software Acquisition Pathway

References/Bibliography

URLs are valid as of the publication date of this report.

[Alves 2016]
Alves, Nicolli S. R.; Mendes, Thiago S.; de Mendonça, Manoel G.; Spínola, Rodrigo O.; Shull, Forrest; & Seaman, Carolyn. Identification and Management of Technical Debt: A Systematic Mapping Study. *Information and Software Technology*. Volume 70. Pages 100-121. 2016. https://doi.org/10.1016/j.infsof.2015.10.008

[Ampatzoglou 2015]
Ampatzoglou, Areti; Ampatzoglou, Apostolos; Chatzigeorgiou, Alexander; & Avgeriou, Paris. The Financial Aspect of Managing Technical Debt: A Systematic Literature Review. *Information and Software Technology*. Volume 64. Pages 52-73. 2015. https://doi.org/10.1016/j.infsof.2015.04.001

[Arce 2014]
Arce, Iván; Clark-Fisher, Kathleen; Daswani, Neil; DelGrosso, Jim; Dhillon, Danny; Kern, Christoph; Kohno, Tadayoshi; Landwehr, Carl; McGraw, Gary; Schoenfield, Brook; Seltzer, Margo; Spinellis, Diomidis; Tarandach, Izar; & West, Jacob. Be Flexible When Considering Future Changes to Objects and Actors. In *Avoiding the Top 10 Software Security Design Flaws.* IEEE Center for Secure Design. September 2014.

[Avgeriou 2016]
Avgeriou, Paris; Kruchten, Philippe; Ozkaya, Ipek; & Seaman, Carolyn. *Managing Technical Debt in Software Engineering.* In *2016 Dagstuhl Reports*. April 2016. https://drops.dagstuhl.de/opus/volltexte/2016/6693/pdf/dagrep_v006_i004_p110_s16162.pdf

[Avgeriou 2023]
Avgeriou, Paris; Ozkaya, Ipek; Chatzigeorgiou, Alexander; Ciolkowski, Marcus; Ernst, Neil; Koontz, Ronald J.; Poort, Eltjo; & Shull, Forrest. Technical Debt Management: The Road Ahead for Successful Software Delivery. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. 2023.

[Behutive 2017]
Behutiye, Woubshet; Rodríguez, Pilar; Oivo, Markku; & Tosun, Ayse. Analyzing the Concept of Technical Debt in the Context of Agile Software Development: A Systematic Literature Review. *Information and Software Technology*. Volume 82. Pages 139-158. 2017. http://dx.doi.org/10.1016/j.infsof.2016.10.004

[Besker 2018]
Besker, Terese; Martini, Antonio; & Bosch, Jan. Managing Architectural Technical Debt: A Unified Model and Systematic Literature Review. *Journal of Systems and Software*. Volume 135. Pages 1-16. 2018. https://doi.org/10.1016/j.jss.2017.09.025

[Brown 2010]
Brown, Nanette; Nord, Robert; Ozkaya, Ipek; Kazman, Rick; & Kruchten, Philippe. Managing Technical Debt in Software-Reliant Systems. Pages 47-52. In *FoSER '10: Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research.* 2010. https://dl.acm.org/doi/10.1145/1882362.1882373

[CISQ 2019]
Consortium for Information & Software Quality (CISQ). *List of Weaknesses Included in the CISQ Automated Source Code Quality Measures*. June 2019. https://www.it-cisq.org/pdf/cisq-weaknesses-in-ascqm.pdf

[CISQ 2021]
Consortium for Information & Software Quality (CISQ). Software Quality Standards—ISO/IEC 5055. *Automated Source Code Quality Measures*. March 2021. https://www.it-cisq.org/standards/code-quality-standards/

[DAU 2023a]
Defense Acquisition University (DAU). LOG 0270 Introduction to DoD Software Lifecycle Management. *DAU website*. July 31, 2023. https://icatalog.dau.edu/mobile/CourseDetails.aspx?id=12675

[DAU 2023b]
Defense Acquisition University (DAU). ACQ 1700 Agile for DoD Acquisition Team Members. *DAU website*. July 11, 2023. https://icatalog.dau.edu/mobile/CourseDetails.aspx?id=12313 https://icatalog.dau.edu/mobile/CourseDetails.aspx?id=12675

[DoD 2020a]
Department of Defense (DoD). DoDI 5000.88. *Engineering of Defense Systems*. November 2020. https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/500088p.PDF

[DoD 2020b]
Department of Defense (DoD). DoDI 5000.87 *Operation of the Software Acquisition Pathway*. October 22, 2020. https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/500087p.PDF

[DoD 2022]
Department of Defense (DoD). *Engineering of Defense Systems Guidebook*. February 2022. https://ac.cto.mil/wp-content/uploads/2022/02/Eng-Defense-Systems_Feb2022-Cleared-slp.pdf

[DoD 2023a]
Department of Defense (DoD). *Defense Technical Risk Assessment Methodology (DTRAM) Criteria Volume 6.4.* May 2023. https://www.cto.mil/wp-content/uploads/2023/05/DTRAM-6-4-May2023.pdf

[DoD 2023b]
Department of Defense (DoD). *Risk, Issue, and Opportunity Guide.* September 2023. https://www.cto.mil/wp-content/uploads/2023/09/RIO-2023.pdf

[DoD 2023c]
Department of Defense (DoD). *Software Engineering for Continuous Delivery of Warfighting Capability.* April 2023. https://www.cto.mil/wp-content/uploads/2023/06/SWE-Guide-April2023.pdf

[DoD 2023d]
Department of Defense (DoD). *Systems Engineering Plan (SEP) Outline Version 4.1.* May 2023. https://ac.cto.mil/wp-content/uploads/2023/05/SEP-Outline-4.1.pdf

[Fernández-Sánchez 2017]
Fernández-Sánchez, Carlos; Garbajosa, Juan; Yagüe, Agustín, Yagüe; & Perez, Jennifer. Identification and Analysis of the Elements Required to Manage Technical Debt by Means of a Systematic Mapping Study. *Journal of Systems and Software.* Volume 124. Pages 22-38. 2017. https://www.sciencedirect.com/science/article/pii/S0164121216302138

[GAO 2022]
United States Government Accountability Office (GAO). *Leading Practices: Agency Acquisition Policies Could Better Implement Key Product Development Principles*. March 2022. https://www.gao.gov/products/gao-22-104513

[GAO 2023]
Government Accountability Office (GAO). *Defense Software Acquisitions Changes to Requirements, Oversight, and Tools Needed for Weapon Programs*. GAO-23-105867. July 2023. https://www.gao.gov/assets/gao-23-105867.pdf

[Izurieta 2019]
Izurieta, Clemente & Prouty, Mary. Leveraging SecDevOps to Tackle the Technical Debt Associated with Cybersecurity Attack Tactics. Pages 33-37. In *TechDebt '19: Proceedings of the Second International Conference on Technical Debt*. May 2019. https://doi.org/10.1109/TechDebt.2019.00012

[Jaspan 2023]
Jaspan, Ciera & Green, Collin. Defining, Measuring, and Managing Technical Debt. *IEEE Software*. Volume 40. Number 3. 2023. Pages 15-19. https://doi.org/10.1109/TechDebt.2019.00012

[Junior 2022]
Junior, Helvio Jeronimo & Travassos, Guilherme Horta. Consolidating a Common Perspective on Technical Debt and its Management Through a Tertiary Study. *Information and Software Technology*. Volume 149. Number 6. 2022. http://dx.doi.org/10.1016/j.infsof.2022.106964

[Kruchten 2019]
Kruchten, Philippe; Nord, Robert; & Ozkaya, Ipek. *Managing Technical Debt.* Addison-Wesley Professional. 2019.

[Lefever 2021]
Lefever, Jason; Cai, Yuanfang; Cervantes, Humberto; Kazman, Rick; & Fang, Hongzhou. On the Lack of Consensus Among Technical Debt Detection Tools. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP).* May 2021. https://arxiv.org/abs/2103.04506

[Lenarduzzi 2020]
Lenarduzzi, Valentina; Besker, Terese; Taibi, Davide; Martini, Antonio; & Fontana, Francesca Arcelli. A Systematic Literature Review on Technical Debt Prioritization: Strategies, Processes, Factors, and Tools. *Journal of Systems and Software*. Volume 171. 2020. https://doi.org/10.1016/j.jss.2020.110827

[Li 2015]
Li, Zengyang; Avgeriou, Paris; & Liang, Peng. A Systematic Mapping Study on Technical Debt and its Management. *Journal of Systems and Software*. Volume 101. Number 3. Pages 193-220. 2015.

[NDAA 2021]
117th Congress. *National Defense Authorization Act for Fiscal Year 2022*. December 2021. https://www.congress.gov/bill/117th-congress/senate-bill/1605/text

[OMG 2018]
Object Management Group (OMG). *Automated Technical Debt Measure*. September 2018. https://www.omg.org/spec/ATDM/1.0/PDF

[OUSD(C) 2021]
Under Secretary of Defense (Comptroller). Volume 11A: "Reimbursable Operations Policy." In *Department of Defense Financial Management Regulation (DoD FMR).* DoD 7000.14 - R. May 2021. https://comptroller.defense.gov/fmr/

[Ozkaya 2022]
Ozkaya, Ipek & Nord, Robert. *10 Years of Research in Technical Debt and an Agenda for the Future* [blog post]. *SEI Blog*. August 22, 2022. https://insights.sei.cmu.edu/blog/10-years-of-research-in-technical-debt-and-an-agenda-for-the-future/

[Rios 2018]
Rios, Nicolli; de Mendonça, Neto; Manoel. Gomes; & Spínola, Rodrigo Oliveira. A Tertiary Study on Technical Debt: Types, Management Strategies, Research Trends, and Base Information for Practitioners. *Information and Software Technology.* Volume 102. Pages 117-145. 2018. https://doi.org/10.1016/j.infsof.2018.05.010

[Shull 2023]
Shull, Forrest. *U. S. Leadership in Software Engineering & AI Engineering: Critical Needs & Priorities Workshop – Executive Summary*. October 2023. https://insights.sei.cmu.edu/library/us-leadership-in-software-engineering-ai-engineering-critical-needs-priorities-workshop-executive-summary/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	November 2023	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Report to the Congressional Defense Committees on National Defense Authorization Act (NDAA) for Fiscal Year 2022 Section 835 Independent Study on Technical Debt in Software-Intensive Systems	FA8702-15-D-0002

6. AUTHOR(S)

Ipek Ozkaya, Forrest Shull, Julie Cohen, & Brigid O'Hearn

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2023-TR-003

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100	n/a

11. SUPPLEMENTARY NOTES

Funded by OUSD (A&S) per FY22 NDAA Section 835

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DOPSR, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

A team from Carnegie Mellon University's Software Engineering Institute (SEI) conducted an independent study to satisfy the requirements of the Fiscal Year 2022 National Defense Authorization Act (NDAA) Section 835, Independent Study on Technical Debt in Software-Intensive Systems.

This report describes the conduct of the study, summarizes the technical trends observed, and presents the resulting recommendations. The study methodology includes a literature review, a review of SEI reports developed for program stakeholders, deep dives on program data from SEI engagements with Department of Defense (DoD) programs, and interviews conducted using the 10 study elements specified in Section 835(b).

The study concludes that programs are aware of the importance of managing technical debt. Furthermore, a number of DoD programs have established practices to actively manage technical debt. During this study, the DoD published several guidance documents that begin to include technical debt and technical debt management as an essential practice for successful software development. Study recommendations include that the DoD must continue to update policy/guidance and empower programs to incorporate technical debt practices as part of their software development activities while enabling research in improved tool support and data collection.

14. SUBJECT TERMS	15. NUMBER OF PAGES
technical debt, FY22 National Defense Authorization Act Section 835	49

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102