DISSERTATION

AVOIDING TECHNICAL BANKRUPTCY IN SYSTEM DEVELOPMENT: A PROCESS TO REDUCE THE RISK OF ACCUMULATING TECHNICAL DEBT

Submitted by

Howard Kleinwaks

Department of Systems Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Engineering

Colorado State University

Fort Collins, Colorado

Fall 2023

Doctoral Committee:

Advisor: Thomas Bradley
Co-Advisor: Ann Batchelor

Gregory Marzolf
Daniel Wise
John F. Turner

ABSTRACT

AVOIDING TECHNICAL BANKRUPTCY IN SYSTEM DEVELOPMENT: A PROCESS TO REDUCE THE RISK OF ACCUMULATING TECHNICAL DEBT

The decisions made early in system development can have profound impacts on later capabilities of the system. In iterative systems development, decisions made in each iteration produce impacts on every future iteration. Decisions that have benefits in the short-term may damage the long-term health of the system. This phenomenon is known as technical debt. If not carefully managed, the buildup of technical debt within a system can lead to technical bankruptcy: the state where the system development can no longer proceed with its lifecycle without first paying back some of the technical debt. Within the schedule constrained development paradigm of iteratively and incrementally developed systems, it is especially important to proactively manage technical debt and to understand the potential long-term implications of decisions made to achieve short-term delivery goals.

To enable proactive management of technical debt within systems engineering, it is first necessary to understand the state of the art with respect to the application of technical debt methods and terminology within the field. While the technical debt metaphor is well-known within the software engineering community, it is not as well known within the systems engineering community. Therefore, this research first characterizes the state of technical debt research within systems engineering through a literature review. Next, the prevalence of the technical debt metaphor among practicing systems engineers is established through an empirical survey. Finally, a common ontology for technical debt within systems engineering is proposed to enable clear and

concise communication about the common problems faced in different systems engineering development programs.

Using the research on technical debt in systems engineering and the ontology, this research develops a proactive approach to managing technical debt in iterative systems development by creating a decision support system called List, Evaluate, Achieve, Procure (LEAP). The LEAP process, when used in conjunction with release planning methods, can identify the potential for technical debt accumulation and eventually technical bankruptcy. The LEAP process is developed in two phases: a qualitative approach to provide initial assessments of the state of the system and a quantitative approach that models the effects of technical debt on system development schedules and the potential for technical bankruptcy based on release planning schedules.

Example applications of the LEAP process are provided, consisting of the development of a conceptual problem and real applications of the process at the Space Development Agency. The LEAP process provides a novel and mathematical linkage of the temporal and functional dependencies of system development with the stakeholder needs, enabling proactive assessments of the ability of the system to satisfy those stakeholder needs. These assessments enable early identification of potential technical debt, reducing the risk of negative long-term impacts on the system health.

ACKNOWLEDGEMENTS

Although it is my name on the authorship line of this dissertation, I could not have accomplished this work completely on my own. There are many who contributed ideas, thoughts, and critical comments without which this work not have been able to be completed.

First, I would like to thank my advisors, Professor Ann Batchelor and Dr. Thomas Bradley. Without their help and patience through weekly meetings, this dissertation would not have come to fruition. Their guidance the critical components of the process provided clarity and definition that, at times, was very much needed. I also want to thank the other members of my committee for their time and valuable insights and recommendations throughout the entirety of this effort.

Thank you to my colleagues at the Space Development Agency, specifically Frank Turner, Matt Rich, and Mike Butterfield, who not only listened to me talk about the need for a process like LEAP but who also helped me to develop the critical concepts. Thank you to Modern Technology Solutions, Inc. for enabling me to participate in the program while working full time.

Finally, and most importantly, thank you to my family: my wife Sue and my children Ellie and Jake. Thank you for understanding when I needed to spend time writing and working on my school work instead of doing other activities and for giving me the time to complete this degree. Your support has made this possible, and I love you and I can’t thank you all enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 - INTRODUCTION

1.1 Motivation and Background

This research is motivated by a broad set of ongoing and evolving changes in the field of systems engineering. The increased importance of software-intensive systems and rapid development of technological capabilities have enabled a reduction in the time to market for systems [1]. The ability of competitors to quickly release products has shifted the driving motivation of systems development from decreasing the cost of the system to increasing the speed with which the system is delivered [2]. The environments in which the systems are developed are increasing volatile, uncertain, complex, and ambiguous (VUCA) [3]. Traditional systems engineering methodologies, such as the Waterfall method [4], treat requirements as fixed and therefore the changes associated with a VUCA environment can result in increases to budget or schedule [5]. The combination of these factors has produced an increased emphasis on requiring "increasing flexibility, innovation and rapid capability development" in systems development [6]. However, the ability to develop a flexible system often conflicts with the ability to rapidly deploy the system [7].

Flexibility can be accomplished by engineering an agile system or by using agile systems engineering methods. The two concepts are not the same [8]. An agile system is a system that can adjust to changes in its environment and intended usage [9]. While customers may desire agility in their system, they often do not want to pay for features that are unused in base cases [10]. Building an agile system may result in a more complex system with higher costs and longer development schedules for the first delivery [7], which conflicts with the desire to increase the speed of delivery.

In contrast, agile systems engineering seeks to add flexibility and agility to the processes used to develop the system [8]. Traditional sequential development models define the requirements up front and deliver the capability at the end. Iterative, incremental, and agile methods contain processes to adjust and adapt to user feedback received on repeated deliveries of incrementally developed capability [11]. Iterative and incremental development methods have been in use for many years, having been used on the X-15 hypersonic jet and NASA's Project Mercury [12], and are becoming more commonplace within the system engineering community [13]. Iterative and incremental methods are being used to rapidly develop multiple prototypes that can receive and adjust to user feedback [14] instead of the traditional approach that tries to develop one perfect prototype [3].

The push towards more flexible systems engineering development methods is driven, in part, by the desire to shorten development cycles. The desire to decrease the time to market results in developers seeking to deploy capabilities quickly. Developers may use approaches that require the least amount of work and may implement technical compromises and shortcuts for the sake of expediency [7]. These compromises include activities such as prioritizing functional requirements over non-functional requirements [15]. Especially pernicious is the fact that technical shortcuts may appear to be successful initially, justifying their use. However, these compromises may slow projects down over time [16] due to the long-term impacts of the early decisions. These impacts on product schedule, cost, and performance can be understood through the metaphor of technical debt.

The technical debt metaphor was first introduced by Ward Cunningham in 1992 as a method to explain to his management the need for refactoring software. He stated that "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly

with a rewrite" [17]. Technical debt can impact a program when "the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt" [17]. An accumulation of technical debt in the system results in system developers working on fixing the debt instead of continuing the development of the system.

Many authors have sought to expand and improve on the definition of technical debt within the realm of software engineering. However, even though the issues associated with technical debt are experienced by systems engineering practitioners [18], the technical debt terminology is not prevalent within the published systems engineering literature [19] and is not commonly used by systems engineering practitioners [18]. The lack of common terminology to describe similar problems prevents the identification and use of common solutions [20]. This dissertation adopts the definition of technical debt from [19]:

> *"Technical debt is a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a system."*

The definition identifies the critical aspects of technical debt: technical compromises that yield initial benefits but cause later problems. Much like the technical debt metaphor, additional related terms, such as 'technical compromise' and 'health of a system', are not well defined in the literature [21]. Additional terms, such as 'rework' are used to describe similar conditions to technical debt. A full ontology of technical debt within systems engineering needs to be created to enable proper communication about the associated issues.

This dissertation seeks to understand the impacts of technical debt on systems developed using incremental and iterative design methods, especially focusing on the ability to proactively identify potential sources of technical debt during feature development that may prohibit the ability to

satisfy stakeholder needs. To do so requires the ability to associate stakeholder needs with technology and feature development in both the temporal and functional dimensions.

1.1.1 Iterative and Incremental Development

Traditionally, a system lifecycle is associated closely with the development method – the set of processes used to manage the system development, such as Waterfall [4], Spiral [22], or adaptive and iterative methods. However, the selection of the development method is only one part of the system lifecycle. Managing a system throughout its lifecycle requires the definition of a complete technical strategy, consisting of the development method, the development strategy, and the delivery strategy [23]. The development method defines the processes, such as how change is handled, system decision points, and the handling of risk. The development strategy defines how successive versions (if any) of the system will be developed. The delivery strategy defines if the system will be delivered all at once or through a series of releases. Any development method can be used with any combination of development strategy and delivery strategy. Table 1-1 provides examples of several different development strategies.

Table 1-1. Examples of strategies and development methods

Technical Strategy	Development Method	Development Strategy	Delivery Strategy
Adaptive/Agile: flexible development cycles to adapt requirements and prioritize development tasks based on frequent stakeholder feedback	**Scrum**: The requirements for each sprint are fixed and flow through design, implementation, and test, with connections between design and test [24]..	**Iterative**: Features and components developed in one sprint may be refined in future sprints.	**Multiple**: A functional, and potentially releasable, system is delivered at the end of each sprint
Sequential Development: system development progresses through a defined series of phases	**Waterfall**: The requirements are fixed at the beginning and tested at the end [4]	**All-at-once**: A single iteration of the system is delivered at end of development	**Once**: A single delivery at the end of system development
Successive Deployment: incremental delivery of capabilities	**Spiral**: Risk based system assessment performed prior to initiation of each increment [25]	**Incremental**: Each spiral builds upon the previous spiral to deliver additional capability	**Multiple**: A delivery is released at the end of each increment

Development strategies may be all-at-once, incremental, or iterative. All-at-once strategies deliver the entire system at one time. Incremental strategies start with a known set of high-level requirements, which are then assigned to stages of development (increments). Each increment delivers part of the overall capability [11]. An overall, connected view of the requirements is critical in this development strategy, as the decisions made on early increments will result in constraints on the capabilities delivered in future increments. For example, the Space Shuttle delivered its engines in an early increment. Once the thrust output of the engines and the number of engines was set, the maximum mass of the rest of the system was fixed and could not be changed, limiting the overall payload capacity of the Space Shuttle [26].

Iterative strategies accept that the requirements have greater volatility and are not completely known at the start of the system development [26]. As the development progresses, the requirements for future iterations are generated from sources such as user feedback, technological advances, and the results of previous iterations. Throughout the system development, several requirements which may have been unclear during the initial iterations become more defined through the future iterations [27]. Iterative methods are also referred to as evolutionary methods and Agile methodologies use iterative development strategies [11].

Iterative development strategies often "stress delivering the most value as early as possible and constantly improving it throughout the project lifecycle based on user feedback" [11]. With flexible processes, the system developer can react to changing user requirements and needs as the system development progresses. However, there will still be uncertainty about whether requirements will change and the impact of those changes [5]. Whether using incremental or iterative methods, the initial development cycles will impose constraints on future work and the work done in each increment or iteration "limits design options for subsequent developments"

[26]. There is the risk that the system, focusing on delivering value to the user, implements the 'easiest' set of components, which break when future changes are required [25]. Understanding the risk to the system requires understanding the connections between iterations.

Within a single iterative design cycle, the requirements for the iteration are selected based on the current state of the system capabilities and the priorities of the stakeholders. The requirements for each iteration define the iteration design. The iteration design is the implementation of the requirements which provides the capabilities to the user. The design may augment or constrain the current system capabilities depending upon the selection of requirements and the success with which they were implemented. Based upon the updated capabilities delivered by the implemented design, the requirements for the next iteration are prioritized and selected, and may include rework. This pattern, illustrated in Figure 1-1, repeats for each iteration.

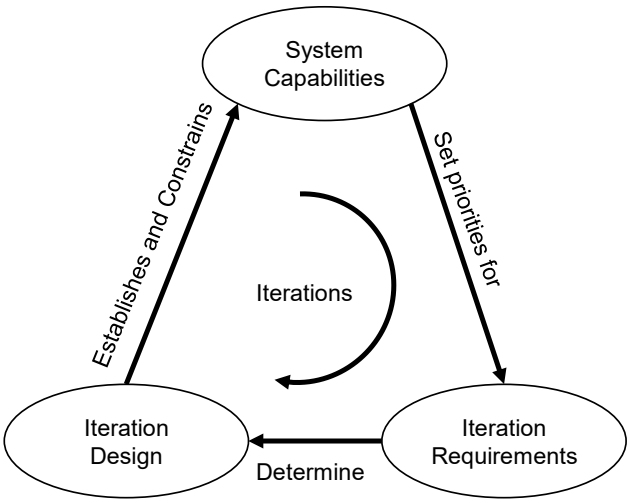

Figure 1-1. Iterative development cycle

Each iteration delivers a subset of the system requirements. The process of establishing the requirements for each iteration is known as release planning. Release planning consists of determining which features to implement in each iteration or series of iterations, based on a prioritization scheme [28]. Release planning may occur once, such as in the case of incremental

development with fixed requirements, or may occur prior to each iteration. Traditional systems engineering methods, such as the Waterfall model [4], spend significant time on upfront release planning – attempting to determine a full and complete set of requirements prior to beginning work. Agile approaches spend less time on upfront planning and more time iterating on the system designs and in integration [29]. Agile approaches can focus on selecting requirements that maximize value delivered to the user in the shortest amount of time [30], enabling the rapid delivery of capability and decreasing the time to market.

The prioritization scheme used to select the requirements for each iteration needs to balance the user value proposition with the limitations that may be imparted on the future development of the system. Many existing prioritization schemes give insufficient weight to non-functional requirements and to the dependencies between functional requirements and non-functional requirements. Similarly, the dependencies between functional requirements evolve with each iteration and traditional risk assessment approaches do not account for the risk associated with these dependencies [31]. If dependencies between functional and non-functional requirements are not considered during the release planning activities, then there is the chance that the value-based prioritization will result in a state where future iterations are impossible to complete, which will require the rework of already completed features or the failure of the system. Managing the dependencies between iterations is critical to the success of the system.

Agile development practices, which are designed to be highly iterative and flexible, first appeared in software engineering. Systems engineering is closely related to software engineering and has a history of borrowing processes form software engineering, including development methods such as Waterfall [4], Spiral [22], and Agile [13]. The developers of large systems are moving towards more flexible techniques, including the increased use of iterative and incremental

methods, in efforts to increase both the speed of delivery and the utility of the developed system. The following sections demonstrate how specific organizations are transitioning to the use of incremental and iterative methods.

1.1.1.1 United States Department of Defense

The United States Department of Defense (DoD) is one of the largest acquirers of complex systems and systems of systems. These systems often require development of new capabilities, which result in significant uncertainties in the ability to deliver on time and on budget. The DoD has traditionally specified the requirements at the start of the program. Due to the contracting mechanisms used, changing these requirements as the system development proceeds has the potential for introducing large schedule and cost increases [32]. The DoD is attempting to reduce its cycle time, defined as the time from the start of the program until the program fields its initial operational capability (IOC) [33]. The DoD is continuously seeking to improve the quality of the systems that it produces, for example, by adopting methods that include users more frequently during the development process [11]. Recent DoD systems engineering guidelines emphasize the use of iteration throughout the system development processes [34]. Government Accountability Office (GAO) reports have explicitly stated that "programs can put themselves in a better position to succeed by implementing incremental acquisition strategies that limit the time in development" [35]. Examining the history of the DoD acquisition process, shown in Figure 1-2, provides an overview of the progression towards more iterative and incremental methods.

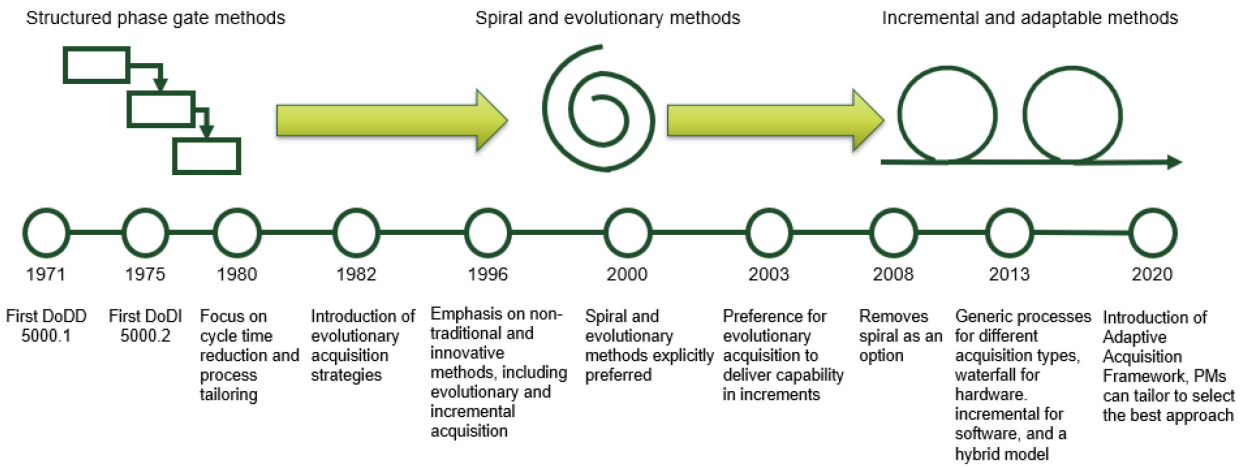

Figure 1-2. History of the acquisition methods in the DoD, sourced from [12] [36] [37] [38] [39] [40]

The DoD acquisition process is controlled by two primary documents: DoD Directive (DoDD) 5000.1 and DoD Instruction (DoDI) 5000.2. The DoD first issued DoDD 5000.1 in 1971 and DoDI 5000.2 in 1975 as a method to control rising defense acquisition costs. The documents described three phases of major defense programs – initiation, full-scale development, and production/deployment with decision gates in between each phase. They also emphasized early test and evaluation, using existing commercial hardware, and making tradeoffs between cost, schedule, and performance where practical. The initial revisions focused on structuring the acquisition process and reducing the cycle time, including the introduction of evolutionary acquisition strategies in 1982 and emphasizing non-traditional and innovative methods, including evolutionary and incremental acquisition, in 1996. However, these processes met with limited success [36].

In 1998, the average DoD cycle time had increased approximately 25% relative to 1969. By contrast, the automobile industry had reduced cycle time by an average of 50-75% during the same time period. The long cycle times resulted in the DoD fielding of out-of-date components and systems that no longer met the end user needs [41].

In part to address the problem of extended cycle times, the DoD released an updated version of DoDI 5000.2 in 2000, titled the "Operation of the Defense Acquisition System" [12]. This version replaced previous versions and also replaced MIL-STD-498, issued in 1994, which had introduced changes to the approved software development process to improve "compatibility with incremental and evolutionary development models" [42]. The 2000 version of DoDI 5000.2 stated an explicit preference for the evolutionary approach over waterfall methods. Spiral development is also explicitly called out as the preferred method for software development [12].

In 2002, the DoD released DoD 5000.2-R "Mandatory Procedures for Major Defense Acquisition Programs (MDAPS) and Major Automated Information System (MAIS) Acquisition Programs" [43]. This regulation allowed the program manager to choose between a single step approach and an evolutionary approach to program acquisition. However, for the software portions of the system, the regulations required the program manager to plan a spiral development process for both evolutionary and single step acquisition strategies. The 2003 release [37], now termed DoDI 5000.02, stated a preference for evolutionary acquisition to deliver capabilities in increments. The program manager (PM) could choose between spiral development and incremental development, although the instruction highlighted the reduction in risk and decrease in cycle time that can accompany incremental development. Other substantial changes included a change in focus from requirements-based processes to capabilities-based processes and an increased focus on achieving interoperability between the military services through more structured systems architectures [44]. The 2008 version [38] removed spiral development as an explicit option in the evolutionary acquisition approach.

In 2013, the references to the evolutionary approach were removed and a generic process was defined for different types of acquisitions, to include hardware-dominant, software-dominant, and

hybrid systems [39]. The generic approach follows a linear, non-iterative development process. The hardware-dominant program follows a traditional waterfall process. The software-dominant program either modifies the traditional waterfall program by releasing several builds or uses an incrementally fielded process, where a separate request for proposal (RFP) is released for each increment and the increments overlap. The hybrid model mixes the two based on whether or not hardware or software is dominant. In the hardware dominant program, the multiple software builds are combined with the hardware waterfall cycle. In the software dominant program, the hardware components are integrated into the software increments. In 2020, the DoD released a new version of Instruction 5000.02, which established the Adaptive Acquisition Framework (AAF) [45]. The AAF reinforces the desire for PMs to tailor their acquisition methods to the capability and need for which they are acquiring the system, giving additional freedom to the PM for rapidly fielded capabilities. This version of the instruction integrates agile development processes for software acquisitions, but does not mandate incremental, evolutionary, or spiral development processes for other acquisition types. The AAF allows PMs to use the best approaches to meet their needs, instead of requiring specific approaches for all systems.

The continual and ongoing revision of the DoDI 5000.02 guidelines illustrates that the DoD has not solved the problem of delivering systems on-time and on-budget that meet the user's needs. The use of spiral development on programs has been both successful, such as with the Predator Unmanned Aerial Vehicle Program (UAV), and unsuccessful, such as in the case of the Navy's Littoral Combat Ship (LCS) [41]. However, the changes in the directives and instructions show a clear path of moving towards incremental and iterative methods with the goals of reducing cost and time to market and delivering products that are of greater utility to the end user.

1.1.1.2 Space Development Agency

The 2020 version of DoDD 5000.01 states that Defense Acquisition Systems must be responsive such that they can be "deployed to the operational community as soon as possible" and that "approved, time-phased capability needs, matched with available technology and resources, will enable incremental acquisition strategies and continuous capability improvement" [40].

The Space Development Agency (SDA) was created in 2019 to "accelerate the development and fielding of new military space capabilities" [46]. To do so, SDA is creating the Proliferated Warfighter Space Architecture (PWSA), which is a proliferated low Earth orbit (pLEO) constellation of satellites to provide data transport and missile tracking capabilities. SDA is incrementally delivering capabilities within two-year tranches, where a tranche is one iteration of the PWSA. SDA adopted this development strategy to deliver 'good enough' capabilities to the end user sooner as opposed to delivering the 'perfect' capability too late [47].

SDA epitomizes the instructions within the 2020 version of the DoDD 5000.01, developing incremental capabilities that will be deployed operationally as soon as possible. The SDA development process leverages existing commercial capabilities and the rapid timeline ensures alignment with end user needs. Each delivered tranche will improve upon the previous set of tranches and will include updated technology, reducing the impacts of stale technology on end user capabilities [46].

1.1.2 Technical Debt in Iterative and Incremental Development

In iterative and incremental development, the decisions made in early iterations can become constraints on the future design [48]. Managing technical debt requires making decisions that trade immediate value against long term cost. Focusing on value, such as in agile approaches, will lead to choices that accumulate technical debt. Focusing on cost, such as in traditional phased

approaches, will take longer to realize value. Technical debt management requires balancing these factors throughout the release planning cycle [16]. Paying off accrued technical debt requires time and cost that could have been spent on adding new features to the system, and therefore represents an opportunity cost to the system developer [49]. Therefore, opportunities for paying down technical debt within a release need to be prioritized along with the development of new features, in context of the total cost and benefit to the system.

The timeline of technical debt [50] within a system is shown in Figure 1-3. Initially, technical debt can be beneficial to the system, as taking on the debt can enable progress towards meeting system milestones. The developer may be unaware of the debt after it occurs (T1), the state of "blissful ignorance." Once the technical debt is discovered (T2) it may still be an asset to the system. However, at some point, the technical debt reaches the "tipping point" (T3), where it becomes a liability to the system, causing more harm than good. The tipping point occurs when the work involving the technical debt item becomes more difficult to perform or the fees reach an intolerable level. After this point, the system developer can choose the path forward (T4), indicated by the dashed lines in Figure 1-3. The developer can choose to repay the debt and enter remediation, removing the liability. If left unmanaged, the system is defaulting (shown as dashed orange lines) - technical debt continues to accumulate, eventually resulting in technical bankruptcy (T5).

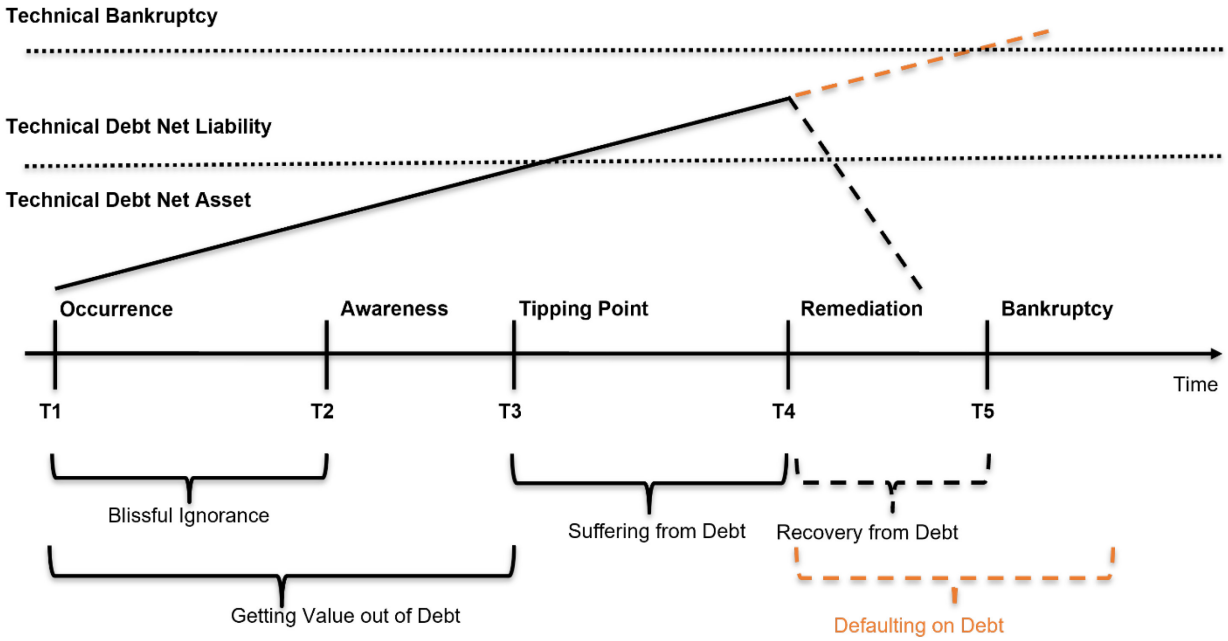

Figure 1-3. Technical debt timeline, adapted from [50]. Dashed lines show potential paths.

To include technical debt as a factor in release planning requires understanding the current state of technical debt within the system, the cost of paying off the debt now, and the probable impact of that technical debt in the future [51]. However, researchers and practitioners have found it challenging to identify and communicate the technical debt within the system [52] and also to estimate the cost and impact of the technical debt [53]. Furthermore, successful release planning also requires the ability to proactively identify technical compromises that may result in the introduction of technical debt. Identification of technical debt's introduction at the time of the compromise enables a complete assessment of the long-term costs of the short-term decision. These long-term costs need to be associated with the satisfaction of stakeholder needs to determine if the system will be successful.

An overall process for managing technical debt within software has been characterized as not well defined [54]. The sparsity of research on technical debt within systems engineering [19] implies that such a process is not well defined for systems engineering either. The basic steps to managing technical debt would include identification, measurement, prioritization, repayment, and

monitoring of the technical debt [55]. This technical debt management process is shown in Figure 1-4.

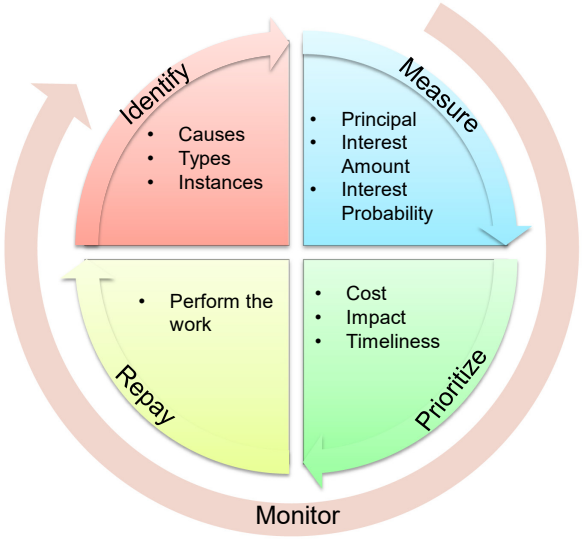

Figure 1-4. Technical debt management cycle

The technical debt management process is presented here as a continuous activity. When an instance of technical debt is identified, the potential impact is measured and the technical debt item is prioritized against the other technical debt items and the features to be implemented. During a release planning event, the technical debt item may be selected for repayment at which point the work is completed. Whether or not the debt is repaid, the technical debt item is still monitored for any changes. The system is continuously monitored for new instances of technical debt, both looking for new instances in the system and assessing the potential impacts of design decisions.

1.1.2.1 Identifying Technical Debt

Technical debt management starts with identifying the current debt within the system by making a list of the technical debt items (TD Item). A TD Item is "One atomic element of technical debt connecting: (1) a set of development artifacts, with (2) consequences on quality, value and cost of the system and triggered by (3) some causes related to process, management, context, or business goals" [56].

Technical debt can be inserted into a system either intentionally or unintentionally and in either a prudent or reckless manner, as Fowler [57] identified with his technical debt quadrant. Tom, Aurum, and Vidgen [52] identified different groups of technical debt based on McConnell's [58] classification. These two viewpoints are combined in Figure 1-5. Negligent technical debt is an example of deliberate and reckless technical debt. It is incurred when a developer knowingly inserts technical debt into the system without any plans to repay the debt. For example, a developer may intentionally skip test cases to save schedule without any plans to run the tests at another time. Strategic technical debt is also deliberate, but is inserted into the system with a plan to pay it down (a prudent decision) [52]. The previous example can be turned from negligent to strategic (or reckless to prudent) by included a reduced version of the test in a future test, thereby ensuring that the functionality is tested and still saving the initial schedule.

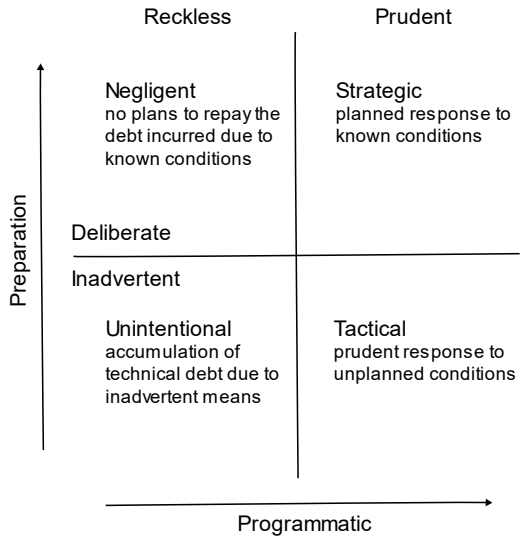

Figure 1-5. Technical debt quadrant, adapted from [57] and [52]

Inadvertent debt is debt which the developer does not know that they are incurring. Unintentional debt is unknown to the developer at the time that it is incurred and can only be discovered later. Therefore, unintentional technical debt cannot be prudent – there is no plan in place to account for repaying the debt. Tactical technical debt lies in the prudent/inadvertent

quadrant of Figure 1-5. This type of debt is described by Fowler as "Now we know how we should have done it" [57] and Tom et al. define tactical technical debt as short-term technical debt that is taken on to meet a milestone [52]. Here, tactical technical debt is defined as that which involves a prudent response (a plan to repay the debt) due to unplanned conditions. Whereas strategic debt has long-term consequences in response to known decisions, tactical debt is short-term in nature and due to issues, such as late discovery of an issue that implements a work-around to make the release.

Figure 1-5 also identifies that the occurrence of the different groups of technical debt can be based on the developer's preparation and on programmatic decisions. Technical debt can move from inadvertent to deliberate based on the developer's preparation. Inadvertent debt is often due to an underprepared developer – they do not know that what they are doing will cause problems. A prepared developer understands the results of their actions and therefore could plan accordingly. Reckless and prudent debt are due to programmatic decisions – how the debt is planned (or not planned) to be handled within the system development cycle.

1.1.2.2 Measuring Technical Debt

In keeping with the financial debt metaphor, TD Items are often assigned values of principal and interest. The concept of technical debt principal has attracted various definitions, including the cost or effort to fix the TD Item [59] [60] [61], the effort to rework the artifacts such that they have their optimal implementation [62], and the savings that the original shortcut provided [56]. Generally, principal relates to the cost to implement the 'proper' solution instead of the shortcut that resulted in technical debt.

Interest on technical debt relates to additional work or cost that the system incurs due to the presence of technical debt [54]. This additional effort can be due to lower maintainability [63] [60]

[62] and the increased level of effort required to restore the system back to its debt-free state [61]. Interest compares the level of effort over time to the level of effort that would have been required had a different decision been made initially [64].

Technical interest is a probabilistic concept composed of both the interest amount and the interest probability. The interest amount is the estimate of the amount of extra work that will be needed due to the presence of the TD Item [51]. The interest probability is the likelihood that the technical debt will cause additional work and may be time-dependent [51] [60].

The measurement of technical debt has proven to be a difficult problem. Measuring technical debt requires the ability to estimate the principal, interest amount, and interest probability [65]. The computed value of the technical debt must then be converted to terms that can be easily understood in the course of the system development. Nord et al. [16] calculate the cost of a release as summation of the cost of new features (the implementation cost) and the cost of rework, where the rework cost is a function of the interdependencies within the system and the change propagation throughout the release. Abad and Ruhe [66] use real options analysis to determine the net present value of requirements decisions. Ampatzoglou et al. [62] identify the use of other techniques, such as portfolio management, value-based approaches, and non-financial implementations. Seaman and Guo [67] assign values of high, medium, and low to the principal, interest, probability, and interest amount for each TD Item when it is created with more detailed estimates performed only when needed. Curtis et al. [49] estimate technical debt principal in the system as a function of the number of problems that must be corrected and the time and cost to correct each one. Ampatzoglou et al. [68] developed a quantitative model relating the size of the interest and the principal for a given TD Item. These various methods show that a consensus

method for measuring technical debt has not been reached and that it remains an open research question.

1.1.2.3 Prioritization, Repayment, and Monitoring of Technical Debt

Once each TD Item has been measured and both its principal and interest determined as functions of time, then the TD Item can be prioritized. Removing TD Items will often compete with the ability to add new features and to correct defects in the system, however, there is not a consensus approach on how to prioritize TD Items [69]. Considerations during prioritization include the cost to repay a TD Item, the impact of the TD Item if it is not repaid, and the timeliness with which the TD Item needs to be repaid prior to the impact being realized. Repayment involves performing the work to correct the TD Item and monitoring is the process by which TD Items are tracked, such as through a technical debt manifest [70], by tracking changes in principal and interest, and through cost-benefit analysis of repaying each TD Item [71].

1.2 Reflections on Technical Debt in Systems Engineering

Technical debt, while originating in the field of software engineering, also impacts the field of systems engineering. The United States Government Accountability Office (GAO) "provides Congress, the heads of executive agencies, and the public with timely, fact-based, non-partisan information that can be used to improve government and save taxpayers billions of dollars." [72] As such, the GAO provides reports on the state of development of several complicated systems and systems of systems procured for the United States Government. Table 1-2 shows several systems assessed by the GAO with issues that can be associated with technical debt, even though the term technical debt is not explicitly used in all the reports.

Table 1-2. Examples of technical debt within United States Government systems, as assessed by the author

ID	System	Org	Technical Compromise	Long-term Impact on System Health
1	James Webb Space Telescope (JWST) [73]	NASA	Reduction of ability to work on other NASA projects to free up funds for JWST	Extra funding sources required for other NASA projects that faced risk of cancellation
2	Multiple [73]	NASA	Contractors authorized to work before final contract agreement and requirements definition reached	Poorly defined requirements, creating requirements debt that increases the risk to program cost and schedule
3	Artemis [73]	NASA	Failure to document decision-making tools	Inability to track mission success
4	Mobile User Objective System (MUOS) [74]	DoD	Development and deployment of compatible user terminals not prioritized	Advanced capabilities of satellite system are unused and satellite lifetime considerations require the purchase of additional satellites
5	MUOS [11]	DoD	Failure to modernize ground system software	Obsolescence of 72 percent of the software in 2014, failure of operational test and evaluation in 2015 due, in part, to cybersecurity concerns in the ground system
6	GPS Modernization [74]	DoD	Required use of the more secure military (M-code) GPS signal prior to availability of the M-code cards	Use of M-code requires upgrades to systems that receive a GPS signal. Delays in the production of the M-code cards resulted in the extension of modernization efforts across the DoD, affecting the schedules of multiple systems
7	F-35 [75]	DoD	F-35 aircraft purchased prior to certification	Simulator delays resulted in the postponement of operational testing and the start of full rate production. F-35 aircraft continue to be purchased, however, increasing the risk of higher retrofit costs if there are issues.
8	Joint Space Operations Center (JSpOC) Mission System (JMS) [11]	DoD	Incomplete software requirements and lack of opportunities for user feedback	JMS was found not operationally effective during its operational test and was cancelled in 2019. [76]

Table 1-2 shows a wide range of long-term impacts from short-term decisions. Some of these decisions were intentionally made to improve technical performance (the transition to the M-code GPS signal), save cost (F-35 simulator transition), and save schedule (authorization of contractors to start work prior to reaching a final contractual agreement). Other issues were likely the result of the accumulation of unintentional decisions, such as failing to adequately document the decision-making processes on Artemis and not prioritizing the user terminals on MUOS. In the case of JMS,

the build-up of technical debt was so severe that it resulted in technical bankruptcy and program cancellation.

The programs cited in Table 1-2 can be placed within the technical debt quad chart as shown in Figure 1-6. Program 1, the JWST, is an example of deliberate and prudent debt. The decision to sustain the telescope was made intentionally, with a plan to take funding from other programs and to gather extra funding from congress. Likewise, the GPS upgrade, program 6, is an example of prudent debt – there was a known impact on the systems to upgrade to the new M-code system. However, the debt was inadvertent, due to the delays from unforeseen supply chain issues. Program 3, the failure of the Artemis program, is an example of inadvertent reckless debt. The program failed to document their processes, but this was not an example of a deliberate decision. Rather, this is an example of a mistake. The other programs, are all examples of reckless, deliberate technical debt – making decisions that affect the long-term health of the system without a plan to pay down the debt in place. Program 7, the F-35 simulator, falls into this category due to the decisions to purchase aircraft prior to certification, which increases the risk of the need to retrofit the aircraft. Program 5, the MUOS failure to modernize the ground system was a result of intentionally sticking with the requirements and obsolete software, instead of updating the requirements, with no plan for modernization.

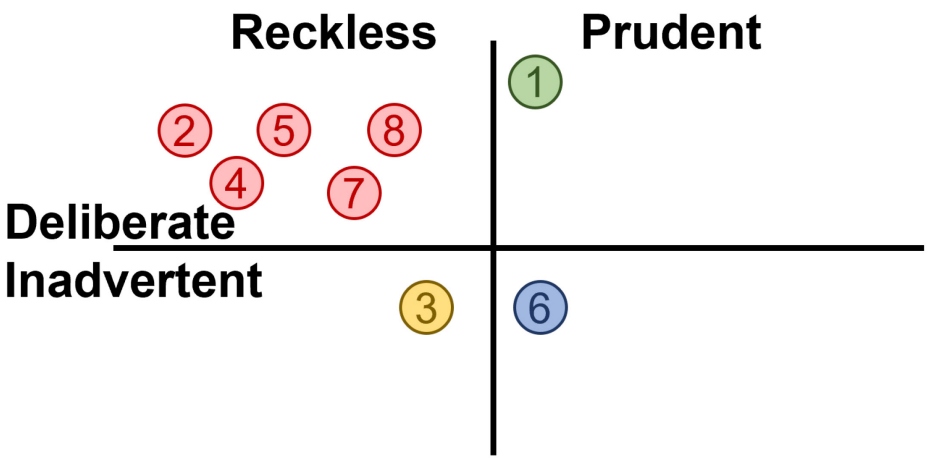

Figure 1-6. U.S. Government program technical debt categorized notionally and with the judgement of the author

These programs demonstrate that decision making around, and management of, technical debt is clearly an issue within the scope of systems development and is not contained solely to the realm of software development. As systems engineering moves to more agile and rapid processes, technical debt will become a larger issue. Although the term technical debt is used within the context of systems engineering [77] [78] [34], there is not a common ontology that can be used to discuss the problem [79]. There is also a lack of empirical evidence documenting the occurrence of technical debt within the systems engineering lifecycle and the traditional lifecycle models do not provide adequate tools for managing technical debt [19].

1.2.1 Technical Bankruptcy in Systems Engineering

Left unattended, technical debt can grow within a system, eventually forcing the system into a state of technical bankruptcy. In this state, the system can no longer proceed with its lifecycle without first repaying some or all of the technical debt [21]. Technical bankruptcy can occur when the project budget or schedule is exceeded [63] or when the technical debt has reached a point where the system can no longer support future development [60]. Technical debt that accumulates within the system can delay updates and new versions of the system, resulting in unfilled feature

requests [80]. If a technically bankrupt project does not pay down its technical debt, then the project risks cancellation [62].

Within the US DoD, programs that experience significant cost and/or schedule overruns experience a Nunn-McCurdy breach. In 2009, Congress passed legislation requiring that programs that experience Nunn-McCurdy breaches be terminated unless the Secretary of Defense provides written certification of the program to Congress. Causes for Nunn-McCurdy breaches include overly optimistic assumptions, misunderstanding of requirements, and changes to requirements [35]. A Nunn-McCurdy breach is a realization of technical bankruptcy.

The Joint Space Operations Center (JSpOC) Mission System (JMS) is an example of a system where technical bankruptcy led to system cancellation. JMS was designed to provide space command and control and situational awareness capability for the Air Force. JMS planned to use an incremental delivery method with three increments. Increment 1 planned to provide the basic structure for the program. Increment 2 planned to add capabilities to the user such that JMS could replace the legacy system. Increment 3 planned to augment the Increment 2 capabilities with data from highly classified programs. The program was started in 2009 and Increment 2 was scheduled to deliver by the end of fiscal year 2014 [81]. The Air Force Operational Test and Evaluation Center (AFOTEC) tested JMS in 2016 [82], 2017 [83], and 2018 [84] and found the following results:

- The risk of interoperability with other systems, specifically the Space Fence, was increased due to the late delivery of JMS Increment 2 and the level of interoperability testing was insufficient (2016 and 2017)
- Service Pack (SP) 7 was not operationally tested because it would not be used for mission critical functions (2016)

- The SP9 developmental test campaign found numerous critical deficiencies, reducing the scope of the operational delivery. The resulting delivery was “not operationally effective or suitable for its Space Situational Awareness (SSA) mission” [84] (2018)
- Many problems could have been prevented with better organization and communication between the program office and the development team throughout the entire system lifecycle (2018)
- The SP11 schedule did not provide time to fix the issues with SP9, address lessons learned from SP9 testing, or account for constraints due to SP9 and SP11 concurrent development. These issues caused AFOTEC to determine that the SP11 schedule was not executable [84] (2018)

At the time it was cancelled, JMS was 42% over budget and three years behind schedule [81]. The cancellation of JMS can be viewed as a result of poor performing program. However, when viewed from the technical debt perspective, there are clear indications that the technical debt in the system built up until the system became technically bankrupt. The author’s assessment of the technical debt associated with the JMS system is shown in Figure 1-7, with green indicating the technical debt principal, orange indicating the technical debt interest, and the blue line indicating the technical bankruptcy threshold – the state at which project replanning to address the technical debt is required.

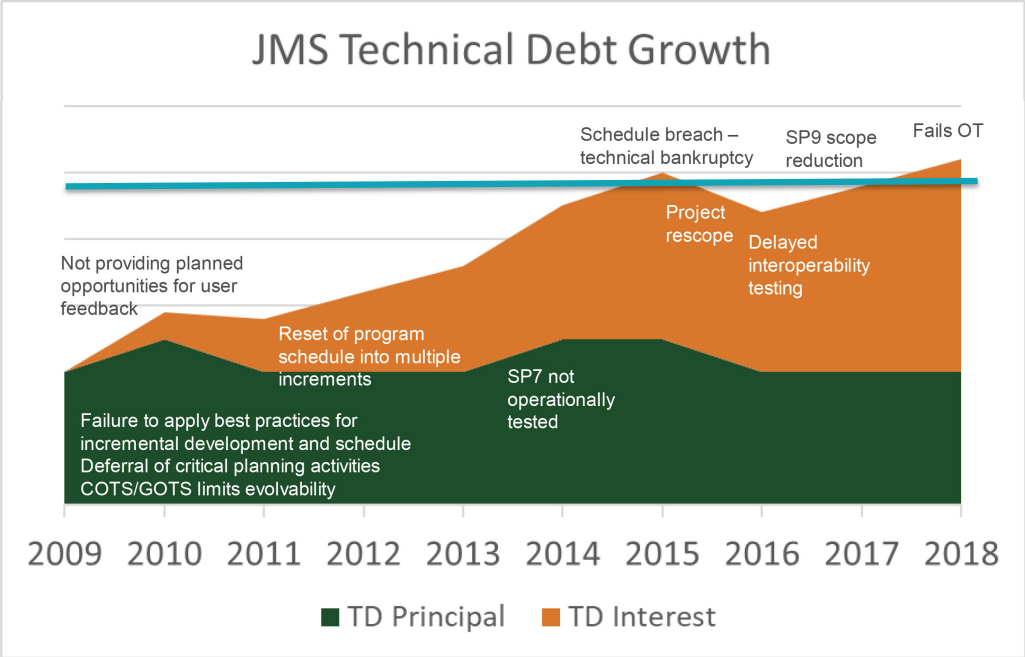

Figure 1-7. Author's assessed growth of technical debt principal (green) and interest (orange) within the JMS program, based on GAO reports [11] [82] [83] [84] [85]

JMS was initially planned as a single acquisition that would be delivered through five releases over a five-year period, from 2011 to 2016. The GAO assessed that while the capability would be delivered over five releases, it was not applying best practices for incremental development. Specifically, the JMS system "plans to proceed without knowledge of all critical technologies and deferral of other planning activities" [85]. While using an incremental delivery method, the JMS developers were not adequately looking ahead to identify critical technologies and capabilities that would support future releases. Therefore, the design had the risk of being dependent upon capability that would not exist [85]. This situation is an instance of architectural technical debt which is due to architectural decisions that produce insufficient quality in the system [54]. The JMS developers chose not to fully develop the system architecture, specifically the dependencies between components and the capabilities required of each of those components. In 2011, JMS took the GAO recommendation and restructured to release in multiple increments, repaying some of the architectural technical debt [11].

JMS planned to use only existing software, both commercially available and government-provided [11]. However, the lack of a fully defined architecture resulted in a failure to identify the insufficiencies of the existing software to meet all needs of the system. The use of the existing software incurred additional technical debt due to the limitations on the ability of the system to evolve [86]. The delays in interoperability testing with Space Fence resulted in integration technical debt – an increased effort required to connect systems and components together [79]. These sources of technical debt accumulated in the system, resulting in additional development and accompanying schedule delays [11]. If the JMS program office had performed an initial, detailed trace of the critical capabilities, they would have seen that the available software tools would need to be modified and could have appropriately included those modifications in both the schedule and the budget.

In 2015, the GAO assessed the JMS schedule and identified several deficiencies, including artificial start dates and illogical connections. JMS did not follow the GAO's best practices for maintaining schedules and the GAO found that the resulting schedule was insufficient to determine the ability of JMS to meet its schedule milestones [87]. The poor schedule practices are an example of reckless and deliberate technical debt: JMS made a deliberate choice not to follow the best practices (a reckless decision). Following schedule best practices may have resulted in early identification of risks to delivery. Instead, the delivery of the system was substantially delayed, resulting in a schedule breach in 2015 [11]. This schedule breach is an instance of technical bankruptcy and resulted in a realignment and rescoping of JMS.

JMS failed its operational test in 2018, being declared neither effective nor suitable for its mission [84]. Within the DoD, operational tests evaluate the system under real-world conditions as used by operational users to determine factors such as effectiveness, reliability, maintainability,

and usability [88]. Developing a system that will pass operational test requires the inputs of the operational users to ensure that their needs are being met. JMS planned to involve users early in the development process and to keep them continually engaged. However, it failed to do so in practice, which is another example of reckless and deliberate technical debt. JMS also did not provide users the opportunity to provide feedback on working software components during development [11]. The decision not to include users in the development process accrued domain debt – the misalignment between the application and the domain in which it will be used. Domain debt interest is paid "in terms of user satisfaction and usability" [89] and a significant amount of domain debt will result in a system that is not usable. The domain debt and other technical debt increased to the level that the JMS project was cancelled in 2019.

1.2.2 State of Technical Debt Research in Systems Engineering

Given the importance of the systems described above and the ubiquity of technical debt as a metaphor in software engineering (a closely aligned field), it would be reasonable to assume that technical debt is a well-researched topic within the field of systems engineering. However, a systematic literature review [19] shows that there is little published research that uses the terminology of technical debt in conjunction with systems engineering. The concepts of technical debt are identified, but there is not a clear and well understood definition [79] or associated ontology. A consistent ontology enables clear communication and discussion of common problems and solutions. Agreement upon a common ontology will lead to the establishment of common metrics to measure technical debt. Once technical debt can be measured, then control systems can be put in place to manage and handle technical debt as it grows beyond tolerable thresholds. Control systems aid communication with the stakeholders about the status of the system and can give indications that the system is approaching technical bankruptcy. However, effective controls

need to be based on a clear understanding of the parameters that are measured [90]. A concise and clear definition of technical debt, its sources, and associated terminology needs to be developed within systems engineering.

Similar to the lack of definition of technical debt, there is not a concise process for identifying technical debt within a system. Sangwan et al. identify the need to determine the dependencies of architectural elements within a system [91], however they do not provide detailed information on the process by which these dependencies are identified. Simply citing a dependency is not sufficient – there needs to be a deeper understanding of how the components of systems are interconnected and how the technical debt may propagate through the system. Technical debt can be contagious: the technical debt and its impact can spread throughout the system, including hidden effects [92]. The technical debt introduced in one part of the system can impact other parts of the system and can accumulate through the reuse of components [80]. The resulting impacts may not be known to the system developers. For example, if the power system of an electric car is changed to use cheaper batteries with a faster delivery cycle, how does that impact the total range of the car and on-board fast charging hardware? Can the debt incurred in that decision be made up in a future development iteration, such as by redesigning the aerodynamic profile to increase range? Is it worth making that change? Understanding the impact of a change to the system requires fully understanding the interdependencies of the system components and the ability to complete a component based on the state of completion of its dependent components.

1.2.2.1 Accounting for Technical Debt in Release Planning Methods

The iterations within iterative development are planned through the release planning process, where the 'best' set of features are chosen to implement within each iteration [93]. Each iteration is referred to as a release, as it represents capability that could be released to the end user. Release

planning methods and optimization strategies exist to help the system developer decide when to implement features and requirements within iterative and incremental development [28]. In these development paradigms, there is often a conflict between early value creation and the minimization of later rework [91]. Several recent works provide methods for optimizing development paths to maximize value or to minimize rework.

Nord et al. [16] utilized design structure matrices (DSMs) to highlight the architectural dependencies of features and to model the cost of rework due to technical debt in release planning. Sangwan et al. [91] extended this work and developed a method using mixed-integer linear programming models to minimize total cost, maximize early value, and to find an optimal combination of features. Their model is based on the initial creation of a dependency matrix between the architectural elements and customer requirements and includes time-based discounts for value and estimates of rework costs. The model uses simplified cash flow to determine the rework effort and does not include significant uncertainty modeling.

Oni and Letier [28] created a model to analyze uncertainty for fixed-date release cycles. They utilized Bayesian probability to model the uncertainty of both the value of a particular feature and the effort required to complete it. The uncertainty is derived from expert opinion and is not mathematically modeled. The model reflects the uncertainty that a feature will be completed within the release to which it was assigned, giving the model a time-based dimension. The model assumes that work items are independent and does not account for technical debt. Schmid [94] provides a method for determining the release where a technical debt item should be repaid, based upon a probabilistic estimate of the occurrence of the TD Item and the impact of the TD Item.

In incremental and iterative methods, technical debt has the tendency to increase as schedule pressure mounts, especially towards the end of a fixed-time release cycle. The fixed-time release

cycles are often controlled by external factors, such as a company's marketing team promising a delivery date or a satellite launch vehicle's schedule. Therefore, the dates often cannot be compromised. Planning for these releases requires a careful assessment of what can and cannot be accomplished within the set timeframe and makes them more likely to accumulate technical debt in the rush to deliver. Planning the release, therefore, should account for the possibilities of technical debt building up in the system and also should include the capacity to pay down the technical debt in later releases. While several authors have created optimized release planning methods to account for the tradeoff between value and cost [94] [91] [28], none of them have fully integrated technical debt within their models. These models fail to account for the impact of technical debt and interest on a feature's development cost and value. The models also do not provide a way to proactively identify technical debt when a technical compromise is made. Finally, the models do not associate the ability to deliver capabilities with the satisfaction of stakeholder needs – it is assumed that if value is delivered then the stakeholders will be satisfied, but this does not account for the temporal component of the value delivery.

Therefore, release planning models that include technical debt must have the following capabilities:

- A thorough understanding of the type of architectural dependencies between components;
- The ability to proactively assess the probability of creating technical debt in future releases;
- The ability to estimate the probabilistic technical debt impact within a release as a function of time; and,
- The ability to estimate the capability of the project to repay technical debt as a probabilistic function of time.

These models need to capture the state of these characteristics of the system at each release planning event or each stage in the development process to account for changing external conditions.

1.3 Research Agenda

On the basis of the above reflections on the state of the field, we can understand that there is a need to manage technical debt throughout the system development lifecycle. Left unchecked, technical debt can accumulate within a system and drive the system to technical bankruptcy. The first step in managing technical debt within a systems lifecycle is to understand the current state of research into technical debt within systems engineering. There is little published work on technical debt specific to the field of systems engineering and the research that does exist does not provide significant empirical evidence to understand the role that technical debt plays within a system lifecycle [19]. Furthermore, there is no clear ontology for technical debt within systems engineering, evidenced by the lack of a common definition of technical debt [79]. A clear understanding of terminology is required to enable precise discussions about the impacts of technical debt on the system lifecycle.

Systems and projects must be able to identify the features and requirements which have significant dependencies later in the release cycle in order to understand the impact that changes to these components will have on the future state of the system. The features must be directly associated with the satisfaction of stakeholder needs in both the temporal and functional dimensions. However, a sufficient model tying this information together does not exist today. A sufficient model requires two separate parts: a description of the system, including the component-level dependencies among components that may incur technical debt, and a planning model that

takes into account the probabilistic occurrence of technical debt and its impact on the ability to deliver a system that satisfies the stakeholders in both the temporal and functional dimensions.

The first part of the model requires a description of a system focusing on the component-level dependencies. One way to describe the system is through the use of the Systems Modeling Language (SysML). This language produces a standard set of diagrams. The diagrams can be used to lay out the system, identifying connections, interfaces, and data flows [95]. While these diagrams are useful for identifying the structure of the system, they do not identify the development-level dependencies between the components, especially when one factors in the temporal development order. When considering the impact of technical debt, the temporal development order becomes critical. In a schedule-constrained release cycle, if one component does not finish on time or does not meet its performance requirements, it will impact the ability to complete subsequent components and still stay on schedule. In iterative development, each iteration can only build on what was completed in the previous iteration. The chronological order of development becomes critical to understanding the impact of technical debt.

Project schedules will give the chronological order of system development but they are not sufficient by themselves to identify occurrences of technical debt. Design structure matrices list the functional dependencies of components and can be ordered to show temporal dependences [96]. However, they do not account for partial completion and cascading impacts. A mathematical relationship needs to be established between the level of completeness of one component and the probability of completing future components. Establishing these relationships will identify the ability of one component development plan to recover from incomplete prior components. For example, consider the case where a car is being manufactured. Each part of the car is manufactured independently, but the assembly of the car itself cannot complete until all parts are delivered. If

the timeframe for the assembly of the car is padded with margin, then it may be able to handle late delivery of a part or two with little impact. If there was no margin on the assembly of the car, then a late part delivery would cause the assembly of the car to finish late. The magnitude of the impact depends on the component that was late. If the battery is delivered late, there may not be a large impact since it is easily installed at the end of the build. However, if the engine is late then its impact may be larger due to its integral role and the significant dependencies that other steps have on the assembled engine. Similar examples can be created to consider cost and performance implications. These dependencies become more important as you consider evolving a system through iterative build and release strategies.

Once the component-by-component dependencies are identified, then the development of the system can be planned, using release planning methods. These methods need to account for the potential occurrence of technical debt, and its subsequent impact on the ability to complete a release and future releases. Sangwan et al. [91] utilize design structure matrices to identify the dependencies and map the impact of change, however, they assume that all dependencies are equal. Schmid [94] provides a method to estimate the cost of a technical debt item, based on the probability of it impacting future releases. The total cost across all releases can be compared to the initial cost to fix the TD Item (the principal) to determine if the TD Item should be repaid in the current release. This method relies on the validity of the probability estimates and does not account for the interdependencies of the components. Oni and Letier [28] optimize a release plan based on the net present value of the release and include uncertainty conditions. Their method accounts for fixed-time releases. However, it does not include the impacts of technical debt, and how the failure to include or complete one component in a release may impact the rest of the release cycles. These

models need to be evolved to account for the interdependencies of the components and the impacts of technical debt.

This research proposes to identify the current state of understanding of technical debt within systems engineering based on both a literature review and a survey of systems engineering practitioners. Upon completion of this research, the results will be used to create a common ontology for technical debt within systems engineering to enable clear and precise communications. After establishing the baseline ontology, a process will be created to proactively identify potential technical debt within iterative and incremental systems development. This process will help to manage and predict technical debt within systems engineering and therefore will reduce the risk of technical bankruptcy. This work will be performed through investigation of the following research questions.

1.3.1 Research Questions

1.3.1.1 RQ1: How prevalent is the technical debt metaphor within systems engineering?

This question drives at the current state of knowledge of technical debt in the field of systems engineering. Answering this question will form a baseline level of knowledge and will inform the work going forward. RQ1 is addressed through the following subordinate research questions.

RQ1.1: What is the current state of research on technical debt within systems engineering?

This question seeks to understand the state of published research using the technical debt metaphor in systems engineering fields.

Task 1.1.1: Perform a literature review of technical debt within systems engineering to answer the following research questions derived from RQ1.1:

RQ1.1.1: What is the prevalence of the technical debt metaphor within systems engineering research?

RQ1.1.2: How is technical debt defined for systems engineering?

RQ1.1.2.1: What types of technical debt are associated with systems engineering?

RQ1.1.2.2: What are the sources of technical debt in systems engineering?

RQ1.1.3: Where in the systems engineering lifecycle does technical debt occur?

RQ1.2: How prevalent is the concept of technical debt and the use of the metaphor among systems engineering practitioners?

This question seeks to understand the use of the technical debt concept and metaphor by systems engineering practitioners. The use of terms and concepts among practitioners and academia are not always the same.

Task 1.2.1: Conduct a survey of systems engineers to gather empirical evidence

Task 1.2.2: Process the survey results to identify key sources and types of technical debt within systems engineering lifecycles and also the stages of the lifecycle which are most susceptible to technical debt

RQ1.3: What common ontology should be used to describe technical debt within the field of systems engineering?

Upon completion of the research associated with RQ1.1 and RQ1.2, develop an ontology to define technical debt and describe its likely sources within the systems engineering field. This ontology can then be used within the rest of this work to establish a baseline for communication.

The ontology of technical debt from software engineering will be modified based upon the results of RQ1.1. and RQ1.2.

Task 1.3.1: Create a definition of technical debt, principal, interest, and other terms as necessary that is tailored to the field of systems engineering. Existing definitions may be used if they are deemed appropriate to the field.

Task 1.3.2: Identify types of technical debt applicable to systems engineering. Based on the results of the survey, clearly identify the key types of technical debt, including their projected impacts.

1.3.1.2 RQ2: How can potential sources of technical debt be identified during the system lifecycle?

This question seeks to understand how to identify technical debt during system development. For technical debt to be a useful tool to guide system development, it must be able to be identified. While the identification of the exact sources of technical debt will vary from project to project, the establishment of a repeatable process will aid systems developers. RQ2 is addressed through the following subordinate research questions.

RQ2.1: How is technical debt identified within software engineering?

Task 2.1.1: Perform research to understand how technical debt is identified within the field of software engineering. This research includes identifying the types and sources of technical debt as well as processes and procedures for identifying technical debt currently in use.

RQ2.2: What process can be used to identify potential technical debt sources within systems engineering?

Task 2.2.1: Create a process to identify the dependencies between components, specifically focusing on the impact of the chronological development process on the ability to complete components and to deliver required capabilities. This process will identify qualitative methods for describing the relationship between components and will provide examples to aid the process. This work will be done in conjunction with the Space Development Agency, whose rapid, incremental development cycle will serve as a testbed for the process development.

1.3.1.3 RQ3: How can technical debt be used as a guide in release planning?

This question seeks to understand how to include technical debt within the context of release planning for iterative and incremental development cycles. Inclusion of technical debt considerations as a decision support system for release planning models should limit the risk of technical bankruptcy.

Task 3.1: Establish a quantitative model that is a companion to the qualitative process established in Task 2.2.1. This model will create the probabilistic relationship between components.

Task 3.2: Relate the quantitative model to the release planning process and demonstrate how it can be used as a decision support system for release planning.

1.3.1.4 RQ4: How can the process and model be used to avoid technical bankruptcy?

This question seeks to understand how to use the created models and processes as tools to avoid technical bankruptcy. Therefore, technical bankruptcy must be defined in the context of the model. Using this definition, the model and processes can be validated through real-world applications. Finally, since a model and a process are only useful if they can be communicated and used by others, a simplified version of the model needs to be created to enable conceptual discussions.

Task 4.1: Create a definition of technical bankruptcy within the context of the process and model outputs. This task will enable the users of the model to identify areas of concern when performing their release planning, such that they can get ahead of the problems before they occur.

Task 4.2: Utilize the developed process and model at the Space Development Agency and report on the results. The work at SDA will serve as a test ground for the process and model, enabling validation of the work.

Task 4.3: Create a simplified way of presenting and communicating the process and model. Systems models that are simple tend to be the most useful by the community. The Vee-model is easy to visualize and understand, increasing its utility. This task seeks to identify a simple way to communicate the key concepts behind the process and model to increase its general utility.

1.4 Structure of this Dissertation

The rest of this dissertation is structured as follows. Chapter 2 discusses RQ1, providing insight into the prevalence of technical debt within systems engineering and proposing a technical debt ontology for systems engineering. Chapter 3 answers RQ2, introducing the List, Evaluate, Achieve, Procure (LEAP) process as a method to proactively identify potential sources of technical debt within system development. Chapter 4 addresses RQ3, enhancing the LEAP process to include probabilistic models and demonstrating how this version can be used as a decision support system for release planning. Chapter 5 discusses RQ4, providing examples of using the LEAP process to proactively identify potential technical debt sources in real-world scenarios. Finally, Chapter 6 concludes the dissertation and provides recommendations for continued work.

This dissertation contains several works that have previously been published in journals or presented at conferences. In these cases, the works are reproduced in whole within this dissertation

and have been reformatted to meet the dissertation style guidelines; however, the content has not been altered.

CHAPTER 2 – TECHNICAL DEBT IN SYSTEMS ENGINEERING

2.1 Introduction

The previous chapter introduced the concept of technical debt and provided examples of its occurrence within large scale systems engineering projects. However, the reports on these project outcomes do not explicitly call out technical debt as a driving factor in project delays or failures. Instead, various project-unique causes are cited which required project-unique solutions. If, however, a common lexicon existed for technical debt, then relationships between these projects may be found and common solutions identified. The apparent lack of a common ontology for similar problems prevents the sharing of management and mitigation strategies [20]. Such an ontology is only enabled if there is abundant research on technical debt within systems engineering and if practitioners use technical debt terminology.

To assess these concerns, this chapter addresses Research Question (RQ) 1: *How prevalent is the technical debt metaphor within systems engineering?* RQ1 is decomposed into three subordinate research questions, which are individually addressed in the following sections:

- RQ1.1: *What is the current state of research on technical debt within systems engineering?*
- RQ1.2: *How prevalent is the concept of technical debt and the use of the metaphor among systems engineering practitioners?*
- RQ1.3: *What common ontology should be used to describe technical debt within the field of systems engineering?*

Addressing RQ1.1 will provide insight into the current state of technical debt research within systems engineering. This assessment of current research is foundational to the creation of an ontology by providing terms and definitions used with the academic community. Addressing

RQ1.2 will provide empirical evidence regarding the use of the technical debt metaphor. If it is commonly used amongst systems engineering practitioners, then there should already be a lexicon that can be leveraged to create a common ontology. These two research questions will establish the prevalence of the technical debt metaphor.

Assessing the prevalence of the metaphor is not enough to enable the sharing of technical debt mitigation and management strategies. Instead, a common ontology needs to be provided such that practitioners can clearly communicate ideas and problems. Addressing RQ1.3 will develop a proposed ontology, leveraging the results of the previous two research questions. The development of a technical debt ontology will foster greater communication by standardizing terminology and preventing the development of multiple descriptions for similar problems. If this ontology can be adopted by the larger community, it will become easier to collaborate on solutions to common problems.

2.2 RQ1.1: What is the Current State of Research on Technical Debt within Systems Engineering?

Assessing the current state of research into technical debt within systems engineering is a critical step to understanding the prevalence of the metaphor within the field. If the metaphor is commonly used and understood then the logical conclusion is that it should appear throughout the published body of knowledge. Therefore, a systematic literature review was conducted in early 2022, in accordance with Task 1.1.1: *Perform a literature review of technical debt within systems engineering*. The literature review was designed to determine an assessment of the prevalence of the technical debt metaphor based on the number of articles explicitly referencing both technical debt and systems engineering. Further analysis included assessing the definition of technical debt in systems engineering, the types and sources of technical debt in systems engineering, and the

occurrence of technical debt within the systems engineering lifecycle. This review identifies the current state of research in the field and provides a baseline for the rest of this dissertation. The literature review was published in *Systems Engineering* in 2023 [19] and is reprinted here.

2.2.1 Technical Debt in Systems Engineering – A Systematic Literature Review [19]

2.2.1.1 Abstract

The metaphor of "technical debt" is used in software engineering to describe technical solutions that may be pragmatic in the near-term but may have a negative long-term impact. Similar decisions and similar dynamics are present in the field of systems engineering. This work investigates the current body of knowledge to identify if, and how, the technical debt metaphor is used within the systems engineering field and which systems engineering lifecycle stages are most susceptible to technical debt. A systematic literature review was conducted on 354 papers in February, 2022, of which 18 were deemed relevant for inclusion in the study. The results of the systematic literature review show that the technical debt metaphor is not prevalent within systems engineering research and that existing research is limited to specific fields and theoretical discussions. This paper concludes with recommendations for future work to establish a research agenda on the identification and management of technical debt within systems engineering.

2.2.1.2 Introduction

Cunningham introduced the technical debt metaphor to explain the need for refactoring software to his management. He stated, "shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite" [17]. Taking on technical debt can benefit a project, as long as the debt is not allowed to grow. Much like traditional (financial) debt can be a source of risk when it is not repaid, technical debt can be a source of risk "when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that

debt" [17]. With this statement, Cunningham defined the concept of technical debt interest – the additional time required when working with code that was "not-quite-right."

Multiple secondary studies have researched the history of academic research into technical debt (TD), its types, and its causes in the field of software engineering. Li et al. [60] performed a systematic mapping study in 2015, finding a total of 94 studies on technical debt, with a total of four studies published prior to 2008 and at least 15 studies published per year since 2010. Verdecchia et al. [97] found 47 primary studies related specifically to architectural technical debt from 2009 to 2017, with the number increasing in the later years. Lenarduzzi et al. [69] found 44 studies published between 2010 and 2020 that apply to technical debt prioritization. Melo et al. [53] analyzed 61 primary studies published between 2010 and 2020 related to the use of technical debt requirements in software engineering, with 71% of those studies published after 2015. These secondary studies show that interest in technical debt in the software domain is increasing and specializing.

Software engineering has traditionally served as a reservoir of analogous processes and tools for systems engineering. Several systems engineering lifecycle models, including the Waterfall model [4] and the Spiral model [22], started as software development lifecycle models. Agile methods are becoming more prevalent in systems engineering as well [13]. These lifecycle models migrate from software development to systems engineering due to the similarities between the two disciplines: both involve the design and development of complex systems with multiple interfaces and components that must be managed.

Leveraging analogous tools and processes from software engineering into systems engineering also interjects some of the same problems. Decisions made in the course of system development to satisfy near-term objectives may not be optimal for the long-term health of the system, adding

technical debt to the system. Given the similarity between the fields, we might expect that systems engineering would see similar types and sources of technical debt as software engineering.

Published systems engineering research discusses similar concepts to technical debt, although it is not always explicitly defined as technical debt. Bahill developed a process to identify unintended consequences on other systems due to the decisions made during the development of a particular system [98]. De Lessio et al. created a process to "identify the main uncertainty drivers potentially affecting the future lifecycle performance of their systems" [99]. Etemadi and Kamp identify that decisions made during early strategy phases of a project can have impacts on schedule and schedule growth [32]. Bowlds et al. show that poor documentation leads to increased maintenance costs and effort in software and hardware systems and component obsolescence is a concern in both fields [100]. Boehm and Behnamghader discuss how inadequate systems engineering resources can lead to "exponentially-large amounts of TD due to poorly-defined interfaces, unaddressed rainy-day use cases and risks, and premature commitments to hopefully-compatible but actually-incompatible COTS products, cloud services, open-source capabilities, and hopefully-reusable components" [101]. Sharon et al. correlate systems engineering management and project management and identify that "careful management of the relationships between the product and the project is crucial to the successes of a project that aims to deliver a defined product" [102]. These sources all discuss critical elements of technical debt – unintended consequences, the impact of early decisions on long-term health of the system, and the need for adequate resourcing and careful decision making. However, only Boehm and Behnamghader explicitly use technical debt in their descriptions.

The lack of a common taxonomy within these sources motivated this work. While numerous secondary studies of technical debt exist within the field of software engineering, the authors are

unaware of any such systematic studies within the field of systems engineering. The goal of this paper is to determine the current state of research on technical debt within systems engineering through a systematic literature review.

This goal led to the definition of the following research questions (RQs):

- RQ1: What is the prevalence of the technical debt metaphor within systems engineering research?
- RQ2: How is technical debt defined for systems engineering?
- RQ2.1: What types of technical debt are associated with systems engineering?
- RQ2.2: What are the causes of technical debt in systems engineering?
- RQ3: Where in the systems engineering lifecycle does technical debt occur?

This paper is structured as follows: Section 2.2.1.3 presents the search methodology; Section 2.2.1.4 provides the results of the search; Section 2.2.1.6 presents a discussion of the results; and Section 2.2.1.6.4 presents the conclusions and opportunities for future work.

2.2.1.3 Methodology

This section defines the methodology used to conduct a systematic literature review of systems engineering technical debt, including the search strategy. The systematic literature review was conducted based on the guidelines in Kitchenham [103]. The results of the search are discussed in the following sections.

2.2.1.3.1 Search Strategy

The search string used to conduct the literature review included the technical debt phrase and any extension of the word system, as follows:

("tech* debt" AND "system*")

For systems that did not allow the use of wildcards, the search string was set to:

("technical debt" AND "system")

Searches were limited to the title and the abstract. The wild card character (*) was used to account for different derivations of technical debt and system (i.e., systems) in the results. The search was applied to the following online databases: *IEEE eXplore, Science Direct, Wiley Online Library,* and *Springer Link.* Due to the limitations of the *Springer Link* online search tool, the search on that site was limited to the title only.

Table 2-1 presents the inclusion and exclusion criteria that were applied as part of the search. Articles had to meet both inclusion criteria to be considered for the literature review.

Table 2-1. Literature review inclusion and exclusion criteria

Type	Criteria	Field(s)
Inclusion	Article discusses technical debt in context outside of software engineering	Title, Abstract, Full Text
Inclusion	Article identifies causes or types of technical debt in the context of an engineering system or identifies the appearance or impact of technical debt within a system life cycle	Title, Abstract, Full Text
Exclusion	Article is not available in English	Title, Abstract
Exclusion	Article refers to financial debt	Title, Abstract
Exclusion	Article is not peer reviewed (blogs, tutorials, speeches are excluded)	Title, Abstract
Exclusion	Article discusses technical debt solely in the context of software engineering	Title, Abstract, Full Text
Exclusion	Article describes a specific product for use in detecting or managing technical debt	Title, Abstract, Full Text
Exclusion	Duplicate articles	Title
Exclusion	Technical debt in systems engineering is not the primary focus of the article	Title, Abstract, Full Text

The search was conducted in February, 2022, included all articles available up to that time and resulted in the identification of 354 articles. The inclusion and exclusion criteria were first applied to the titles and abstracts and then the full text of the remaining articles was assessed. Application of the criteria to the full text resulted in the identification of 18 papers to be included in the

assessment, as shown in Table 2-2. The preponderance of excluded papers referenced technical debt in the context of software engineering.

Table 2-2. Search results at each stage of evaluation

Evaluation Stage	# of Articles Found	Articles Excluded by Criteria
Online database search	354	Not available in English: 3 Refers to financial debt: 47 Solely in context of software engineering: 202 Describes a specific product: 15 Duplicate article: 4 Technical debt in systems engineering is not the primary focus: 41
Title and abstract criteria	39	Solely in context of software engineering: 15 Duplicate article: 2 Technical debt in systems engineering is not the primary focus: 4
Full reading	18	
Quality assessment	18	

Following the full text reading, a quality assessment was performed on each paper. The quality assessment criteria presented in Lenarduzzi et al. [69] were used to assess the received papers, which the exception of quality assessment criteria 1: "Is the paper based on research (or is it merely a 'lessons learned' report based on expert opinion)?". Since the goal of this study is to examine the prevalence of the technical debt metaphor in systems engineering, "expert opinion" articles have value in this context. The remaining criteria include assessments of the clarity of goals, the collection of data, and the value of the study.

All 18 articles that passed the full text reading assessment also passed the quality assessment.

2.2.1.3.2 Data Extraction

Table 2-3 illustrates the mapping between the research questions and the data that was extractable from the final set of 18 articles on systems engineering technical debt. To answer RQ1, data on the field of study, the type of study conducted, and whether the study applies a new usage of technical debt metaphor were extracted. These data inform our understanding of the prevalence

of the technical debt metaphor within systems engineering. The information identifies not only the areas of active research but also the extent to which new methodologies for dealing with technical debt are studied.

Table 2-3. Research questions and associated data extracted from the qualifying articles

Research Question	Data Type	Description
RQ1	Field of study	The field within which the research was conducted
RQ1	Type of study	Is the research based on survey data, case studies, or author experience?
RQ1	Usage of technical debt	Is the research reporting on existing usage of the technical debt metaphor or proposing a new use?
RQ2	Technical debt language	Is the research adapting software engineering terminology to systems engineering or using systems engineering terminology?
RQ2.1	Technical debt types	Types of technical debt applied to systems engineering identified in the research
RQ2.2	Technical debt causes	Causes of technical debt applied to systems engineering identified in the research
RQ3	Life cycle models	Identified systems engineering life cycle models where technical debt occurs
RQ3	Life cycle stages	Stages of the life cycle models where technical debt is instantiated, where interested occurs, and when it is (or is not) paid back

Data supporting RQ2 include the prevalence in the paper of unique systems engineering terminology and the inclusion of new types (RQ2.1) and causes (RQ2.2) of technical debt. These data assist in identifying differences in technical debt occurrences between systems engineering and software engineering. The data extracted from the articles to answer RQ2.1 are compared to the set of software technical debt types from Li et al. [60]. The causes of technical debt retrieved in support of RQ2.2 are mapped to the types of technical debt.

Studies that identified the occurrence of technical debt within the systems engineering life cycle models provide data to answer RQ3. Data extracted from each article include the type of lifecycle model and the stages within the lifecycle model where the technical debt initially occurs, where its impact is felt, and where it must be paid back.

2.2.1.3.3 Threats to Validity

The findings of a literature review may be affected by numerous factors. Although large digital databases were used for the search for articles for this study, it is possible that relevant articles were not included in the search. The word "system" was required to be in the title or abstract, this restriction may have eliminated studies in closely related fields that do not emphasize the term (such as aerospace engineering). The researcher may have a bias on the study selection and the data extraction. The potential bias on study selection was mitigated by using a clear set of selection criteria, as outlined in Section 2.2.1.3.1. Potential bias at the data extraction stage was mitigated by pre-identifying the data types to be extracted. The descriptions of each data type, shown in Table 2-3, were used to guide the extraction of data. For example, the type of study was classified as either empirical survey data, case study data, or author experience. These descriptions provide the guidelines for coding the data from each article.

2.2.1.4 Results and Discussion

This section presents the results of the literature review in response to the research questions. The initial search returned 354 articles, of which only 18 passed the criteria for inclusion in the literature review. The publication dates of the search results spanned 2009-2022. The IEEE eXplore database returned 172 results, which was the most of any of the databases searched. By contrast, a similar search for ("requirement" AND "system") in the document title in the IEEE eXplore database returned 1,898 results for the same timeframe. A similar search on "systems engineering" in the document title returned 3,401 results. The limited number of search results related to technical debt indicates that technical debt is not a well-studied concept within systems engineering.

Figure 2-1 shows the breakdown of the passing articles by topic of study in relation to the research questions. All articles applied to RQ1 and some articles addressed multiple research questions.

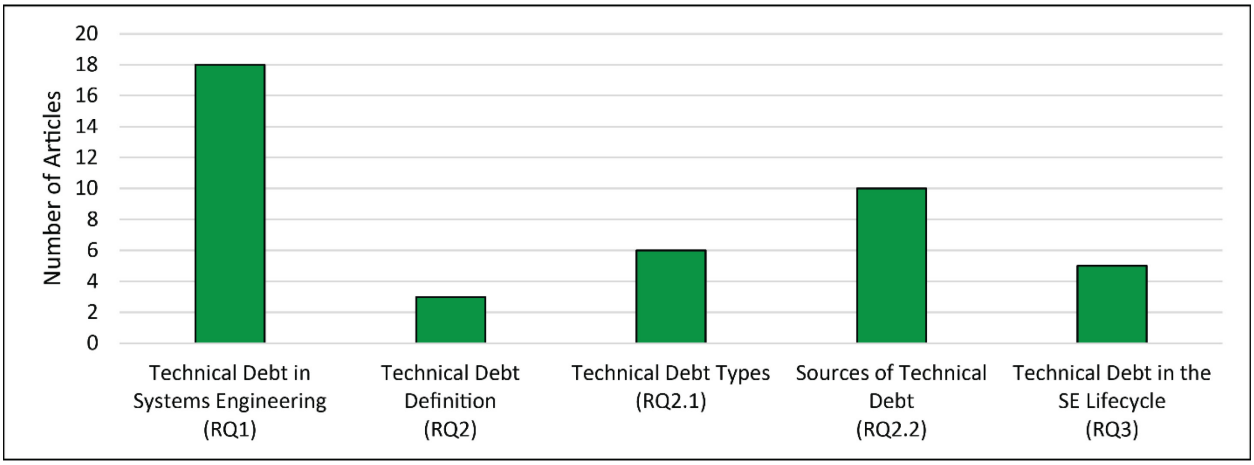

Figure 2-1. Topics of study in selected articles applied to research questions

2.2.1.4.1 RQ1: Prevalence of Technical Debt in Systems Engineering

Figure 2-2 provides an overview of the articles that passed the criteria for inclusion in the literature review, broken down by field of study and data source.

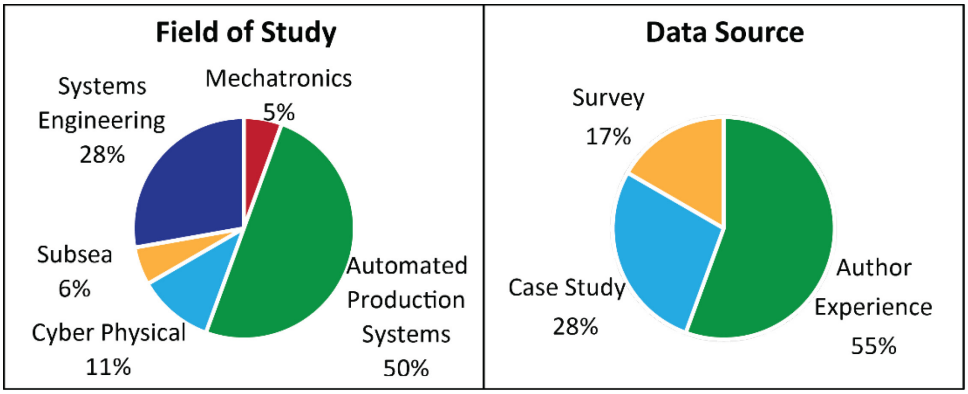

Figure 2-2. Overview of selected articles by field of study (left) and data source (right)

Of the eighteen sources that are applicable to this study, nine focus on the application of technical debt in the development of automated production systems [104] [105] [106] [107] [108] [109] [110] [111] [112]. These studies build upon each other and seek to identify the causes and types of technical debt across the electronic, mechanical, and software components of automated

production systems. For example, Vogel-Heuser and Bi [113] expand their research from the realm of automated production systems to mechatronic systems in general and gather empirical data on the occurrence of technical debt within the mechatronic system life cycle, the types of technical debt that are present, and the causes of the technical debt.

Five of the 18 papers focused on technical debt in traditional systems engineering. Rosser and Norton [79] and Rosser and Ouzzif [70] provide a systems engineering view of technical debt. In both papers, the researchers map technical debt types from the software engineering field to the systems engineering field and identify new types of system-engineering-centric technical debt, such as depreciation debt. Fairley and Willshire [114] identify high level applications of technical debt to different systems engineering life cycle models and provide some methods for assessing the accrual of technical debt during the development cycle. Fairley [64] identifies how technical debt accrues due to rework in iterative and incremental system lifecycles. Storrle and Ciolkowski [89] argue that technical debt needs to be considered at the domain-level and define domain debt as "the mis-representation of the application domain by an actual system." They argue that domain debt, which applies at the system level, needs to be considered alongside other technical debt types, and provide a case study example of sources and effects of domain debt.

The selected articles used a combination of case studies and surveys as empirical data sources. Slightly more than half the articles were primarily derived from author experience, which includes theoretical data, and the rest gathered empirical data from case studies and surveys. Figure 2-3 shows the breakdown of the articles by the field of study and associated data source.

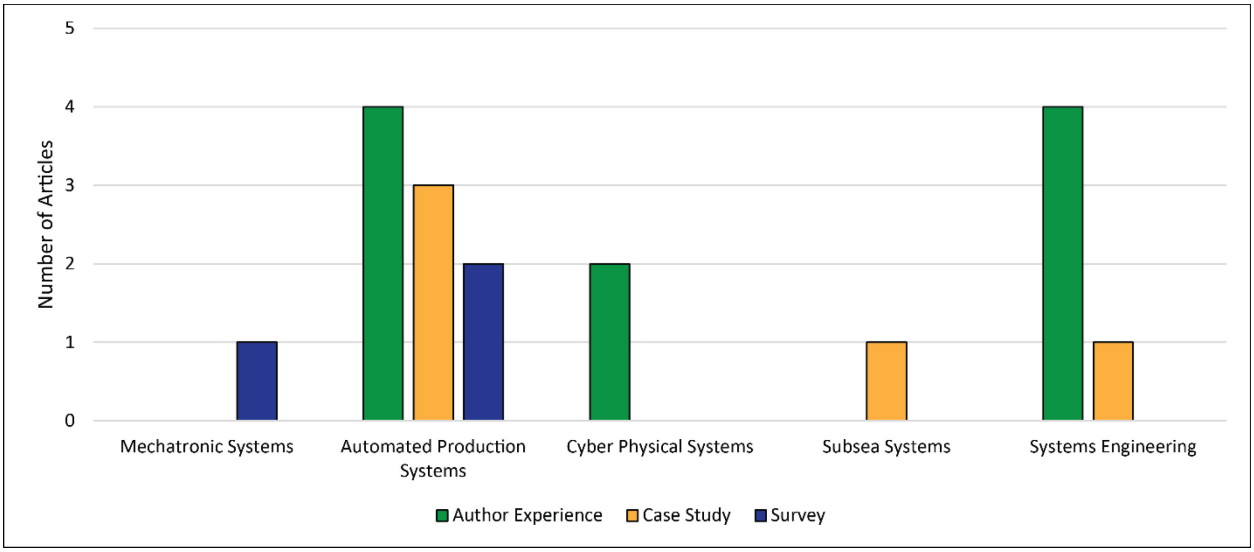

Figure 2-3. Data source by field of study

From these results and in reference to research question 1, we can conclude that the technical debt metaphor is not broadly utilized within published works on systems engineering. Of the thousands of systems engineering papers published during the study period, only 18 were identified as pertaining to technical debt in systems engineering. The majority of the 18 articles focus on technical debt in specific systems and not on the application of technical debt to the broader concept of systems engineering. The most extensive set of articles focus on the domain of automated production systems. These systems provide a reasonable basis for extension to the general systems engineering field due to the interdependencies of multiple engineering disciplines. The literature on technical debt in automated production systems is primarily based on a foundational set of case studies and industrial surveys, which provide an applied and empirical underpinning to the research to date.

The few articles that focus on the broader scope of systems engineering do not provide significant empirical evidence for their conclusions. The lack of empirical data supporting the analysis of technical debt within systems engineering highlights the need for research into this emerging field, including the generation of such data.

2.2.1.4.2 RQ2: Definition of Technical Debt in Systems Engineering

The definition of technical debt has evolved since Cunningham first coined the metaphor. McConnell defined technical debt as "the obligation that a software organization incurs when it chooses a design or construction approach that's expedient in the short term but that increases complexity and is more costly in the long term" [58]. Review of the selected articles revealed that the majority use a modified version of the definition of technical debt from Li et al. [60]. Li defines that "Technical debt is a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a software system" [60]. The use of the term "long-term health" instead of costs expands McConnell's definition to include non-cost impacts on the system. Fairley and Willshire [114] define technical debt as the "difference between planned or reported product delivery and actual delivery." Biffl et al. [112] define technical debt as "violations in the engineering data model or engineering data instances compared to an intended data model architecture for data integration in systems engineering." Rosser and Norton [79] do not explicitly define technical debt, but state that "a consensus-based definition for system technical debt has not yet emerged." They instead identify multiple classes and types of technical debt for consideration in systems engineering.

The selected studies show that technical debt is an evolving concept and its application to systems engineering is expanding. 17% of the studies included a new or heavily modified definition of technical debt. Of the five articles that focused on general systems engineering, two (40%) provided a new definition of technical debt. These results reinforce that convergence on the definition and scope of technical debt has not yet emerged.

Given these results, we conclude that the software-based definition of technical debt provided by Li et al. is a valid starting point for a systems engineering-centric definition of technical debt.

To make the definition applicable for systems engineering, we suggest removing the software focus of the definition, replacing it with a systems focus as follows: “Technical debt is a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a *system*.”

These findings illustrate that there is a lack of a consensus definition for system technical debt, and more broadly a lack of a common ontology for discussing technical debt within systems engineering.

2.2.1.4.2.1 RQ2.1: Types of Technical Debt in Systems Engineering

The technical debt landscape consists of issues related to software evolvability and maintainability [115]. These issues may be internal to the system and invisible to the user, such as architectural issues, or they may be issues directly visible to the user such as poor usability. Kruchten et al. [116] define a set of concepts to assist in identifying what issues should and should not be considered technical debt:

- Technical debt does not consist only of bad quality
- New usages of a system can create technical debt
- Defects are not necessarily technical debt
- Unfinished or postponed work is not technical debt
- Work to be done in the future is not technical debt

These concepts informed the review of technical debt types in the selected articles.

Li et al. [60] identify ten types of technical debt prevalent in software engineering. This typology served as the baseline point for comparison in the review of the selected articles. While some articles identified different types of technical debt, many of those can be mapped into the

baseline set. For example, we assert that all types of technical debt associated with flaws in software designs, whether they are in a database or source code, can be allocated to the code debt type. Technical debt due to hardware assembly and software build processes were grouped together into the "build debt" type and renamed to "build/assembly debt". Within a systems engineering framework, these types of debt are similar since they both are centered on the steps to integrate and create the system. Table 2-4 presents the summary of that process of mapping the literature to an expanded typology of technical debts.

Table 2-4. Types of technical debt identified in selected articles

Technical Debt Type	Definition	Source(s)
Architectural	"Caused by architecture decisions that make compromises in some internal quality aspects, such as maintainability" [60]	[108] [111] [113] [79] [70] [117]
Automation	Technical debt associated with the automated machinery used in hardware systems [70]	[70]
Build/Assembly	"Flaws in a software system, in its build system, or in its build process that make the build overly complex and difficult" [60]	[113] [79]
Code	"Poorly written code that violates best coding practices or coding rules" [60]	[108] [111] [112] [79]
Commissioning	Related to the commissioning and start-up of automated production systems [113]	[113]
Configuration	Hardware configuration and testing can be affected by availability of the systems [70]	[79] [70]
Defect	"Defects, bugs, or failures found in software systems" [60]	[79] [70]
Depreciation	Effect of aging system, outdated components, and the need to replace/update them [70]	[79] [70] [86]
Design	"Technical shortcuts that are taken in detailed design" [60]	[111] [113] [117]
Documentation	"Insufficient, incomplete, or outdated documentation in any aspect of software development." [60]	[108] [111] [113] [79] [70] [86]
Domain	"The misrepresentation of the application domain by an actual system" [89]	[89]
Implementation	Errors in hardware implementation resulting from poor instructions [70]	[113] [70]
Infrastructure	"Sub-optimal configuration of development-related processes, technologies, supporting tools, etc." [60]	[111] [113]
Integration	Use of "non-standard connections, outdated or proprietary interfaces, and infrequently used standards." [27]	[79] [70] [86]
Modeling and Simulation	Technical debt associated with the models and simulations used to support a system [70]	[79] [70]
Operations / Maintenance	"Any kind of handicap with adverse effects on the product or system maintenance" [111]	[113]

Technical Debt Type	Definition	Source(s)
Organic	"Refers to any combination and degree of technological, systemic, project, and program decisions, behaviors, and practices made by the workforce, management and/or senior/executive leadership of the organization responsible for introductions of new technologies and systems and/or the sustainment of existing systems" [86]	[86]
Requirements	"Distance between the optimal solution to a requirements problem and the actual solution, with respect to some decision space" [118]	[113] [79] [70] [86]
Start-up	"Refers to shortcuts taken in the startup process of the product or system." May be specific to the field of automated production systems [113]	[111]
Test	"Shortcuts taken in testing" [60]	[111] [113] [79] [70]
Versioning	"Problems in source code versioning" [60]	[113] [86]

In summary, the selected articles identified a total of 21 types of technical debt, many of which are new types of technical debt compared to the baseline list. Figure 2-4 shows the occurrence of each identified type of technical debt in the selected articles. The most commonly referenced types are "architectural technical debt" and "documentation technical debt". The largest occurring types of technical debt map to the types identified by Li et al., as shown in Table 2-4.

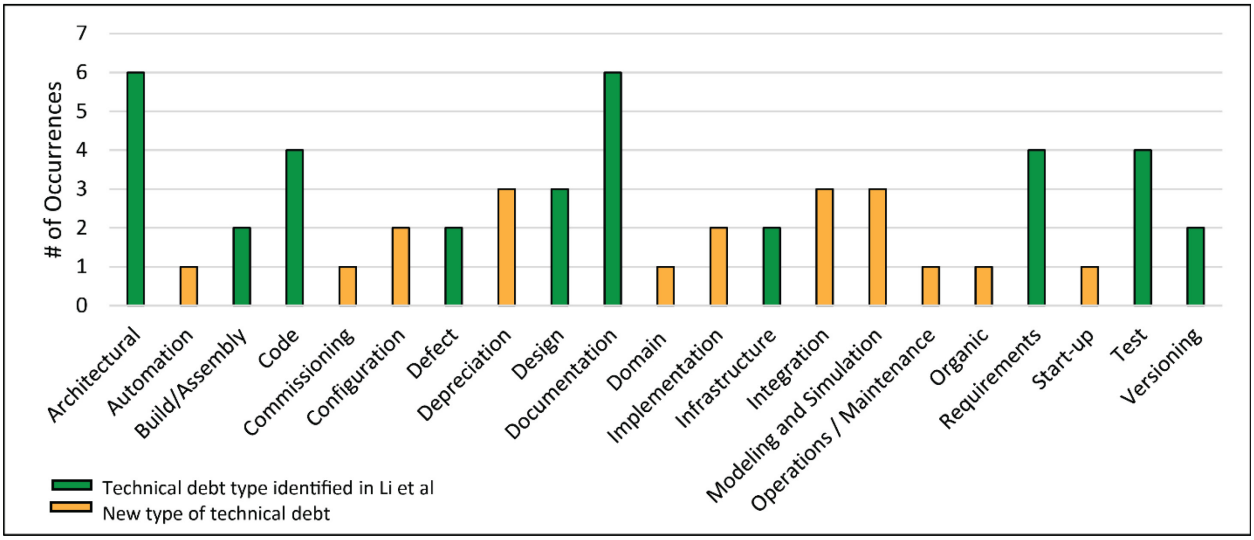

Figure 2-4. Occurrence of technical debt types in selected articles

Six of the eleven new types of technical debt are defined within only one source, and twelve of the total types of debt are defined within two or fewer sources. These findings show that there

exists a lack of a consensus categorizations for technical debt within systems engineering. Several of the identified types of technical debt, such as commissioning debt, are unique to a specific field. The lack of empirical data throughout many of the articles prevents an independent evaluation of the rationale for each type of technical debt. Additional data would be required to verify and assert a comprehensive list of the types of technical debt.

2.2.1.4.2.2 RQ3: Causes of Technical Debt in Systems Engineering

Ten of the articles identified causes of technical debt. Table 2-5 aggregates the identified sources into major categories and maps those categories to the different types of technical debt identified in Table 2-4.

Table 2-5. Technical debt causes in selected articles

Technical Debt Cause	Description	Technical Debt Type(s)	Source(s)
External factors	Factors external to the system, such as cost and schedule limitations and external priorities that result in the pressure to take short-cuts in any stage of the system lifecycle	Domain, Architectural, Implementation, Build, Code, Modeling and Simulation, Requirements, Test, Organic	[111] [113] [70] [114] [89] [64]
Short cut modifications	Poorly documented modifications, such as those made in the field and not recorded and correction of hardware deficiencies through software, that lead to multiple versions of the system	Implementation, Configuration, Infrastructure	[113] [70]
Poor requirements	Poorly specified requirements and poor initial assumptions about the system may lead to rework due to incorrect implementation	Requirements, Domain	[114] [64]
Inadequate resources	Inadequate resources and lack of proper budgets may result in work arounds that accumulate debt	Organic	[111] [114] [64]
Multiple disciplines and parallel development	Multiple disciplines require synchronized planning. Lack of synchronization or knowledge of the state of the other disciplines can cause divergent designs and result in rework. Different performance indicators and misaligned timelines can contribute to divergent requirements and designs.	Design, Architectural, Organic	[111] [112] [113]

Technical Debt Cause	Description	Technical Debt Type(s)	Source(s)
Deviations from standards and poor or incomplete work	Failure to implement to a standard, either external or internal, and developing poor (i.e., bad architecture or containing multiple defects) or incomplete work (i.e., incomplete refactoring) leads to technical debt. Performing simplified analyses when more detailed analyses are required can result in an insufficient design.	Design, Code, Architectural, Requirements, Build, Defect	[108] [111] [70] [64] [117] [119]
Environmental changes	Changes to the system's environment due to regulatory or standards changes or evolution of the domain result in shortfalls in the current system that need to be remedied. Similarly, decay of the system's design and evolution of technology beyond the system's capabilities result in performance shortfalls.	Domain, Depreciation	[70] [89]
Third-party products	Third-party or commercial products may not completely satisfy the system requirements.	Design, Architectural, Domain, Requirements	[111] [89]
Human factors	Personnel may have a lack of knowledge or skill that contribute to the need for rework. Lack of communication between personnel can also contribute to rework.	Organic	[111] [113]
Proposal effects	Solutions generated in a time and cost constrained manner, such as for a proposal, can take shortcuts in detailed analysis and cost-savings measures that result in the need for rework later.	Architectural, Design	[117]

Synthesizing the selected articles provides a substantial listing of causes of technical debt. External factors, deviations from standards, and poor work are the most common causes of technical debt (mentioned by six different articles). However, all articles lack empirical data linking the causes to the types of technical debt or, more importantly, to the impact on the system lifecycle.

2.2.1.5 RQ3: Technical Debt in the Systems Engineering Lifecycle

As shown in Figure 2-1, five of the studies addressed the role of technical debt in the systems engineering lifecycle, and can therefore be referenced in answering RQ3. Fairley and Willshire [114] discuss the management of technical debt in the linear-predictive (Waterfall), incremental-predictive (incremental), and iterative-adaptive (agile) systems lifecycles. In linear-predictive lifecycles, they propose to identify and review the technical debt at review gates at the end of each

stage. Similarly, they propose to review technical debt at the end of each increment in an incremental-predictive lifecycle and suggest merging technical debt burndown into the future increments of an iterative-adaptive lifecycle. Dong et al. [107] describe how technical debt created in one discipline affects the others throughout the life cycle of automated production systems, but they do not apply technical debt concepts directly to the stages of the life cycle. Vogel-Heuser and Bi [113] show how technical debt is most prevalent early in the lifecycle of mechatronic systems and how different types of technical debt emerge throughout the lifecycle of these systems. Requirements, architecture, design, variants, and version technical debt occur within the specification and design stages, code and test technical debt occur during the development stage, and defect, start-up, and maintenance technical debt occur during the startup and operations stages. They also note that infrastructure and documentation technical debt, which are two of their more prevalent types of debt, can occur in any stage of the lifecycle. Callister and Andersson [117] describe the occurrence of technical debt in multiple stages of subsea system development using a Collapsed Vee lifecycle model. They provide a case study of two projects, showing technical debt occurrences in the tender (proposal) phase, the system definition phase, and the detailed design phase. Fairley [64] assesses technical debt in the context of an iterative and incremental lifecycle. He identifies that in a well-managed project, technical debt accrued in one increment should be paid back in the next increment. They assert that there may be the need to devote an entire increment to paying back technical debt.

Although the mechanisms by which technical debt appears in the systems engineering lifecycle were not addressed in a large number of the articles, those that did address it showed that technical debt can occur in many stages of the systems engineering lifecycle. Vogel-Heuser and Bi [113] provide a complete description of technical debt occurrence throughout the lifecycle of a

mechatronic system. There is an opportunity in the field to map technical debt within the most common systems engineering lifecycles, based on empirical data, to identify the stages that are most susceptible to technical debt accumulation. The effect of early-stage technical debt on end stage capabilities could be quantified in key systems engineering applications such as aerospace and energy systems. For example, Schutz [120] argues that there is insufficient research on technical debt in the domain of smart grids.

2.2.1.6 Discussion

2.2.1.6.1 Implications of the Sparsity of the Technical Debt Metaphor within Systems Engineering

Although technical debt is illustrated here to be not well-researched in the context of systems engineering, its existence is known and it is identified as a problem. Technical debt as it affects systems engineering projects is an open area of research to which future work can contribute, especially with empirical data gathering.

The results of RQ1 show that the technical debt metaphor is not prevalent within systems engineering research. The limited usage of the metaphor does not imply that technical debt does not occur within systems engineering. Rather, the limited usage of the metaphor implies that systems engineering lacks a common ontology to discuss the types of problems that result from technical debt. The lack of commonality can be seen in the different applications and definitions of technical debt.

The implication of the confusion in definitions can be seen by examining two products from the International Council on Systems Engineering (INCOSE): *Project Manager's Guide to Systems Engineering Measurement for Project Success* [78] and *Needs, Requirements, Verification, Validation Lifecycle Manual* [77]. Each of these publications provide a different definition of technical debt:

- "The promise to complete a technical shortcoming in the future while declaring it complete today." [78]
- "What occurs when a project team uses a quick short-term solution that will require additional development work later to meet the needs of stakeholders." [77]

The first definition does not include the concept of technical debt interest – that there will be additional work required due to the technical shortcoming. The accumulation of technical debt interest is what makes technical debt dangerous to systems development. In contrast to the second definition, the first definition does not allow the system developer to properly understand the potential impacts of their decisions on the ability to complete the system in the future.

Common ontologies produce commonly understood definitions of terms and enable communication of ideas. Establishing such an ontology for technical debt in terms of systems engineering will lead to increased communication about and identification of technical debt within the systems development process. Once technical debt is identified then it can be measured, and perhaps controlled. The sparsity and inconsistency of the technical debt metaphor in systems engineering may weaken communication and collaborative control of technical debt in systems development.

2.2.1.6.2 Research Agenda for a Systems Engineering-Centric View of Technical Debt

To use the technical debt metaphor as a tool for managing systems engineering projects requires additional research to fully understand the implications of technical debt within systems engineering. We propose the following research agenda for a systems engineering-centric view of technical debt:

- Baseline the knowledge of both the concept behind technical debt and the use of the metaphor within systems engineering by gathering empirical data in key systems

engineering applications. Such data could be gathered through a survey of industry and academia and would be used to understand the current state of technical debt within systems engineering.

- Develop a systems engineering-centric ontology of technical debt to provide clear communication of the concepts, causes, and interrelationships among technical debts. The types and causes of technical debt identified in this literature review and through the gathering of additional empirical data can be mapped to standard systems engineering processes and lifecycles to develop a comprehensive set.
- Develop techniques to identify causes of technical debt within the systems engineering lifecycle. These techniques can be used to find technical debt before it accrues to the point of affecting the system.
- Determine a method to quantify the future impact of technical debt. Technical debt must be measurable for it to be controlled. Existing tools and techniques for technical debt management need to be examined and, if necessary, expanded upon such that they can be applied to systems engineering.
- Test and validate the methods to identify and measure technical debt through application to systems engineering projects and programs.

2.2.1.6.3 Implications for the Management of Technical Debt in Systems Engineering Processes

Despite the relative obscurity of technical debt in the systems engineering literature, it's presence can dominate the performance of systems engineering projects. Left unchecked, it will lead to technical bankruptcy – the state where the project can no longer proceed on time and budget without first paying down the debt. Technical bankruptcy can result in significant schedule delays, cost increases, or reductions in performance. In 2019, the United States Government

Accountability Office (GAO) issued a report on four systems that suffered from technical debt and experienced technical bankruptcy (even though the terms "technical debt" did not appear in this report). Impacts of the increase in technical debt included schedule breaches, failures in developmental test, failures in operational test, obsolete software, inadequate cybersecurity, incomplete systems engineering, and significant software rework [11]. Each of these issues resulted in cost growth or schedule delays and some of the systems experienced failures of development or operational testing.

The GAO report recommended focusing on including users early in the development process. Doing so is a form of "Requirements Technical Debt" management. Users are better positioned to give feedback on the long-term usability impacts of decisions that are made early, and as such, can identify causes of technical debt as they occur. Early identification is a key technique for managing any potential problem sources, including technical debt.

Many of the studies reviewed here focus on identification of technical debt, fewer focus on the systems engineering processes, tools, and techniques that can be used to manage technical debt within systems engineering processes, especially as systems engineering processes move to become more agile [13]. With agility comes change and with change comes the potential for technical debt. Without express management, technical debt build up will lead to technical bankruptcy.

2.2.1.6.4 Conclusions and Future Work

In this systematic literature review, we searched for published articles that described the effect of technical debt in systems engineering. Eighteen articles were retrieved and analyzed to help answer research questions on the prevalence and definition of technical debt within the systems engineering field and its impact on the systems engineering lifecycles.

The results of the review show that technical debt is not a well-studied concept within systems engineering. We have developed a more comprehensive set of the types and sources of technical debt, and have derived a set of critical elements for managing technical debt within systems engineering: a common ontology, empirical data, and management tools and processes. These findings illustrate that by enrichening the metaphor of technical debt for systems engineering, we can enable the consideration of technical debt as a part of system development. Management of the creation and impact of technical debt during system design development is a critical aspect of minimizing the risk of technical bankruptcy.

This research will continue through the implementation of the proposed research agenda. An empirical survey of practicing systems engineers will provide evidence for the prevalence of technical debt within systems engineering. The results of this survey can be used to understand the phases within the systems engineering lifecycle where technical debt is created and observed. This data can then be used to develop effective technical debt management and mitigation tools for use in systems engineering.

2.2.1.7 Appendix A: Selected Articles

The articles that passed the literature review process are listed in Table 2-6. Full bibliographical information for each article is available in the references section.

Table 2-6. Selected articles

Author(s)	Article Name	Reference
Biffl, Ekaputra, Luder et al.	Technical Debt Analysis in Parallel Multi-Disciplinary Systems Engineering	[112]
Brenner, Weippi, and Ekelhart	Security Related Technical Debt in the Cyber-Physical Production Systems Engineering Process	[119]
Callister and Andersson	Evaluation of System Integration and Qualification Strategies using the Technical Debt metaphor; a case study in Subsea System Development	[117]

Author(s)	Article Name	Reference
Cha, Dong, and Vogel-Heuser	Preventing Technical Debt for Automated Production System Maintenance Using Systematic Change Effort Estimation with Considering Contingent Cost	[104]
Dong, Ocker, and Vogel-Heuser	Technical Debt as indicator for weaknesses in engineering of automated production systems	[107]
Dong and Vogel-Heuser	Modelling Industrial Technical Compromises in Production Systems with Causal Loop Diagrams	[106]
Dong and Vogel-Heuser	Cross-disciplinary and cross-life-cycle-phase Technical Debt in automated Production Systems: two industrial case studies and a survey	[105]
Fairley	Assessing, Analyzing, and Controlling Technical Work	[64]
Fairley and Willshire	Better Now Than Later: Managing Technical Debt in Systems Development	[114]
Ocker, Seitz, Oligschlager, Zou, and Vogel-Heuser	Increasing Awareness for Potential Technical Debt in the Engineering of Production Systems	[110]
Rosser and Norton	A Systems Perspective on Technical Debt	[79]
Rosser and Ouzzif	Technical Debt in Hardware Systems and Elements	[70]
Storrle and Ciolkowski	Stepping Away from the Lamppost: Domain-Level Technical Debt	[89]
Vogel-Heuser and Bi	Interdisciplinary effects of technical debt in companies with mechatronic products — a qualitative study	[113]
Vogel-Heuser and Neumann	Adapting the concept of technical debt to software of automated Production Systems focusing on fault handling, mode of operation and safety aspects	[108]
Vogel-Heuser and Rosch	Applicability of Technical Debt as a Concept to Understand Obstacles for Evolution of Automated Production Systems	[111]
Vogel-Heuser, Rosch, Martini, and Tichy	Technical debt in Automated Production Systems	[109]
Yang, Michel, Wage, Verma, Torngren, and Alelyani	Towards a taxonomy of technical debt for COTS-intensive cyber physical systems	[86]

2.2.2 Addressing RQ1.1

This literature review addresses RQ1.1. The review identified that technical debt is a well-researched field within software engineering but is not a well-researched field within systems engineering. The research that does exist is largely applied to specific types of systems and a wholistic view of technical debt impact across the field of systems engineering is not presented in the published body of knowledge. There is a lack of common definitions and terminology for technical debt within systems engineering, exacerbated by the fact that there is not a consensus definition of the base technical debt term [79]. The definition of types and sources of technical

debt is not consistent across the published research, with unique types identified based on the type of system assessed. This overspecification of types can lead to confusion and prevent the use of common management methods. Technical debt is identified as occurring throughout the systems lifecycle, but is not mapped to a specific lifecycle or stages. Such a mapping could increase the understanding of the impacts of technical debt in the system lifecycle.

2.3 RQ1.2: How Prevalent is the Concept of Technical Debt and the use of the Metaphor Among Systems Engineering Practitioners?

Full understanding of the prevalence of the technical debt concept and metaphor requires more than just a literature review. If the concept is well-researched but is not well-used in practice, then it cannot be considered prevalent. Alternatively, the concept of technical debt may be well-used but not well researched. The components of technical debt may be known to practicing systems engineers but they may not use the same terminology. To assess the knowledge and usage of the technical debt concepts and terminology, a survey of practicing systems engineers was conducted during the summer of 2022. The literature review identified that published systems engineering research did not adequately address technical debt. The survey was designed to identify if technical debt is a concern for systems engineers in accordance with Task 1.2.1 and Task 1.2.2: conducting the survey and processing the results to identify causes of technical debt and the occurrence within the system lifecycle. The survey results were presented at the *2023 INCOSE International Symposium* and published in the conference proceedings [18]. The paper from the conference is reprinted here.

2.3.1 An Empirical Survey on the Prevalence of Technical Debt in Systems Engineering [18]

2.3.1.1 Abstract

The technical debt metaphor is used within software engineering to describe technical concessions that produce a short-term benefit but result in long-term consequences. Systems engineering is subject to these concessions, yet there is a limited amount of research associating technical debt with systems engineering. This paper provides the results of an empirical survey investigating the prevalence of technical debt in systems engineering, including the occurrence of technical debt, the use of the metaphor, and the distribution of technical debt within the systems engineering lifecycle. The results of the survey show that while technical debt is common in systems engineering and occurs throughout the lifecycle, the metaphor and terminology of technical debt is not consistently applied. These results emphasize the need to enrich the usage of the technical debt metaphor within systems engineering to enable the management of technical debt and to reduce the risk of technical bankruptcy.

2.3.1.2 Introduction and Background

Modern technology and the digital engineering transformation are increasing the emphasis on delivering flexible systems more rapidly [6]. Agile systems engineering methods [13] and iterative and incremental development strategies [23] are used to increase flexibility and to limit the cost and schedule increases traditionally associated with requirements changes [3]. While Agile processes can be mapped to the system development lifecycle [1], the increased emphasis on shorter times to market can result in "system sponsors and stakeholders… encourag[ing] developers to take shortcuts early in the development process in order to get system capabilities deployed quickly" [7].

If not carefully managed, the decisions made during the planning and execution of iterations can have far reaching consequences on the future state of the system. The work in each iteration places design constraints on future iterations [26] which can result in more expensive changes later in the development cycle [121] or the failure to meet performance objectives. Projects may start work prior to fully understanding the problem in order to deliver a working system faster and then rely on user feedback to improve the system to better meet the users' needs. However, the initial decisions made to produce early value may result in severe inefficiencies in the implemented system, such as lower usability and increased rework later in the development schedule [63]. This phenomenon is known as technical debt.

The technical debt metaphor was introduced as a method to communicate the need to refactor software code to remove short-cuts that were put in place to meet a goal, such as a scheduled release, before those short-cuts could add up to larger problems within the system [17]. Much like financial debt, technical debt accrues interest, which manifests as increased development timelines, increased project cost, and/or rework later in the development cycle. Unmanaged technical debt may lead to technical bankruptcy – the state where system development cannot continue without first repaying back the technical debt [60].

Since 2008, published research on technical debt in the field of software engineering has steadily increased [54]. Technical debt has been classified into multiple types [69] [70], different causes have been identified [122] [58], and multiple measurement techniques have been suggested [65] [16] [67] [66]. This research, however, has been primarily constrained to the field of software engineering [19]. Despite the fact that systems engineering has borrowed many concepts from software engineering, including lifecycle models and development approaches [1], there is not a substantial amount of published research on technical debt with systems engineering [19].

The concepts behind technical debt are not new to systems engineering. Terminology such as 'rework' has been used to define similar problems. Guenov and Barker [123] applied axiomatic design theory and design structure matrices to identify design conflicts that result in delays due to unplanning iterations and rework. Boehm, Valerdi, and Honour [124] discuss the reduction in rework that can be achieved by applying systems engineering to software-intensive systems. Broniatowski and Moses [125] define a "rework potential" to measure the rework associated with design choices. Raman and D'Souza [126] developed a decision learning framework that, in part, addresses the uncertainty of architectural design decisions, including those that may lead to more effort than an optimal solution. Shallcross et al. [127] discuss the use of set based design to limit premature design decisions which may result in expensive rework. Siyam, Wynn, and Clarkson [128] identify a need to evaluate how changes in processes affect the value of a system later in the lifecycle. Bahill [98] developed a process to deal with unintended consequences. These research papers all define similar problems to technical debt - minimizing the amount of effort required to correct a technical issue through early detection and mitigation. However, none of the cited works include the term "technical debt." Instead, each paper uses their own terminology to describe the problem.

Recognizing that the lack of a common ontology prevents a common understanding of the problem [129], Kleinwaks, Batchelor & Bradley [19] conducted a systematic literature review to determine the prevalence of the technical debt metaphor within published systems engineering research. They concluded that the technical debt metaphor is not prevalent in published papers on systems engineering, that there is not a consensus definition for technical debt within systems engineering, that there is little empirical evidence on the impact of technical debt within systems

engineering, and that a common ontology for technical debt in systems engineering has not been established.

Kleinwaks, Batchelor, and Bradley [19] recommend gathering empirical data to understand the use of the technical debt metaphor by practicing systems engineers to supplement the research in the literature review. This paper provides the results of an empirical survey following this recommendation. The survey was constructed to answer the following research questions:

- RQ1: Does technical debt occur within systems engineering, and if so, what is its impact?
- RQ2: What are the causes of technical debt within systems engineering?
- RQ3: How prevalent is the use of the technical debt metaphor among systems engineering practitioners?
- RQ4: Where does technical debt occur within the systems engineering lifecycle?

The rest of this paper is structured in four sections. First, the research methodology is presented. Next, the primary findings of the survey are presented. Then, the findings are discussed in the context of the research questions. Finally, the paper is concluded and concepts for future work are presented.

2.3.1.3 Research Methodology

2.3.1.3.1 Study Method

This research was conducted using an online survey tool to collect responses to a series of questions designed to assess the respondents' familiarity with situations that can be classified as technical debt, their familiarity with the metaphor of technical debt, and the stages in the systems engineering lifecycle where technical debt occurs.

Participants in the study were recruited through email solicitations and social media postings sent to specific groups of systems engineers, including a corporate systems engineering community of practice, a graduate university systems engineering department, the local INCOSE chapter, and the authors' LinkedIn networks. These participant groups were selected based on experience with systems engineering as well as the ability of the authors to contact the group members.

The survey was conducted anonymously, however, some basic demographic questions, such as current position and years of experience, were asked in order to inform the data analysis process. The survey was first released on July 14, 2022 and was closed on August 31, 2022. 50 respondents replied to at least one question in the survey.

2.3.1.3.2 Data Analysis

The collected data reports were generated with an anonymous respondent identifier that was matched to the responses for each question. Respondents were not required to answer every question and therefore percentages are reported based on the number of respondents who answered the question and not on the total number of participants in the survey. Several questions allowed the respondent to select multiple responses; in these cases, the percentages are reported as the number of respondents who selected that answer and therefore the percentages may add up to be greater than 100%.

2.3.1.3.3 Threats to Validity

The internal validity of a study is the measure of how well the collected data corresponds to the research questions [130]. The internal validity is assessed by examining the potential biases that may arise within the study formulation, including the development of the research questions and the survey questions. To limit biases in the development of the research questions, gaps in the current state of academic research on technical debt in systems engineering [19] formed the basis

of the research questions. Multiple researchers reviewed and developed the survey questions to confirm that they mapped to the research questions. Upon completion, a professional systems engineer evaluated the survey questions and the authors refined the questions based upon the engineer's feedback. Terminology was carefully selected in Question Group 2 to avoid the use of the term "technical debt" in the questions to minimize previous familiarity (or lack thereof) with the term from biasing the answers prior to the introduction of the technical debt metaphor within the survey. The data was collected using an online survey tool that allowed for anonymous responses to prevent biases in reviewing and analyzing the data.

The external validity of the study is the measure of how well the research findings can be extended from the sample group to the general population of interest [130]. In this study, the sample group was recruited through email and social media postings. The general population of interest is the set of professional systems engineers, across all disciplines. The majority of respondents indicated background in similar industries, especially the defense industry. This factor has the potential to bias the results towards the defense industry, and therefore the results of the survey may be more generalizable to that subset of professional systems engineers. Another concern prior to the execution of the survey was that a potential bias may arise if software engineers responded to the survey, due to the familiarity of the technical debt metaphor within systems engineering. This concern is addressed in later in this paper.

Given the lack of published research on, and common definitions of, technical debt within systems engineering, it is possible that the respondents do not represent a valid source of knowledge for providing responses regarding the occurrence of technical debt within the systems engineering lifecycle. This threat to the study validity is mitigated by providing the survey

participants with a common definition of technical debt prior to asking questions in Question Group 3 and Question Group 4.

2.3.1.3.4 Survey Questions

Table 2-7 lists the questions included in the survey, along with a mapping to the research questions. The starred question numbers indicate questions that allowed multiple answers.

Table 2-7. Survey questions

#	Question	RQ
1.1	What is your current position?	N/A
1.2	How many years of professional experience do you have?	N/A
1.3	How many years of experience do you have as a systems engineer?	N/A
1.4	What industry do you currently work in?	N/A
2.1	Have you ever worked on a system where a less than ideal short-term solution to a problem created negative long-term impacts on the system? Negative impacts may include issues such as difficulty meeting requirements, decreased ease of use, and increased system maintenance.	RQ1
2.2*	What negative long-term impacts have you experienced from less-than-ideal short-term solutions? Select all that apply.	RQ1
2.3	Do negative long-term impacts arise primarily from decisions to implement less than ideal short-term solutions (e.g., as way to reach a project completion milestone) or from the accumulation of unintentional decisions (e.g., as the by-product of poor requirements)?	RQ2
2.4	When making the decision to implement the less-than-ideal short-term solution, were there any considerations of the potential for negative long-term impacts?	RQ2
2.5*	Which reasons explain why a systems engineer would implement a less than ideal solution that has benefits in the short-term but negative long-term impacts? Select all that apply.	RQ2
2.6	Have you ever had to correct system issues that were due to less-than-ideal short-term solutions that had negative long-term impacts?	RQ1
2.7	If you have had to correct negative long-term impacts of a decision, how did the effort to correct the negative long-term impacts compare to the effort that would have been required to implement the ideal original solution?	RQ1
3.1	Prior to this survey, how familiar were you with the term technical debt?	RQ3
3.2	How frequently do you use the term technical debt in your daily work?	RQ3
3.3	How familiar are your co-workers with the term technical debt?	RQ3
3.4*	In what engineering contexts have you used or heard the term technical debt? Select all that apply.	RQ3
4.1*	In which stage(s) of the systems engineering lifecycle is technical debt most likely to be created (the decision is made to implement then less than ideal solution)? Select all that apply.	RQ4
4.2*	In which stage(s) of the systems engineering lifecycle is the impact of the technical debt (additional work due to the less-than-ideal solution) most likely to be observed? Select all that apply.	RQ4
4.3*	In what stages of the lifecycle is creating technical debt (deciding to implement the less-than-ideal solution) acceptable? Select all that apply.	RQ4
4.4*	In what stages of the lifecycle is creating technical debt (deciding to implement the less-than-ideal solution) unacceptable? Select all that apply.	RQ4

Question Group 1 (QG1) included basic demographic questions to identify the professional background and experience of the participants. Question Group 2 (QG2) contained questions that were designed to identify if survey participants had experience with technical debt without using the term "technical debt." Instead, these questions used the terms "less than ideal short-term solution" and "negative long-term impacts." This terminology was specifically chosen to convey the concepts behind the technical debt metaphor, without relying on the metaphor to convey the meaning. In this way, it is possible to assess the participants experience with the conditions that give rise to technical debt without biasing the answers towards familiarity with the metaphor.

After completing QG2, the respondents were provided with the following definition of technical debt: "Technical debt is a metaphor reflecting technical compromises that can yield short-term benefits but may hurt the long-term health of a system" (Kleinwaks, Batchelor & Bradley 2023). If a respondent indicated that they were not familiar with technical debt, they were provided a short example.

Question Group 3 (QG3) assessed the respondents' familiarity with the technical debt metaphor, introducing the terminology into the questions. These questions were designed to assess the frequency with which the terminology is used in professional situations. Question Group 4 (QG4) assessed the respondents' view of the impact of technical debt in the following phases of the systems engineering lifecycle: needs analysis, requirements definition, preliminary design, critical design, integration, verification and validation, and operations. These phases were chosen since they occur in all system development, regardless of the development method used. Agile and iterative development cycles include the same phases; however, the phases are repeated more frequently. The questions in QG4 asked respondents to consider the lifecycle phases where technical debt is likely to be created and observed, and in which lifecycle phases it is acceptable

and unacceptable to create technical debt. These questions were designed to identify the lifecycle stages where technical debt identification and management is the most important in preventing technical bankruptcy.

2.3.1.4 Research Findings

This section presents the main findings of the survey.

2.3.1.4.1 Participant Demographics

QG1 asked the respondents to provide information about themselves and their background as systems engineers. The results are shown in Figure 2-5. The left chart shows the breakdown of the participants by their current position. The middle chart shows the breakdown of the participants by their current industry. The right chart shows the participants' total professional experience (Total) and their experience as a systems engineer (SE). The chart is colored based on the participant's current position. For example, the chart shows that 10% of the participants classified themselves as management with 5-10 years of experience as a systems engineer.

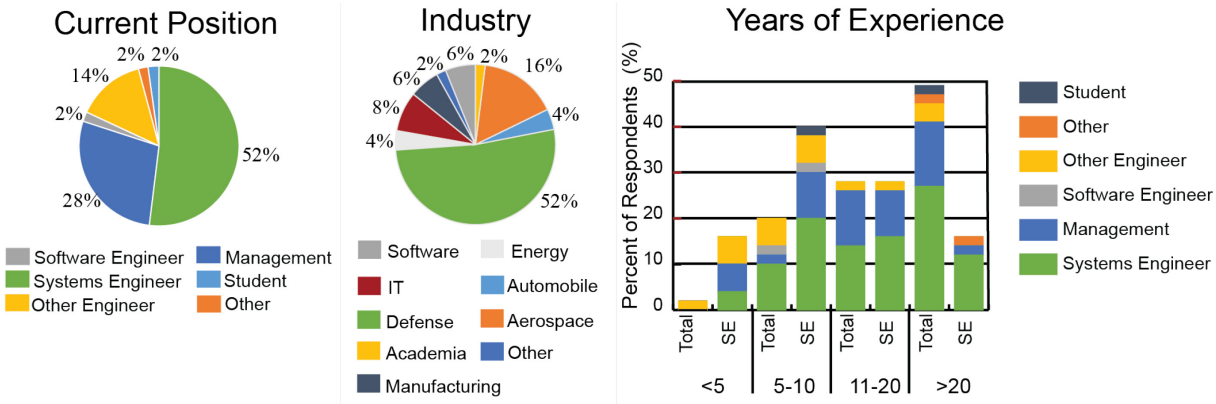

Figure 2-5. Demographics of survey respondents

The majority (52%) of the survey respondents listed systems engineer as their current position (shown in blue in Figure 2-5). 84% of the respondents reported more than 5 years of experience as a systems engineer, indicating that the respondents have substantial backgrounds in the field,

even if they are not currently serving as a systems engineer. These results show that the survey reached the targeted audience of experienced and professional systems engineers. Of note is that only 2% of the respondents listed themselves as a software engineer. One potential concern with the survey was familiarity with the technical debt term due to experience with software engineering. While later results will show that there is likely carryover of terminology from the software engineering field, the limited number of participants who identified as software engineers reduces the concern that the results are biased based on a large number of responses from software engineers.

The majority of respondents (68%) work in the Aerospace and Defense industries. The large section of respondents with similar backgrounds has the potential to bias the results towards those industries.

2.3.1.4.2 Technical debt is common in systems engineering

Question 2.1 asked if the respondents experienced the conditions that are defined as technical debt, without using the metaphor. 100% of participants responded that they had worked on such as system. Question 2.6 asked if participants had to correct issues associated with technical debt. 86% percent of the respondents stated that they have had to correct issues caused by less-than-ideal short-term solution.

The answers to question 2.1 and 2.6 clearly indicate that technical debt is a common occurrence within systems engineering. Every respondent experienced negative long-term effects due to short-term decisions, and the large majority of the respondents have corrected issues associated with these decisions. In other words, the respondents have repaid technical debt.

2.3.1.4.3 Technical debt accrues interest

Technical debt is typically measured in terms of principal and interest. The principal represents the amount of effort that would have been required to implement the ideal solution [62] and the interest refers to additional effort to implement that same solution at a later time, due to the presence of the less-than-ideal solution [68]. Question 2.7 addressed the presence of technical debt interest in systems engineering. If it is more difficult to correct the problems with a less-than-ideal solution than it would have been to initially implement the ideal solution, then it can be inferred that the technical debt has accrued interest. 79% of the respondents to question 2.7 stated that it was either more effort (36%) or significantly more effort (43%) to correct the issues after the less-than-ideal solution was implemented, as shown in Figure 2-6. These data indicate that technical debt accrues interest within systems engineering.

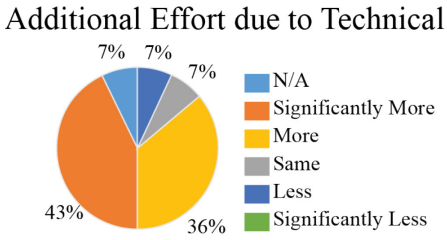

Figure 2-6. Additional effort required to correct technical debt compared to the effort to implement the ideal solution originally

Six survey respondents answered "no" to question 2.6, indicating that they never had to correct issues associated with technical debt. Of those six respondents, three answered that question 2.7 was not applicable to them (N/A in Figure 2-6), two did not answer question 2.7, and one respondent answered that correcting the issue required less effort. These answers are deemed to have no impact on the overall conclusions from this question, namely that technical debt does accrue interest.

2.3.1.4.4 Technical debt has multiple long-term impacts

Question 2.2 asked the participants to specify what negative long term impacts they had observed from implementing less than ideal short-term solutions. Participants were able to select more than one answer and the results are shown in Figure 2-7. "Failure to meet performance objectives" and "Substantial rework of an earlier part of the system" were the most common responses. Only 4% of the participants selected "Other", indicating that the answer choices well covered the negative impacts due to technical debt.

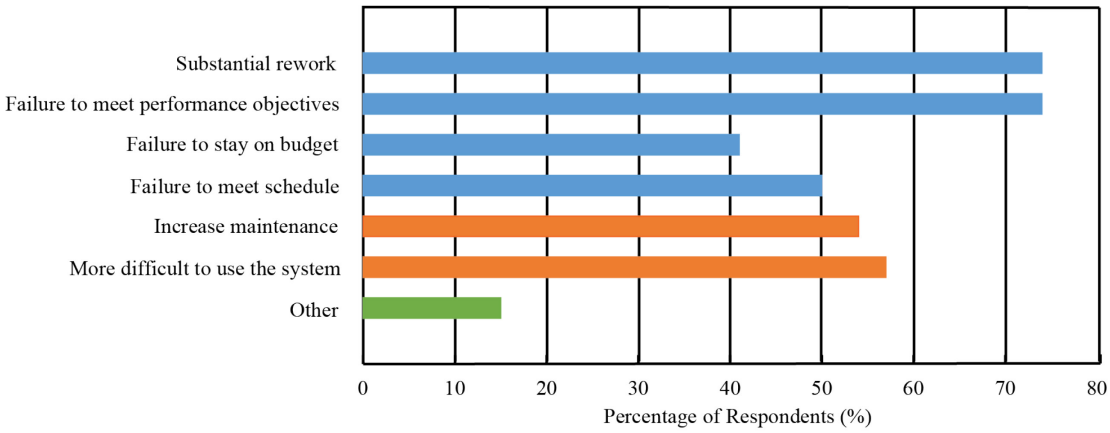

Figure 2-7. Negative long-term impacts of technical debt

These data indicate that there is not a single impact of technical debt on a system but rather that the impact is felt in multiple areas. The answer choices cover two areas: those that occur during system development, shown in blue in Figure 2-7, and those that occur after the system is deployed, shown in orange in Figure 2-7. Over 50% of respondents indicated that negative long-term impacts occur in both of these areas. From these data, it can be concluded that technical debt is something that will need to be managed throughout the system lifecycle.

2.3.1.4.5 Technical debt is driven by schedule and cost pressures and both intentional and unintentional decisions

Questions 2.3, 2.4, and 2.5 addressed the reasons why a project would take on technical debt. 79% of the respondents indicated that potential long-term consequences were considered when making short-term decisions. These long-term consequences were determined to arise from both intentional and unintentional decisions, as shown in the left side of Figure 2-8. The right side of Figure 2-8 shows the reasons for accruing technical debt. Over 80% of the respondents stated that schedule pressure contributes to the decisions to introduce technical debt into the system. Over 60% of the respondents stated that cost pressure contributes to the introduction of technical debt. Technical compromise was selected by 36% of the respondents. These results indicate that cost and schedule are the primary factors that drive a system to make technical compromises and therefore incur technical debt.

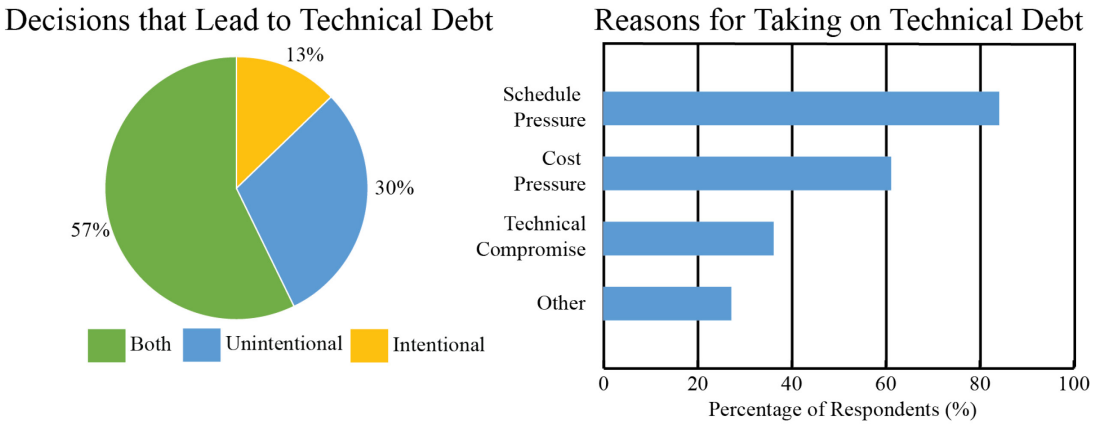

Figure 2-8. Rationale for accruing technical debt

Participants were allowed to select multiple answers for question 2.5, including identifying other reasons for taking on technical debt. Other responses included acceptance of a prototype, political pressure from management and other external sources, the lack of consideration of long-term goals and impacts in the daily decisions, and the inability to react to previous instances of

technical debt. Failure to react to previous instances of technical debt is an indicator that a system may be on a path to technical bankruptcy.

2.3.1.4.6 The technical debt metaphor is not common terminology in systems engineering

QG3 assessed the participants' familiarity with and usage of the technical debt metaphor after providing all participants with a common definition of technical debt. The left side of Figure 2-9 shows the self-assessed familiarity with the metaphor, broken out by the participant's years of experience as a systems engineer. 47% of respondents stated that they were very or extremely familiar with the metaphor and 30% of respondents stated that were either slightly familiar or not at all familiar with the metaphor.

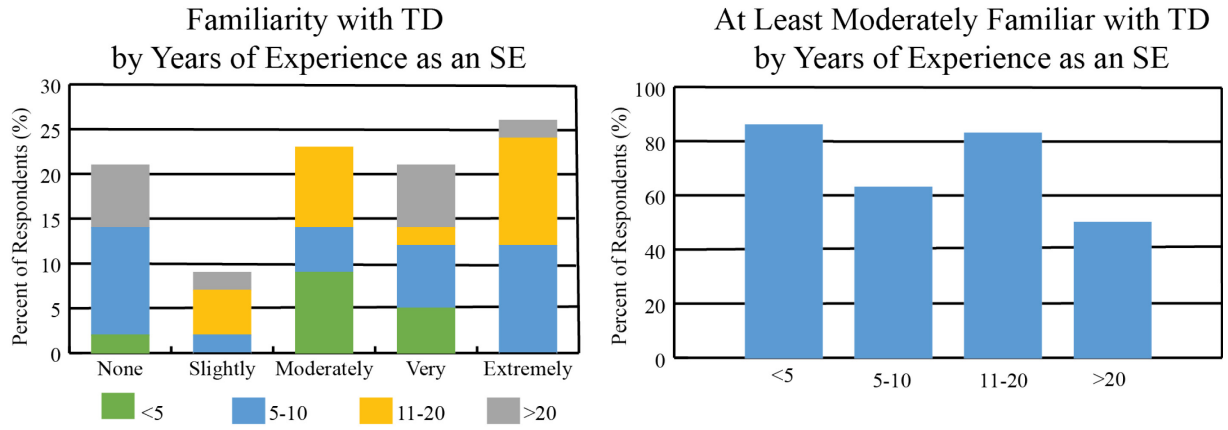

Figure 2-9. Participant familiarity with the technical debt metaphor

When examined through the lens of years of experience as a systems engineer, some interesting trends appear. The right side of Figure 2-9 shows the percentage of respondents who are either moderately familiar, very familiar, or extremely familiar with technical debt based on the respondents' years of experience as a systems engineer. The percentages are based on the total number of respondents with the stated years of experience. For example, seven respondents had less than five years of experience as a systems engineer. Of those respondents, six stated that they were at least moderately familiar with technical debt, resulting in a value of 86%.

While there is not enough data to make conclusive arguments, it can be seen that the less experienced (< 20 years of experience) systems engineers tend to be more familiar with technical debt than the very experienced systems engineers (> 20 years of experience). This could be a result of the small sample size (only eight respondents had > 20 years of experience); however, it could also indicate that the technical debt terminology is better known to less experienced systems engineers due to the relative newness of the terminology. Technical debt research in software engineering accelerated around 2008 [60]. Systems have become more software intensive [131] and familiarity with software engineering is now part of recommended systems engineering graduate school curriculum [132]. It is possible that these trends contribute to a greater familiarity with the metaphor among less experienced systems engineers.

Figure 2-9 shows that overall, there is familiarity with the technical debt metaphor among systems engineers. However, familiarity with a term is not enough to establish that the term is a common part of the lexicon. Therefore, participants were asked to identify how frequently they use the technical debt metaphor and in which technical contexts it is used. These results are shown in Figure 2-10. The left side of Figure 2-10 shows the usage of the technical debt metaphor. Only 26% of the participants reported using the technical debt metaphor frequently (gray) or very frequently (yellow), and 56% of the participants reported not frequently using the metaphor (blue). These results indicate that the metaphor, while it may be familiar to systems engineers, is not commonly used.

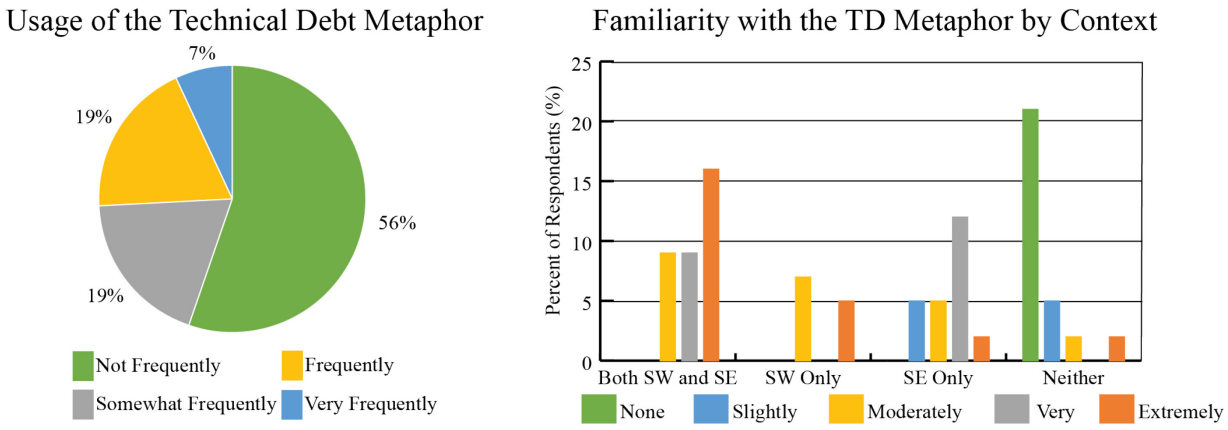

Figure 2-10. Usage of and familiarity with the technical debt metaphor in various contexts

The right side of Figure 2-10 shows the answers to question 3.4, which assessed the contexts in which participants have used or heard the technical debt metaphor. 35% of the respondents reported that they have used or heard technical debt in both systems engineering (SE) and software engineering (SW) context. Only 23% of the respondents said they have only used or heard the term in just the SE context and 12% of the respondents stated that they have used or heard the term in just the SW context. A likely interpretation is that the familiarity with the technical debt metaphor from software engineering produces carryover usage in the field of systems engineering. Of note is that the respondents who indicated usage in both the SE and SW context also indicated higher levels of familiarity with the technical debt metaphor. From these results, it can be concluded that the technical debt metaphor is present in the systems engineering lexicon, however, it is not a frequently used component of that lexicon.

2.3.1.4.7 Technical debt occurs throughout the system lifecycle

QG4 focused on technical debt in the system lifecycle. The participants' responses, shown in Figure 2-11, demonstrate that technical debt occurs throughout the system lifecycle, both in terms of its creation and its impact. The left chart in Figure 2-11 shows the design phases where technical debt is most likely to be created and most likely to be observed, according survey responses. These

data show that technical debt is more likely to be created during the design phases of the system and that the impact is more likely to be observed during the integration, verification and validation, and operations phases. These results show why technical debt is dangerous to a program – it is created based on decisions made in early phases, but the impacts are not felt until later phases, when it is more difficult to correct the issues.

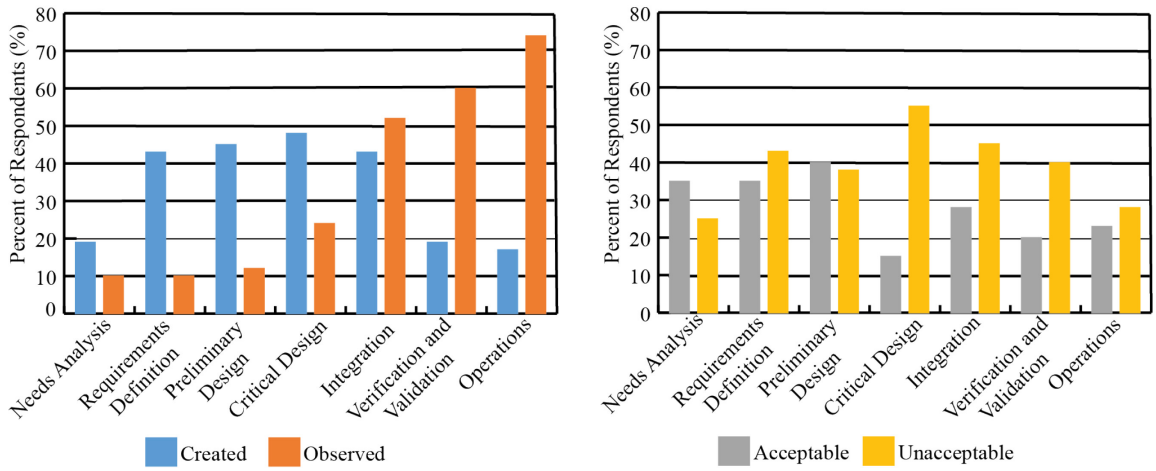

Figure 2-11. Technical debt in the system lifecycle

Questions 4.3 and 4.4 asked if there are specific phases within the system engineering lifecycle where it is more or less acceptable to create technical debt. The right side of Figure 2-11 shows the responses to these questions. The results show that participants generally viewed technical debt created in the early phases of the program to be more acceptable, with technical debt created during critical design to be the most unacceptable. However, the critical design phase is the phase that was indicated as the most likely place for technical debt to be created. These data support the premise that unmanaged technical debt is dangerous to a system. Technical debt is created where it is deemed unacceptable to do so, since creation in those phases is likely to drive to poor outcomes for the system. Therefore, it is critical to manage the creation of technical debt to prevent later impacts.

2.3.1.4.8 Technical debt can be beneficial

The phrasing of the technical debt metaphor implies that the consequences of technical debt are always negative. If true, then it would be expected that the survey respondents would never have indicated that creating technical debt was acceptable. However, as shown in the right side of Figure 2-11, over 30% of respondents identified that technical debt is acceptable to create in the early stages of the system lifecycle. Why would technical debt creation be acceptable?

While the survey did not ask this question, a reasonable answer is that technical debt creation is acceptable if it provides a benefit to the development of as system. The initial technical debt metaphor highlighted this aspect of technical debt stating "A little debt speeds development so long as it is paid back promptly with a rewrite" [17]. Taking on technical debt can enable a system to achieve critical results, such as delivering on schedule, even if compromises are made in the design. However, without a plan to repay the debt, it may spiral out of control and result in technical bankruptcy.

2.3.1.5 Discussion

This survey provides an empirical basis for understanding the prevalence of the technical debt metaphor in the field of systems engineering. The results can be used to draw several conclusions based on the research questions.

2.3.1.5.1 RQ1: Impact and occurrence of technical debt in systems engineering

The survey results clearly indicate that technical debt commonly occurs within systems engineering. The impacts of technical debt, such as increased effort and increased rework, were clearly identified by survey participants. Participants identified factors that can lead to technical bankruptcy, such as failure to meet cost and schedule, as impacts of technical debt. Participants

identified that technical debt creation during early system development phases can be acceptable, indicating that there can be benefits to taking on technical debt.

The confirmation of technical debt as a contributor to project success and failure means that it needs to be managed within the systems lifecycle. Tools need to be created to identify, manage, and monitor technical debt to minimize its impact. If a system developer waits until the impact of technical debt is seen in the system, it may be too late or too expensive to correct the issues. The survey results show that technical debt is more likely to be observed later in the system lifecycle, when it is more expensive to correct problems [121]. Therefore, technical debt needs to be monitored from the start of the system and should be repaid soon as possible.

2.3.1.5.2 *RQ2: Causes of technical debt within systems engineering*

Multiple factors contribute to technical debt; however, schedule pressure was cited as the top cause by the survey respondents. Schedule pressure is a significant concern in iterative development programs. As systems embrace Agile development strategies, they are often faced with fixed-duration development periods (sprints). Each sprint is intended to deliver a potentially releasable product [24]. This combination naturally exerts pressure on the developer to release a working system and can result in the developer taking shortcuts, intentionally or unintentionally, in order to make the delivery timeline. Proper planning involves sequencing tasks based on both the functional value delivered to stakeholders and on the temporal value delivered to the system. Understanding both the functional and temporal dependencies in the system development is critical for avoiding the need to incur technical debt. Supporting requirements, such as quality requirements (maintainability, reliability, etc.), must be given proper weight such that future iterations can begin with all the required infrastructure in place, even if they are not perceived as

high-value to the stakeholder. Otherwise, the future iterations are likely to need to take shortcuts, and thereby take on technical debt, to account for the missing components.

Another major driver of technical debt is cost pressure. The system may reach budget limits that require compromise in one area or another. For example, insufficient funding for testing may result in insufficient tests being performed on the system. The lack of testing may then result in an underperforming system. Budget allocations must be sufficient to enable proper system development, or else the system risks accruing technical debt.

While technical compromise was not cited by as many respondents as cost and schedule pressure, it was still cited as a cause of technical debt by over 30% of the respondents. Technical compromise means that the system developer makes technical concessions in one area to enable satisfaction of technical goals in another area, such as reducing the size of a satellite antenna to satisfy the mass constraints. If the full impacts are not assessed, the technical concessions can result in a system that cannot meet its overall performance goals.

2.3.1.5.3 RQ3: Use of the technical debt metaphor among systems engineering practitioners

The respondents to the survey stated that they had a broad range of familiarity with the metaphor of technical debt and that they were more familiar with it than their coworkers. However, they also responded that they do not frequently use the metaphor. These results indicate that the metaphor is not prevalent among systems engineering practitioners. Yet, the responses to QG2 indicate that the impacts of technical debt were observed by all survey respondents. This apparent disconnect highlights that the impacts associated with technical debt are real, but that it is not part of the lexicon of systems engineering. Instead, systems engineers use terms such as rework [125] and unintended consequences [98], however a detailed examination of the terminology in current use was outside the scope of this survey.

The survey results show that systems engineers understand some aspects of technical debt, such as the implications of short-term decisions on the long-term health of the system. However, the lack of general usage of the metaphor implies that the full richness of the technical debt metaphor is not used or understood. Simply delaying work does not result in technical debt and identifying the potential for rework does not quantify the impact on the future state of the system.

The use of inconsistent vocabulary creates barriers to effective communications even amongst practitioners in the same field [129]. The technical debt metaphor, through its use of concept such as principal, interest amount, and interest probability, can create a consistent vocabulary to allow systems engineers to quantify the impact of decisions. The quantified impact can then be used to support the decision-making processes during system development. Technical debt ontologies have been proposed within software engineering [133]; however, even the definition of technical debt is not agreed upon within systems engineering [79]. The results of this survey indicate that the technical debt terminology is not widespread within the systems engineering field, and this may be due, in part, to the lack of a consistent ontology. Development of such an ontology, specific to systems engineering applications, will aid in furthering the understanding of the impacts of technical debt and developing strategies for managing technical debt when it occurs.

2.3.1.5.4 RQ4: Occurrence of technical debt within the systems engineering lifecycle

The survey results show that technical debt is more likely to be created early in the systems engineering lifecycle and also more likely to be observed late in the systems engineering lifecycle. This combination results in an accumulation of interest on the technical debt and is what makes technical debt expensive to the systems developer.

Of particular interest is the combination of the most respondents stating that technical debt is likely created during critical design and the most respondents stating that it was unacceptable to

create technical debt during critical design. These data indicate that systems engineers may "know what they are doing is wrong" during the critical design phase, and yet they do it anyway – intentionally creating technical debt to get the design completed. If there is no plan to manage and pay back this technical debt, then it can be harmful to the system. This technical debt will then likely appear in the integration and/or operations phases. These results also indicate how technical debt can arise – schedule pressures and other outside influences can force the system developer to take those short cuts to complete the design by a set time. These data reinforce the need to manage and monitor technical debt. It is when the most critical elements of the development occur that taking on technical debt is most likely, and also the most unacceptable.

2.3.1.6 Conclusion and Future Work

Kleinwaks, Batchelor, and Bradley [19] proposed a research agenda to develop a systems engineering-centric view of technical debt. This agenda includes:

- Gathering empirical data to baseline the usage of the technical debt metaphor and the impacts of technical debt within systems engineering applications;
- Developing an ontology of technical debt for the field of systems engineering, developing methods and techniques to identify causes and occurrences of technical debt within systems development, developing processes and methods to measure technical debt; and,
- Verifying and validating the processes developed through application to systems engineering problems.

This survey represents the first step in the above research agenda and its results form the basis from which the above research agenda can be continued. The survey provides an empirical basis for the usage of technical debt within the systems engineering field and future work will continue to develop this usage. The survey results will guide the development of the ontology of technical

debt by providing area of emphasis where common language is required. For example, the prevalence of the impact of technical debt is clear from the survey results, but respondents do not use the same terminology. Additional surveys can be conducted to determine the terminology that is used in practice, which will further inform the development of the ontology.

This survey has provided a substantial amount of empirical evidence leading to the following key conclusions:

- Technical debt is common in systems engineering applications, but the associated terminology is not frequently used.
- Technical debt results in problems with system performance, cost, and schedule and bears interest – it requires more effort to correct the problem then it would have taken to do it correctly in the first place
- Cost and schedule pressure are the primary drivers of technical debt
- Technical debt is created early in the system lifecycle and observed late in the system lifecycle

The impacts of technical debt on a system are real and substantial. By enriching the usage of the technical debt metaphor within systems engineering, a common language can be used to manage and reduce those impacts. This research will continue to fulfill the above research agenda to provide a mechanism for managing technical debt to reduce the risk of technical bankruptcy.

2.3.2 Addressing RQ1.2

A summary of the results of the survey, originally presented at the *2023 INCOSE International Symposium* [134], is shown in Figure 2-12. The major conclusions from the survey are:

- Technical debt is common in systems engineering, but not commonly discussed

- Schedule pressure is the leading cause of technical debt
- Technical debt results in rework and performance issues
- Technical debt is generated early in the systems lifecycle but not observed until late in the system lifecycle

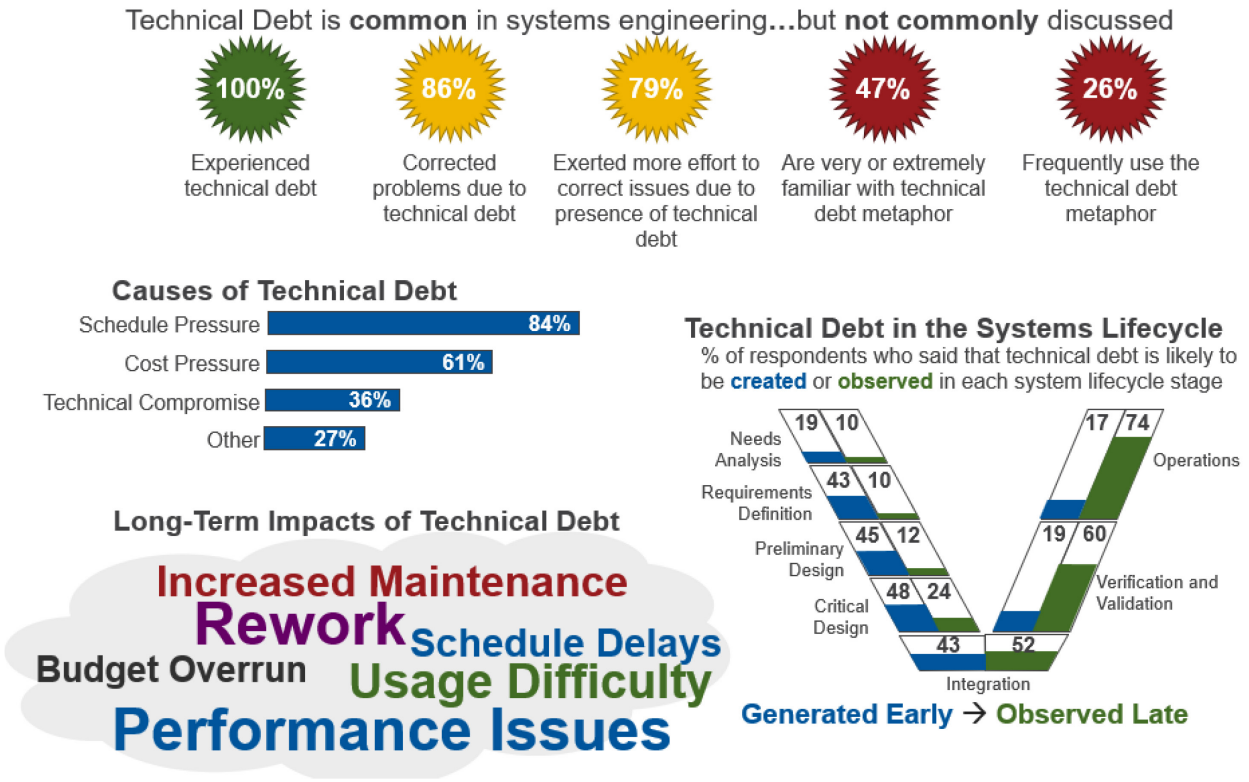

Figure 2-12. Summary of results from the survey on the prevalence of technical debt from [134]

The top graphic in Figure 2-12 illustrates the conclusion that technical debt is common in systems engineering but not commonly discussed. Every survey participant experienced technical debt, but less than half were familiar with the metaphor and only 26% of the participants frequently use the metaphor. The lack of common usage of the terminology prevents sharing of methods to mitigate and manage technical debt. As shown in the bar chart in Figure 2-12, the survey identified that the primary cause of technical debt is schedule pressure – system developers are pressured to make technical compromises to release the system on time. Cost pressure, which requires technical compromises to save budget, is also a significant contributor to technical debt. The word cloud on

the bottom left of Figure 2-12 highlights the long-term impacts of technical debt, with more popular responses written in larger font. The survey identifies that rework and performance issues are the two most common impacts of technical debt, followed closely by usage difficulty. The graphic on the bottom right of Figure 2-12 shows where technical debt is likely to be created (left side, blue) and observed (right side, green) throughout the system lifecycle. The numbers indicate the percentage of survey respondents who stated that technical debt was likely to be created or observed in the identified lifecycle stage. The most common stages for technical debt to be created were requirements definition, design, and integration, which constitute the early stages of the system lifecycle. The most common stages for technical debt observation were verification and validation and operations, which are the later stages of the system lifecycle. This result indicates why technical debt can be dangerous to a system – it is often not found until the later lifecycle stages where it is more expensive to correct [135]. Therefore, proactive techniques for identifying and managing technical debt are required to minimize its impact.

The survey provides results to address RQ1.2 and concludes that even though the concept of technical debt is familiar to systems engineering practitioners they do not regularly use the technical debt metaphor. These results indicate that the lack of a common lexicon may be preventing the communication of mitigation and management strategies.

2.4 RQ1.3 What Common Ontology Should be used to Describe Technical Debt Within the Field of Systems Engineering?

The literature review and the survey make clear two results:

1. Technical debt within systems engineering is not well researched and technical debt terminology is not commonly used
2. Technical debt is a problem within systems engineering

The facts that technical debt within systems engineering is not well researched and that the terminology is not commonly used could lead one to believe that technical debt is not a problem within systems engineering. However, the survey results state otherwise – 100% of the respondents have experienced technical debt. Therefore, it can be concluded that the problems associated with technical debt, such as the long-term impacts of short-term decisions, are researched, but not under the same terminology. A quick literature review finds work on rework [112] [136] and unintended consequences [98] that are similar to the concepts associated with technical debt. However, the definitions of these terms are not agreed upon, with eight different definitions of rework found in [136].

Therefore, enabling a discussion about technical debt within systems engineering first requires the development of a lexicon that can be shared amongst practitioners. Many authors have proposed definitions of specific components of technical debt within systems engineering [92] [137] [138], however these are mostly geared towards creating taxonomies to classify technical debt types or components. Although there is greater agreement among terms within software engineering, there is not a general ontology that can be applied to systems engineering [21]. An ontology is a set of definitions while a taxonomy is a system for classification [129]. An ontology provides a common lexicon for practitioners to use to discuss similar problems and solutions. A taxonomy can then evolve from the ontology. For example, a taxonomy of technical debt types within systems engineering can evolve from a systems engineering technical debt ontology.

This section addresses this problem and answers RQ1.3 by creating an ontology for technical debt in systems engineering. In accordance with Task 1.3.1, the ontology provides a definition of technical debt, principal, interest, and other key terms specific to systems engineering. This ontology of technical debt for systems engineering was created and published in the *IEEE Open*

Journal of Systems Engineering in September, 2023 [21]. This ontology is based on the published research in systems engineering and software engineering and informed by the results of the literature review and survey conducted as part of this dissertation. The ontology paper is reprinted here.

2.4.1 An Ontology for Technical Debt in Systems Engineering [21]

2.4.1.1 Abstract

The technical debt metaphor is used to describe the long-term consequences of engineering decisions made to achieve a short-term benefit. The metaphor originated in the field of software engineering and has begun to migrate to other fields, including systems engineering. The usage of the metaphor, its associated terminology, and basic definitions vary both within the software field and within the greater engineering community. The lack of consistent definitions inhibits the ability of system developers to understand and control technical debt within their system developments. This paper presents an ontology for technical debt, focusing on the field of systems engineering. By providing a set of concise and consolidated definitions, this ontology enables precise discussion of technical debt and associated techniques for mitigating its impact within systems engineering.

2.4.1.2 Introduction

Technical debt (TD), originally defined within the context of software engineering [17], is becoming a standard part of the technical lexicon, used by system engineers [77], program managers [139], and corporate executives [140]. But what exactly is technical debt? It has been variously defined as the long-term impact of compromises made for short-term benefit [60], the difference between the planned system capabilities and the actual system capabilities [114], a promise to complete work in the future [78], the acceptance of a short-term solution that will create

additional work in the long-term [77], and all the "technical work that has to be completed in the future" [140]. Further complicating the problem is the use of different terms, such as rework [125] and unintended consequences [98], to define similar problems. Even these terms do not have consistent definitions, as up to eight different definitions of rework have been found within the same paper [136]. These conflicting sources make it clear that a common definition of TD does not exist, neither in the broader research community nor specifically within the field of systems engineering [79].

The definitions of the components of TD also vary from author to author. Tom et al. [52] mapped the components of TD to the associated forms of TD, showing that multiple components can be classified into more than one form of debt. Li et al. [60] define the "cause" of TD as "the reason for the existence of technical debt", which corresponds to the "precedent" defined by Tom et al. [52]. Rios et al. [137] use the term "consequence" to identify the impacts of TD on the system while Tom et al. discuss the impacts in terms of the "attributes of technical debt." Alves et al. [133] define an ontology of TD types, but do not provide details on terminology beyond those types.

To address the terminology differences, several authors have developed taxonomies of TD. A taxonomy provides methods to classify items while an ontology provides definitions of those items [129]. Taxonomies are necessary to enable the classification of different TD types, however, an accepted ontology is required to provide the basis for those taxonomies. Yang, Verma, and Anton [141] recently defined a taxonomy focused on the incorporation of custom off the shelf (COTS) products into complex systems. They expand Kruchten's TD landscape [115] to include an additional 'Accountability' access and define several factors leading to different types of TD. Tom et al. [52] also define a taxonomy of TD, including methods for classifying the TD based on precedents, outcomes, and attributes. They define several attributes of TD, such as technical

bankruptcy, but do not provide a full ontology and their definitions do not necessarily extend beyond the field of software engineering. Alves et al. extended their earlier work to provide a taxonomy of TD types [142]. Several other authors have proposed taxonomies related to TD [137], [138], [92], but the taxonomies focus on classifying TD and do not provide consistent definitions that can be uses across industries [20].

Furthering the problem, TD is not well researched within systems engineering literature [19]. The authors have provided empirical evidence that TD does occur within systems engineering, even if the terminology is not widely used [18]. This survey identified that technical debt is more likely to be created early in the system lifecycle and its impacts are more likely to be observed later in the system lifecycle. The lack of a common ontology for common systems engineering problems prevents the systematic identification of similar research and therefore the sharing of tools and techniques to manage TD and to mitigate its impact throughout the system lifecycle [20]. With TD occupying significant portions of corporate technology portfolios [140], the management of TD is increasing in importance and value. Within specific systems engineering contexts, the problem of TD is increasing with the push to release products on shorter timelines [2] and an increased emphasis on prioritizing value delivery over non-functional requirements [15]. These pressures can result in developers taking shortcuts [7] and systems that break more easily when changes are required and which are more difficult to maintain, both of which are symptoms of unpaid TD [25].

Based on this examination of the state of the art in the field, it is clear that there is a need for a common ontology for TD. While multiple taxonomies exist, the authors are unaware of a comprehensive ontology of TD, particularly when considered in the field of systems engineering. Establishing a common language for TD is a key step to enable cooperation between the business

and technology sectors of a company [143] and to enable communication between practitioners throughout the field. This paper develops such an ontology for the field of systems engineering, which will enable a consistent discussion about TD and its management within systems engineering [129]. Standardization of terminology and definitions will lead to knowledge sharing and the development of measurement and management techniques.

Communication between practitioners is especially important as systems become increasingly complex and combined into systems of systems. In these cases, TD can be incurred in one system and then compound throughout the systems of systems. With increased complexity, identifying the source of the problem so that it can be remedied can become difficult and expensive, especially since the source of the TD may be far removed from its impacts. These factors are exacerbated within systems engineering, compared to software engineering, due to the increased interactions with external influences that may be outside the control of the system developer. Therefore, an ontology that provides a common basis for discussions across the entire system context is critical to managing systematic risks.

The rest of this paper is structured as follows. Section 2.4.1.3 presents the proposed ontology for TD for systems engineering. Section 2.4.1.4 discusses the use of the ontology and Section 2.4.1.5 concludes the paper and presents concepts for future work.

2.4.1.3 Technical Debt Ontology for Systems Engineering

2.4.1.3.1 Technical Debt Concept Map

The development of an ontology for TD starts with a conceptual understanding of technical debt. The following example, originally provided in a survey on the prevalence of TD within systems engineering [18], demonstrates how TD can impact the development of a system.

"Sydney is a test engineer tasked with writing test procedures to ensure that each part being manufactured is of sufficient quality. Sydney has substantial experience working with the parts and the test equipment. Sydney writes test procedures that outline the steps to execute the tests such that executing the tests should take one hour on each part. Following these procedures, Sydney can verify the quality of each part in one hour. Months later, Sydney is promoted and Jody is given the responsibility of testing the quality of the parts. Jody is new to the company and to the specific product line. Jody follows Sydney's test procedures, but Jody takes two hours to test each part, instead of one, reducing the overall throughput of the test team. Why?

The test procedures were written at a level relevant for Sydney's use and not for someone with less experience on the product line. Doing so saved Sydney time, but also increased the amount of time that someone unfamiliar with the testing would need to test each part, which introduced technical debt into the system. While Sydney saved time in creating the procedures, the system took on debt in the form of a less than ideal set of test procedures. The debt impacts the system when it takes Jody longer to test each part and slows down the process. In this case, paying back the debt requires rewriting the test procedures such that they are at a sufficient level for any engineer, regardless of experience, to be able to use efficiently. The system suffered from delays due to the increased time to evaluate each part and also from the time to rewrite the test procedures. This technical debt can impact the project schedule, the cost of the project, and also the quality of the outputs. Was this example helpful in explaining technical debt?"

This example highlights the major concept of technical debt: a technical compromise made to achieve a short-term benefit creates additional costs in the long-term. To visually explain the concept of TD, Izurieta et al. [144] developed a conceptual map of TD for software engineering which was extended by Rios et al. [137]. The concept maps visualize the major components of TD and associate these components with the system and business goals. While a useful aid in understanding the fundamental concepts of TD, these maps contain some notable deficiencies, including the lack of a feedback loop between TD and the system performance.

To address these concerns, a modified concept map of TD within the context of systems engineering has been developed based on a synthesis of TD components identified in the literature and interactions between the system and its stakeholders. This concept map is shown in Figure 2-13. In this concept map, the business goals exert pressure on the system and its developers, who

are then forced to make a technical compromise. This technical compromise can yield a short-term benefit, which satisfies the business goals, but which may create TD.

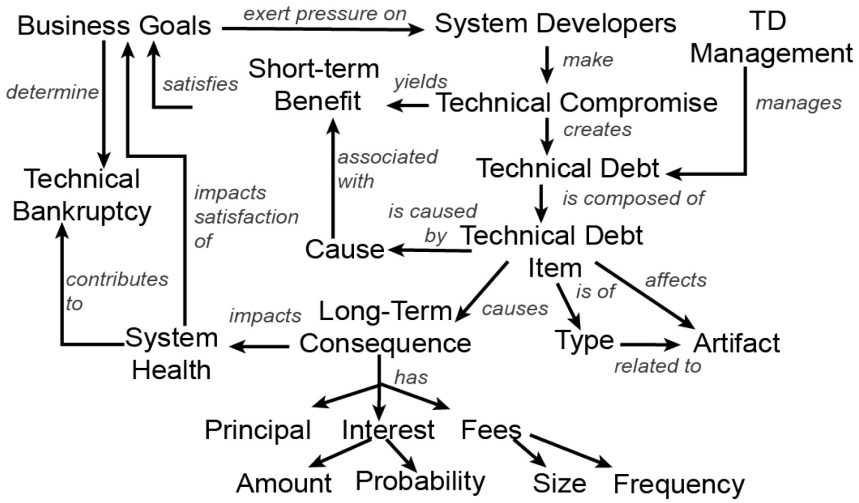

Figure 2-13. Conceptual map of technical debt for systems engineering, based on [29] and [13]

Technical debt is composed of one or more technical debt items (TD Item), which have several attributes, including the affected artifact, the type of TD, the cause of the TD Item, and the long-term consequences. The cause is associated with the short-term benefit that satisfies the business goals. The long-term consequence is measured in principal, interest, and fees and impacts the system health – the ability of the system to meet its performance objectives. These impacts on the system health can also impact the satisfaction of the business goals, which leads to additional pressure on the system or to technical bankruptcy. TD management is an activity associated with the control of TD items. This map shows the feedback between TD and system health as an indicator of system performance.

The concept map defined in Figure 2-13 provides a starting point for the creation of a TD ontology by identifying the relationship between the critical components of TD. This ontology is designed to provide common terminology and definitions that are focused to the systems engineering field. It leverages terminology from the software engineering field where possible.

However, the ontology also redefines terms and introduces new terms as necessary to clarify the definitions and usages within systems engineering specific applications.

2.4.1.3.2 Background Terminology

This section defines the background concepts and terminology used to establish the ontology.

2.4.1.3.2.1 System Dimensions

The development of a system can be characterized along three major dimensions: budget, schedule, and performance, where performance is defined as the combination of the system scope (what the system does) and its quality (how well it does it). These dimensions are linked together through a concept similar to the "Iron Triangle" of program management [145]: stakeholders must conduct tradeoffs between the three dimensions. For example, the customer can define the scope through a requirements specification and can define a delivery timeline. The system developer then determines the cost of the project that provides the developer with a sufficient value. The value is not necessarily a profit-driven parameter; a project may have other value to a system developer, such as development of new technology. Alternatively, if the stakeholder asks for a product within a specified budget and on a specified schedule, then the scope of the deliverable may need to change. In this case, the stakeholder must conduct a tradeoff between the achievable scope and available budget and schedule.

The triangle concept can be represented visually, as shown in Figure 2-14, where the vertices are performance (P), profitability ($), and speed to market (T). The area of the triangle represents the value of the system. As the values of the dimensions change, the vertices will move, altering the system value. The farther the vertices are from the center of the triangle, the larger the area of the triangle and therefore the larger the value provided by the system: faster time to market, higher profitability, and better performance all deliver higher value. Increased costs cause the profitability

vertex to move left, which lowers the area of the triangle and decreases the overall value. Similarly, realized cost savings increase the profitability, moving the vertex to the right and increasing the overall value.

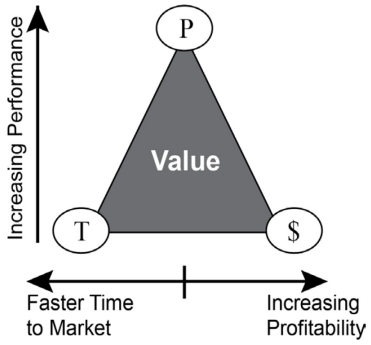

Figure 2-14. Interconnected system dimensions showing an estimation of system value

2.4.1.3.2.2 Phases

System lifecycles flow through characteristic stages, regardless of the development strategy that is employed [146]. Different strategies result in different frequency and numbers of iterations through the system lifecycle. Broadly speaking, the systems lifecycle can be divided into two phases: system development and system deployment. The development phase might consist of the following stages, adapted from [146]:

- Needs Analysis: definition of system capabilities to satisfy stakeholder needs
- Requirements Definition: decomposition of stakeholder requirements into system requirements
- Preliminary Design: development of design specifications to prove the ability of the system to meet requirements
- Critical Design: detailed design of the system
- Integration: implementation and integration of the components of the system

- Verification: verification that the components and the integrated system meet the requirements

The deployment phase might consist of the following stages:

- Validation: validation that the integrated system meets the stakeholders needs
- Operations: post-development phases of the system, consisting of production, use, maintenance, and retirement of the system

Within this ontology, the development phase will be used to refer to the activities leading up to system validation and the deployment phase will be used to refer to activities that occur during and after system validation, including production, operations, maintenance, and retirement.

2.4.1.3.3 Technical Debt Definition

Cunningham introduced the concept of TD stating:

> *"Although immature code may work fine and be completely acceptable to the customer, excess quantities will make a program unmasterable, leading to extreme specialization of programmers and finally an inflexible product. Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object- oriented or otherwise."* [17]

He used the term as a metaphor and not a definition. Since its introduction the metaphor has proven useful in explaining the impact of technical decisions in terminology familiar to personnel who are not involved in the system development.

With its increased use, there is a need to provide a consensus definition of the TD metaphor. Many authors have provided definitions of TD, especially within the realm of software engineering. These definitions have subtle differences and nuances; however, the following components are common across the definitions:

- TD occurs due to decisions made for short-term benefit that have long-term negative consequences [54] [147] [148] [110] [149] [65].
- Taking on TD involves making a compromise in one area to achieve a benefit in another area (e.g., reducing the quality of testing to save schedule) [110] [66] [51] [62] [58] [89]
- The effect of taking on TD is an increased amount of work in the future [66] [150] [49].

Some authors propose alternate definitions of TD, including presenting it as a gap between the actual and should-be state of a system [114], [52], [120], the existence of incomplete or immature components [78], [67], or work not yet done [77]. We assert that these definitions do not reflect the central tenant of the Cunningham's initial concept – that the decisions made today may result in increasing consequences tomorrow.

Rosser and Ouzzif provide a systems engineering based definition: "Expedient engineering decisions in requirements, architecture, design, documentation, integration and test are made to gain short term advantage, with similar negative effects on productivity and quality as have been shown in software" [70]. This definition is cumbersome and does not detail the negative effects, instead relying on a foreknowledge of the application of TD within software engineering.

Jones et al. define TD as consisting 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 more costly or impossible" [151] This definition does not define what an "expedient construct" is and whether it is due to poor design or intentional choices. Additionally, this definition states that TD only impacts the system when future changes are required. However, as will be discussed later, there is a component of TD associated with the use of a system.

To enable clear communication, a concise and easily understood definition of TD is preferred. Therefore, the definition of TD for systems engineering proposed by Kleinwaks, Batchelor, and Bradley [19] is adopted here:

Definition 1: *Technical debt is a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a system.*

Definition 1 identifies the TD metaphor – the application of the concept of TD to describe potential problems within a system. The metaphor is used to talk about the abstract concept without necessarily relating it to concrete numbers and measurements. However, the term "technical debt" is also commonly used to refer to "the complete set of TD items" [56] within the system and as a value representing something the system "owes". When used in this context, the term technical debt takes on a different meaning, as listed in Definition 2. To limit the confusion, the term "TD metaphor" is used in the conceptual context and the term "technical debt" or "TD" is used in the quantitative context.

Definition 2: *Technical debt is the quantitative impact on the long-term health of the system accrued as the result of a technical compromise made to achieve a short-term benefit.*

These definitions of TD consist of four main components: technical compromises, short-term benefits, and the potential for negative impacts, and the long-term health of the system. The following sections explain these components in more detail.

2.4.1.3.3.1 Technical Compromises

Referencing Figure 2-14, compromises can be made that affect one or more of the system dimensions. For example, the stakeholder can compromise on budget by adding funding or

compromise on schedule by delaying delivery until the system reaches the specified level of quality. Technical compromises are defined in Definition 3.

Definition 3: *A technical compromise is a concession made in the performance dimension, either in scope or quality.*

Only decisions that require concessions in the performance dimension are included in this definition. Decisions such as increasing the development timeline to enable the full realization of the system design (concession on schedule, benefit on performance) do not constitute TD.

2.4.1.3.3.2 Short-term Benefit

A system-level benefit is an increase in system capability in one of the three dimensions. A benefit in schedule would be the reduction in the time required to release a product. A benefit in performance would be the increase of capability in one area of the system. A short-term benefit is one that quickly realizes the benefit for the stakeholders and the system developers. For example, releasing a product two days earlier is a short-term benefit. A long-term benefit would be one that is not manifested until later in the system lifecycle. For example, an increase in system documentation may produce a benefit by reducing the complexity of system level maintenance and a corresponding increase in the performance dimension. Generating the documentation during the system design phase results in a long-term realization of the benefit. The actual calendar times associated with short-term and long-term are subjective and dependent upon the system being developed.

Decisions that do not yield short-term benefits do not constitute TD. For example, the decision to invest in the development of a new factory instead of running additional shifts at the current factory provides a long-term benefit instead of a short-term benefit, and therefore does not constitute TD.

2.4.1.3.3.3 Potential for Negative Impacts

Unlike financial debt, TD has intrinsic uncertainty about when it will need to be repaid and exactly how large the cost will be to repay the debt. Cunningham captured this concept when he said “The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt” [17]. If system developers have to interact with the portion of the system that has TD, then they will have to expend additional effort to develop that portion of the system. However, if developers never interact with that component, then the technical compromise will not impact the system development. Definition 1 captures the probabilistic nature of TD – there is a chance that the debt will need to be repaid and also a chance that the debt may not negatively impact the system. The probabilistic nature of TD must be considered when making the initial technical compromise.

2.4.1.3.3.4 Long-term Health of the System

The result of the technical compromise is often a long-term impact on the system health, if the technical concessions are not restored. Cunningham recognized this fact when he stated “Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object- oriented or otherwise” [17]. If the technical concessions that are made are left uncorrected, then the system health may become compromised over time. Kothamasu et al. define the health of a deployed system as the ability of the system to stay in an operable condition [152], characterized by margins in design specifications, lack of observable damage to the system, system reliability and performance parameters that are within the required bounds, and lack of any issues that would compromise the integrity of the system [153]. However, technical compromises affect both the development and operational phases of the system lifecycle [18] and therefore an updated system health definition covering both phases is required.

Definition 4: *The system health is the ability of the system to meet its objectives in the performance dimension without changes to the budget or schedule dimensions.*

During system development, objectives in the performance dimension include designing and implementing the system in line with the scope and quality requirements. After the system is deployed, these objectives include meeting the quality requirements, such as usability and maintainability. A system that fails to meet either set of objectives within its planned schedule and budget is unhealthy. In development, an unhealthy system requires additional funds and/or schedule to deliver the required performance. After deployment, an unhealthy system underperforms its requirements, especially in areas of maintenance and usability.

To be considered TD, the impacts on the system health must be long-term. The use of the long-term qualifier implies that the impacts will be remain in the system unless they are corrected. A short-term impact of a decision is resolvable and, if resolved, may have no significant impact on future changes. As such, this type of decision is an alternative design choice and does not incur TD [66]. For example, a test is specified to be conducted with a flight model of a satellite component. However, the component is delayed, and in order to keep the test schedule an engineering model of the satellite component is used instead. This decision represents a technical compromise – the exact flight unit is not tested. However, if the engineering model is of sufficient quality, the test results will be valid and not require retesting and there is no long-term impact.

2.4.1.3.3.5 Impact of Technical Debt on the System Dimensions

Figure 2-15 shows the progressive impact of TD on the overall value of the system. Section 1 of the diagram shows the baseline system, with target performance (P), profitability ($), and speed to market (T) objectives resulting in a defined system value. TD affects the long-term health of the system through a concession made in the performance dimension. The system may still meet the

overall scope requirements, but the concessions may make additional changes more complicated. This system state is represented in section 2 of the diagram. Here, a small amount of TD has been introduced into the system (the arrow labelled TD) which does not have a significant impact on the overall value of the system. Section 2 represents system states such as taking on prudent, deliberate debt, which is debt incurred with a known repayment plan, to meet a specified release date, where the reduction in performance is acceptable. In fact, not all TD taken on during the course of system development is detrimental, as debt incurred intentionally to meet a deadline may benefit the system [116]. Prudent TD can be recovered in future releases.

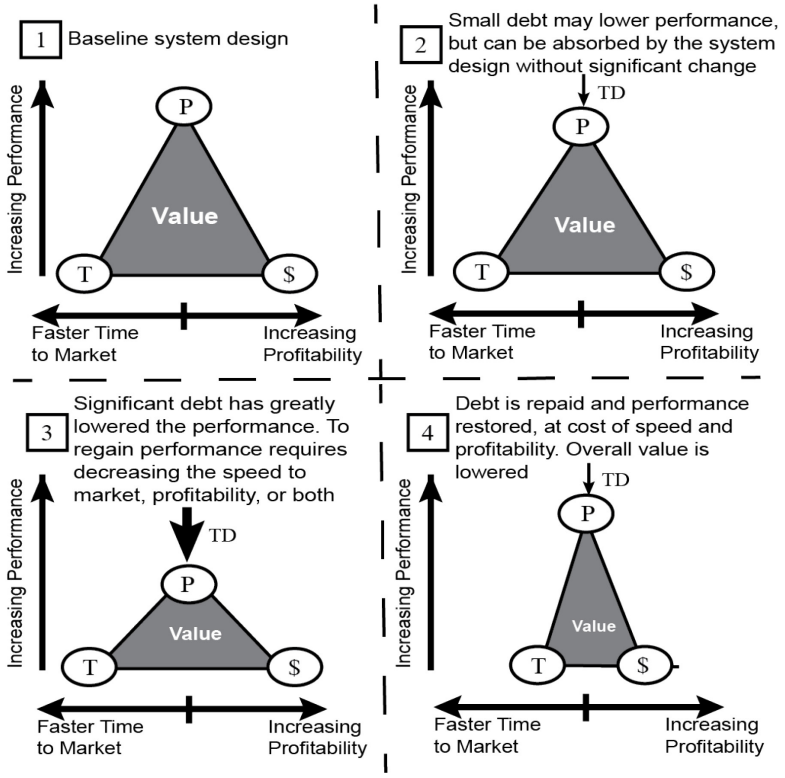

Figure 2-15. Impact of TD on project schedule, performance, and cost during project execution. Restoring system performance requires reducing the time to market or the profitability of the project.

In section 3 of Figure 2-15, the TD has grown significantly, indicated by the larger arrow, drastically reducing both the system performance and the overall system value. The performance can no longer be recovered without adjustments to the other vertices. This state arises from the accumulation of TD, either through reckless and inadvertent means, where technical debt is

incurred without a defined repayment plan [57], or through the failure to follow the TD repayment plan. Section 4 of Figure 2-15 shows how the system performance can be recovered, by paying down the TD. The schedule and profitability vertices have both been moved inwards to prop up the performance vertex, representing an increase in time to market (schedule delays) and a lower profitability (increasing cost). While the performance has been restored, the overall value of the system (the area of the triangle) is reduced.

This analysis shows how TD, which is incurred in the performance dimension, can have impacts on the other system dimensions. The system dimensions are interconnected and therefore require a system-level view of TD in order to mitigate the impact of the debt. This ontology provides the communication framework necessary to support the system-level view.

2.4.1.3.4 System Technical Debt

"System technical debt" refers to the set of TD items present in the system. It is used to address the accumulation of TD from multiple sources within a system and separates the definition of the total TD (system TD) from the TD associated with each TD item (technical debt). The system TD provides a method to quantitatively understand the accumulation of TD within the system.

Definition 5: *System technical debt is the concrete set of TD items present in the system.*

2.4.1.3.5 Technical Debt Measurement Units

Addressing TD quantitatively requires that TD be measured in a consistent unit across all occurrences. A consistent measurement enables comparison of the impact of multiple TD items. TD has been measured as the financial cost [65], the amount of time required to do the work [68], and the work required to be performed [67]. Each of these terms have varying degrees of usability in systems engineering, and roughly align to the three system dimensions of budget (financial cost), schedule (time required), and performance (work required). Definitions 1 and 3 state that

TD starts with a technical compromise in the performance dimension of the system that results in impacts on the long-term health of the system. Definition 4 states the health of the system is the ability to meet requirements in the performance dimension. The health of the system can be measured in the performance dimension by assessing the change in performance required to achieve the objectives. Since both the technical compromise and the long-term health can be measured in the performance dimension, TD should also be measured in the performance dimension.

The units used to measure the performance dimension will be different across different systems. For example, some systems may choose to measure the performance dimension based on the number of labor hours required to complete the work. Other systems may measure the performance dimension in terms of the lines of code that need to be written and other systems may use the number of verified requirements. Still other systems may convert the performance dimension into strict financial terms. With respect to TD, the specific unit does not matter, so long as each TD item is represented in the same unit for comparison and the unit used is understood to the system development team. Given the freedom of a system to report instances of TD in their own units, the term UNIT will be used within this ontology as the measurement of the TD.

Definition 6: *The UNIT of technical debt is a quantified measurement of a change in the performance dimension of the system.*

2.4.1.3.6 Technical Bankruptcy

Technical concessions may increase the cost to develop new features and the costs to maintain the system [63]. The project development schedule may be exceeded due to the impact of the technical concessions. The impacts in the performance dimension may be so severe that the system development cannot continue or that the maintainability and reliability of the deployed system is

insufficient [62]. Reaching any of these conditions puts the system into a state of technical bankruptcy, defined as follows:

Definition 7: *Technical bankruptcy is the state where the system can no longer proceed with its lifecycle until some, or all, of the system technical debt is repaid*

Systems that are technically bankrupt are no longer able to support future development without first repaying some or all of the existing system TD [60]. This situation can occur when the effort required to repay system TD exceeds the capacity of the development team. The development team will not be able to make progress on the system, resulting in a bankrupt state. Bankrupt systems are no longer able to either verify or validate their requirements within the system development timeline and budget [63].

A system may also reach technical bankruptcy once it is deployed due to an accumulation of technical fees. Technical fees, which define the increased difficulty in using the system due to the presence of unpaid principal, impact the quality of the delivered product. An excessive accumulation of fees will make the system unusable until the TD that resulted in the fees is corrected.

Systems reach technical bankruptcy when the technical costs, associated with system development and system use exceed system benefits, such as delivering on time and budget. Technically bankrupt systems require an increase in the budget or schedule dimension or a reduction in the expectations in the performance dimension to emerge from bankruptcy.

2.4.1.3.7 Technical Debt Item

A technical debt item (TD Item) is a concrete instance of TD within a system that connects the technical concession and its consequences on system artifacts [56]. The TD item represents the

concession that was made as part of the technical compromise and is used to track the impacts on the long-term health of the system. The following sections discuss each of the TD Item attributes.

2.4.1.3.7.1 Description

The description provides a narrative of the technical concession and the steps required to restore the system in the performance dimension.

2.4.1.3.7.2 Consequence

The consequence of the TD item refers to the potential impacts on the long-term health of the system [144]. It consists of a narrative description of the impacts and the quantitative measures of principal, interest, and fees.

2.4.1.3.7.3 Principal

The existing definitions of the principal vary, but are typically centered on the effort required to correct the issue causing the TD [60], [68], [69], [59]. Ampatzoglou et al. define it as "the effort that is required to address the difference between the current and the optimal level of design-time quality" [62]. Izurieta et al. state that principal "refers to the cost or effort (measured monetarily or in time units) necessary to restore a software artifact back to health" [61]. Avgeriou et al. identify principal as the "cost savings gained by taking some initial approach or 'shortcut' in development (the initial principal). Or the cost it would take now to develop a different or better solution (the current principal)" [56].

This quick review of the literature identifies two major methodologies for calculating the principal: it is either the UNIT to implement the optimal solution originally (the savings from the original concession), or it is the current UNIT to implement the optimal solution now. The principal measures the initial concession made as part of the technical compromise – it is the UNIT of system performance that is given up in order to achieve the desired benefit. The principal is like

the principal in financial debt – it does not increase with time, although payments can be made to reduce the principal and therefore the following definition is adopted.

Definition 8: *The principal, P, is a measurement of the concession made in the performance dimension to achieve a short-term benefit.*

2.4.1.3.7.4 Interest and Fees

The long-term impact of TD on a system's users is different than the impact on the system's developers. System users may experience decreased usability, maintainability, or reliability of the system. These impacts are likely to occur each time the system is used and to be the same magnitude with each occurrence. System developers may experience the same issues, but also may experience increased difficulties in continuing the system development to meet performance requirements. The impacts seen by developers occur when the system is modified, either due to the natural development process or due to changes in the system requirements. These impacts tend to be less predictable both in occurrence and in the magnitude of the impact. Therefore, the long-term impact of the technical concession needs to be considered from both perspectives. This consideration results in two separate quantities: interest and fees.

Interest and fees can be distinguished based on how they impact the system. Interest is based on impacts to the development of the system – if the technical concession results in increased costs, schedule, or difficulty in making modifications to the system, then the system has accrued interest. Interest is variable – both the interest amount and the interest probability are functions of the state of the system. Fees are based on impacts observed during usage of the system – if the system is more difficult or complicated to use as a result of the technical concession, then the system has incurred a fee. The magnitude of the fee is constant; however, the fee must be paid by the user every time that the impacted artifact is used.

Fees are paid by the user and interest is paid by the system developer. The system developer must repay the principal. The repayment of the principal constitutes the correction of the original technical concessions and removes the technical debt. This repayment must be done by the developers and then the updated system is released to the users.

2.4.1.3.7.4.1 Interest

Interest on TD is traditionally defined as the extra effort required to modify the system due to the presence of deficiencies [60], [54], [149], [62], [68]. TD interest has also been defined as the work to correct a deficiency [61], the additional work to implement new functionality [56], and the additional work to maintain the system due to the presence of the deficiency [63], [154].

TD interest results from the lower design-time quality of a component (poor documentation, low maintainability, etc.) that requires additional effort in subsequent development efforts. TD interest can be contagious [92] – each new component that interacts with a component containing TD may require additional effort to develop and may then carry forward that interest into its successor components (compounding the interest). If a sub-optimal component is included in the architecture instead of correcting the component (the principal), each new application that connects to that component would suffer from its sub-optimality [69]. The build-up of dependencies on the sub-optimal component results in overall sub-optimal performance and increased work to add new components (the interest).

Within the systems engineering context, the TD interest refers to the long-term impacts on the system as encountered by the system developers. The interest will accrue in the performance dimension of the system and can impact both the scope and the quality of the system. The interest definition, therefore, is limited to the impact on the system developers.

Definition 9: *The interest, I, is the expected value of additional UNITs incurred by the system developers in the performance dimension due to the presence of unpaid principal.*

Applying a direct financial analog to TD would require the definition of an "interest rate" for TD, which would associate the total amount to be repaid with a known growth rate of the debt. However, a relationship between TD principal and interest that would apply to all projects has not been defined. Ampatzogolou et al. suggest that such a rate cannot be defined, since the specific growth of TD interest depends on aspects unique to each system such as the system implementation, the system context, and the maintenance activities performed [68]. Due the complexities in calculating and predicting the effort associated with interest, Seaman et al. [51] divide the interest into two categories: interest amount and interest probability.

2.4.1.3.7.4.2 Interest Amount

The interest amount reflects the long-term change in the performance dimension that is traceable to the original concession (the principal) [51].

Definition 10: *The interest amount, a, is the additional UNITs incurred by the system developers in the performance dimension due to the presence of unpaid principal as a function of the state of the system.*

The interest amount represents the impact of the principal on the future state of the system. For example, if the principal was incurred due to a decision to not complete documentation, then the interest amount would be an increase in the effort required to update that part of the system in future iterations. The interest amount is measured in the same UNIT as the principal. The interest amount is a function of the state of the system development and the development timeline. For example, a system may initially have few interfaces and components, and the ability to work around the initial concession is small. As the system grows, the number of interfaces increases and

the impact of the initial concession spreads to a larger number of interfaces and components. Therefore, the change in the performance dimension has increased due to the larger number of impacted components, increasing the interest amount. The interest amount may also decrease due to changes in the system development, such as removing an interface.

2.4.1.3.7.4.3 Interest Probability

Unlike financial debt, which has a known schedule of payments and interest, TD interest may or may not be realized. Once the technical compromise is made, the principal exists in the system. The technical compromise may be made in a component of the system that never has to be altered again, and therefore the compromise does not need to be resolved. The interest probability accounts for the likelihood of the interest being realized [51]. For example, a system may choose smaller batteries that reduce the upgradability of the system. However, if those upgrades are not implemented, then the interest is never realized.

Definition 11: *The interest probability, r, is the probability that the interest amount will be realized as a function of the state of the system.*

Like the interest amount, the interest probability is also a function of the state of the system and the development timeline. As the system development changes, especially in iterative design cycles, the probability of the interest being realized may change. For example, an incompletely implemented standard may initially have a low interest probability if the component that implements the standard is isolated. If the system design changes such that the component is no longer isolated, then the interest probability will increase.

2.4.1.3.7.4.4 Fees

Users of a system use a released version of the system and therefore the system's capability in the performance dimension is largely fixed. Design choices made in the system development may result in a less-than optimal experience for the user. Activities may take longer than they should due to underperforming hardware or due to poor user interface design. Capabilities may not be fully implemented and require work-arounds by the user. The system may not be easily maintained or may not be as reliable as it should have been. These issues all tend to occur in the quality aspect of the performance dimension. Unlike TD interest, these issues occur with each use of the system. The total impact of the issues is dependent upon the number of times that the system is used and is not based on the effort required to add capability to the system or to modify the system design.

Therefore, these impacts on the health of the system are separated out from the TD interest and are instead termed TD fees. Fees are the recurring costs of using a system containing TD and are measured in UNIT every time that the system is used. Izurieta et al. [144] defined this concept as *recurring interest*. An example of a fee occurs when a poorly developed user interface results in several extra minutes spent inputting system parameters in a software system. Every time the user has to input the parameters, users will have to "pay the fee" of those extra minutes, until the principal on that TD Item (reworking the interface) is repaid by the developers. A fee is defined as follows:

Definition 12: *The fee, f, is the amount of additional UNIT incurred by the user with each use of the system due to the presence of technical debt.*

A system that performs poorly does not necessarily have fees. Fees must be associated with a technical compromise, and as such, an instance of TD. For example, a race car that is slower than its competitors does not necessarily have any fees associated with the use of the car – it is just not

as well designed as its competitors. However, a cost savings compromise made to use a metal frame instead of a composite frame which reduces the gas mileage would be an example of a fee – the user (the driver) must perform additional pit stops every time the car is raced.

2.4.1.3.7.5 Balance

The balance, B, is the summation of the principal and the interest and represents the total UNITs required to repay the TD item. The balance does not include the fees (either realized or anticipated) in the system, as fees are not repaid. The expected value of the balance is calculated as shown in Equation 2-1. The subscript, *t*, indicates the parameters that change with time.

$$\overline{B_t} = P + a_t * r_t \qquad (2\text{-}1)$$

2.4.1.3.7.6 Total Cost

The total cost, C, in terms of UNIT, due to the TD item is inclusive of the balance and the fees. The total cost is a time-dependent value, as it includes the interest, which is a function of the state of the system, and the expected fees. Fees are fixed in magnitude, but the number of fees, n, will change with time. The expected value of the total cost is calculated as shown in Equation 2-2. The subscript, *t*, indicates the parameters that change with time.

$$\overline{C_t} = B_t + f * n_t \qquad (2\text{-}2)$$

2.4.1.3.7.7 Artifact

The artifact is the part of the system that is affected by the TD [144]. A TD item may impact multiple artifacts – the principal may be associated with one artifact while the interest and fees may be associated with a different artifact. An artifact may be a piece of documentation, a component of the system, a test case, or any other part of the system itself.

Definition 13: *An artifact is the part of the system affected by technical debt.*

2.4.1.3.7.8 Cause

The cause of a TD item defines the reasons why the technical compromise was made [144]. It consists of two attributes: the specific cause and the cause category. The cause provides traceability of the TD item to the original decision which can then be used in forensic evaluations.

2.4.1.3.7.8.1 Specific Cause

The specific cause of a TD item is the short-term benefit provided to the system developers, stakeholders, or users that is realized through a technical concession. The specific cause includes the rationale for why achieving the short-term benefit required a technical concession. For example, a technical compromise may be made such that a program increment can be released on time. In this example, the specific cause is the on-time release of the program increment. The rationale defines why a technical concession had to be made to release the increment on schedule, such as supply chain issues forcing a switch to a different, less reliable part.

Definition 14: *The specific cause of a technical debt item is the short-term benefit realized through the technical concession.*

2.4.1.3.7.8.2 Cause Category

The cause category provides a general categorization of the cause. The cause category is defined as follows:

Definition 15: *A cause category is the dimension of the system development where the short-term benefits are achieved as a result of the technical concession.*

Kleinwaks, Batchelor, and Bradley [18] conducted an empirical survey of systems engineering professionals. This survey included questions on reasons why a system developer may incur TD. Over 80% of the respondents identified schedule pressure as a reason, over 60% of the respondents

identified cost pressure as a reason, and over 30% of the respondents identified technical compromise as a reason. Kruchten et al. similarly identified schedule pressure as the primary cause of TD [115]. These results lead to the following cause categories:

- Schedule: consists of pressures put on the technical solution due to the need for the system to meet schedule. For example, any TD incurred such that the system can meet its scheduled release date is caused by the schedule category.
- Cost: consists of pressures put on the technical solution due to the need for the system to stay on budget. For example, technical concessions associated with the use of a cheaper part are associated with the cost category.
- Performance: consists of technical concessions made in one area to achieve technical benefits in another area of the system. For example, a satellite system may choose to use a less performant antenna such that system mass requirements are met.

These categories mirror the system dimensions defined in Section 2.4.1.3.2 . As evidenced by Figure 2-15, pressure on any of the dimensions may result in movement in the other dimensions. Figure 2-16 shows an example of this process. Stakeholders, such as management executives, may put pressure on the system to release earlier in order to beat a competitor to market. This pressure pulls the schedule vertex (T) to the left as shown in section 1 in Figure 2-16. Without other resources, the movement of the schedule vertex would result in a corresponding decrease in system performance (P), shown in section 2 of Figure 2-16. This reduction indicates that a technical compromise is required, which introduces TD to the system. To restore the system performance, the cost vertex ($) is moved left, decreasing the system profitability, shown in section 3 of Figure 2-16. Therefore, the stakeholders would have to accept a tradeoff in the system – either a decreased performance and the introduction of TD or decreased profitability due to an increase in costs. Note

that the profitability factor here does not account for the potential future benefits of releasing the system earlier to market.

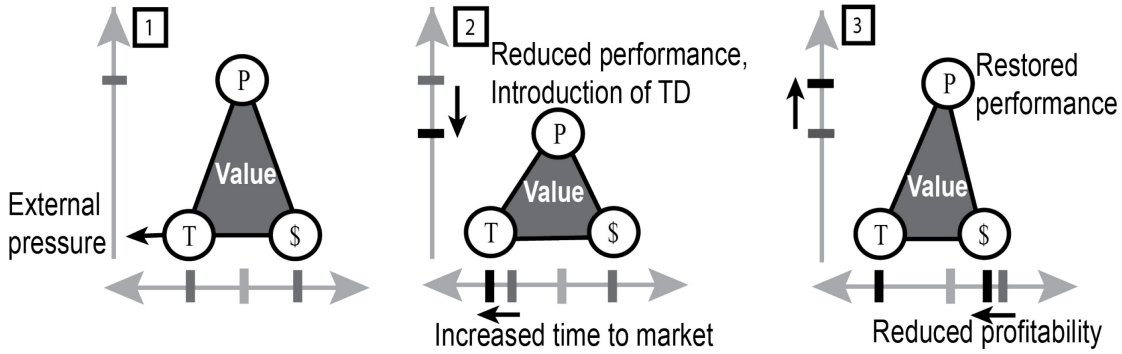

Figure 2-16. Example of schedule pressure creating TD

2.4.1.3.7.9 Type

The type of a TD item provides a means to categorize the TD item. TD items with similar types may have similar causes or similar methods for repaying the TD. Examples of different types of TD can be found throughout the literature and include items such as architectural TD [97], domain debt [89], and requirements debt [118]. TD occurs in various stages throughout a system's lifecycle and for various reasons. Classification of TD into different types assists in understanding and managing it, however, too many disparate types risk diluting the strength of the TD metaphor [115]. A definition for a TD type, such as that provided in Definition 16, can assist in restricting the accumulation of differing TD types.

Definition 16: *A technical debt type is a classification of technical debt based on the artifacts that are negatively affected by the technical concessions made to realize a short-term benefit.*

This definition restricts a TD type to be associated with specific artifacts and the technical concessions that are made. Domain debt, defined as the "misrepresentation of the application domain by an actual system" [89], is associated with the documentation of stakeholder needs and the system requirements. Technical concessions that result in domain debt can include limiting

user interactions to save development time. Defect debt, defined as any defect found within the system [60], would not be a type of TD according to Definition 16. Defect debt can be mapped to an artifact, such as the source code, but not to technical concessions. Defects are the result of poor work and are not inserted into the system to realize a short-term benefit.

2.4.1.4 Discussion

The ontology provided in this paper provides a starting point for developing a common framework for discussing TD within systems engineering. This commonality is critical to enable the sharing of methods and processes for identifying and mitigating the impacts of TD. The need for a common set of definitions can be seen by examining a listing of types of TD.

Kleinwaks, Batchelor, and Bradley [19] identified the types of TD found within published systems engineering research. Recognizing that creating too many types of TD risks diluting the strength of the metaphor [115], the types of systems engineering TD were reevaluated in context of this TD ontology. This evaluation resulted in the consolidation of the TD types as shown in Figure 2-17, with the types classified as interest bearing (associated primarily with impacts during system development), fee bearing (associated primarily with impacts during system usage), or both interest and fee bearing. Several of the identified types of TD proved to be instances of other types of TD. For example, versioning debt is an instance of documentation debt and not a separate type of TD. Automation debt, build/assembly debt, depreciation debt, and infrastructure debt were originally listed as different types of TD [19]. These types of TD all impact the same artifacts – the supporting tools used to develop the system. Therefore, according to the ontology, they represent different facets of the same type of TD. Figure 2-17 shows the subtypes as italicized items under the new parent type, which is listed in bold. After application of the definition of a TD type, several types of TD listed in [19] were found to not be TD types: defect, operations and

maintenance, and organic. These items reflect causes or impacts of TD instead of types of TD. This short example demonstrates the utility of the ontology – it provides clear guidelines of what is and is not TD and can prevent over classification, which impedes communication and the development of effective management strategies [115].

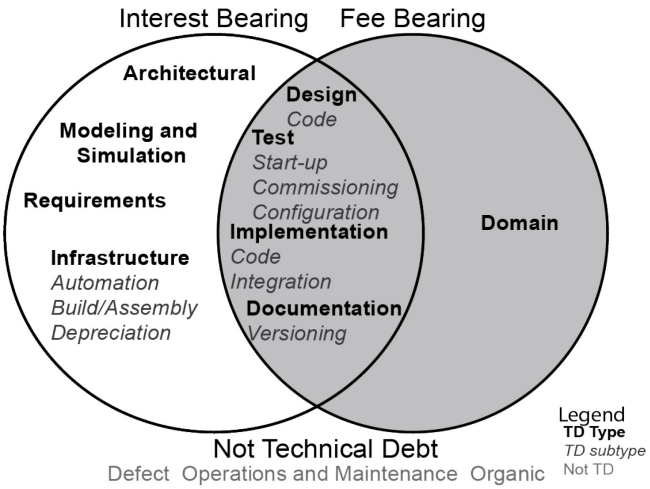

Figure 2-17. Consolidation of TD types from [19], organized by interest and fee bearing status.

2.4.1.5 Conclusions

Kleinwaks, Batchelor, and Bradley [19] proposed a research agenda for understanding TD in the context of systems engineering. This agenda includes baselining the knowledge of TD in the field of systems engineering through empirical data collection, developing a systems engineering ontology of TD, developing techniques to identify causes of TD within systems engineering, developing methods to quantify and predict the impact of TD within systems development, and verifying and validating these methods.

The first agenda item was addressed through a survey on the prevalence of TD in systems engineering [18]. The research question presented in this paper address the second agenda item – identifying an ontology of TD within the context of systems engineering. This research presents a starting point for the development of a complete ontology. It introduces and defines the key terms, with clear explanations. These explanations and definitions begin the creation of a common

lexicon and provides practitioners with the semantics necessary to create clarity in communications.

The ontology presented here is not complete. Further work needs to be performed to create taxonomies of TD types and of specific causes of TD. Too many classes of either TD types or specific causes can dilute the strength of the TD metaphor [115]. Future research needs to provide guidance on how to classify TD such that it is precise enough to be meaningful without overspecification.

The socialization of this ontology, of which this paper is the first step, will provide a starting point for clear and concise terminology usage within the field of systems engineering, which is a necessary step towards mitigation of the risks associated with technical debt and the prevention of technical bankruptcy.

2.4.2 Addressing RQ1.3

The technical debt ontology addresses RQ1.3 by providing a common ontology for technical debt within systems engineering. To demonstrate the utility of the ontology, the following sections provide example applications, including developing a consolidated set of types of technical debt within systems engineering, an example application of the ontology to the development of a notional satellite system, and a discussion on items that do not constitute technical debt based on the ontology.

Application of the ontology to the list of types of technical debt found in the literature review addresses Task 1.3.2 by providing concise definitions of types of technical debt applicable to systems engineering. The survey results are used to map these types of technical debt into the systems engineering lifecycle to identify the artifacts and phases affected by each type.

2.4.2.1 Technical Debt Types in Systems Engineering

With the ontology defined, it can be applied to multiple cases to demonstrate its utility. As an example, the types of technical debt in systems engineering identified in [19] can be reevaluated using the definitions contained in the ontology. Table 2-8 lists the types of technical debt in systems engineering identified in [19] after applying the ontology and associates the type with the affected artifact. In accordance with the definitions, Table 2-8 identifies the types of technical debt that are interest bearing (associated primarily with impacts during system development), fee bearing (associated primarily with impacts during system usage), or both interest and fee bearing. The following sections define these technical debt types in more detail. Several types of technical debt identified in [19] were found not to be technical debt after application of the ontology. Section 2.4.2.3 discusses why these types do not constitute technical debt.

Table 2-8. Technical debt types in systems engineering

Technical Debt Type	Artifact where Technical Concessions are Made	Interest or Fee bearing
Architecture	System architecture	Interest
Design	Component specifications and detailed system design	Interest and Fee
Documentation	System documentation	Interest and Fee
Domain	Statement of needs, system requirements, system architecture, test plans	Fee
Implementation	System components and integrated system	Interest and Fee
Infrastructure	Supporting equipment for system components, the integrated system, and verification and validation tests	Interest
Modeling and Simulation	Supporting models and digital twins	Interest
Requirements	System requirements	Interest
Test	Test artifacts	Interest and Fee

2.4.2.1.1 Architecture Debt

Architecture debt is incurred through technical compromises that impact the system architecture, such as decisions related to the structure and implementation of interfaces in the system [97]. Architecture debt can result in impacts on the quality attributes of the system, such

as maintainability and evolvability [155]. Architecture debt is present in both systems engineering and software engineering, and its use in systems engineering is a direct analog of its use in software engineering. Similar architectural decisions and processes are made in both fields. Technical concessions made in the system architecture result in the accrual of interest on those concessions. Architecture issues such as incompletely implemented standards, adoption of immature system components, and reliance on outdated architectures make developing and improving the system more complicated.

Examples of architecture debt include [97]:

- The Minimum Viable Product that Stuck: the focus on delivering value quickly to stakeholders results in the generation of a minimum viable product (MVP), which minimally meets the requirements. The adoption of the MVP as the baseline for the architecture can then result in immature system implementations that result in future complexities.
- The Workaround that Stayed: Architectural workarounds are introduced in the system, such as creating temporary interfaces between components. Over time, these workarounds become critical features of the system, even though the workaround may not have been implemented to the required standards.
- Re-inventing the Wheel: Components are developed when existing components with similar functions are available. This type of architectural debt is the result of build versus buy decisions and can cause future impacts due to the additional work associated with testing a new developmental item.

- Architectural Lock-in: Architectural components become deeply embedded in the system design, making them costly or impossible to replace. The use of these components can incur performance costs in the future if they cannot be upgraded.
- New Context, Old Architecture: The context in which a system is deployed may change. The failure to keep the system architecture updated with changes in the context can result in new fees being applied to the system every time it is used.

Architecture debt is interest bearing. It reduces the future evolvability and development of the system and the architecture becomes more difficult to change as system development progresses.

2.4.2.1.2 Design Debt

Design debt is incurred through technical compromises that impact the design of the system [110]. Design debt can be incurred during preliminary design or detailed design and can result from "an under-focus on qualities such as maintainability and adaptability, or subsequent piecemeal design with an absence of refactoring" [52]. Design debt can occur in hardware and software designs. Parts of code debt as identified in software engineering apply to design debt, such as violations of good object-oriented design [133].

Design debt is interest bearing and fee bearing. Concessions made during design will have impacts on the ability to make future modifications to the system. The design choices may also result in fees where design compromises result in a system that is more difficult to maintain or use.

2.4.2.1.3 Documentation Debt

Documentation debt refers to all issues pertaining to the documentation of the system, including poor documentation, version control, and configuration management. Versioning debt is included in documentation debt as version control and configuration management are related to document management. Documentation debt occurs due to "insufficient, incomplete, or outdated" [60]

documentation and applies to any of the documents used to design or maintain a system or a system component [70], such as user manuals, test plans, or source code comments.

Documentation debt incurs both interest and fees. Poor documentation related to the development of the system, such as insufficient source code commenting, incurs interest since it makes the future development of the system more complicated and costly. Poor documentation related to the usage of the system, such as an incomplete user manual, can make user training more complicated, incurring a fee at each training event.

2.4.2.1.4 Domain Debt

Domain debt is the "misrepresentation of the application domain by an actual system" [89] and impacts the system itself. The impact of this type of technical debt is seen when a system fails validation – the developers built a system that does not meet the needs of the end users. Domain debt can be caused by poor requirements and poor stakeholder involvement during development or by changes in the system context and use cases after development [63].

Domain debt results in fees – the user experiences low usability on a product that may not meet their needs. Domain debt may not impact the development of the system, however, decisions made during development can incur domain debt if they drive the system to a solution that is not representative of its intended use and operational domain [89].

2.4.2.1.5 Implementation Debt

Implementation debt occurs while the system is being built - after the design is completed and prior to the release of the system. During the implementation and integration phases, the system developer is often faced with decisions about how to implement the system. These choices do not change the system design or architecture. For example, a software system may choose an inefficient algorithm or produce code without following proper coding practices [60], resulting in

larger rework times the next time the software is used. In a hardware system, the use of a customized part and proprietary interfaces can make future updates more complicated [70].

Implementation debt is interest and fee bearing. The concessions made during system implementation and integration may make future system development more challenging, as in the case of an incomplete implementation of a standard. These concessions may make using the system more difficult, as in an under-implemented interface between two components. The latter case may result in unreliable or low-rate data flow across the interfaces, thereby increasing the time it takes to transmit data from a remote system.

2.4.2.1.6 Infrastructure Debt

Systems are not developed in a vacuum, instead they rely on infrastructure and supporting processes. Infrastructure includes the development tools used by the system developer, the supporting systems at the deployment location, and third-party components used by the developer within their system. Infrastructure debt includes technical concessions made in the configuration of the tools that are used to support the development and deployment of the system [60]. These concessions can be made in areas involving the depreciation of parts and components and the automation of machinery [70]. In software systems, concessions made with the automated build pipelines, such as the failure to include adequate cybersecurity scanning tools to save cost, constitute infrastructure debt. Technical compromises made to enable the use of commercial off-the-shelf (COTS) products, such as accepting lower system performance or more complicated integration, also constitute infrastructure debt [86].

Infrastructure debt is interest-bearing. Concessions made in the infrastructure related to the development of the system will incur interest when the development is more complicated, such as the need to modify a COTS product to meet the system requirements. Poor infrastructure in a

deployed system is a result of design decision and is perhaps more properly characterized as design debt.

2.4.2.1.7 Modeling and Simulation Debt

Digital engineering relies on a substantial increase in the use of modeling and simulation tools to verify system-level performance through analysis. Digital tools are used to verify requirements, assess system performance, and predict system behavior. If these models and simulations are performed to improper fidelity levels, are not maintained in parallel with the system under development, or are poorly documented, then the model's predictions will differ from the system reality. In these cases, the technical concessions made in the model development will impact the health of the system, as the system may need to be redesigned to meet the required performance parameters. Modeling and simulation debt is the result of technical concessions made in the digital engineering environment for a system [70].

Modeling and simulation debt is interest-bearing. Concessions made in the modeling phase can make the development of the actual system more challenging and result in redesigns. For example, if an overly simplistic thermal model of a satellite is used, then the satellite may be designed with insufficient radiators. During thermal testing, the satellite would not perform as expected and would require a redesign.

2.4.2.1.8 Requirements Debt

Requirements debt has been defined as the "distance between the optimal solution to the requirements problem and the actual solution, with respect to some decision space" [118], issues with requirements formatting and content [70], and trade-offs in the requirements specification [66] or implementation [133]. There is disagreement in the published literature about whether or

not requirements debt is a type of technical debt [149]. In systems engineering, requirements debt should be considered as a type of technical debt.

Requirements debt occurs when concessions are made during the formulation of the system requirements. Requirements debt can have large impacts on various aspects of the system, such as the design, implementation, and operations and maintenance, and could be seen as an analog of domain debt. However, domain debt occurs when the system developer does not implement the requirements in accordance with the stakeholder's expectations and requirements debt occurs during the formulation of the requirements themselves. Requirements debt results from compromises made when creating the system requirements and not from poorly formatted requirements. However, it can occur when a poorly formatted requirement results in an inaccurate understanding of the requirement. The lack of clear requirements can result in a system that requires redesign, additional acceptance testing, or multiple rounds of validation. Missing requirements can result in the need to add unbudgeted work to the system to complete it to specifications.

Requirements debt is interest bearing. Technical concessions made during requirements development, such as the failure to include all relevant stakeholders during elicitation and the failure to verify the requirements to confirm that they are complete and conflict-free, can have large impacts later in the system development process.

2.4.2.1.9 Test Debt

Test debt is incurred due to technical concessions made in the development of test artifacts and in conducting testing activities [133]. Test debt can occur due to shortcuts taken when executing the tests [60] or due to insufficient coverage of the system functions and behaviors in the set of test cases [70]. Specific instances of test debt include commissioning and start-up debt (shortcuts

in commissioning and startup process of automated systems [113]) and configuration debt (issues with the hardware configuration and availability for testing [70]).

Test debt is interest bearing as tests are part of the development process of the system. Test debt can lead to test failures which will lead to additional development work to correct the system. Test debt is also fee bearing. If the system is not fully tested, then inadequate performance may not be discovered during the operational phase, resulting in increased difficulty when the system is used.

2.4.2.2 Example usage of the Ontology

The use of the technical debt ontology for systems engineering is best understood through an example. This example considers the development of a notional communications satellite with a primary radio-frequency (RF) antenna for communications with users on the ground. The satellite is launching as part of a rideshare and therefore has a strict mass limit and a strict schedule. The satellite must be ready in twelve months or else it will miss the launch, and will have to sacrifice the budget allocated to the launch. If the satellite is over mass or fails the specified environmental testing, then it will not be allowed to connect to the launch vehicle. The RF antenna needs to connect to an existing set of ground terminals and ground antennas (more than one combination), without modifying those systems. Therefore, there are minimum performance requirements on the antenna. Further complicating the development of this system is the fact that a satellite cannot be serviced once it is launched, with the exception of software updates. This example examines the impacts of technical debt on the development path of the satellite. For this example, technical debt principal, interest, and fees will be measured in UNITs of time. Table 2-9 contains the details of the technical debt items created and observed throughout this example, identified by the step number associated with each paragraph. The column C/O indicates if technical debt is created (C)

or observed (O) and the column CC indicates the cause category for that technical debt item, with S used to indicate schedule, C used to indicate cost, and P used to indicate performance.

1. The first step in the system development process is identifying the stakeholder needs. The stakeholders identify an exemplar terminal and antenna to which the satellite needs to connect. The stakeholders do not identify any other constraints and, to enable the quick start of the system development on a tight timeline, the system developer does not pursue any further information. Domain debt is created.
2. Following the identification of the stakeholder needs, the satellite requirements and system architecture are defined. The requirements for the satellite include the overall mass limit. The architecture requires the development of a new antenna for the satellite; however, an existing radio can be used. The new antenna is an evolution from an existing antenna and the mass and power of the existing antenna are used in the initially proposed design. The exemplar terminal specifications that were provided represent a worst-case scenario for the antenna performance. No size requirements are flowed down to the antenna manufacturer as the previous design was small enough to fit within the launch vehicle restrictions. Requirements and modeling and simulation debt are created.
3. During the preliminary design of the system, the initial antenna models are revisited. It is determined that the antenna does not produce enough gain to close the link with the exemplar terminal. The design has to be reworked, resulting in an increase in size and mass of the antenna. Modeling and simulation debt is observed and design debt is created.
4. The critical design of the system takes longer than expected due to the larger mass of the antenna, which produced a need to rework the guidance systems. The mass increase also resulted in the need to increase the size of the satellite reaction wheels, which further

increased the mass of the system. To counteract the increase in mass, a battery is removed from the system, which decreases the duty cycle of the antenna. Design debt is observed and domain debt is created.

5. In the integration phase, the command-and-control software for the satellite and its payload are implemented from the design. Due to the short timelines, best practices for the development of the software are not followed and the documentation produced for the software is minimized. Implementation and documentation debt are created.
6. Following integration, the satellite is verified and validated, including the execution of the tests required by the launch vehicle provider prior to allowing the satellite to be integrated on the launch vehicle. To save time, radiation testing is not performed. Instead, the requirements are verified by analysis. Test debt is created.
7. Eventually, the satellite is qualified for launch and is placed into orbit by the launch vehicle. As the satellite enters operations, it becomes apparent that it does not meet the needs of the users. It is limited in duty cycle and experiences a limited ability to close the link with a majority of the terminals that it was supposed to support. Radiation events occur semi-frequently, causing the satellite to go out of service due to the need to reset the system. The software is complicated to update and patch, resulting in longer outages whenever an error occurs. Domain, test, and documentation debt are observed.

This example shows how technical debt can build up within a system through decisions made to achieve short-term benefits. As a result, the system was more expensive to develop and is less usable by its end-users, which could result in a monetary failure of the system.

Table 2-9. Example creation and observation of technical debt

Step	TD Type	C/O	Compromise	CC	Potential Consequence	Principal	Interest/Fee
1	Domain	C	Failure to identify all operational needs and constraints	S	System may not work for all user terminals, in all weather conditions, or at all ranges	Seven work days saved by not completing needs analysis	Fee: inability to transfer all data in one pass causes user to wait for a second orbit to complete transfers; due to orbital geometry delay can be 90 minutes to 12 hours
2	Requirements	C	Failure to consider all sources of requirements in development of subsystem requirements	S	Rework of design if launch vehicle constraints are not met	Two weeks earlier start on design	Interest Amount: four weeks of redesign if constraints are not met Interest Probability: 50%
	Mod/Sim	C	Simplified models used to assess system design	S, C	Rework of the design if simplified models are incorrect	Two months earlier start on design	Interest Amount: One month of redesign effort Interest Probability: 30%
3	Mod/Sim	O	N/A	N/A	N/A	N/A	Interest Paid: time spent to rework design, which reduces schedule margin and increases cost
	Design	C	Acceptance of larger antenna mass	P	Larger mass makes attitude control, launch deployment, and power management more complicated	Three weeks saved on antenna design optimization	Interest Amount: Two weeks of effort to redesign the reaction wheels and attitude control software Interest Probability: 90%
4	Design	O	N/A	N/A	N/A	N/A	Interest Paid: time spent to rework design, increasing cost and reducing schedule margin
	Domain	C	Lowering of battery capacity	P	Lower duty cycles reduce the usability of the system for the end user	Six weeks of effort to optimize other areas of the satellite to meet new power requirements	Fee: lower duty cycle results in ability to only service one user per orbit, creating operational delays from 90 minutes to 12 hours based on orbital geometry
5	Implementation	C	Not following software best practices for peer review	S	Increased risk of software errors and increased difficulty in updating software	Four weeks of development time saved	Interest Amount: Two days of additional work for each update Interest Probability: 75%

Step	TD Type	C/O	Compromise	CC	Potential Consequence	Principal	Interest/Fee
	Documentation	C	Reduction in software documentation	S	Increased risk of software errors and increased difficulty in updating the software	Four weeks of development time saved	Interest Amount: Four days of additional work for each update Interest Probability: 60%
6	Test	C	Radiation tests not performed	S	Susceptibility of system to single event upsets may decrease the usability of the system resulting in technical fees	Two weeks of test time	Fee: Single event upsets cause a reset of the satellite every day, which may result in missed connection opportunities or data loss, creating operational delays from 90 minutes to 12 hours based on orbital geometry
7	Domain	O	N/A	N/A	N/A	N/A	Fee Paid: Links to most terminals only close at high elevation angles, reducing effective communication times and increasing the time for a customer to receive service
	Test	O	N/A	N/A	N/A	N/A	Fee Paid: Single event updates cause frequent system resets and outages, increasing the wait time for customer to receive service
	Domain	O	N/A	N/A	N/A	N/A	Interest Paid: Increased timeline for patching and updating the satellite software due to poor documentation

2.4.2.3 Is It Technical Debt?

When evaluating potential technical debt within a system, it is simple to describe every problem as a type of technical debt. After all, most problems encountered in a system development will eventually cause issues. However, this generalization of problems can lead to two undesirable conclusions [115]:

1. Fine-grained distinctions between different types of technical debt prevent the development of efficient tools to manage technical debt; and,
2. Over-generalization prevents the application of the right tools, such as risk and schedule management techniques, to the right problem.

If every technical problem is not technical debt, then there needs to be a methodology for determining what is and what is not technical debt. The definition of technical debt from [21] will be used to address several situations that should not be considered technical debt. For an issue with the system to be considered technical debt it needs to meet the following criteria:

- A technical concession was made (trading schedule for budget does not constitute a technical concession);
- There was a short-term benefit received as a result of the concession;
- There is the potential for damage to the long-term health of the system; and,
- The technical concession can be associated with a system artifact.

Table 2-10 provides example categories of items that are often confused with technical debt but fail to correspond to the definitions provided in the ontology.

Table 2-10. Examples of items that are not technical debt

Category	Definition	Why it is Not Technical Debt	Could be Technical Debt If
Incomplete work	Unfinished work that is on schedule [58], new features, or additional functionality [115]	Not the result of a technical compromise	Work is left undone to meet a release, thereby representing a technical compromise
Defects and failures	Software bugs, poor work, errors in the system [49]	Not inserted into a system to achieve a benefit; tend to have an immediate impact on the system instead of a long-term impact [116]	Defects arise due to misinterpretation of the requirements; the defects are then a symptom of requirements debt
Technical compromise with no long-term impacts	Reducing the performance of the system through a technical compromise that still enables the system to meet its objectives	No long-term impacts since the system still meets all the objectives	Unexpected long-term impacts arise from the concessions
Poorly designed or implemented systems	Systems that are properly built to their requirements, but the design is poor	No technical compromises are made in the development of the system	Misunderstanding of the customer needs led to the initial requirements, resulting in domain debt

Using the definitions in the ontology, several types of technical debt identified in [19] no longer fit the requirements to be considered technical debt. These types include defect debt, operations and maintenance debt, and organic debt. Defect debt is defined as any defect found within the system [60]. However, as identified in Table 2-10, defects themselves are not technical debt, and therefore defect debt is not a type of technical debt. Instead, defects should be viewed as symptoms of other types of technical debt [49]. Operations and maintenance debt is defined as "any kind of handicap with adverse effects on the product or system maintenance" [112]. This definition is overly broad and the adverse effects on maintenance, which are technical fees, could arise from any of the other types of technical debt. Therefore, operations and maintenance debt should not be a separate category to avoid overspecification. Organic technical debt refers to the impact of external factors, such as decisions by management, on the system [86]. This definition implies that these external factors force the system developers to make technical concessions due to driving

concerns, such as limited budgets and speed to market. Per the ontology, these external factors are causes of technical debt and not a specific type of technical debt themselves.

2.5 Technical Debt in the Systems Engineering Lifecycle

Section 2.4.2.1 states that the types of technical debt are defined based upon the artifacts that they impact. By evaluating the stages in the system lifecycle based on where the affected artifacts are created and used, it is possible to map the types of technical debt to the system lifecycle. Figure 2-18 shows a mapping of the system lifecycle stages used in the survey by Kleinwaks, Batchelor, and Bradley [18] to the types of technical debt identified above. In the figure, each stage outputs an artifact, which then may be used as an input to another stage. For example, the statement of needs is used as an input to the requirements definition stage, the verification and validation stage, and the operations stage. These input/output relationships are shown via the blue arrows. The technical debt type that may be created in the development of each artifact is shown in red italicized text to the left of the artifact. The types of technical debt that impact the development or the usage of the artifact are shown in bold purple text to the right of the artifact. For example, the preliminary design stage produces the component specifications. During the development of these specifications, technical compromises may occur that result in design debt – the design of the specification is non-optimal. Therefore, design debt is shown in red italics to the left of the component specification icon in Figure 2-18. The process of developing the component specification itself may be more complicated due to the presence of requirements, architecture, or modeling and simulation technical debt. These types of technical debt are shown in bold purple text to the right of the specification icon in Figure 2-18. If technical compromises existed in the development of the requirements, such as not validating the requirements for consistency in order to meet schedule, then it may be impossible to develop proper component specifications. The

system developer would have to go back to the stakeholders to renegotiate the inconsistent requirements, which increases the execution time.

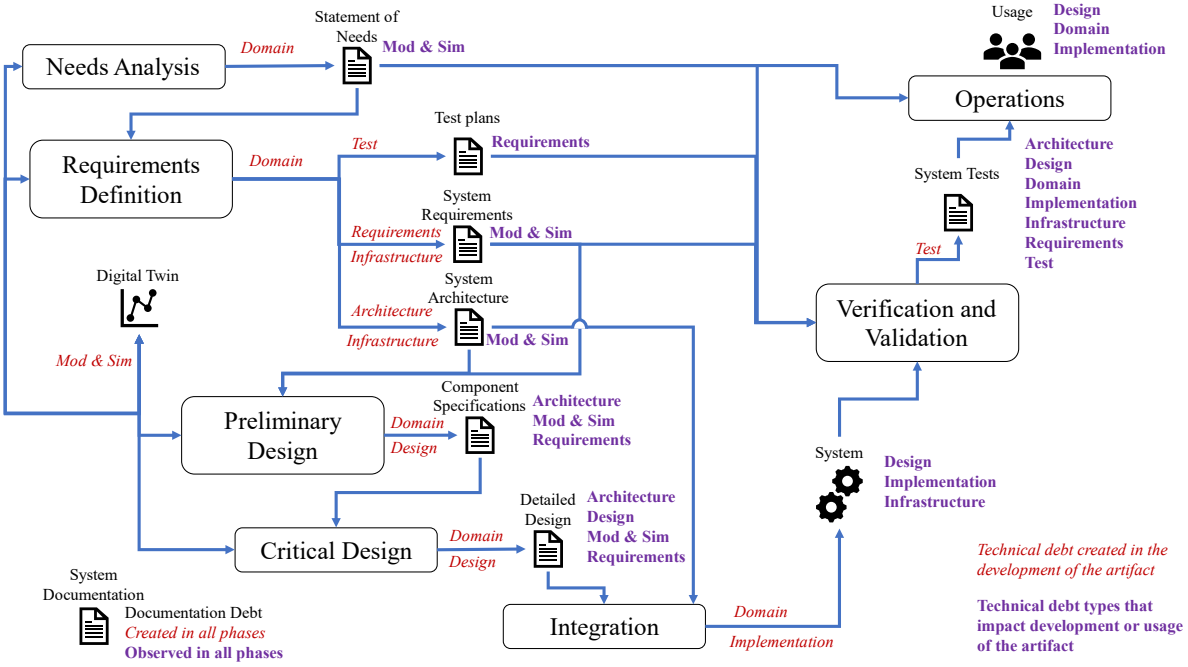

Figure 2-18. Technical debt type creation and observation based on impacted artifacts throughout the system lifecycle

Of note in Figure 2-18 is that modeling and simulation debt (Mod & Sim) is created during the development of a digital twin. The term digital twin here is used to refer to any models or simulations used during the development of the system. The digital twin provides inputs to and receives outputs from the needs analysis, requirements definition, preliminary design, and critical design stages. The system documentation is both generated and used throughout the system lifecycle. Therefore, documentation debt may be created or observed at any point in the lifecycle. The documentation artifact is not shown connected to the stages for clarity in the diagram. Table 2-11 defines the rationale for the depictions shown in Figure 2-18.

Table 2-11. Creation and impact of technical debt types

Technical Debt Type	Creation	Impact	Artifact(s)
Architecture	The system architecture is created at the end of the requirements analysis, which occurs during the requirements definition stage [146].	Compromises in the system architecture such as incompletely implemented standards can make system design more complicated. Similarly, these incomplete interface implementations can impact the ability to verify and validate the system.	System architecture
Design	The design consists of the component specifications and the detailed design, which are created during the preliminary and critical design stages [146].	The design directly impacts the system implementation and the creation of the system components and the integrated system. Compromises in the design may make the system harder to implement, require redesign late in the process, or make the system more difficult to use.	Component specifications and detailed system design
Documentation	Documentation occurs throughout the system development. Compromises in any source of documentation can create technical debt, such as poor user manuals, failure to capture requirements rationale, and poor source code commenting.	Documentation is used throughout the system lifecycle and any concessions made in the documentation may impact the ability to design, develop, and utilize the system.	System documentation
Domain	Domain debt occurs due to a misunderstanding of the stakeholder needs and operating environment. Domain debt is created throughout the system lifecycle system any time a technical compromise is made that results in a system that does not properly meet the stakeholder needs.	Domain debt is not observed until late in the system development cycle, when the system is validated by the stakeholders. Concessions made early, such as forgoing user input to meet schedule, show up late in a system that fails validation or does not operate as expected.	Statement of needs, system requirements, system architecture, test plans
Implementation	Implementation debt occurs due to technical concessions made while implementing the detailed design. These concessions can be intentional or unintentional.	Implementation debt impacts the system itself and begins to be observed in the integration stage as one part of the implementation can affect others. Concessions made in the implementation of the system can impact the verification and validation of the system and the operations.	System components and integrated system
Infrastructure	Infrastructure debt broadly captures the impacts on the elements in the system context, beyond the system itself, that are required to support the development of the system. It is created in the requirements phase when the infrastructure requirements are specified.	The impact of infrastructure debt is felt in the integration and verification and validation stages. If concessions are made with respect to the tools used to build or test the system, then the processes may not be as efficient or may require a redesign of the system to fit the available tools.	Supporting equipment for system components, the integrated system, and verification and validation tests

Technical Debt Type	Creation	Impact	Artifact(s)
Modeling and Simulation	Modeling and simulation debt is created during the process of defining models, simulations, and digital twins that support the design of the system. An example of a concession made in the modeling and simulation process is using low fidelity models to decrease model run time.	Digital twins are used to support the development of the system needs, requirements, architecture, and designs. Therefore, any concessions made in the digital twins will impact these processes.	Supporting models and digital twins
Requirements	The requirements are created during the requirements definition stage [146]. Requirements debt occurs if the system does not properly validate the requirements to ensure that they are conflict free, understood, and support the stakeholder needs.	Requirements debt impacts the design of the system and the verification and validation of the system. Concessions made in the requirements can result in a system design that is inaccurate, which can then result in the failure of verification testing.	System requirements
Test	Test debt is associated with the test plans and processes used to test the system. It is created during the development of the test plans as part of the requirements definition and again during the verification and validation stage, where the detailed test plans and procedures are run.	Test debt impacts the verification and validation stage. During this stage the system is tested to ensure that it meets the requirements and concessions made during the test development may result in additional tests if the test results are inadequate.	Test artifacts

Figure 2-19 relates the types of technical debt to the systems engineering Vee. Dark green areas of the Vee indicate areas where that type of technical debt is likely to be created (the left side of Figure 2-19) or observed (the right side of Figure 2-19). Technical debt is more likely to be created on the left side of the Vee and observed on the right side of the Vee, in accordance with the results presented in [18].

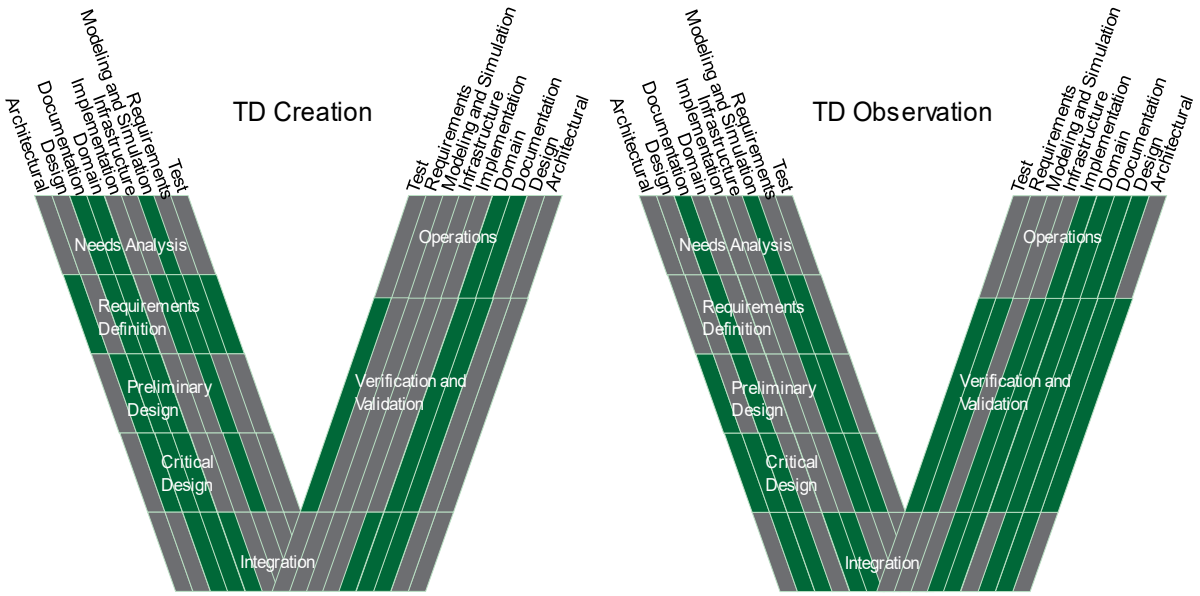

Figure 2-19. Technical debt creation (left) and observation (right) by type in the systems engineering lifecycle

While the general trend shown in Figure 2-19 matches that in [18], one key difference is the number of types of technical debt observed in the Operations stage. According to [18], over 70% of the survey respondents stated that technical debt is likely to be observed in the Operations stage. Figure 2-19 shows only four types of technical debt (design, documentation, domain, and implementation) being observed in the Operations stage. There are two reasons for this difference. First, the definitions provided in the technical debt ontology make a clear difference between those types of technical debt that affect the development stage (those that cause interest) and the deployment stage (those that cause fees). These concepts were not provided to the survey participants. Second, the survey participants were not provided with the detailed breakdown of

types of technical debt. Any technical compromise in the system can result in difficulties in operations, including under-performance and more complicated maintenance. However, the definition of domain debt, the failure of the system to meet the user's needs, covers most of these circumstances. Domain debt can be created at multiple steps throughout the systems engineering lifecycle leading to the observation of the technical debt in the Operations stage. Domain debt combined with design and implementation debt, which cover concessions made during the design and build of the system, produces the majority occurrences of technical debt in the operations phase.

2.6 Conclusion

This chapter addresses RQ1: *How prevalent is the technical debt metaphor within systems engineering?* Through the literature review and empirical survey, it was determined that the metaphor of technical debt is not widely used either in published research or in practice. However, the impacts of technical debt on a system development are commonly observed in the field of systems engineering. Therefore, there is a need to establish a common lexicon to enable discussions about technical debt, its impact, and its management. The ontology for technical debt [21] created as part of this dissertation provides the first known comprehensive lexicon for technical debt within systems engineering.

Application of the ontology allows types of technical debt to be defined while minimizing the risk of overspecification. By associated a type of technical debt with the affected artifact, similar management and mitigation techniques can be developed. Finally, the likelihood of creating and observing technical debt throughout the system lifecycle is defined, both in terms of empirical data from survey participants and through analysis of the presented types of technical debt. This data

supports the conclusion from the survey that technical debt is likely to be created early in the system lifecycle and observed late, but that it can occur anywhere.

This chapter provided an overview of the current state of technical debt research within systems engineering. The provided ontology, tailored specifically to systems engineering, serves as a starting point for practitioners to develop a shared language that can be used to determine methods and processes to mitigate and manage technical debt throughout the system lifecycle. The next chapters of this dissertation provide a process by which technical debt can be proactively assessed and then mitigated within the context of system development.

CHAPTER 3 – IDENTIFICATION OF TECHNICAL DEBT IN THE SYSTEM LIFECYCLE

3.1 Introduction

The system lifecycle defines how a system progresses from its initial conception through design and development and into operations, sustainment, and disposal. Each system ends up following a unique lifecycle, however, the lifecycle stages and process are defined by the technical strategy used to execute the lifecycle. The technical strategy consists of the development method, development strategy, and delivery strategy used to complete the system [23]. Iterative delivery strategies execute similar lifecycle stages repeatedly, increasing the overall value delivery of the system with each iteration. In these delivery strategies, technical debt can be a major contributor to delays, as the concessions made in early iterations may significantly impact the ability to complete work in later iterations [25]. Therefore, identifying technical debt within the system lifecycle is a critical capability that will enable improved satisfaction of stakeholder needs.

This chapter addresses Research Question 2: *How can potential sources of technical debt be identified during the system lifecycle?* As discussed in Chapter 2, technical debt occurs throughout the system lifecycle, often created in the early stages of the lifecycle and identified late in the lifecycle. This dichotomy is what makes technical debt a threat to the success of system development. Late detection of technical debt during system development increases the expected value of the technical debt interest and therefore the overall cost (in technical debt UNIT) required to repay the debt.

The late detection of technical debt can be mitigated by developing proactive mechanisms to identify potential sources of technical debt early in the system lifecycle and through detected

management of technical debt once it is discovered. This chapter focuses on the identification of technical debt in the lifecycle through two subordinate research questions:

- RQ2.1: *How is technical debt identified within software engineering?*
- RQ2.2: *What process can be used to identify potential technical debt sources within systems engineering?*

Addressing RQ2.1 provides insight into the tools currently used within the software engineering field to identify technical debt. The majority of technical debt research occurs within software engineering [19], and therefore understanding the methods used in that field may provide insights into appropriate methods for systems engineering. Addressing RQ2.2 will identify an existing process that can be used to identify technical debt within systems engineering, if possible. If such a process does not exist, then one will be created to enable proactive identification of potential technical debt in systems engineering.

3.2 RQ2.1: How is technical debt identified within software engineering?

This section provides a review of existing methods of technical debt identification within the published literature. There is limited research on technical debt within systems engineering [19]. Associated issues, such as rework, see limited published research on mitigation techniques within program development and specific to systems engineering [136]. Therefore, in accordance with Task 2.1.1, technical debt identification methods within software engineering are surveyed and the identified methods are assessed for their applicability to systems engineering.

3.2.1 Existing Methods of Technical Debt Identification in Software Engineering

It has already been established that the current state of technical debt research is primarily focused on the field of software engineering [19]. This section performs a review of technical debt

identification methods within software engineering from which knowledge may be derived for adaptation and application to the field of systems engineering.

Technical debt in software can be identified through automated approaches or through manual approaches. Automated approaches use tools to analyze artifacts to find indicators of potential technical debt, primarily focusing on source code analysis [156]. Manual approaches ask the system developers and stakeholders to identify instances of technical debt. While more time consuming, manual approaches can find sources of technical debt outside of the source code and provide additional context for each instance of technical debt [156]. Either automated or manual approaches can be used with different technical debt identification methods. Table 3-1 summarizes the methods for identifying technical debt in software engineering found within the literature.

Table 3-1. Methods for identifying technical debt within software engineering

Method	Description
Code Analysis	Using automated tools to identify existing problems in source code [157]
Self-admitted Technical Debt	Identification of problems in source code based upon the comments submitted by the system developers [158]
Requirements Validation	Ensuring that each requirement conforms to the organization's guidelines. Requirements validation confirms that the requirements are written clearly and unambiguously in the documentation style required by the project rules and that the interpretation will meet the stakeholders' needs [159]
Architectural Analysis	Evaluation of the system architecture to assess interconnectedness and potential disconnects among the components through modularity and dependency analyses [97]

3.2.2 Applicability of Software Engineering Methods to Systems Engineering

Systems engineering shares a lot of characteristics with software engineering, and therefore it a reasonable assumption that the methods for identifying technical debt within software engineering could be applied to systems engineering. Although the methods identified in Table 3-1 are not directly analogous to systems engineering, components of each method can be applied.

Code analysis investigates software for the following problems [157]:

- Code smells: software implementations that do not follow good object-oriented design practices;
- Modularity violations: software modules that are supposed to be independent develop co-dependencies;
- Design patterns and grime buildup: identification of software classes that fail to follow established design patterns and accumulate grime (code not related to the design pattern) and rot (breakage of the integrity of the design pattern); and,
- Automatic static analysis issues: tools automatically review source code for violations of best practices that may lead to future issues in software quality.

These techniques could be migrated to systems engineering with minor modifications, although their automatic implementation may be more complicated. For example, modularity violations can occur between hardware components in addition to software components. Hardware designs can accumulate grime and rot when the components fail to follow the established design patterns and interfaces. Smells can occur if system components do not follow good design practices, such as not including proper mass growth allowances on a satellite system build. These techniques may become more relevant with the rise of digital engineering and model-based systems engineering, where structured designs are more amenable to automatic scanning tools.

Self-admitted technical debt analysis is a process by which system developers admit that they have contributed technical debt to the system, through comments such as “FIX ME” or “TO DO” in the source code [158]. Within systems engineering, self-admitted technical debt is the equivalent to self-identification of shortcuts taken in a design process, such as redlines in a test procedure or the results of engineering review boards where a technical compromise is accepted and logged into the system.

Requirements validation is currently a part of systems engineering best practices [159]. If applied prior to the baselining of requirements, then the process can help identify and prevent technical debt from arising – the validation will clarify ambiguous requirements statements or intents and can therefore prevent domain debt. If the requirements validation is not performed until after the requirements have been baselined, then the validation process can reveal requirements debt and domain debt that is already in the system.

Architectural analysis methods include [97]:

- Modularity analysis: an assessment of the independence of the system functionalities;
- Dependency analysis: an assessment of the dependencies between components, with an emphasis on identifying 'irregularities' such as circular dependencies, typically conducted through the use of design structure matrices (DSM);
- Human analysis: identification by the system developers;
- Compliance checking: analysis of the difference between the designed and the implemented architecture; and,
- Change impact analysis: analysis of alternatives designs with the goal of limiting the development of technical debt.

These techniques can also be adapted for use within systems engineering. System components are often modular and have dependencies upon each other. Identifying these dependencies and their potential impacts in the future is a critical part of technical debt assessments.

3.2.3 Addressing RQ2.1

Research Question 2.1 sought to understand how technical debt is identified within software engineering. It was anticipated that a technique, or techniques, from software engineering could

be extended to systems engineering to easily create a method for the proactive identification of technical debt. However, the evaluation of the current technical debt identification and management methods within software engineering did not reveal a proactive method that could be extended to systems engineering. Certain aspects of these methods, such as the modularity and dependency analyses of architectural analyses, can be migrated to systems engineering, but a usable end-to-end process was not identified.

Therefore, to reduce the risks of technical bankruptcy associated with technical debt, a proactive process needs to be established that can convert negligent debt into strategic debt. A rapid, easy-to-use, proactive process for identifying technical debt could enable more widespread use, promoting the assessment of technical debt when decisions are made. Such assessments turn negligent debt into strategic debt and reduce the risk of reaching technical bankruptcy. This process has the following objectives:

1. Identification of the system features that support the realization of the stakeholder needs;
2. Identification of the dependencies between the features and needs and between features and features, in the temporal and functional dimensions;
3. Assessment of the potential impact of a technical concession on the satisfaction of the stakeholder needs; and,
4. Easy to use, update, and understand by both system developers and stakeholders.

Such a process can be used throughout the lifecycle of the system, including in the design and implementation stages. Iterative use of the process identifies the appropriate needs and features at each level of development and supports decision making in the performance, cost, and schedule dimensions.

3.3 RQ2.2: What process can be used to identify potential technical debt sources within systems engineering?

The previous section reviewed the technical debt identification methods used within software engineering and concluded that a new systems engineering-centric process is required for technical debt identification within systems engineering. The creation of this process starts with developing an understanding of the technical debt identification timeline.

3.3.1 Technical Debt Identification Timeline

The technical debt context map in Figure 2-13 identifies a pathway by which technical debt occurs within a system. Figure 3-1 shows how this pathway is realized within a system development and associates the expected repayment cost with each step in the pathway. The stakeholder demands a short-term benefit from the system developers. The developers evaluate alternate solutions to provide this benefit, which may require technical concessions. The developers then select a solution and implement it, delivering the short-term benefit to the stakeholder and possibly introducing technical debt into the system. If not managed, then at some point in the future, the long-term consequences of those technical concessions are observed in the system, by both the developers and the stakeholders.

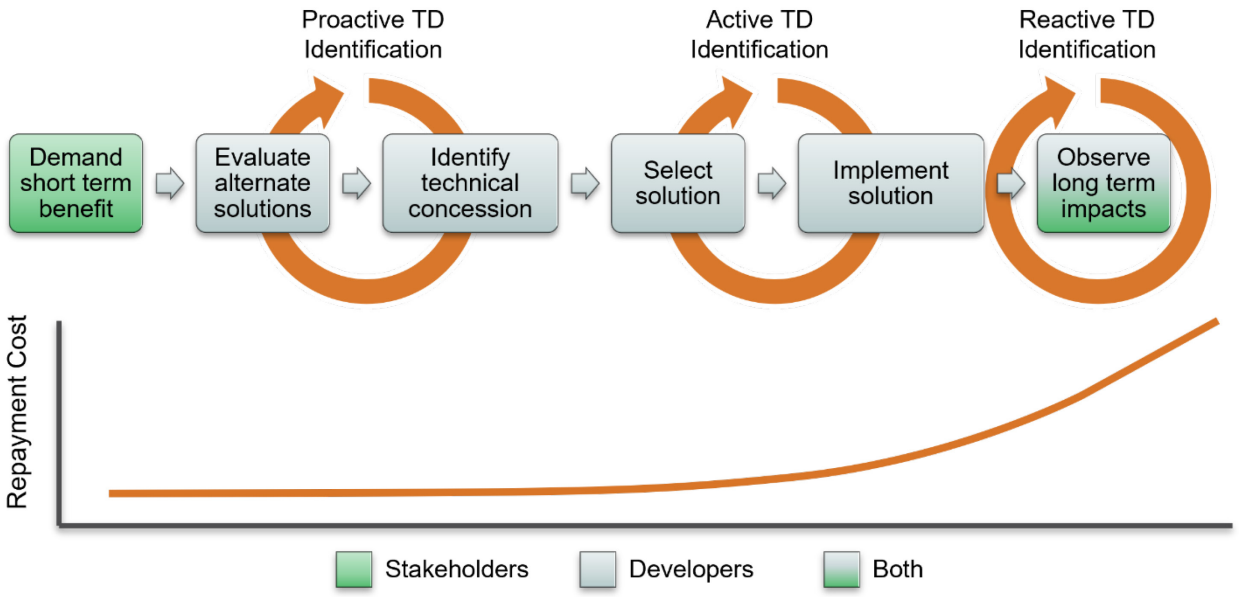

Figure 3-1. Technical debt identification method application in the occurrence of technical debt. Repayment cost based on [135]

The sequence shown in Figure 3-1 indicates three timeframes where technical debt can be identified. During the evaluation of alternative solutions and the identification of technical concessions, the potential consequences of the concessions can be identified. Techniques and methods such as requirements validation are used during this phase. Finding the technical debt at this phase of the implementation is proactive – it identifies the potential technical debt prior to a decision to make a technical concession. Identifying technical debt at this phase enables the ramifications of the concessions to be considered as part of the selection process.

The second timeframe where technical debt can be identified is during the selection and implementation of the solution. Identification of technical debt at this timeframe is active – the technical debt is identified as it is put into the system. For example, the developers may note the impacts of the concessions during implementation, such as identifying that a standard is incompletely implemented. Methods such as tracking self-admitted technical debt can be applied in this stage, if the developers track the concessions that they make. Active technical debt

identification, if properly managed, enables the technical debt item to be tracked as soon as it is created and therefore enables mitigation of its long-term consequences.

The third timeframe for technical debt identification occurs once the long-term consequences of the technical concession are observed within the system. This type of technical debt identification is reactive – it finds technical debt that is already within the system. Code analysis techniques are used in this stage to identify existing technical debt. Reactive identification of technical debt can assist in explaining system-level behavior but results in more expensive repayment of the debt.

Architectural analysis techniques can be applied across all stages identified above and provide methods to assess the interdependencies of the system. By applying these techniques early in the system lifecycle, the developer can prevent locking in architectural decisions that will require significant effort to correct in later stages.

3.3.2 The Need for Proactive Technical Debt Identification

As shown by the cost graph at the bottom of Figure 3-1, the cost of repaying technical debt increases the later in the system development cycle that it is identified [135]. Therefore, there is a need to enable early and proactive identification of technical debt to limit its impact on the system development, across all three system dimensions: cost, schedule, and performance.

Mapping the groups of technical debt identified in Section 1.1.2.1 onto the technical debt timeline yields Figure 3-2. Figure 3-2 shows a notional alignment of the groups of technical debt on the timeline and Table 3-2 provides a definition of each of the stages and entry gates identified in the figure. Prudent technical debt, both strategic and tactical, skips the blissful ignorance phase, since the intentional decision to take on the technical debt implies that the technical debt is identified when it enters the system. Negligent technical debt, while a deliberate decision, does not

include a repayment plan, and therefore the technical debt may be forgotten about or may not be shared with other system developers, resulting in a period of time where the debt is hidden and the system is ignorant of its presence. Prudent technical debt is less likely to induce technical bankruptcy, due to established repayment plans. Therefore, transition point T5 (Potential for Technical bankruptcy) is removed from the strategic and tactical timelines.

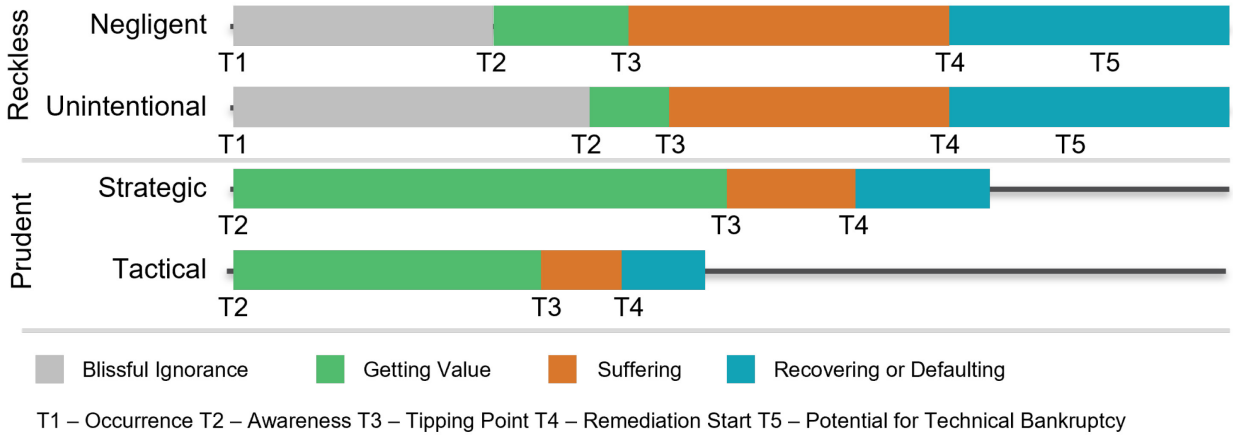

Figure 3-2. Notional timeline of technical debt occurrence by group

Table 3-2. Stages and gates of the technical debt timeline

Stage	Definition	Entry Gate
Blissful Ignorance	The developer is unaware of the technical debt in the system	T1 - Occurrence: the entry of technical debt into the system
Getting value out of debt	The developer is aware of the technical debt, but the technical compromise is still yielding a short-term benefit and the long-term impacts on the system health have not manifested	T2 - Awareness: the technical debt has been identified
Suffering from debt	The long-term impacts on system health outweigh the short-term benefits	T3 - Tipping Point: the technical debt has begun to cause issues in the system development lifecycle
Recovery from debt	The technical debt is paid back and the long-term impacts are mitigated	T4 - Remediation Start: the technical debt is managed and on a known repayment plan
Defaulting on debt	The technical debt is not sufficiently paid back, and the system enters technical bankruptcy	T5 - Potential for Technical Bankruptcy: the technical debt impacts have increased to the point where the system cannot continue with its lifecycle until the debt is repaid or reduced

The timing of the transition points (labelled with "T") in Figure 3-2 provide insight into the notional phased benefits of each of the technical debt groups. Negligent technical debt has a shorter period of value return, a longer period of suffering from the technical debt, and a quicker potential for technical bankruptcy compared to strategic technical debt. The timeframes for tactical technical debt, both in terms of getting value and suffering from the debt are shorter than those of strategic technical debt.

To limit the impact of technical debt, the durations of the blissful ignorance, suffering, and recovering phases need to be minimized. The blissful ignorance phase allows for unknown and unmanaged accumulation of technical debt interest, thereby increasing the potential future cost of repayment. The suffering phase is when the system is impacted by the technical debt and the performance is decreased. Reducing recovery timelines enables a quicker return to delivering value. Therefore, the earlier that technical debt is identified within a system, the less harmful it is likely to be. Proactive technical debt identification allows for identification of technical debt prior to its introduction into the system. These methods can move technical debt from negligent and unintentional to strategic – early identification enables plans to be put into place to mitigate the impact of technical debt.

Proactive methods are necessary due to the volatile, uncertain, complex, and ambiguous (VUCA) environment into which most systems are deployed [3]. Environmental changes can force the system to be used in unintended ways or to react to unintended inputs. Careful design is required to implement, operate, maintain, and monitor systems in these environments [150]. Decisions that are made regarding the system design and architecture incur cost and accrue debt since they lock in a system configuration and need to be actively managed [65]. However, these decisions are often made in short-time frames without consideration of the future consequences.

The later that technical debt associated with these decisions is discovered, the more expensive it is likely to be to correct, as the cost to implement a change becomes significantly more costly with each program phase [135]. During systems development, technical debt is likely created during the critical design phase where it is also unacceptable to create the technical debt [18]. This seeming paradox is an indicator of the creation of negligent technical debt – system developers create technical debt to complete the design even though they know that it is not the best solution.

Technical debt can also be "contagious" within a system. As the system grows, technical debt can spread, "infecting" other parts of the system [92]. For example, an electrical system can be designed to connect components in series or in parallel to achieve a required reliability of 0.95. For a system where a "short cut" decision is made to connect an uncertain number of components in series, the interest on this technical debt item (the decision to implement in series instead of in parallel) grows with every new component added. Each new component connected in series is affected by an electrical failure in any of the connected components, reducing the reliability of the system, as shown in Figure 3-3. In this example, each component has an individual reliability of 0.99. When connected in series, the reliability of component F is reduced to 0.94, due to the dependencies on the previous components. When connected in parallel, the reliability of component F stays at 0.99. With the series connection, a redesign would need to occur to meet the requirement, either by improving the reliability of several components in the chain or by redesigning the entire system. The cost of this redesign goes up with the number of components connected.

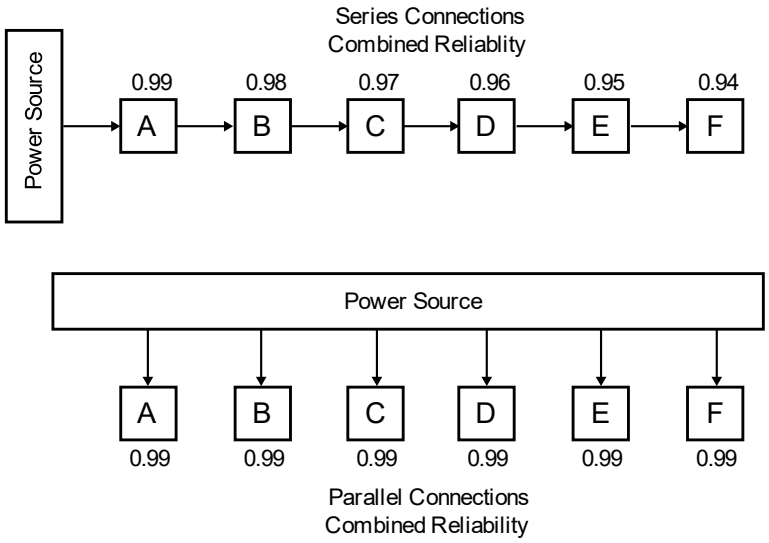

Figure 3-3. Series and parallel implementation of components showing the change in reliability of each component

Based on this analysis of the state of the art of technical debt identification methods, it is clear that the majority of the methods used within software engineering are reactive identification methods. Most of these approaches "focus on defect detection and avoidance, rather than a strategic management of key infrastructure decisions, especially in the context of architecture" [16]. Techniques such as source code analysis can only be applied to detect technical debt that is already within the system and these techniques do not inform the decision paths as the system is developed, potentially resulting in large accumulations of technical debt that are expensive to correct [16].

Proactive methods of technical debt identification, where potential technical debt is identified during the analysis of design choices and ahead of design implementation, are necessary to minimize the impact of technical debt within the system. Active methods detect technical debt when it occurs and make it easier to manage, but the debt has already found its way into the system. Reactive methods find existing technical debt. Only proactive methods can prevent the introduction of technical debt within the system.

3.3.3 The LEAP Process

A proactive process for technical debt identification within systems engineering could not be found within the published research. Section 3.2.3 established criteria for a proactive technical debt identification process. Techniques found within software engineering are mostly reactive and do not meet these criteria. Therefore, the List, Evaluation, Achieve, and Procure (LEAP) process was developed, in accordance with Task 2.2.1. This process provides mathematical methods to associate stakeholder needs with technology development timelines. By directly associating the need dates and the development timelines, the process clearly indicates which technologies affect the ability to deliver capabilities needed by the stakeholder on schedule. This association enables the identification of technologies that are likely to produce technical debt by determining the dependencies of each capability on the technology. The technologies with more capability dependencies are likelier to be larger sources of technical debt if compromises are made in their development. The LEAP process was defined in a paper presented at the *2023 INCOSE International Symposium* [160] and is reprinted here.

3.3.3.1 LEAP – A Process for Identifying Potential Technical Debt in Iterative System Development [160]

3.3.3.1.1 ***Abstract***

Systems engineering has seen a rise in the use of iterative methods to design and develop both hardware and software systems which allow for system refinement to be responsive to user needs. However, focusing on items with high value to the user can result in technical debt, where technical compromises made for short-term gain impact the long-term health of the system. Current methods for identifying technical debt focus on finding existing technical debt items within a system and not on proactive identification of technical debt during the iterative system planning process. This

paper presents a novel technique to identify technologies that impact the ability of the system to satisfy the needs of its stakeholders. The method is used to evaluate different choices of technological implementations in both the temporal and the functional dimensions to reduce the risk of incurring technical debt which prevents the successful delivery of the system.

3.3.3.1.2 Introduction

System development and operating environments have become increasingly volatile, uncertain, complex, and ambiguous (VUCA) [3]. VUCA environments require both the development of agile systems and the use of agile systems engineering methods. Iterative development methods, which include agile systems engineering methods, add flexibility and agility to the system development processes [8]. Although not novel techniques, iterative development methods enable the system developer to respond to the VUCA environment by releasing new iterations of the system [11].

The use of iterative development methods is driven, in part, by the desire to shorten development cycles. This desire can result in stakeholders "encourag[ing] developers to take shortcuts early in the development process in order to get system capabilities deployed quickly" [7]. This pressure can result in a system developer making technical compromises to meet schedule [2] and prioritizing perceived high-value functional requirements over quality requirements [15]. These compromises, while appearing to be successful early, may slow the system development down over time, as their impacts need to be overcome [16]. This phenomenon is known as technical debt.

Technical debt is a metaphor that identifies how technical compromises made for short-term benefit can have long-term impacts on the health of the system [19]. Systems that are iteratively developed are at additional risk of technical debt, since decisions made early in the development cycle impose additional constraints on the later iterations [26]. There is a risk that the system

developers will select the “easiest” set of components to deliver value to the stakeholder, resulting in a system that breaks when changes are required in the future [25].

Existing methods for managing technical debt focus on the identification of technical debt within the system and the repayment of that debt in future iterations. While important to maintaining a healthy system, these methods do not provide techniques that enable the stakeholders and system developers to assess the impact of their decisions on the ability of the system to deliver the required capabilities on the required timelines. Therefore, this paper introduces a new process called LEAP – List, Evaluate, Achieve, and Procure. The LEAP process provides a structured method to identify critical technologies that enable the stakeholders’ desired capabilities. The identification of the technologies is followed by a mathematical analysis to determine the impact of incomplete or late technology development on the ability to deliver the capabilities on time. The analysis can be used to examine different development paradigms, identifying choices that may cause long-term impacts through a delay in the development of a key technology. The LEAP process proactively identifies sources of technical debt prior to incurring the debt.

The rest of this paper is structured in three sections. First, an overview of related work on technical debt management and prediction is presented. Next, the LEAP process is described in detail. Finally, the process is discussed and the paper is concluded with a presentation of recommendations for future work.

3.3.3.1.3 Related Work

Iterative system development consists of either fixed-scope or fixed-time iterations. Fixed-scope iterations deliver a known scope in each iteration but the time it takes to deliver that scope may vary. Fixed-time iterations deliver a variable scope in each iteration as the work is constrained

by the length of the iteration [28]. Pressures associated with releasing the iteration on time make fixed-time iterations especially susceptible to technical debt.

Agile methodologies, such as Scrum, use fixed-time iterations known as sprints to plan and execute the work of developing the system. Agile frameworks, such as the Scaled Agile Framework (SAFe), add program increments to the sprints and plan the high-level features delivered in the next set of sprints [161]. Program increments are an example of release planning, a steady cadence of planning for successive releases. Release planning methodologies often stress the need to deliver value to the stakeholders as early as possible. SAFe uses a methodology called 'weighted shortest job first', where tasks are selected based on the ratio of delivered value to the time to complete the task [30]. These prioritization methods produce an increased stress on delivering short-term benefit at the risk of damaging the long-term health of the system.

Release planning models and methods exist to try to optimize the features that are included in each release [28]. In iterative development, there is often a tradeoff between early value creation and future rework [162], requiring the use of optimization methods to determine the best course of action. Nord et al. [16] modeled the interdependencies of architectural features to estimate the rework costs. Sangwan et al. [91] extended this work to optimize release plans by minimizing total cost, by maximizing early value, or by finding an optimal combination of the features. This model does not estimate the uncertainty associated with the development of each feature or any technical debt incurred during feature development. It does provide a method to assess cost and value trade-offs, but does not track these trade-offs against the temporal need dates of the stakeholders. Oni and Letier [28] created a model that examines uncertainty associated with fixed-time release cycles. They use expert opinion to determine the uncertainty associated with completing a feature in a specific release cycle and then probabilistically estimate the ability to complete the release as

planned. However, they do not model the propagation of delays of each feature on the delivery of future releases. Schmid [94] provides an analytical method to determine the optimal release to repay a technical debt item, based on the interest amount and interest probability of the item. However, this method does not assist in identifying the technical debt items.

Based on this examination of the state of the field, it is clear that a technique is required to enable proactive identification of technical debt sources. Early identification of potential technical debt allows a system developer to take measures to address the source of technical debt prior to it impacting the health of the system. Proactive management of potential technical debt sources reduces the risk of incurring technical debt through incomplete technology development.

3.3.3.1.4 The LEAP Process

Rapid and successful iterative development requires the inclusion of mature technologies in each iteration [162]. Identifying mature technologies requires understanding the functional and temporal dependencies of capabilities on those technologies. This process, known as List, Evaluate, Achieve, and Procure (LEAP), allows the identification of technology areas that require investment to ensure that they will be available for use on the desired timeline.

The LEAP process consists of four major steps:

1. List: list the strategic and tactical capabilities and technologies required to achieve the system objectives, based on the stakeholders' functional and temporal requirements
2. Evaluate: evaluate the ability of the state of the art to meet both the functional and temporal stakeholder requirements
3. Achieve: identify the gaps between the current development timelines and the capability need dates and provide or leverage resources to close those gaps, on both tactical (short-term) and strategic (long-term) timelines

4. Procure: produce a solicitation to acquire a system to provide capabilities, ideally limiting the amount of non-recurring engineering (NRE) included within the system development

Figure 3-4 provides an overview of the LEAP process. In the figure, solid lines indicate forward progress and dashed lines indicate feedback paths. The following sections discuss the details of each step.

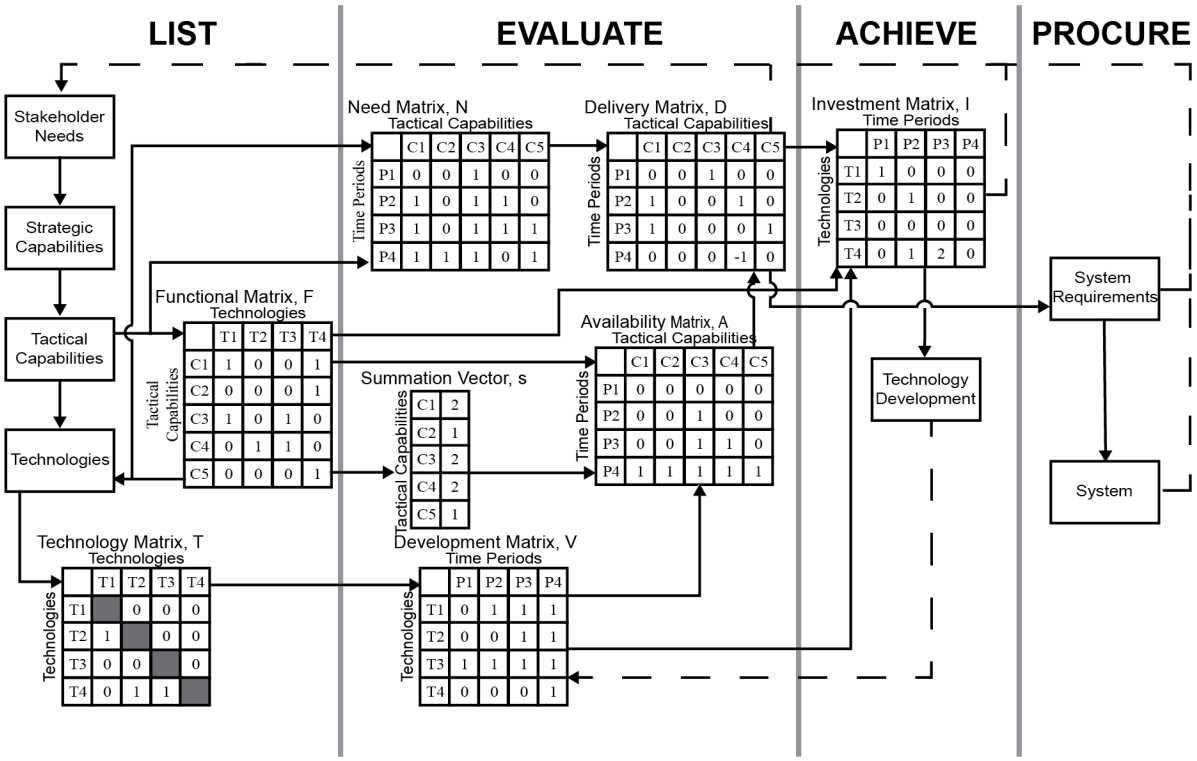

Figure 3-4. The LEAP process

Table 3-3 provides the ontology of terms used in the LEAP process and defines the symbology used throughout the rest of this paper.

Table 3-3. Definition of LEAP terms and symbols

Term	Definition
Strategic Capability	Derived from the end users, strategic capabilities identify the intended behavior of a system in its operational environment. Strategic capabilities identify the full desired capability and the minimum viable product.
Tactical Capability	Tactical capabilities represent the ability of a system to provide part of a strategic capability that delivers value to the end user. A tactical capability is required at a specific time.
Technology	Methods and devices resulting from the practical application of knowledge. A technology is a tangible product (hardware or software) that is delivered at a specific time.

Term	Definition			
System	A set of hardware, software, and/or physical and logical interfaces developed based on technologies with the intent of delivering capabilities.			
Symbols	T	Technology Matrix	I	Investment Matrix
	N	Need Matrix	J	Hadamard identity matrix
	F	Functional Matrix	t	Number of technologies considered
	s	Summation vector	c	Number of tactical capabilities considered
	S	Summation Matrix	p	Number of time periods considered
	V	Development Matrix	∘	Hadamard product operator
	A	Availability Matrix	H	Heaviside function
	D	Delivery Matrix	⊗	Outer product operator

3.3.3.1.4.1 LEAP Phase 1: List

The List phase decomposes the stakeholder needs into the strategic and tactical capabilities and identifies the technologies required to support these capabilities. The stakeholder needs are first broken down into strategic capabilities, which define the long-term objectives and the end capabilities provided to the user. However, developing only to the long-term strategic capabilities could result in delays of system development. Therefore, the strategic capabilities are broken down further into tactical capabilities, in a process similar to the Agile software development process of breaking down an epic into features. Tactical capabilities are designed to be achievable within a single procurement. The tactical capabilities serve as the primary point of analysis for the rest of the LEAP process.

With the tactical capabilities defined, the technologies required to support these capabilities are identified. Each tactical capability is supported by one or more enabling technologies. Technologies can vary in scope, but are generally broken down to the level of items that can be separately developed. Each technology may support the development of one or more tactical capabilities.

The Functional Matrix captures the dependencies of the tactical capability on the enabling technologies, an example of which is shown in Figure 3-5. The Functional Matrix is a (c x t) matrix,

with the tactical capabilities listed in the rows and the technologies listed in the columns. If the tactical capability depends on the technology, then a one (1) is entered in the cell. For example, in the Functional Matrix shown on the left side of Figure 3-5, tactical capabilities C1 and C3 both depend on technology T1. The Functional Matrix is the connecting fabric between the technologies and the capabilities and will be used as a baseline in the following analysis.

Functional Matrix, F

Tactical Capabilities \ Technologies	T1	T2	T3	T4
C1	1	0	0	1
C2	0	0	0	1
C3	1	0	1	0
C4	0	1	1	0
C5	0	0	0	1

Technology Matrix, T

Technologies \ Technologies	T1	T2	T3	T4
T1		0	0	0
T2	1		0	0
T3	0	0		0
T4	0	1	1	

Figure 3-5. Input matrices of the list phase of the LEAP process

The critical technologies identified in the Functional Matrix may be dependent upon other technologies. Identifying these dependencies, including their required order of development, is necessary to properly estimate the timelines on which technologies will be available. Therefore, a design structure matrix (DSM) is created for the technologies. The DSM is a square matrix, where the technologies are listed in both the rows and the columns. If information flows from the item in column *j* to the item in row *i*, then a one (1) is placed in cell (*i, j*). Therefore, any entries in the DSM with a one (1) indicate a dependency of the row on the column. If the rows are arranged in chronological order of development, then the dependencies located below the diagonal indicate the forward flow of information (from column to row), and the dependencies above the diagonal indicate feedback mechanisms [163].

In the LEAP process, the DSM is referred to as the Technology Matrix, which is a (*t* x *t*) matrix. In the Technology Matrix, the technologies are listed in both the rows and the columns. An

example of the Technology Matrix is shown on the right side of Figure 3-5. The one (1) in the first column of the second row indicates that information flows from T1 to T2. Therefore, T2 depends on T1. Additionally, technology T4 depends on T2 and T3. T3 and T1 have no dependencies. In this example, the matrix only contains entries below the diagonal, indicating that there are no feedback dependencies.

After creation, the Technology Matrix is partitioned to reduce the dependencies and set the order of technology development. Partitioning attempts to reduce the matrix to a lower triangular matrix to remove any of the feedback dependencies. This effort results in a reordering of the matrix and a possible adjustment to development order [164]. After partitioning, the order of the matrix represents the notional chronological development order. If the partitioning does not result in a fully lower triangular matrix, then there are cyclic dependencies between technologies. The Technology Matrix serves as a guideline for creating the Development Matrix, which is described in the following section.

3.3.3.1.4.2 LEAP Phase 2: Evaluate

The Evaluate phase uses the output from the List phase and determines if the tactical capabilities can be satisfied based on the current state of technology development. The first step is to establish the Need Matrix, which defines the required delivery timelines for each tactical capability to meet the needs of the end user. The Need Matrix (N) is a (p x c) matrix and is shown on the left side of Figure 3-6. The time periods in which needs or development occur are labelled as P1, P2, P3, and P4. A one (1) is entered in any cell where the capability is required in the time period and a zero (0) is entered in a cell where the capability is not required in the time period. Capabilities may not be needed in every time period and may also no longer be needed after a specific time. In the Need

Matrix in Figure 3-6, capability C1 is first needed in P2 and is required through P4, capability C2 is not needed until P4, and capability C4 is only needed in P2 and P3.

Need Matrix, N

Tactical Capabilities

Time Periods	C1	C2	C3	C4	C5
P1	0	0	1	0	0
P2	1	0	1	1	0
P3	1	0	1	1	1
P4	1	1	1	0	1

Development Matrix, V

Time Periods

Technologies	P1	P2	P3	P4
T1	0	1	1	1
T2	0	0	1	1
T3	1	1	1	1
T4	0	0	0	1

Figure 3-6. Input matrices of evaluate phase of the LEAP process

The partitioned Technology Matrix, T, is used as an input to the Development Matrix (V), which establishes the timelines for technology development. The Development Matrix, shown on the right side of Figure 3-6, is a (t x p) matrix. A one (1) is entered in any cell where the technology is expected to have a technology readiness level (TRL) of at least six (6), in accordance with the best practices for rapid development identified by Tate [162]. In Figure 3-6, technology T1 reaches TRL 6 in P2. Therefore, it will be available in P2, P3, and P4. Technology T4 is not expected to reach TRL 6 until P4.

The completion of the Development Matrix represents the end of the set of inputs that must be provided by subject matter experts. The Evaluation phase takes these inputs and computes the next set of matrices to identify expected delivery dates of the tactical capabilities. These dates are used to identify the technologies that may impact the on-time delivery of tactical capabilities.

The first computation is the Availability Matrix (A), which is a (p x c) matrix. This matrix combines the Development and Functional Matrices together to determine when each tactical capability will be available. The Functional Matrix (c x t) uses the technologies as the columns and the Development Matrix (t x p) uses the technologies as the rows. The dot product of a row of the Functional Matrix and a column of the Development Matrix results in the count of the number

of technologies that support the capability (a one (1) in the Functional Matrix) and are available in a time period (a one (1) in the Development Matrix). If the technology either does not support the capability or is not available in the time period, then it will not contribute to the dot product. For example, to determine how many technologies that support capability C3 in P1, the dot product is taken between the third row of the Functional Matrix (shown in Figure 3-5) and the first column of the Development Matrix (shown in Figure 3-6). As shown in Equation 3-1, the result is 1, indicating that one technology that supports the capability is ready in P1.

$$F[2,:] \cdot V[:,0] = [1 \quad 0 \quad 1 \quad 0] \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = 1 \qquad (3\text{-}1)$$

Multiplying the Functional and Development Matrices together produces a *(c* x *p)* matrix. This matrix is transposed to produce a (*p* x *c*) matrix, where each cell represents the number of developed technologies that support each capability (columns) in the time period (rows). However, this product is insufficient to determine if the capability will be available, as it does not indicate how many technologies are required to support the capability. This number is found by summing the values in each row in the Functional Matrix to produce the Summation vector s, as shown in Equation 3-2. The Summation vector is length *c*.

$$s = \begin{bmatrix} \sum_{i=0}^{t} F[0,\mathrm{i}] \\ \vdots \\ \sum_{i=0}^{t} F[\mathrm{c},\mathrm{i}] \end{bmatrix} \qquad (3\text{-}2)$$

The Summation vector is turned into a matrix of the same dimensions as the transpose product of the Functional and Development matrices (*p* x *c*) by taking the outer product of the Summation vector and a row vector of all ones with length *p*, indicated by 1_p in Equation 3-3. The outer product produces the Summation Matrix (S).

$$S = 1_p \otimes s \quad (3\text{-}3)$$

An example of the Summation Matrix is shown on the left side of Figure 3-7. The Summation Matrix is a (p x c) matrix, with the number of technologies required to support each capability listed in the cells. In this example, capability C1 requires two technologies, while capability C2 only requires one technology.

Summation Matrix, S

Tactical Capabilities

	C1	C2	C3	C4	C5
	2	1	2	2	1
	2	1	2	2	1
	2	1	2	2	1
	2	1	2	2	1

Availability Matrix, A

Tactical Capabilities

Time Periods	C1	C2	C3	C4	C5
P1	0	0	0	0	0
P2	0	0	1	0	0
P3	0	0	1	1	0
P4	1	1	1	1	1

Delivery Matrix, D

Tactical Capabilities

Time Periods	C1	C2	C3	C4	C5
P1	0	0	1	0	0
P2	1	0	0	1	0
P3	1	0	0	0	1
P4	0	0	0	-1	0

Figure 3-7. Calculated matrices of the evaluate phase of the LEAP process

The Summation Matrix provides the number of required technologies, and the product of the Functional and Development matrices provides the number of available technologies. Subtracting the two matrices determines if enough technologies are available to fully support the capability. Equation 3-4 defines the function to compute the temporary availability matrix (a), a (p x c) matrix.

$$a = (FV)^T - S \quad (3\text{-}4)$$

In the temporary availability matrix, a zero (0) indicates that the capability is available in the time period and a negative number indicates that it is not available. The use of negative numbers and zeros is confusing and can produce undesired mathematical results later in the process. Instead, it is desirable to have the Availability Matrix contain a one (1) when the capability is available and a zero (0) when it is not. Therefore, the temporary availability matrix is modified by using the Heaviside function [165], which converts the input values as shown in Equation 3-5.

$$H(x) = \begin{cases} 0, x < 0 \\ 0.5, x = 0 \\ 1, x > 0 \end{cases} \quad (3\text{-}5)$$

Prior to the application of the Heaviside function, 0.5 is added to each value in the temporary availability matrix. This addition serves to ensure that values of zero (0) will become one (1). In matrix notation, 0.5 is multiplied times a (p x c) Hadamard identity matrix, J, which consists of a one (1) in every entry in the matrix [166]. The resulting matrix is added to the temporary availability matrix such that negative values in a will remain negative and zero (0) values in a become positive. Applying the Heaviside function to each cell in the resulting matrix produces the Availability Matrix (A), which will have values of either zero (0) or one (1). Equation 3-6 shows the final equation for calculating the Availability Matrix.

$$A = H((FV)^T - S + 0.5J) \quad (3\text{-}6)$$

An example of the Availability Matrix is shown in the center of Figure 3-7. A value of one (1) indicates that the capability is available in the time period, while a value of zero (0) indicates that the capability is not available. In this figure, it can be seen that capabilities C1 and C2 are not available until P4. Both capabilities depend on technology T4, which, according to the Development Matrix in Figure 3-6, is not available until P4. Capability C3, on the other hand, depends on technologies T1 and T3 which are both available by P2, and therefore, C3 is available in P2.

The final step in the Evaluation phase is to calculate the Delivery Matrix, which is a (p x c) matrix. The Delivery Matrix, shown on the right side of Figure 3-7, indicates if the capability will be available on the timelines identified in the Need Matrix without any intervention. The Delivery Matrix has the same dimensions as the Availability Matrix and is calculated by subtracting the Availability Matrix from the Need Matrix, as shown in Equation 3-7.

$$D = N - A \quad (3\text{-}7)$$

This subtraction results in the following possible values in each cell in the Delivery Matrix:

- 1: indicates that the capability is late to need - it is not available in the time period and it was needed in time period
- 0: indicates that the capability is available when needed or the capability is not available and was not needed
- -1: indicates that the capability is available and is not needed, either because the capability is available early or is available but no longer required

The Delivery Matrix shown in Figure 3-7 shows that capability C1 is not available when needed in P2 or P3, but is available in P4, at which point there is still a need for the capability. Capability C2 is delivered on time. Capability C4 is not ready when needed in P2 and is still available in P4, even though it is no longer needed.

3.3.3.1.4.3 LEAP Phase 3: Achieve

The Delivery Matrix provides a traceable indication of the ability of the developer to deliver capabilities on the required timelines. This evaluation is made based on the timelines established in the Development Matrix. However, an organization has the ability to influence the timelines of key technologies by funding research, partnering with other organizations, and encouraging development by industry. The Achieve phase of LEAP focuses on identifying the key areas for investment by any of the means specified above. The first step in the Achieve phase is to identify the capabilities that will be late to need by finding the negative values in the Delivery Matrix. These values can be traced through the Functional Matrix to identify the technologies that support

the capability. The technologies can be traced through the Development Matrix to identify which specific technologies will be late to need.

This traceability is mathematically performed by calculating the Investment Matrix. The Investment Matrix determines which technologies are delivered late to need, and how many capabilities those technologies are impacting in each time period. The Investment Matrix is a (t x p) matrix and is initially calculated as shown in Equation 3-8:

$$i = (DF)^T \quad (3\text{-}8)$$

The Investment Matrix multiplies the rows of the Delivery Matrix, which contain the time period where each capability is available, by the columns of the Functional Matrix, which contain the technologies supporting each capability. This multiplication results in the identification of the technology available within a time period in support of the capabilities. However, without further adjustment, the produced term does not properly account for whether or not the technology is planned to be ready for that time period.

Accounting for the planned development of the technology requires an element-wise multiplication by the Development Matrix, known as the Hadamard product [166]. First, the Development Matrix is subtracted from a (t x p) Hadamard identity matrix (J), which only retains cells where the technology is not planned to be developed. The Hadamard product of this matrix and the matrix i calculated in Equation 3-8 removes any technology identifications where the technology is planned to be developed from the Investment Matrix. The final Investment Matrix equation is shown in Equation 3-9.

$$I = (DF)^T \circ (J - V) \quad (3\text{-}9)$$

An example of the Investment Matrix is shown in the lower right of Figure 3-8. A non-zero cell in the Investment Matrix indicates the number of capabilities that the technology impacts by not being developed in the time period. For example, in Figure 3-8, technology T4 will not complete development until P4 and therefore impacts one capability in P2 and two capabilities in P3. A negative number in the Investment Matrix indicates that the technology contributes to early delivery of capabilities, and therefore could be delayed with minimal impact to meeting stakeholder needs.

Investment Matrix, I

Technologies \ Time Periods	P1	P2	P3	P4
T1	1	0	0	0
T2	0	1	0	0
T3	0	0	0	0
T4	0	1	2	0

Figure 3-8. Investment Matrix, calculated in the achieve phase of the LEAP process

Once the technologies that are late to need are identified, the organization evaluates which type of investment is appropriate to accelerate the development. Achievement is considered in both tactical (short-term) and strategic (long-term) viewpoints. The Delivery Matrix identifies which capabilities will fall short, whether they will fall short in the next time period, or in a time period farther in the future. The Investment Matrix indicates how critical each technology is to meeting the overall needs and requirements of the organization in both the short and the long-term. This viewpoint leads to the early identification of key technologies required for the support of future iterations.

3.3.3.1.4.4 LEAP Phase 4: Procure

The final phase in LEAP is the Procure phase. In the Procure phase, the organization selects which capabilities, and therefore the associated technologies, to include in the next iteration of

system development. A reduction in the complexity of the system requires minimizing NRE occurring within the iteration. The first three steps of the LEAP process, when performed iteratively, identify the tactical and strategic investments that the organization can use to develop these technologies in response to the stakeholder needs. In a perfect world, all technologies would be available on schedule and no NRE would need to occur within a release. However, NRE often does need to occur within a given release development. Therefore, the technologies need to be carefully chosen to minimize risk.

The Procure phase includes an assessment of the current state of available technology against the stakeholder needs, starting with the Delivery Matrix. From there, the organization selects which capabilities need to be included in the release in association with the stakeholders. Other considerations, such as system constraints, are included to ensure that an achievable system is designed and deployed within the required schedule. During this iterative process, the risk associated with delaying technology development is considered. The final product is a set of system requirements for the next iteration of system development.

3.3.3.1.4.5 Iterations within the LEAP process

The LEAP process is an inherently iterative process. The Procure phase is when the organization commits to developing a system and locking in the associated schedule, cost, and capabilities. Therefore, it is necessary to iterate within the List, Evaluate, and Achieve phases repeatedly prior to entering the Procure phase. Figure 3-4 shows the feedback paths as dotted lines.

The iterative process begins after the determination of the Delivery Matrix in the Evaluate phase. The Delivery Matrix indicates which components will currently be late to need. Stakeholders review this information to reconsider and to alter the specified need dates as necessary in the Need Matrix. Additional iterations occur after the completion of the Achieve

phase. The Achieve phase accelerates technology development, which can adjust the Development Matrix. The results of the Achieve phase are returned to the stakeholders, who may adjust their needs and the associated capability requirements.

Finally, the Procure phase feeds back into the List phase, as each procurement results in a selection of implemented and developed capabilities. These capabilities will impact the priorities and needs for the next iteration.

3.3.3.1.4.6 Assessing the Potential for Technical Debt with LEAP

Technical debt within iterative releases can appear in two ways: a release may not deliver all of its intended capabilities or technology may not be ready in time for use in a release. Managing this technical debt requires identification of all of the capabilities and technologies needed in the current release and in future releases. The dependencies between these capabilities must be identified in both the temporal and the functional dimensions.

The LEAP process provides a clear indication of technical debt potential, primarily through the Investment Matrix. This matrix shows the impact of the late delivery of a technology on future capabilities. Larger numbers in the Investment Matrix indicate the technologies that are likelier sources of technical debt in the system, especially when they occur in later time periods. The Achieve phase attempts to reduce the accumulation of technical debt by investing in specific technology development.

The LEAP process can also be used to assist in assessing changes made during the course of system development. Particularly in a cost and schedule constrained environment hard trades must be made on system performance. These trades may occur during the design of the procurement or during program execution. Sometimes these trades may require the acceptance of lesser performance while other times they require selecting one technology over another. The impact of

these trades can be evaluated by modifying the Development Matrix and then seeing the changes propagate into the Delivery and Investment Matrices, paying particular attention to later time periods. These matrices can identify the long-term impacts of short-term decisions and therefore proactively identify potential sources of technical debt.

For example, consider a delay to the development of technology T1 in the matrices shown in Figure 3-6 that pushes its development time period from P2 to P3. The Technology Matrix shows that technology T2 is dependent upon technology T1, and therefore the delay in T1 delays the development of T2 by one time period. This delay cascades into the delivery of capabilities. Figure 3-9 shows the changes to the development timeline and the associated impacts, highlighting the changes in the gray squares. Clearly, the delay in the development of technology T1 impacted the delivery capabilities C3 and C4. The delay in C4 is not due directly to the delay of T1. Instead, it is the associated delay of T2 that causes C4 to slip, thus revealing a source of technical debt that otherwise may been missed.

Development Matrix, V

Time Periods

Technologies	P1	P2	P3	P4
T1	0	0	1	1
T2	0	0	0	1
T3	1	1	1	1
T4	0	0	0	1

Availability Matrix, A

Tactical Capabilities

Time Periods	C1	C2	C3	C4	C5
P1	0	0	0	0	0
P2	0	0	0	0	0
P3	0	0	1	0	0
P4	1	1	1	1	1

Delivery Matrix, D

Tactical Capabilities

Time Periods	C1	C2	C3	C4	C5
P1	0	0	1	0	0
P2	1	0	1	1	0
P3	1	0	0	1	1
P4	0	0	0	-1	0

Investment Matrix, I

Time Periods

Technologies	P1	P2	P3	P4
T1	1	2	0	0
T2	0	1	1	0
T3	0	0	0	0
T4	0	1	2	0

Figure 3-9. Impact of delay in Technology T1 development time on delivered capabilities

3.3.3.1.5 Discussion

The LEAP process is a newly developed process. Kleinwaks et al. [167] provide an example of its use to determine the right mix of research and development investments within the Space Development Agency (SDA). The LEAP process is iteratively applied in conjunction with large contract procurements and smaller research investments to develop and field technologies on two-year timelines. They conclude that the LEAP process enabled identification of technologies in need of investment in order to meet stakeholder need dates. In response, SDA made several research investments designed to accelerate technology development. The new development timelines were inserted into the LEAP process and the Delivery and Investment Matrices updated. Examination of the updated Delivery Matrix shows that satisfaction of stakeholder needs will occur sooner, but that further acceleration of technology development is still required. The Investment Matrix reveals the critical technologies that require additional investment and the impact they have on overall satisfaction of stakeholder needs in the required timelines. By using LEAP in an iterative process, SDA is able to carefully choose the requirements for its next iteration of satellites, which are developed on short, two-year cycles. These procurements necessitate steps to minimize schedule risk, including the use of mature technologies whenever possible [162]. LEAP identifies those technologies that require significant development ahead of their inclusion in a system procurement. If there is flexibility in the requirements selected for each iteration, then the technology can be developed outside of the main system iteration and therefore the risk to the iteration is reduced.

Although the LEAP process is newly developed, it builds off of agile and iterative development processes. Traditional agile and iterative development processes do not intrinsically consider technical debt and do not directly link the development of each technology with the ability to

satisfy the stakeholders' needs in both the functional and temporal dimensions. In Agile development, the iterations are often fixed in time, but they allow the value delivered in each iteration to change, limiting the ability to properly forecast the time at which a set of requirements will be satisfied. The selected features are often based on delivering the most immediate value to the stakeholder which can produce systems that are complicated and more expensive to change in later increments. Spiral development focuses on incrementally adding capability and using evidence-based risk assessments to continue development. However, the method does not provide processes to identify the long-term impacts of technology development timelines, instead relying on expert assessments [25]. The LEAP process augments these development methods by providing a proactive and objective method to assess the timeframe of stakeholder need satisfaction based on the maturity of supporting technologies.

However, the LEAP process itself is also under continuous refinement. The process deals in "absolutes" – technologies and capabilities are assumed to be fully developed in a specific time period. In reality, there are variances in schedules, cost, and performance that need to be accounted for when planning out the development cycles. The LEAP process relies on the ability to perform a full functional breakdown of the stakeholder needs into the supporting technologies. In practice, it can be challenging to determine the proper level of detail in each category to enable conclusive analysis of the outcomes. Future development and additional use cases will aid in clarifying the proper dividing lines between capabilities and technologies.

3.3.3.1.6 Conclusions and Future Work

The LEAP process, as presented in this paper, provides a novel approach to identifying the future impact of technology development on system capability. By identifying both the functional and temporal needs of the stakeholders and the developmental timelines of critical technologies,

the LEAP process deterministically assesses which technologies require investments to enable the on-time delivery of the needed capabilities. This assessment enables the reduction of technical debt input into the system by clearly identifying the impacts of technology completion and investment decisions. The user of the LEAP process can identify which technologies have the most impact on the ability to implement capabilities in future iterations of their system and therefore realistically assess multiple possible investment pathways for overall value and contribution to program success. Early identification of these technologies can reduce the risk of future rework and therefore prevent increases in development costs [121].

The ability to understand the impact of investments in critical technologies on the delivery of system capabilities has far-reaching implications for multiple users. Those users interested in the long-term development of a system can use the LEAP process to understand how to phase their investments over time and where they need to encourage the participation of other organizations to spur technology development. Users with shorter-term horizons can apply the same process to guide decisions to minimize the impact on the future system. The LEAP process can be used to determine corporate investments that are likely to pay the largest dividends and can also be used to assess the required technology developments that will prevent the system from entering technical bankruptcy.

The LEAP process as presented is undergoing continuous improvement. Topics for additional development of the process include the following:

- Adding a prioritization matrix to enable ranking of capabilities in each time period. The effect of this additional multiplication would be to change the values in the Investment Matrix to give insight into how valuable the late capabilities are to the stakeholders. While providing valuable insight, prioritization also has the potential to skew the results to a

preferred solution based on the input values. The impact of prioritization on the outputs of the process needs to be investigated in further detail.

- Adding probabilistic dependencies to the calculations to indicate the probability of delivering a capability on time. This methodology would allow for estimates of delivery based on probabilistic analysis instead of the binary condition presented in this paper. The probabilistic analysis facilitates the development of optimal development timelines and strategies for iterative releases through Monte Carlo simulations.
- Directly linking the Technology Matrix to the Development Matrix. This capability would enable the Development Matrix to be generated automatically from a sequenced Technology Matrix resulting in additional capability for automation and optimization.
- Continued verification and validation of the LEAP process. The LEAP process is newly developed and therefore there is not a significant amount of usage data on its successes and failures. Kleinwaks et al. [167] provide an example use of the LEAP process at the Space Development Agency. However, additional verification and validation of the process and its conclusions is required, in additional real-world scenarios. The authors plan to continue examining the use of LEAP in multiple case-studies and industries.

3.3.4 Addressing RQ2.2

The LEAP process identified in [160] provides a qualitative mechanism to identify potential technical debt sources within system development, addressing RQ2.2. By using the process, stakeholders can assess the impacts of decisions related to the timelines of technology development on the ability to deliver capabilities. With a detailed functional breakdown, critical technologies can be identified, even if those technologies do not provide high value to the user. Then, when schedule pressures or cost pressures increase, the impact of a compromise on a specific technology

can be assessed within the iterative system lifecycle. Technologies with large impacts are more likely to introduce technical debt into the system if compromises are made in their development. The process as defined in [160] satisfies the objectives identified in the Section 3.2.3 for consideration as a proactive process for identifying technical debt as shown in Table 3-4.

Table 3-4. LEAP satisfaction of proactive process for technical debt identification

	Objective	LEAP Capability
1	Identification of the system features that support the realization of the stakeholder needs	The Functional Matrix maps the stakeholder needs to the system features
2	Identification of the dependencies between the features and needs and between features and features, in the temporal and functional dimensions	The Technology Matrix maps the dependencies between features and the Functional Matrix maps the functional dependencies between needs and features. The Need Matrix shows the temporal dimension of the needs and the Development Matrix shows the temporal dimension of the features.
3	Assessment of the potential impact of a technical concession on the satisfaction of the stakeholder needs	The Delivery Matrix shows how the ability to temporally satisfy a stakeholder need changes with the delivery timeline of a technology (temporal dimension), which may change as a result of a technical concession (functional dimension). The Investment Matrix shows the impact level of a specific technology.
4	Easy to use, update, and understand by both system developers and stakeholders	The LEAP process can be implemented in a spreadsheet or simple software script and produces simple values of "delivered or not". Appendix A contains a sample Python implementation of the LEAP process.

Based on this assessment, the LEAP process meets the requirements to be a proactive process for identifying technical debt. However, as defined in [160], it is a qualitative process and its use as a decision support system for release planning is limited. The next chapter focuses on updating this process to be probabilistic, accounting for technical debt within the technology development timelines, and integrating the process into release planning, both within an iterative system development and also within a single iteration.

3.4 Conclusion

This chapter addresses RQ2: *How can potential sources of technical debt be identified during the system lifecycle?* Unfortunately, the tools enabled within software engineering to identify technical debt are not directly applicable to systems engineering and are primarily reactive tools. Therefore, their application identifies technical debt after it has been inserted into the system. To minimize the impact of technical debt, proactive methods need to be implemented such that mitigation plans can be put in place immediately, moving the technical debt from negligent to strategic.

The LEAP process is proposed as a proactive method of identifying potential sources of technical debt. It enables the system developer to assess the impact of a delay in technology development on the ability to meet the temporal and functional requirements of the system stakeholders. By identifying which technologies drive delays in the delivery of capability to stakeholders, the process can proactively identify potential technical debt sources. If a technology's development path does not support the stakeholder delivery dates, it will be identified through this process. By altering the values in the LEAP Development Matrix, the system developer can assess how different investments affect the ability to deliver capability, highlighting situations where delivery of high value items at the expense of infrastructure can end up delaying the satisfaction of the stakeholder needs. This process highlights where investments should be made to achieve on time delivery, but does not directly model the impact of technical debt created by one technology on its successors.

The next chapter updates the LEAP process to address these considerations by quantitatively modeling the impact of technical debt and including probabilistic values in the LEAP matrices.

These updates enable the LEAP process to be used to assess the likelihood of satisfying the stakeholder needs in the presence of technical debt.

CHAPTER 4 – USING TECHNICAL DEBT AS A GUIDE IN RELEASE PLANNING

4.1 Introduction

Iterative system development methods are often used when the requirements for the system are uncertain or when the environment is complex and changing [28]. By releasing a system in multiple iterations, the requirements can be adjusted as system context and environment evolve. Planning these iterations is the domain of release planning: the act of selecting which features of the system to include in each release to maximize value to stakeholders while accounting for constraints and dependencies [93]. While release planning is frequently used in conjunction with Agile software development strategies, systems are also iteratively and incrementally developed. Release planning in systems engineering is not a direct analog to release planning in software engineering as the constraints associated with hardware development add complexity to the increments and iterations [13]. This added complexity increases the need for proactive identification of technical debt within systems engineering release planning.

There are many different methods for release planning, each of which tries to produce the "best" combination of features for each release. The definition of "best" varies based on the method, criteria, and viewpoints of the stakeholders. The available set of software release planning models are not thoroughly validated through practice, and have difficulties with less than certain requirements or when stakeholders are not available to provide context [168]. These methods are also limited in their application of uncertainty [28], especially with regards to the impact of technical debt. Technical debt introduces uncertainty into the system due to the nature of the technical debt interest – it is uncertain whether or not the interest will be realized and how large the impact will be. Release planning methods involving technical debt often focus on paying back

the technical debt as part of the release [94]. However, the impact of technical debt on the ability to complete future features, such as the increased level of effort required to implement a feature that is subject to significant technical debt, needs to be included in the release plan. Including these impacts enables accurate representations of the ability to complete the feature list within a given release. Proactive identification of potential technical debt sources should change the priority of feature implementation within a release planning cycle to limit the future impact. The earlier that a potential issue is addressed, the less costly it will be to correct that issue [135].

Based on this background, there is a need to account for technical debt within release planning methods, especially within systems engineering contexts. However, as Chapter 2 identified, technical debt is not a well-researched phenomenon within systems engineering and the author was unable to locate any iterative development methods within systems engineering or software engineering that proactively account for technical debt. The LEAP process, introduced in Chapter 3, provides a capability to proactively identify potential technical debt sources. Therefore, integration of the LEAP process into release planning will account for technical debt. This chapter provides this integration to address RQ3: *How can technical debt be used as a guide in release planning?*

This research question is addressed through the accomplishment of two tasks. Task 3.1 establishes a quantitative version of the LEAP process to create a probabilistic analysis model. Task 3.2 demonstrates how the quantitative LEAP process can be used as a decision support system for release planning.

4.2 Quantitative LEAP Process

The qualitative LEAP process, presented in Chapter 3, provides a starting point for enabling the assessment of technical debt within release planning. It provides a mechanism to identify the

schedule impact of technical compromises on the satisfaction of stakeholder needs. However, to be used as a resource for release planning, the LEAP process needs to be updated to add two key elements:

- Quantitative assessment of the impact of technical debt on the duration of a task and its successor tasks; and,
- Probabilistic determination of the temporal satisfaction of stakeholder needs.

The quantitative assessment of technical debt impact is required to enable tradeoffs between different release plans. A system developer can model the technical debt introduced into the system against the temporal ability to complete a task, thereby determining not only a preferred development order of features but also determining the impact of injecting or reducing technical debt within the system. Section 4.2.1 discusses the modeling of technical debt within a Monte Carlo schedule analysis process.

The Monte Carlo schedule analysis will generate probabilities for the development timelines of technologies, accounting for their interdependencies and technical debt. These probabilities can then be included in the LEAP process model. However, the basic matrix operations used in the qualitative LEAP process are insufficient to deal with the probabilistic representation and therefore the equations used to generate the outputs of the LEAP process need to be modified. Section 4.2.2 discusses the modifications to the LEAP process to enable probabilistic analysis.

4.2.1 Including Technical Debt in Project Schedule Analysis

Technical debt is introduced into a system as a result of a technical compromise that has long-term impacts [21]. The long-term nature of the impacts is what make technical debt so pernicious. It is not the task where the compromise is made that will require increased effort. Rather, any of the successor tasks, including those several steps removed from the initial technical debt

introduction, could be impacted by the technical compromise. Traditional schedule analysis techniques account for task relationships and variations on task durations, but tend to assume that each task is completed perfectly. Successor tasks are assumed to have the same duration and level of effort regardless of any technical debt introduced by predecessor tasks. However, the presence of technical debt can cause increased effort to complete a task and this impact on successor tasks is not included in traditional schedule analysis techniques. Therefore, a new method for schedule analysis in the presence of technical debt was created and the following paper was submitted to *IEEE Access* in July, 2023 [169]. This paper applies notional technical debt considerations to an aircraft development project to demonstrate how technical debt can be included in schedule analysis.

4.2.1.1 Predicting the Dynamics of Earned Value Creation in the Presence of Technical Debt [169]

4.2.1.1.1 Abstract

Technical debt, the long-term impact of decisions made to achieve a short-term benefit, has a unique impact on a project schedule. Technical debt does not impact the ability to complete the task on which it is incurred but rather impacts successor tasks causing unplanned schedule delays or budget increases. The impact of technical debt is uncertain and therefore must be modeled probabilistically. When unaccounted for and unmanaged, technical debt can build up in the project with increasing impact, eventually forcing forward progress to stop while the technical debt is remedied. Traditional project scheduling methods allow for uncertain task durations but do not provide explicit means of modeling the impacts of technical debt. Instead, they assume that each task is unaffected by the completion status of its predecessors and its duration is only dependent upon the initial estimates. This research addresses this gap by providing a novel model of the

impact of technical debt on the project schedule through estimating the dynamics of value creation in the presence of technical debt. Equations are developed for estimating the probabilistic impacts of technical debt on the generation of earned value. These equations are then inverted and used to calculate task duration in the presence of technical debt and included in a Monte Carlo analysis. Comparisons are made to an existing Monte Carlo schedule analysis and technical debt impacts are explored.

4.2.1.1.2 Introduction

Project managers traditionally handle uncertainty by including cost and schedule margin in their project plans [170]. These margins can be used to mitigate the impact of rework and technical debt within a project. Love defines rework as the "unnecessary effort of re-doing a process or activity that was incorrectly implemented the first time" [171]. Kleinwaks, Batchelor, & Bradley define technical debt as "a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a system" [19]. Within the context of a project, technical debt occurs when decisions made in the completion of one task negatively impact the ability to complete successor tasks on time and on budget. The impact of technical debt is not certain: the compromises made on one task may or may not impact a successor task [51] and the compromises may proliferate throughout the system and cause significant issues [92]. Within this article, technical debt is distinguished from rework by asserting that rework is the result of the poor execution of defined processes and methods and technical debt is the result of shortcuts taken in the requirements development, design, and/or implementation in order to achieve a short-term benefit. Rework requires the repeated execution of existing process and unplanned iterations of existing tasks. Technical debt does not typically require the redoing of a specific task but instead technical debt makes completing successor tasks more complicated, costly, or time-consuming. If technical

debt is not accounted for in project scheduling, then successor task duration may increase unexpectedly, resulting in late completion of tasks compared to stakeholder expectations. However, traditional schedule analysis techniques do not model changes in successor task durations based on the fidelity of predecessor task completions. This article provides a novel mechanism to address this gap and enable more realistic schedule assessments.

Properly assessing project schedules requires the ability to proactively predict risks associated with both technical debt and rework [172]. Monte Carlo simulation is often used to assign probabilistic durations to tasks, assuming that the task will be completed within the bounds of the assigned distribution. However, these simulations can overlook the costs associated with changes to the schedule [170] as a result of technical debt or rework [173]. Several authors have investigated the use of design structure matrices (DSM) to predict the impact of design iterations on project schedule [174] [175] [176] [177]. These techniques can be used to assess the probability of rework occurring within a project and the extensions to schedule that occur. However, they do not model the potential for technical debt. Ma et al. [174] extend DSMs to include a probability of rework and its impact on future tasks in the context of design iterations. However, in many projects, iterations are not planned – the successor tasks must be extended or changed to address the shortcomings of the predecessor tasks. Furthermore, while modeling rework can account for project extensions, it is not the same as modeling technical debt. Rework results in repeated execution of the same tasks. Technical debt may result in longer durations of successor tasks and the potential need for unplanned effort to remove the debt from the system.

Maheswari and Varghese [177] provide a method to use DSMs to determine a project schedule accounting for overlapping tasks. By assessing the necessary condition of task overlap in a project, they demonstrate that tasks do not always abide by strict finish-to-start schedule relationships.

However, their work assumes a fixed value of the overlap time and does not provide a mechanism to calculate when a task reaches that level of completion. Additionally, they assume that the work completes perfectly until the overlap time is reached without considering technical debt's task to task dependencies.

From this review, it is clear that additional techniques to handle the presence of technical debt within a project schedule are required. Failure to model technical debt can result in overly optimistic schedule estimates due to the failure to account for the cascading impact of technical debt interest. The technical debt incurred on one task can compound, impacting multiple successor tasks, resulting in significant delays and cost increases to the project.

In this article, we extend existing project schedule analysis methods to include technical debt analysis. The impact of technical debt incurred on one task on successor tasks is modeled through earned value computations. The earned value equations are inverted to estimate the duration of successor tasks subject to technical debt from their predecessors. With these equations, the impact of technical debt is then included in a Monte Carlo schedule analysis and the results compared to a traditional Monte Carlo schedule analysis. Various impacts of technical debt are explored by altering the parameters in the analysis. This article answers the following research question:

How can technical debt be accounted for within project scheduling activities?

By answering this research question, this article presents a mathematical model that can be used by project managers and schedulers in Monte Carlo schedule analysis techniques. This model uses the technical debt formulation to compute increased duration of successor tasks, thereby providing a more realistic schedule analysis.

The rest of this article is structured as follows: first, overview of related work is provided. Next, the method used to account for technical debt within a schedule is described and is followed by an example application of the method within Monte Carlo schedule analysis. Finally, the results are discussed and opportunities for future work are presented.

4.2.1.1.3 Related Work

Earned value management expresses the project progress in terms of value created, where value is expressed in monetary terms. The creation of value is then used to predict both the project cost at completion and the schedule at completion through linear extrapolation of the current state [178]. While EVM is traditionally effective in cost management, its schedule management component is usually considered insufficient, especially since the schedule is expressed in cost parameters [179]. These weaknesses led to the development of earned schedule (ES) techniques [180]. ES techniques have been shown to be more accurate in predicting the schedule at completion [181] and can be more easily understood, as they measure the earned schedule in units of time (and not cost). Both EVM and ES use the same planned value and earned value curves, which take the form of an 'S-curve', shown in Figure 4-1. The planned value is based on the baselined project development plan, while the earned value is based on measured project progress. EVM techniques measure the difference between the two curves in the vertical direction while ES techniques measure the difference between the two curves in the horizontal direction.

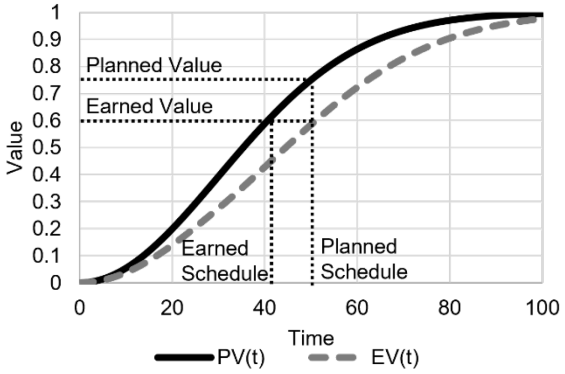

Figure 4-1. Planned and earned value 'S-curves'

Warburton [182] formulated (4-1) to (4-4) to represent planned value (PV) and earned value (EV) curves. Lowercase letters represent the instantaneous value and capital letters represent the cumulative value. Note that (4-4) in [182] contains an error where the negative sign on the first exponential was excluded. That error has been corrected in (4) in this article. The variables used in these equations are defined in Table 4-1.

$$pv(t) = \frac{Nt}{T^2} e^{\frac{-t^2}{2T^2}} \tag{4-1}$$

$$PV(t) = \int_o^t pv(s)ds = N\left[1 - e^{\frac{-t^2}{2T^2}}\right] \tag{4-2}$$

$$ev(t) = \begin{cases} (1-r)pv(t),\ t \leq \tau \\ (1-r)pv(t) + r * pv(t-\tau),\ t > \tau \end{cases}$$

$$ev(t) = \begin{cases} (1-r)\frac{Nt}{T^2} e^{\frac{-t^2}{2T^2}},\ t \leq \tau \\ (1-r)\frac{Nt}{T^2} e^{\frac{-t^2}{2T^2}} + r\frac{Nt}{T^2} e^{\frac{-(t-\tau)^2}{2T^2}},\ t > \tau \end{cases} \tag{4-3}$$

$$EV(t) = \begin{cases} EV_1(t) = \int_0^t ev(s)ds,\ t \leq \tau \\ EV_2(t) = EV_1(\tau) + \int_\tau^t ev(s)ds\,, t > \tau \end{cases}$$

$$EV(t) = \begin{cases} (1-r)N\left[1 - e^{\frac{-t^2}{2T^2}}\right], t \leq \tau \\ N - N(1-r)e^{\frac{-t^2}{2T^2}} - rNe^{\frac{-(t-\tau)^2}{2T^2}}, t > \tau \end{cases} \quad (4\text{-}4)$$

Table 4-1. Variables used in earned value equations

Symbol	Definition
N	The number of tasks occurring in the project
T	The time at which the maximum value of the instantaneous planned value curve occurs
t	The independent time variable
r	Percentage of activities that require extra work
τ	Delay introduced into the project due to extra work

Equation 1 defines the instantaneous planned value function. This equation models a project where the planned value achieved at each point in time, for example, work accomplished each day, initially increases until time T, which is the time at which the maximum instantaneous planned value is reached. After this point, the contributions to planned value in each time period steadily decrease. The cumulative planned value is calculated using (2). This equation, the integral of (1), produces the traditional S-curve, as shown in Figure 1. Equation 3 calculates the instantaneous earned value by assuming that a fraction of the tasks, r, are late by a time τ, thereby delaying the accumulation of value. Equation 4 computes the cumulative earned value as the integral of the instantaneous earned value [182].

This related research forms the basis of the process for accounting for technical debt in the schedule analysis. Building off of the equations for earned value, the time at which a task reaches the necessary conditions for the successor task to start can be established. The r and τ parameters allow for the modeling of delays introduced into a task from its predecessor tasks, a key component of technical debt. Attaching these equations to a Monte Carlo analysis allows for the modeling of the probabilistic aspects of technical debt interest.

4.2.1.1.4 Accounting for Technical Debt in Schedule Analysis

Accounting for technical debt in schedule analysis starts with understanding how to measure task completion. Technical debt occurs when technical compromises are made in the execution of a task in order to achieve a short-term benefit [19]. The technical compromises may impact the scope of the task, resulting in reduced performance relative to its objectives, or in the quality of the task, resulting in lower maintainability, upgradability, sustainability, and other -ilities. These compromises may then impact the ability to complete future tasks on time, on budget, or to their performance specifications [18]. For example, technical debt is incurred when the documentation associated with a system component is reduced (technical compromise) to release on time (short-term benefit). The lack of documentation may make integration and testing of the component more time consuming and more costly (long-term impact). Kleinwaks, Batchelor, and Bradley conducted a survey on the presence of technical debt within systems engineering, concluding that, although the terminology of technical debt is not well used within systems engineering, the impacts of technical debt are substantial [18].

The size of the impact of technical debt, referred to as the interest amount within software engineering [51], is uncertain and dependent upon both the technical compromise and the interconnectedness of the task within the system context. The occurrence of the interest, defined as the interest probability [51], is uncertain – if no changes need to be made to the component carrying the technical debt, then no interest needs to be paid. Technical debt may remain hidden in a system and linger for extended periods of time, compounding the interest amount and resulting in more complicated, or even impossible, design changes.

4.2.1.1.4.1 Utility as Value

When modeling project value for schedule analysis, the use of a monetary metric as the project value can confuse value and utility. While project duration ultimately relates to project cost, the ability of one task to begin work is not related to how much profit that predecessor task generates. Therefore, this article assumes that a value function can be formulated in terms other than financial terms [183]. Specifically, this article models value as the utility of a task to its successor tasks, where utility is measured as the completion percentage of the predecessor task. A successor task may be able to begin work when a predecessor task is not complete (has a utility of less than one (1)), an implementation of the start-start relationship [178] of traditional project scheduling techniques. The value function is modeled as an S-curve, a relationship that has been shown to hold for task duration as well as cost [179] and which enables the time at which the task reaches a specified utility (value) to be found. Therefore, the start time of the successor tasks can be determined, leading to the calculation of the overall project duration.

4.2.1.1.4.2 Modeling Earned Value from Multiple Predecessors

Modeling technical debt impact requires the ability to determine both the interest amount and the interest probability and to account for their impacts on the value creation of a specific task. Since the interest could come from any of the predecessor tasks, it is necessary to determine the contributions to the value of a task that is derived from each of its predecessors. Adopting an S-curve formulation of the value function, modifications to Warburton's equations can be made to calculate the earned value contributions from each predecessor task in turn. As written, Warburton's equations assume that the earned value is contributed evenly from multiple predecessor tasks. The *N* parameter is used to represent the number of predecessor tasks, which changes the magnitude of the overall planned and earned value, but only in aggregation. Each

predecessor task contributes the same portion of the value. This model is appropriate for planned value, which assumes perfect schedules. However, earned value, which attempts to model the actual value creation schedule, must account for the individual impacts of predecessor tasks on the earned value of the successor task. Equations 4-5 and 4-6 show the updated equations for earned value accounting for the impacts of the predecessors. *N* becomes a scaling variable applied evenly to all the predecessor tasks.

$$ev(t) = \sum_{i=0}^{n} \begin{cases} (1-r_i)\alpha_i pv_i(t),\ t \le \tau_i \\ (1-r_i)\alpha_i pv_i(t) + r_i\alpha_i pv_i(t-\tau_i),\ t > \tau_i \end{cases}$$

$$ev(t) = \sum_{i=0}^{n} \begin{cases} (1-r_i)\frac{\alpha_i N t}{T^2} e^{\frac{-t^2}{2T^2}},\ t \le \tau_i \\ (1-r_i)\frac{\alpha_i N t}{T^2} e^{\frac{-t^2}{2T^2}} + r_i \frac{\alpha_i N t}{T^2} e^{\frac{-(t-\tau_i)^2}{2T^2}},\ t > \tau_i \end{cases} \quad (4\text{-}5)$$

$$EV(t) = \sum_{i=0}^{n} \begin{cases} EV_{1i}(t) = \int_0^t ev_i(s)ds,\ t \le \tau_i \\ EV_{2i}(t) = EV_{1i}(\tau) + \int_{\tau_i}^t ev_i(s)ds\,, t > \tau_i \end{cases}$$

$$EV(t) = \sum_{i=0}^{n} \begin{cases} (1-r_i)\alpha_i N\left[1 - e^{\frac{-t^2}{2T^2}}\right],\ t \le \tau_i \\ \alpha_i N - \alpha_i N(1-r_i)e^{\frac{-t^2}{2T^2}} - r_i\alpha_i N e^{\frac{-(t-\tau_i)^2}{2T^2}}, t > \tau_i \end{cases} \quad (4\text{-}6)$$

In (4-5) and (4-6), it is assumed that each predecessor task independently impacts a portion of the successor's task earned value. This portion is controlled by the α variable, which is the percentage of the successor task's earned value that is impacted by predecessor task *i*. The α variables are constrained to add up to one, as shown in (4-7).

$$\sum_{i=0}^{n} \alpha_i = 1 \quad (4\text{-}7)$$

α_0 is the percentage of the successor task's earned value that is not impacted by any predecessor and can be calculated using (4-8).

$$\alpha_0 = 1 - \sum_{i=1}^{n} \alpha_i \quad (4\text{-}8)$$

Figure 4-2 depicts the contribution of multiple predecessors to the earned value of a successor task. In Case 1, each predecessor contributes equally to the earned value of Task D. In Case 2, the individual contributions are not equal, resulting in different values of α. Changing the values of r and τ for each predecessor task will change the overall earned value based on the values of α, which is discussed in the next section.

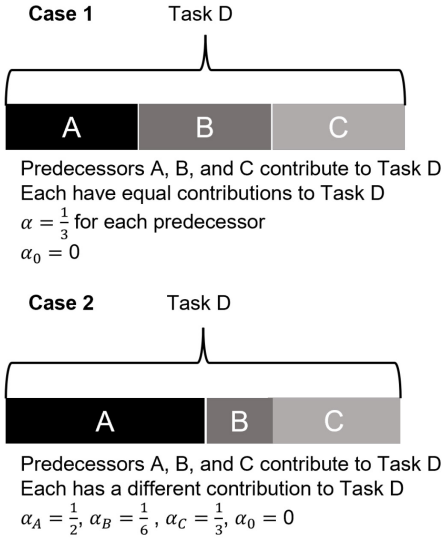

Figure 4-2. Multiple predecessor contribution to earned value

4.2.1.1.4.3 Technical Debt and Earned Value

Warburton's equations can be used to model the impacts of technical debt interest on the system by redefining the variables r and τ. Warburton defines r as the percentage of activities that require rework. Within the multiple predecessor and technical debt context, r is redefined as the percentage of the predecessor task's impact on the successor task that is subject to a delay. This relationship is shown in Figure 4-3. In this figure, $\alpha_A = 0.25$: task A impacts 25% of the earned value of task D. $r_A = 0.25$ and therefore 25% of task A's impact on task D is subject to technical debt interest from task A. Combined, 6.25% of task D is subject to delays due to technical debt interest from task A.

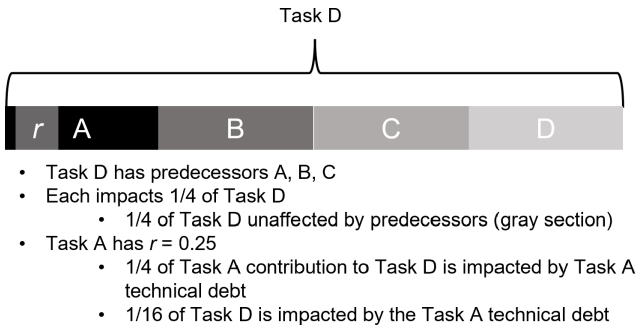

Figure 4-3. Definition of r parameter in the context of multiple predecessors and technical debt

The definition of τ is unchanged from Warburton – it is a measure of the delay introduced to the system due to technical debt interest. It measures how much longer a task takes to complete based on the technical debt introduced by a predecessor task. The impact of changing r and τ is shown in Figure 4-4. Increasing *r* shifts the earned value curve to the right along the time axis but does not significantly change the slope – it changes the time at which the value is accumulated but not the rate. Changing τ changes the slope of the earned value curve thereby affecting the rate of value accumulation along with the time at which the value is earned.

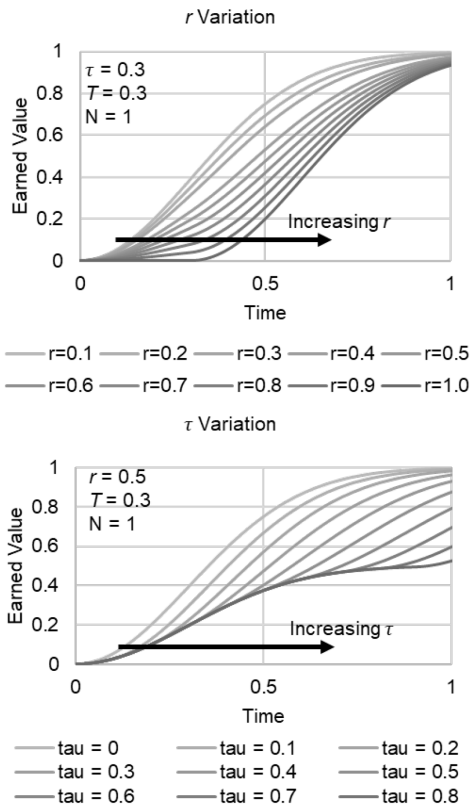

Figure 4-4. Effect of changing r and $\boldsymbol{\tau}$ *on earned value*

In terms of technical debt, r and τ, when combined, represent the interest amount. The interest probability can be modeled through the specification of probability distributions for r and τ and the use of Monte Carlo analysis. The impact of r and τ and the relationship to technical debt is best understood through an example.

Williams [173] defines a schedule for the development of a test aircraft, including the expected duration for each of the tasks. This schedule will be used throughout the rest of this article as an example project. Williams provides a relevant example to technical debt through the discussion of the third management action in his aircraft example: if avionics production is delayed, then a temporary avionics kit may be installed in production aircraft. The technical compromise is to use a non-fully functional avionics kit to achieve the short-term benefit of meeting the task schedule. The long-term impact is the lack of fully functional aircraft and the potential for rework to retrofit

the avionics kit. In Williams' example, 28% of the aircraft had the temporary kit installed, so r = 0.28. Williams does not provide the timeline to produce additional kits, but it is fair to estimate that it would be the same as the avionics production task and range between 12-18 months. Therefore, τ could be estimated through a distribution that produces values in the range of 12-18 months. r and τ would then be applied to the earned value equation for the aircraft assembly along with an estimate of the alpha value – the portion of the aircraft assembly affected by the avionics.

4.2.1.1.4.4 Compounding Technical Debt Interest

One of the most pernicious qualities of technical debt is that the interest compounds. Technical debt may impact multiple successor tasks, may not appear until several successor tasks have completed, and it may grow in impact as it affects more tasks [92]. To model the compounding of technical debt interest, it is necessary to consider all the predecessor tasks as having some contribution to the earned value of the successor task. If direct predecessor tasks are the only ones considered, then there is a chance that the technical debt contribution is underestimated. For example, consider the development of a software interface with three tasks: development of the interface control document (ICD), writing the software code, and integrating the software interface. An ICD may contain documentation debt [60], which includes the under specification of the interfaces. The software developer can take the ICD and perfectly implement it as written, and may not be aware that the interfaces were underspecified. It is not until the next task, the integration of the interface, that the technical debt in the ICD will appear, even though the ICD is not a direct predecessor of the integration task.

To model the compounding of technical debt interest, it is necessary to specify the α, r, and τ values for each possible predecessor for every task. Figure 4-5 shows an example of how to specify the values using two design structure matrices (DSM), based on the aircraft project provided in

Williams [173]. The dependency matrix, on the left of Figure 4-5, indicates the direct predecessor (value of 1) and indirect predecessor relationships (value of 2). The alpha matrix, on the right side of Figure 4-5, indicates the value of alpha for the relationships. These matrices are read like other DSMs, where a value in a cell indicates that the column contributes to the row. The dependencies of task 5, interim avionics, are found by reading down the column. This task has one immediate successor (task 7, assemble d/b aircraft) and several secondary successors. The value of alpha for the immediate successor is 0.1. To maintain the constraint identified in (4-7), the summation of the values in the rows of the alpha matrix must equal one (1). Similar DSMs could be created for r and τ.

Dependencies

ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ID	General Design	Engine Design	Avionics Design	d/b airframe design	d/b engine manufacture	interim avionics	d/b airframe manufacture	assemble d/b aircraft	engine development	engine production	avionics test	avionics flight trials	engine/frame flight trials	airframe production	avionics production	ready to assemble
0																
1																
2	1															
3	1															
4		1														
5	2		1													
6	2			1												
7	2	2	2	2	1	1	1									
8		1														
9		2							1							
10	2		1								1					
11	2		2													
12	2	2	2	2	2	2	2	1								
13	2	2	2	2	2	2	2	1					1			
14	2	2	2	2	2	2	2	2			2	1	1			
15	2	2	2	2	2	2	2	2	2	1	2	2	2	1	1	

Alpha

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
General Design	Engine Design	Avionics Design	d/b airframe design	d/b engine manufacture	interim avionics	d/b airframe manufacture	assemble d/b aircraft	engine development	engine production	avionics test	avionics flight trials	engine/frame flight trials	airframe production	avionics production	ready to assemble
1															
	1														
0.4		0.6													
0.4			0.6												
	0.7			0.3											
0.1		0.7			0.2										
0.1			0.7			0.2									
0.2	0.1	0.1	0.2	0.1	0.1	0.1	0.1								
	0.7							0.3							
	0.4							0.4	0.2						
0		0.7								0.3					
0		0.4								0.4	0.2				
0.2	0.2	0	0.2	0.1	0	0.1	0.1					0.1			
0.2	0	0	0.7	0	0	0	0					0	0.1		
0.2	0	0.7	0	0	0	0	0			0	0	0		0.1	
0.1	0.1	0.1	0	0	0	0	0	0	0.2	0	0	0	0.2	0.2	0.1

Figure 4-5. Specification of compounding technical debt

From these matrices, it becomes clear which tasks may have larger impacts throughout the system. For example, summing the columns in the α matrix will provide a total of the impact percentage of a specific task. Larger values will have higher potential for compounding technical debt interest.

4.2.1.1.4.5 Calculating the Time at Which Earned Value is Reached

Equation 4-6 models the earned value, in the presence of technical debt and multiple predecessors, as a function of time. Therefore, if this equation could be solved for *t*, then the time at which a specified earned value is reached could be analytically determined. However, this equation is a transcendental equation and is not analytically solvable, especially in the presence of an unknown number of predecessor tasks. Numerical techniques could be used; however, they do not lend themselves to easy application.

Examining the shape of the S-curve reveals that there are four distinct sections [184]:

- Stage 1: value accumulation starts out slowly, usually as the project is ramping up
- Stage 2: value accumulates rapidly as more resources are put into the project and work is delivered
- Stage 3: value accumulation slows down as the bulk of the work is completed and resource loading starts to reduce
- Stage 4: additional value accumulation is minimal as tasks are finalized and the project is concluded

These stages are shown in Figure 4-6.

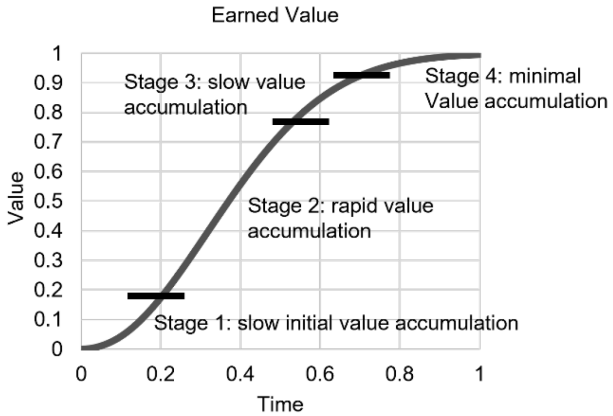

Figure 4-6. Stages of earned value S-curve

Between each of these stages, the concavity of the S-curve changes direction. A piece-wise linear function can be used to approximate the curve, with a separate line for each of the four sections [184]. Determining this piecewise function requires identifying the transition points between the changes in concavity.

The changes in concavity of the function are found by taking the derivatives of the function and setting those derivatives to zero. The derivatives of the earned value function are not directly solvable, due to the presence of multiple exponentials and the unknown number of predecessor tasks. However, the planned value function only contains a single exponential and does not depend on the number of predecessor tasks and therefore the transition points on the planned value curve can be found. Figure 4-7 plots the cumulative planned value (*PV*), instantaneous planned value (*pv*), and the derivative of the instantaneous planned value ($\frac{dp}{dt}$). The solid black lines represent the transition points on the *PV* curve. The three transition points, G_1, G_2, and G_3, can be found by applying successive derivatives of the *pv* curve.

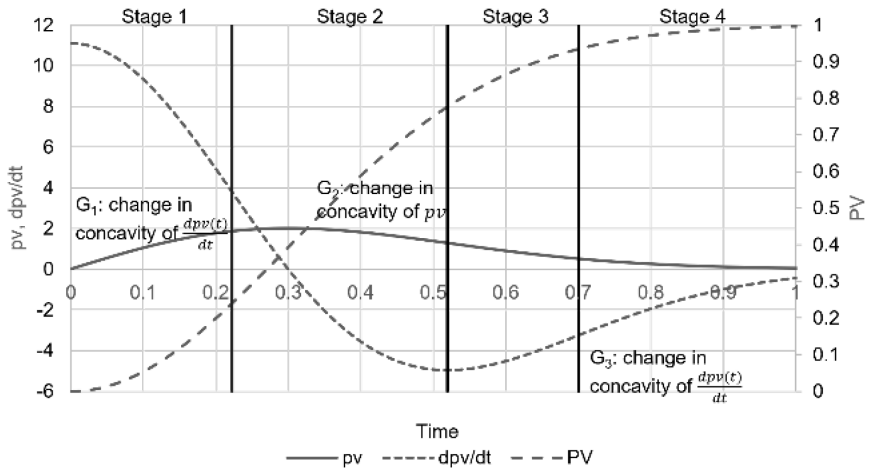

Figure 4-7. Concavity changes indicating transition points between growth stages in planned value

The first transition point to be found is G_2 – the transition from stage 2 to stage 3 [185]. As seen in Figure 4-7, this transition point occurs where the concavity of *pv* changes. Candidates for

changes in concavity are found by finding the roots of the second derivative of a function. Equation 4-9 shows the second derivative of the pv function. Therefore, G_2 is found by solving (4-9) for *t*, which is shown in (4-10)[1]. Only the positive roots are considered in this analysis.

$$\frac{d^2pv(t)}{dt^2} = \frac{N}{T^6} te^{\frac{-t^2}{2T^2}}(t^2 - 3T^2) \quad (4\text{-}9)$$

$$G_2 = \pm\sqrt{3}T \quad (4\text{-}10)$$

Transition points G_1 and G_3 occur when the concavity of $\frac{dp}{dt}$ changes, as shown in Figure 4-7. Therefore, the second derivative of $\frac{dp}{dt}$, which is the third derivative of *pv*, needs to be found and solved. Equation 4-11 shows the third derivative of *pv*, and solution is shown in (4-12). Again, only the positive roots are used in this analysis.

$$\frac{d^3pv(t)}{dt^3} = \frac{N}{T^8} e^{\frac{-t^2}{2T^2}}(t^4 - 6t^2T^2 + 3T^4) \quad (4\text{-}11)$$

$$G_1 = \pm\sqrt{-(\sqrt{6} - 3)T^2}, G_3 = \pm\sqrt{(\sqrt{6})T^2 + 3T^2} \quad (4\text{-}12)$$

With the transition points known, the piecewise linear equation for the planned value (*LPV*) can be found, as shown in (4-13).

$$m_1 = \frac{PV(G_1)}{G_1}$$

$$m_2 = \frac{PV(G_2) - PV(G_1)}{G_2 - G_1}$$

[1] Derivatives and solutions were checked using the Online Equation Solver from Wolfram Alpha, available at https://www.wolframalpha.com/calculators/equation-solver-calculator/.

$$m_3 = \frac{PV(G_3) - PV(G_2)}{G_3 - G_2}$$

$$m_4 = \frac{1 - PV(G_3)}{1 - G_3}$$

$$LPV(t) = \begin{cases} m_1 * t, & t \leq G_1 \\ m_2 * (t - G_1) + PV(G_1), & G_1 < t \leq G_2 \\ m_3 * (t - G_2) + PV(G_2), & G_2 < t \leq G_3 \\ m_4 * (t - G_3) + PV(G_3), & G_3 < t \leq 1 \end{cases} \quad (4\text{-}13)$$

Equation 4-13 can be easily solved for t to determine the time at which a specific planned value occurs. Following the same process to linearize the earned value equations results in unsolvable derivative equations due to the combination of multiple exponentials and the unknown number of predecessor tasks. A possible solution is to use the transition points found in the planned value curve as the transition points of the earned value curve. The reuse of these points will induce error in the time dimension of the linearization, which needs to be characterized. The resulting linearization of planned value and earned value is shown in Figure 4-8. In this case, the linearized earned value plot underestimates the earned value in the stage 4 and overestimates the earned value in stage 1.

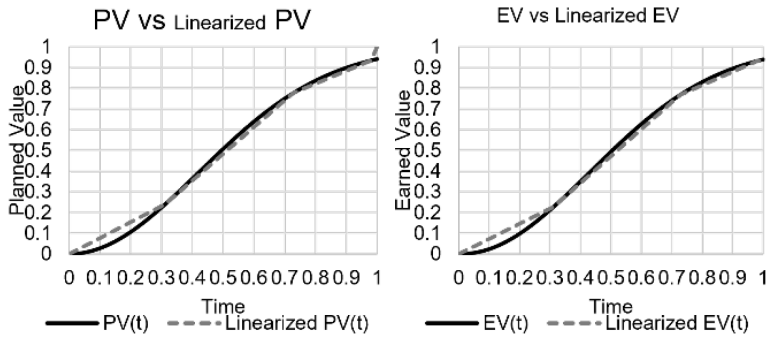

Figure 4-8. Linearized planned and earned value curves using the same transition points

With the transition points set, the linearized earned value equations can be determined and then solved for *t*, as shown in (4-14) and (4-15), where *V* is the desired earned value. The impact of

multiple predecessors is included in the linearization by using the complete earned value (EV) equation (4-6) at each of the transition points.

$$m_1 = \frac{EV(G_1)}{G_1}$$

$$m_2 = \frac{EV(G_2) - EV(G_1)}{G_2 - G_1}$$

$$m_3 = \frac{EV(G_3) - EV(G_2)}{G_3 - G_2}$$

$$m_4 = \frac{EV(1) - EV(G_3)}{1 - G_3}$$

$$LEV(t) = \begin{cases} m_1 * t, t \leq G_1 \\ m_2 * (t - G_1) + EV(G_1), G_1 < t \leq G_2 \\ m_3 * (t - G_2) + EV(G_2), G_2 < t \leq G_3 \\ m_4 * (t - G_3) + EV(G_3), G_3 < t \leq 1 \end{cases} \quad (4\text{-}14)$$

$$t = \begin{cases} \frac{V}{m_1}, V \leq EV(G_1) \\ \frac{(V - EV(G_1))}{m_2} + G_1, EV(G_1) < V \leq EV(G_2) \\ \frac{(V - EV(G_2))}{m_3} + G_2, EV(G_2) < V \leq EV(G_3) \\ \frac{(V - EV(G_3))}{m_4} + G_3, EV(G_3) < V \leq 1 \end{cases} \quad (4\text{-}15)$$

t represents the time at which the task reaches a particular earned value. Successor tasks may be able to start at *t*, however, the task is not necessarily complete at this point in time. The time of task completion is found by calculating when the earned value equals the total planned value for the task. The total planned value is input into (4-15) as *V* and then the task completion time is found. The difference between the task completion time and the original planned duration is the penalty on the task due to technical debt.

With (4-15), it is now possible to determine the time at which a task earns a particular value and the time at which it will finish in the presence of technical debt from multiple predecessors. The algorithm is as follows:

1. Set the values of α, r, and τ for each predecessor task
2. Based on the value of T for the task, determine the transition points G_1, G_2, and G_3 using equations
3. Calculated the earned value at each transition point for each predecessor task using (6)
4. Given the desired earned value V, calculate t from (4-15)

An accuracy assessment of this method is provided in the appendix.

4.2.1.1.5 Application to Monte Carlo Schedule Analysis

The prior analysis shows how to calculate the time at which a task reaches a desired earned value in the presence of technical debt. A Monte Carlo analysis can be used to determine the most likely duration of the entire project, accounting for technical debt along the way. Table 4-2 shows the parameters used in the analysis and recommended random and static variables. The random variables are assigned probability distributions, such as normal or triangular distributions and the accompanying distribution parameters are set as static variables. Static variables are held constant through each trial of the Monte Carlo analysis while random variables are resampled and changed with each Monte Carlo trial. Variables either apply to a singular task, such as the independent duration, or to a pair of tasks, such as r and U.

Table 4-2. Recommended random and static variables for Monte Carlo analysis

Random Variables	Static Variables
D – independent duration for each task	α – % of successor impacted by predecessor. Applies to task pairs α_0 - % of successor that is not impacted by predecessors. Applies to each task

Random Variables	Static Variables
r - % of α that is impacted by technical debt interest. Applies to task pairs. r_0 - % of α_0 that is impacted by self-inflicted technical debt interest. Applies to each task	U – earned value when successor task can start/finish (based on relationship). Applies to task pairs
τ - delay introduced into the successor task due to the technical debt interest on the predecessor task. Applies to task pairs. τ_0 - self-inflicted delay on a task. Applies to each task	T – the time of peak planned value. Applies to each task
	Earliest start time – defines the earliest that a particular task can start development
	Distribution parameters for r, D, τ will be static (e.g., mean and standard deviation for normal distributions)

With the task duration, *D*, expressed as a random variable, it becomes simpler to express the time parameters (*T*, *t*, and τ) as percentages of the task duration, forcing them to have values between zero (0) and one (1). Setting the value of *N* to one (1) treats each task as a single activity. Then, the calculated earned value is the percentage of the planned value and the utility *U* is expressed as a percentage of planned value. This convention allows all the parameters in the Monte Carlo analysis, with the exception of the task duration to be on the same scale, from 0 to 1. It also enables automatic adjustments of the technical debt delay based on the duration of the task. τ is expressed as the percentage of the successor task duration and therefore adjusts with the random selection of the task duration in the Monte Carlo analysis. The actual task duration is found by multiplying the time at which the utility threshold is reached by the duration. This method is shown in the following example.

Williams [173] performed a Monte Carlo analysis for a new airplane development project, including modeling management actions. The tasks, their sequence, and the parameters for the distributions of the task durations are shown in Figure 4-9. This analysis will serve as a test case for the method presented in this article, including updating the analysis to account for technical debt.

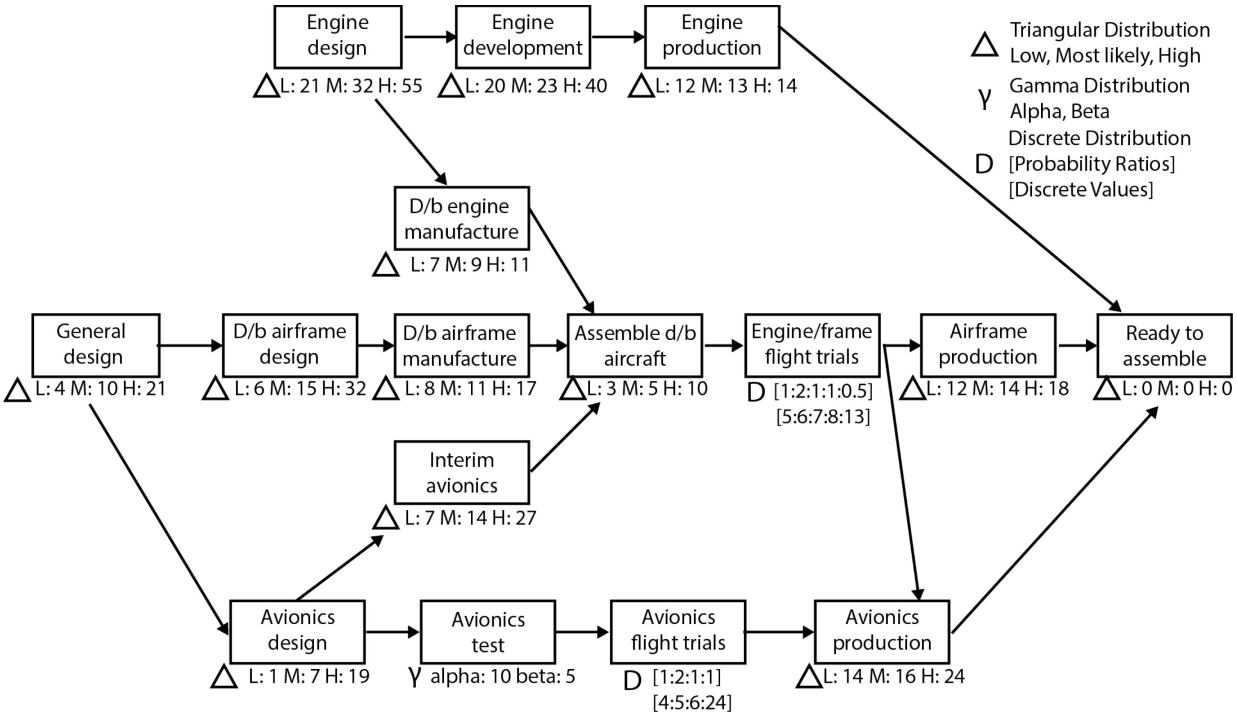

Figure 4-9. Project tasks, durations, and sequence, adapted from [178]

Williams assessed the project duration, found by determining the completion time of the 'Ready to assemble' task, in two cases: the baseline case which only uses the distribution of the durations and a case that represents the application of management actions that cause impacts to subsequent tasks such as "downstream quality issues" [173]. These impacts can be interpreted as technical debt.

Table 4-3 compares the mean duration of the project and the 90% point (the time at which 90% of the Monte Carlo trials show completion of the project) provided by Williams with those calculated using the method presented in this article. The parameters used in this method are also provided for each case. Since planned value curves for each task were not provided by Williams, the value of T used for all tasks was iteratively determined by running the algorithm with different values until results comparable to Williams was achieved. In cases where the planned value curves of each task are known, T would be determined as the point of maximum instantaneous planned

value as defined by Warburton [182]. The high value of τ is used to force t to be less than τ in cases where technical debt is not applied.

Table 4-3. Comparison of results with [173]

Case	Values from [173]	Calculated Values	Parameters Used
Baseline analysis	Mean Duration: 90.5 90% Value Duration: 103	Mean Duration: 90.5 90% Value Duration: 100.5	$T = 0.395$ (all tasks) $U = 1$ (all tasks) $r, r_0 = 0$ (constant value, all tasks) $\tau, \tau_0 = 0$ (constant value, all tasks) $\alpha_0 = 1$ (all tasks) $\alpha = 0$ (all tasks)
Expediting engine design	Not provided	Mean Duration: 88.1 90% Value Duration: 95.7	Same as previous case except: Engine Design Duration: custom triangular distribution where the value is reduced by 1/3 if it exceeds 34, per [173] d/b Engine Manufacture for the dependency on engine design: τ: Uniform Distribution between [0, 0.2] r: Normal Distribution with $\mu = 1, \sigma = 0$ α: 0.5 $\alpha_0 = 0.5$ Engine Development for the dependency on engine design: τ: Uniform Distribution between [0, 0.05] r: Normal Distribution with $\mu = 1, \sigma = 0$ α: 0.5 $\alpha_0 = 0.5$

Case	Values from [173]	Calculated Values	Parameters Used
Increased parallelism	Mean Duration: 87.2 90% Value Duration: 95	Mean Duration: 88.2 90% Value Duration: 95.6	Same as previous case except: Engine Production Add a dependency on engine flight trials with the following parameters: $T = 0.395$ $U = 0.2$ $r = 0$ $\tau = 0$ $\alpha = 1/3$ For the dependency on engine development set: τ: Uniform Distribution between [0, 0.1] r: Normal Distribution with $\mu = 1, \sigma = 0$ α: 1/3 $\alpha_0 = 1/3$

As can be seen in Table 4-3, the new method provides answers that are similar to those provided by Williams. Of note is that a custom distribution for duration had to be applied to account for the management actions associated with expediting the engine development to better map to the method used by Williams. The closeness of the results lends confidence to the baseline algorithms presented in this article.

4.2.1.1.5.1 Implementation

The equations described in the previous sections can be implemented as part of a Monte Carlo schedule analysis. The algorithm requries the user to specify the task duration and technical debt parameters. Static variables, as defined in Table 4-2, have their specific values defined. Random variables, as defined in Table 4-2, have the parameters of their associated probability distributions set. For this algorithm, it is assumed that the sequence of tasks is known. The algorithm is defined as follows:

1. Define the sequence of tasks and establish the predecessor-successor relationships
2. Define the parameters α, T, and U
3. Define the distribution parameters for D, r, and τ

4. For each trial in the Monte Carlo analysis:
    a. Randomly set all values of D, r, and τ using the suppplied distribtuions
    b. For each task;
        i. Determine the earned value at the transition points using (4-6)
        ii. For all predecessors:
            1. Calculate *tu*, the time at which the earned value threshold, U, is reached using (4-15). This value will be between zero and one
            2. Calculate the actual task duration, *td*, by multiplying *tu* times D
            3. Calculate the predecessor end time as predecessor start time plus *tu*
            4. Set the task start time to the maximum predecessor end time
    c. Determine the completion time as the end time of the last task
5. Average the results of the Monte Carlo analysis to produce the results

4.2.1.1.6 Discussion

Using the same example project provided in [173], the impact of technical debt and increased parallelism on the project schedule can be assessed by rerunning the Monte Carlo analysis for conditions assessing both technical debt and increased parallelism. Starting with the baseline analysis case, two different technical debt conditions were run: low technical debt and high technical debt. In the high technical debt case, the technical debt is assumed to affect a higher portion of the successor task and with a larger impact – both r and τ are higher. The distributions used are listed in Table 4-4. The values for alpha were set using the values shown in Figure 4-5. The increased parallelism case sets the values for U to 0.9 for all task dependencies, indicating that a task can start once all of its predecessors have reached at least 90% of their earned value. Figure

4-10 shows the cumulative distribution function for each of the cases. Note that it is possible to calculate durations that are of extreme length due to the probabilistic analysis. Outliers were defined as total project durations above 200 months and these outliers were removed from the results.

Table 4-4. Technical debt and increased parallelism impact on the airplane project

Technical Debt Condition	**Case**	
	No Parallelism (U = 1)	Increased Parallelism (U = 0.9)
No Technical Debt: r: Normal Distribution with $\mu = 0, \sigma = 0$ τ: Normal Distribution with $\mu = 0, \sigma = 0$	Mean duration 90.5 90% duration 101.0	Mean duration 65.2 90% duration 72.1
Low Technical Debt: r: Normal Distribution with $\mu = 0.2, \sigma = 0.05$ τ: Normal Distribution with $\mu = 0.2, \sigma = 0.05$	Mean duration 92.0 90% duration 102.0	Mean duration 68.9 90% duration 76.9
High Technical Debt: r: Normal Distribution with $\mu = 0.5, \sigma = 0.2$ τ: Normal Distribution with $\mu = 0.5, \sigma = 0.2$	Mean duration 108.2 90% duration 123.5	Mean duration 100.7 90% duration 114.5
	No compounding interest	Compounding interest
Low Technical Debt, Compounding Interest: U = 1 for all task R and τ: same as the low technical debt case except for engine design task. All dependencies on engine design have the following distributions r: Normal Distribution with $\mu = 0.5, \sigma = 0.2$ τ: Normal Distribution with $\mu = 0.8, \sigma = 0.1$	Mean duration 104.4 90% duration 123.2	Mean duration 108.0 90% duration 128.4

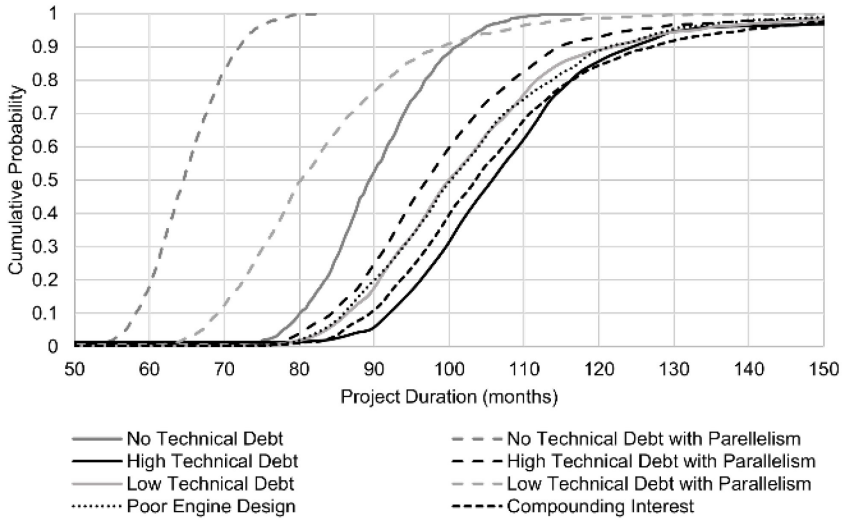

Figure 4-10. Cumulative probabilities of completing the aircraft project under various technical debt and parallelism assumptions

4.2.1.1.6.1 Impact of Increased Parallelism on Project Schedule

The third column and the second through fourth rows in Table 4-4 show the impact of assuming that tasks can start when their predecessors reach at least 90% of their value. Evaluating the start time of successor tasks based on the accumulation of value can significantly decrease the subsequent start time of each task and therefore decrease the overall project duration. Conceptually, this conclusion follows from the evaluation of an earned value curve, such as shown in Figure 4-1, where accumulating the last 10% of the project value can take over 20% of the time. This last 10% of value often does not add value to the successor tasks, and therefore, by starting earlier, the entire project can be accelerated. For example, a software interface between two separate components is typically defined by an interface control document (ICD). To start developing the software interface, it is necessary to have the majority of the ICD complete, but the final version, which may include non-technical aspects such as formatting and obtaining signatures, is not required.

4.2.1.1.6.2 Impact of Technical Debt on Project Schedule

The third and fourth rows in Table 4-4 show the impact of technical debt on the project. In both cases, the mean duration of the project increased when technical debt is assumed to occur on each task. In the 'high technical debt' case increasing the parallelization is not sufficient to bring the schedule back to the original baseline. These results model the impact that technical debt can have on a schedule and highlight one of the deficiencies of traditional Monte Carlo schedule analysis. Every task carries some risk of creating technical debt for its successor tasks, either through design or implementation deficiencies or through a change in the context of the system. Traditional methods add margin for the duration of impacted tasks without actually assessing the downstream impacts. The method presented in this article allows for the project manager to assess both increases in task duration and different levels of impact through setting the distribution and technical debt parameters. By varying these assumptions on individual tasks, the project manager can determine which tasks carry the largest risk associated with technical debt. Evaluating these risks allows a project manager to determine the likelihood that a task moves onto the critical path due to technical debt.

4.2.1.1.6.3 Impact of Compounding Technical Debt Interest

The last row of Table 4-4 shows the impact of compounding technical debt interest. In this scenario, the tasks all demonstrate low technical debt, except for the engine design task. The engine design task is modeled as completing with exceptionally high technical debt, resulting in high values of r and τ. In the second column of Table 4-4, it is assumed that the technical debt interest does not compound, and that the technical debt accrued in the engine design task only affects its direct successor. In the third column of Table 4-4 the technical debt from the engine design task affects all of the possible successors. The results show that compounding the technical debt interest

causes increased delays to the project: a 3.5% increase in mean project duration and a 4.2% increase in the 90% point.

Figure 4-11 shows the average duration and completion time for each task in the low technical debt case with no parallelism, the low technical debt case with a high technical debt engine design and no compounding interest, and the compounding interest case. The task with high technical debt, engine design, does not suffer from significant delays. The technical debt of the engine design impacts the d/b engine manufacture task and the engine development task directly. In the compounding case, additional delays are seen in the engine production, assemble d/b aircraft, and engine/frame trials since the additional dependencies on the technical debt from the engine design are modeled.

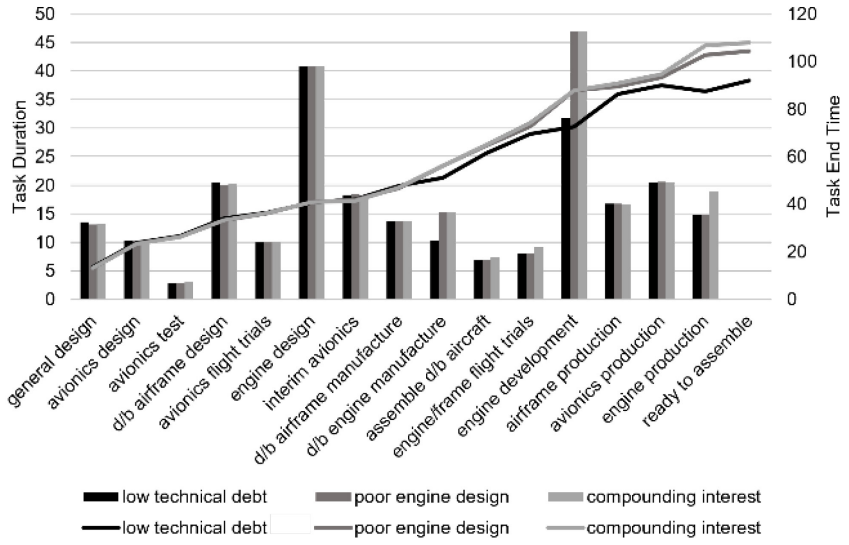

Figure 4-11. Effect of compounding interest on task duration and end time

4.2.1.1.6.4 Quantifying Technical Debt Interest

As defined in (4-6), this method allows for the quantification of technical debt interest. The technical debt interest amount is represented by r and τ and the interest probability is represented through the distribution parameters selected for the Monte Carlo analysis. For each task, the interest amount can be evaluated by assessing the delay in task completion due to the technical

debt of the task predecessors. Using the normalized parameter representation, the task is complete when the earned value reaches a value of one (1). This time, t_c, can be found by calculating *t* using (4-15), with $V = 1$. The interest amount, i_A, is expressed as a percentage of the task duration and is calculated using (4-16):

$$i_A = t_c - 1 \tag{4-16}$$

i_A can be multiplied by the task value to convert it to the value units. If this value is also tracked through the Monte Carlo analysis, then the results of the analysis can be used to predict the expected value of the technical debt interest. Figure 4-12 shows the cumulative probability of the interest amount for the 'engine/frame flight trials' task for the low technical debt with no parallelism, the high technical debt with no parallelism, and the compounding interest cases found in Table 4-4. This task depends on several other tasks with both primary and secondary dependencies, as seen in Figure 4-9.

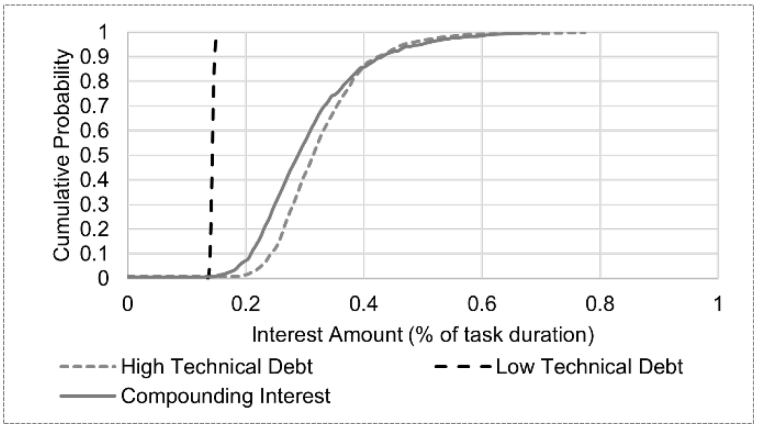

Figure 4-12. Interest amount for 'engine/frame flight trials'

The low technical debt case has a small standard deviation and does not compound the interest; therefore, the predicted interest amount is relatively static. The compounding interest case, which propagates the effects of a single task with large technical debt, incurs close to the same level of interest as the high technical debt case, where all tasks carry technical debt. This result highlights

how technical debt can permeate through the system – a single task can cause cascading delays throughout the rest of the project.

4.2.1.1.6.5 Comparison to Existing Methods

Using the equations and processes defined in this article, it is possible to model increases to the durations of successor tasks based on technical debt introduced in predecessor tasks. This technique is important to schedule analysis as it highlights which tasks need more effective process control methods to prevent the entire project from being delayed.

Compared to existing methods of Monte Carlo schedule analysis, the method presented in this article adds additional capability to evaluate technical debt and its impacts. This method leverages the existing approaches and adjust the duration calculation for each task based on the technical debt parameters. While requiring a larger upfront investment of effort to determine the parameters, the method adds minimal runtime to the analysis, yet produces leading indicators for the project manager.

4.2.1.1.7 Limitations and Future Work

While providing a novel approach to including technical debt contributions in a Monte Carlo schedule analysis, this work is not without its limitations, which can be explored through future efforts. This work assumes that the technical debt parameters remain constant between predecessor and successor task pairs. However, it is likely that the potential impact of technical debt could change based on the state of the predecessor task. This dynamic model could be implemented in future versions of the algorithm. The linearization of the earned value equations introduce error into the analysis, as shown in Appendix A. These equations can be refined and better solutions found to reduce the error. Finally, the major limitation in the work is reliability of the input parameters and estimates. In any schedule analysis, the output is only as good as the original

estimates. The same principle holds with this approach – the overall fidelity of the assessment is based on the accuracy of the input task durations and technical debt parameters. Future work can explore relationships between different task types to established guidelines for the parameters to be used. Additional future work includes verification and validation of the method through application to real project development. These applications will reveal the success of the method in predicting technical debt impacts and the cost-benefit tradeoff of early introduction of technical debt reduction efforts.

4.2.1.1.8 Conclusion

Monte Carlo schedule analysis provides a probabilistic estimate of the duration for completing a project. However, traditional techniques do not consider the impact of the quality of each task on the ability to complete the successor task on schedule. They also tend to assume finish-to-start relationships, which do not accurately represent task sequencing, especially in high level schedules. This article provides a novel method to assess the technical debt of each task and its impact on successors by modeling technical debt contributions and impacts on successor tasks. It also allows for the modeling of relationships where a task starts once its predecessor reaches a specified percentage of its final value. This combination allows for more accurate schedule modeling early in projects based on real world conditions and for the inclusion of technical debt effects. By estimating technical debt impacts on successor tasks, the project manager has the ability to evaluate leading indicators of future delays. Leading indicators provide project managers with time to implement corrective actions, such as increased quality control, while the cost to do so is low. Regularly updating the schedule analysis based on the evaluated technical debt of tasks in progress can identify the risk of delays to future tasks, and therefore the entire project.

Identification of these risks enable project managers to introduce proper mitigation strategies before the risks become issues

4.2.1.1.9 Appendix A: Accuracy Assessment

Given the piecewise nature of the linearization function, it is beneficial to look at the accuracy in each of the four sections. An exhaustive analysis was done examining the linearized earned value functions for values of T, r, and τ for the single predecessor case. All three parameters were varied from 0.1 to 0.9 in steps of 0.1. For all cases, $N = 1$ to enable consistent scaling. The maximum absolute error and the maximum percent error were calculated for each of the four linearization stages for each combination of input parameters. The maximum and average values found are shown in Table 4-5, showing that while the percent errors are large in some cases, the absolute errors are of similar magnitudes for each case. Therefore, the linearization can be considered a valid approximation to the true function.

Table 4-5. Accuracy assessment of earned value linearization

Output Parameter	Input Condition	Stage 1	Stage 2	Stage 3	Stage 4
Maximum absolute error	All	0.057	0.079	0.034	0.579
Maximum percent error	All	318%	141%	33%	554%
Average absolute error	All	0.032	0.022	0.011	0.095
Average percent error	All	107%	11%	3%	25%
Maximum absolute error	$t \leq \tau$	0.052	0.025	0.016	0.042
Maximum percent error	$t \leq \tau$	187%	5%	2%	5%
Average absolute error	$t \leq \tau$	0.028	0.014	0.009	0.021
Average percent error	$t \leq \tau$	98%	4%	2%	2%
Maximum absolute error	$t > \tau$	0.057	0.079	0.034	0.579
Maximum percent error	$t > \tau$	318%	141%	33%	554%
Average absolute error	$t > \tau$	0.049	0.026	0.013	0.098
Average percent error	$t > \tau$	148%	15%	3%	26%

The earned value function itself is piecewise, changing equations when $t = \tau$. Therefore, rows have been added to Table 4-5 showing the results for the cases where $t \leq \tau$ and where $t > \tau$. The largest percent error values are for stage 1. This section of the linearization curve applies when the

calculated earned values are small which can lead to large discrepancies in percent error. The magnitude of the absolute error, while higher that the other sections, is still in the same general range.

Figure 4-13 plots the maximum and average percent errors for each analyzed value of T, r, and τ. From these plots, it can be clearly seen that large values of *r* (center plots) consistently lead to higher error values, while the largest values of the other parameters do not exhibit consistent behavior. Therefore, it can be inferred that the *r* parameter drives the errors when it gets large. The impact of *r* is to shift the earned value plot to the right. Large values decrease the similarity that was assumed when reusing the inflection points from the planned value curve. Figure 4-13 shows that the linearization accuracy is within 10% on average for the final three linearization stages when *T*, *r*, and τ are all less than or equal to 0.5. Note that values of zero in the plot indicate cases that were not realized. For example, high values of *T* did not enter the limited growth phase in the cases tested.

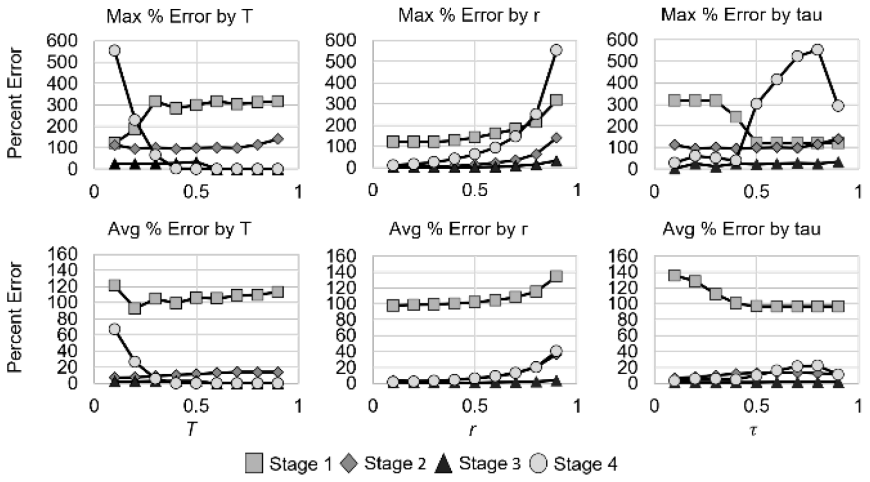

Figure 4-13. Maximum and average percent error of linearization of earned value sliced by T, r, and τ

Although the linearization produces some areas of large percent error, these errors are low in absolute magnitude. Additionally, these errors are likely to be smaller than any errors introduced through the initial estimation of the task duration. Therefore, it can be concluded that the linearization does not cause a significant impact on the overall accuracy of the schedule assessment.

The transition points in the linearization are controlled by the value of T. T changes as the shape of the planned value curve changes and therefore the transition points will change. As seen in the first column of Figure 13, the percent error in the analysis is relatively constant across different values of T, expect for the first and last stage. Therefore, values of T that produce longer first or last stages would produce additional errors.

4.2.1.1.10 Appendix B: Computation Environment

The Monte Carlo analysis in this article was conducted using Python 3.9.7 scripts executed within the Spyder integrated development environment (version 5.1.5). The software was executed on a Dell Vostro 15 7510 computer running 64-bit Windows 11 Pro with a dual 2.30 GHz 11th Gen Intel® Core ™ i7-11800H processor and 16.0 GB of RAM. All cases in this article were executed for 1000 trials and the execution time was between 2.3 and 2.6 seconds

4.2.1.2 Summary of Technical Debt Inclusion in Project Schedule Analysis

The above paper shows how technical debt could be included in a schedule analysis and combined with Monte Carlo methods to generate probabilistic delivery timelines for tasks. In view of the LEAP process, this method provides a mathematical way to develop a probabilistic Development Matrix from the Technology Matrix. The next section defines how the probabilistic Development Matrix is incorporated into the LEAP process.

4.2.2 Quantification of the LEAP process

The mathematics behind the LEAP process [160] relies on the ability to associate the Functional and the Development Matrices through matrix multiplication. When the inputs to the matrices are binary (one (1) or zero (0)), the multiplication of the Functional and Development Matrices provides the number of technologies developed in a time period that support a capability. However, as soon as probabilities are introduced into the Development Matrix, the assumptions underpinning the process break down. The matrix multiplication will sum the probabilities of delivering each technology, which can erroneously produce probabilities of delivering the capabilities of greater than one (1). Therefore, the mathematical process underpinning the LEAP process need to be updated. The following paper, submitted to *IEEE Access* in September, 2023 [186], provides this mathematical update.

4.2.2.1 Probabilistic Enhancement to the Leap Process for Identifying Technical Debt in Iterative System Development [186]

4.2.2.1.1 Abstract

The List, Evaluate, Achieve, Procure (LEAP) process defines a methodology for mathematically associating the delivery of system capabilities with the temporal satisfaction of stakeholder needs while identifying technologies at high risk of imparting technical debt into the system. The original process is qualitative, relying on binary definitions of timelines for technology development – the technology either is or is not developed in a specific time period. The binary definitions allow for rapid high-level assessments of the potential for technical debt. However, they fail to capture more realistic scenarios of uncertain technology development timelines. This paper resolves these issues by introducing probability into LEAP process. This

paper also provides examples of using the probability in the LEAP process and compares the probabilistic (quantitative) and binary (qualitative) models.

4.2.2.1.2 Introduction

Kleinwaks et al. [160] developed the List, Evaluate, Achieve, Procure (LEAP) process to provide a structured approach to identifying technologies that are critical to meeting the stakeholders' needs. This process uses matrix operations to mathematically combine a system functional breakdown with stakeholder needs to identify capabilities that will be delivered late to need and the technologies that drive the delivery timelines. The LEAP process is designed for use within increasingly volatile, uncertain, complex, and ambiguous (VUCA) system development and operating environments [3]. By applying LEAP in an iterative manner, the system developer can identify investments that reduce level of non-recurring engineering (NRE) in system development to enable rapid and successful iterative development cycles [162]. The LEAP process can also be used as a decision support system to assess the long-term impacts of decisions made to achieve a short-term benefit, known as technical debt [19]. Examples of technical debt include minimizing documentation or system modeling and analysis to ensure an on-time release, which can result in increased effort to change the system in the future.

The LEAP process consists of four major steps [160]:

1. List: establish the system definition by decomposing the stakeholder needs into capabilities and perform a functional breakdown of the capabilities into enabling technologies
2. Evaluate: assess the capabilities and technologies to determine the need dates and expected development timelines and compute the ability of the system to meet the needs

3. Achieve: identify the technologies that have the largest contribution to late need satisfaction, either in the current time period or in the future. The system developer can invest in these technologies to reduce the development timeline
4. Procure: include mature technologies within a larger-scale development cycle to develop the system that meets the stakeholders' needs

The ability of the system to meet the temporal needs of the stakeholders is computed using matrix operations. Kleinwaks et al. define the process in detail, including the explanation of the supporting mathematics [160]. The baseline process, referred to in this paper as the qualitative LEAP process, is shown in Figure 4-14.

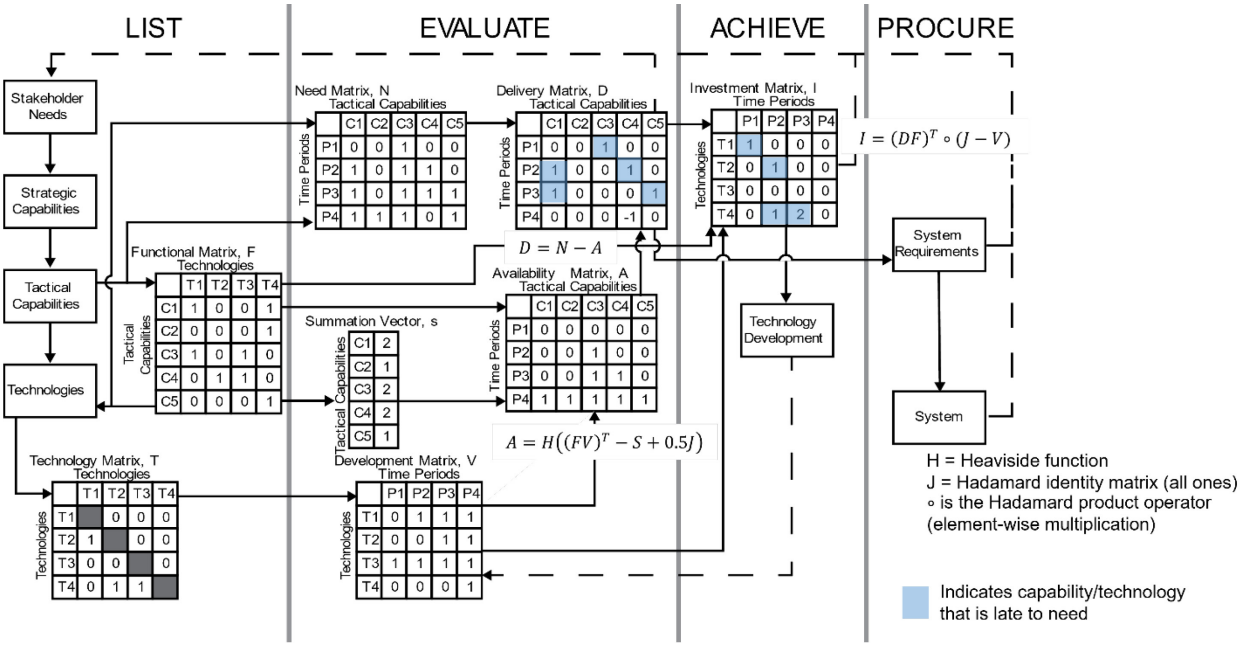

Figure 4-14. The qualitative LEAP process as defined in [160]

The qualitative LEAP process relies on three primary inputs: the Functional Matrix (F), the Development Matrix (V), and the Need Matrix (N). These inputs are uniquely defined for each system of interest, based on an analysis of the stakeholder needs and system requirements. The Functional Matrix defines the functional breakdown of capabilities into supporting technologies. The Development Matrix defines the development timelines for each of the technologies. The

Need Matrix defines the times at which the stakeholders require each capability. In the qualitative LEAP process, the values in each of these matrices are binary: either zero (0) or one (1). In these matrices, one (1) indicates that the rows and columns are connected – the technology supports the capability, the technology will be developed in the time period, or the capability is needed by the stakeholders in the time period.

The binary inputs enable rapid instantiation of the process and support standard matrix multiplication methods. However, the use of binary inputs provides a qualitative assessment of absolute technology development timelines: technologies will or will not be developed in a given time period. Unfortunately, technology development rarely follows well defined timelines. Several methods exist to estimate the duration of a technology development program, such as the critical path method and the program evaluation and review technique (PERT) [187]. Schedule risk analysis combines these methods with Monte Carlo analysis to produce the probability of a technology being developed in a specific time period [188]. To increase the usability of the LEAP process, it needs to be able to input these probabilities into the Development Matrix and to propagate the probabilities through the rest of the analysis. This paper defines updates to the LEAP process and equations to account for probability and to produce the likelihood of delivering capabilities on time to the stakeholders. The updated process is referred to as the quantitative LEAP process.

The rest of this paper is structured in three sections. First, an overview of related work is presented. Next, the quantitative LEAP model is described in detail and an example of its usage and comparison to the qualitative model is provided. Finally, the paper is concluded and recommendations for future work are presented.

4.2.2.1.3 Related Work

Satisfying stakeholder needs is critical to the success of a project. Unfortunately, satisfying these needs often produces schedule and cost pressure on a system developer, resulting in the introduction of technical debt into the system [18]. Increasing the awareness of technical debt upon its introduction to the system is can improve overall project performance [189]. de Almeida et al. connect technical debt prioritization with business processes, and show accounting for business processes affects how technical debt is prioritized [190]. However, they do not provide a generalizable and mathematical approach to link the stakeholder needs and the system development to assist in the prioritization of development.

The LEAP process provides a novel approach for linking the delivery timeline of system capabilities to the times when the stakeholder needs the capability [160]. It can be used in iterative development scenarios or in project planning. An example of its usage for identifying technological investments is provided in [167]. The primary objective of the process is to identify technologies that may exacerbate system development schedules, resulting in the failure to meet stakeholder needs on time. The LEAP process allows technologies that contribute to the late delivery of multiple capabilities to be identified early. Therefore, the LEAP process can provide leading indicators of technical debt and the impact of technical compromises involving these technologies can be assessed. However, the LEAP process in [160] only deals in absolute delivery timelines and needs to be augmented with probabilistic delivery estimates.

Similar work has been performed by other authors investigating the impacts of rework on project schedules. Rework is associated with the repetition of tasks which were not performed to the required quality levels of the project [171] while technical debt is associated with the increased effort required to complete successor tasks [19]. Research on rework includes the connection

between project iterations [176] [191], causes of rework [171] [170], and task and project durations [175] [192]. These overlapping conditions are critical to project success, since a successful project requires acceptable performance in addition to on-time and on-budget delivery [192].

Kim incorporates rework probabilities into a linear programming solution to determine the cost of crashing schedule and the impact on total project duration [192]. Smith and Eppinger identify methods to determine which tasks are contributing the most work in iterative design [191], using off-diagonal rework probabilities [176]. While these methods allow for the successive build-up of downstream impacts, they do not account for increases to a successor task's duration based on the technical compromises made during the execution of the predecessor tasks.

Krishnan, Eppinger, and Whitney analyze the duration of successor tasks based on the overlap with predecessor tasks [175]. They assess that starting a successor task too early may increase the effort and duration of the successor task and may also result in a quality loss of the predecessor activity due to a loss of flexibility in the predecessor task. Their model attempts to determine how many iterations to perform with overlapping tasks. However, in many situations, iterations are not included in a project plan and the model does not provide methods to address the impact on the successor tasks of quality loss in a predecessor task. Maheswari and Varghese [177] address task overlaps but do not quantify the rework duration, identifying the assessment of this duration as a critical area for future work.

Ma et al. recognize that current schedule analysis tools offer only passive management capabilities for rework and that leading indicators of rework potential are required [174]. They identify rework probability, the chance of rework occurring, and rework impact, the impact of each activity, and then apply a learning curve to each iteration to measure its impact. This work is similar in concept to the LEAP method in that it attempts to predict the future impact of rework

on project schedule. However, it focuses on calculating the iterations required within a project and not on the association between delivery timelines and the satisfaction of stakeholder needs.

The methods and techniques identified in this review focus on the technology delivery aspects of a project – estimating when the project will be complete. While they provide quantitative estimates, they do not directly connect the technology delivery timelines to the need dates of the project stakeholders. The original LEAP process performs this association, but is restricted to qualitative estimates. Therefore, enhancing the LEAP process by adding probabilistic methods will provide a quantitative method to mathematically associate the likelihood of capability delivery with the temporal satisfaction of stakeholder needs.

4.2.2.1.4 Including Probabilities in the LEAP Model

The updates to the LEAP model presented in this section focus on including probabilities in the Development Matrix. The Development Matrix defines the timelines on which the individual technologies are developed [160]. Switching the representation of this matrix from binary values (one (1) and zero (0)) to probabilities enables a more realistic modeling of technology development.

4.2.2.1.4.1 Matrix Multiplication with Probabilities

The original LEAP model [160] uses matrix operations to identify relationships and to compute the availability of capabilities. The Availability Matrix (A), which defines whether or not a capability will be available in a specified time period, is computed by first multiplying the Functional Matrix (F) and Development Matrix (V), which gives a matrix containing the number of developed technologies that support each capability in each time period. The total number of technologies that support the capability (S) is subtracted from the product to determine if the capability is complete. Finally, the Heaviside function (H) is used to restrict the output values to

be between zero (0) and one (1). The Availability Matrix calculation is shown in (4-17) and a complete explanation of the supporting mathematics can be found in [160].

$$A = H((FV)^T - S + 0.5J) \quad (4\text{-}17)$$

The critical concept in the Availability Matrix calculation is the combination of the Functional and Development Matrices through matrix multiplication. The dot product of the row of one matrix and the column of the other is used to determine the count of technologies that are developed (column of the Development Matrix) that support the capability (row of the Functional Matrix). The dot product adds the products of each of the corresponding elements of the row and column vectors.

If the Development Matrix includes probabilities instead of binary values, then (4-17) is no longer valid. Assuming that the development of each technology is independent, then the probability of developing a capability *c* in a specific time period *p* is the product of the probabilities of developing each supporting technology *t* in that same time period *p*, as depicted in (4-18).

$$P(C_p) = \prod_i P(t_{i,p:t_i\ supports\ c}) \quad (4\text{-}18)$$

The matrix multiplication FV produces a summation of independent probabilities and not the product, as shown in (4-19). Additionally, (4-19) includes all the cells of each row of the Functional Matrix in the computation. This inclusion creates a problem when $F[i,j] = 0$. When adding the products of each cell, a zero (0) value in F simply eliminates the corresponding value of V from the sum. However, when multiplying the products of corresponding cells by applying (4-18), a zero (0) value in F results in a zero (0) product. In the definition of the Functional Matrix, a zero (0) equates to a technology that does not support the capability, and therefore the V value should be eliminated from the product instead of the reducing the product to zero.

$$FV = \begin{bmatrix} \sum_i^n F[0,i] * V[i,0] & \cdots & \sum_i^n F[0,i] * V[i,p] \\ \vdots & \ddots & \vdots \\ \sum_i^n F[m,i] * V[i,0] & \cdots & \sum_i^n F[m,i] * V[i,p] \end{bmatrix} \quad (4\text{-}19)$$

Based on these observations, standard matrix operations do not meet the requirements for updating the LEAP process to include probabilities in the Development Matrix. The required function must input two vectors of the same size and compute the product of the products of corresponding elements, if, and only if, the element of one vector is non-zero.

Two separate functions are required: one that selects the elements of a vector and one that produces the multiplication of the elements in the matrix. These functions are defined in the following sections.

4.2.2.1.4.1.1 Selecting Elements of a Vector: the k Function

A new function, called the k function, is defined in (4-20) to select and replace non-zero input values. It inputs three values *x*, *y*, and *z*. If *x* is not zero (0), then the k function outputs *y*. If *x* is zero (0), then the k function outputs *z*. The function provides a simple method to select a value based on another input. Equation 4-21 extends the k function to apply to vectors and (4-22) extends it to matrices. A capital K is used to denote the matrix version of the equation. Note that in (4-21) vectors $\vec{u}$ and $\vec{v}$ must be the same length and in (4-22) matrices U and V must have the same dimensions.

$$k(x,y,z) = \begin{cases} y, x \neq 0 \\ z, x = 0 \end{cases} \quad (4\text{-}20)$$

$$\vec{k}(\vec{u},\vec{v},z) = [k(\vec{u}[0],\vec{v}[0],z) \quad \ldots \quad k(\vec{u}[n],\vec{v}[n],z)] \quad (4\text{-}21)$$

$$K(U,V,z) = \begin{bmatrix} k(U[0,0],V[0,0],z) & \cdots & k(U[0,n],V[0,n],z) \\ \vdots & \ddots & \vdots \\ k(U[m,0],V[m,0],z) & \cdots & k(U[m,n],V[m,n],z) \end{bmatrix} \quad (4\text{-}22)$$

4.2.2.1.4.1.2 Multiplying Matrices: the k* Function

The k function provides the first step of the required multiplication process – the elimination of the zero (0) terms from one of the vectors. A second function is required to address the multiplication of the elements in two matrices instead of the summation. The k* function is defined in (4-23). For two vectors, it computes the product of the application of the k function to the corresponding elements of the vectors. Equation 4-24 shows the matrix version of the k* function, denoted with a capital K.

$$k^*(\vec{u}, \vec{v}, z) = \prod_i^n k(\vec{u}[i], \vec{v}[i], z) \qquad (4\text{-}23)$$

$$K^*(U, V, z) = \begin{bmatrix} k^*(F[0,:], V[:,0], z) & \cdots & k^*(F[0,:], V[:,p], z) \\ \vdots & \ddots & \vdots \\ k^*(F[m,:], V[:,0], z) & \cdots & k^*(F[m,:], V[:,p], z) \end{bmatrix} \qquad (4\text{-}24)$$

The K* function has the requirements necessary to combine the Functional and Development matrices when the Development Matrix contains probabilities: it eliminates zero values in the Functional Matrix from the product and also multiplies the elements of the matrices instead of adding them.

4.2.2.1.4.1.3 Application of the K* function

The application of the K* function is shown in Figure 4-15. In the figure, the Development Matrix contains the probabilities of completing each technology in each time period. If this matrix is multiplied by the Functional Matrix F using standard matrix multiplication, the result is the third matrix, FV, located in the lower left of the figure. The red cells indicate results where the probability of delivering the capability in a time period are greater than one. The application of the K* function results in the matrix on the lower right of Figure 4-15. The same cells are highlighted

in red, however, they now have the actual probability values for delivering the capability in the specified time periods.

	P1	P2	P3
T1	0.5	0.75	1
T2	0	0.5	0.75
T3	1	1	1

Development Matrix, V

	T1	T2	T3
C1	1	0	0
C2	1	1	0
C3	0	1	1

Functional Matrix, F

	P1	P2	P3
C1	0.5	0.75	1
C2	0.5	1.25	1.75
C3	1	1.5	1.75

FV

	P1	P2	P3
C1	0.5	0.75	1
C2	0	0.38	0.75
C3	0	0.5	0.75

K*(F,V,1)

Figure 4-15. Application of the K function and comparison with matrix multiplication*

The first row of the final matrix is unchanged between the standard matrix multiplication and the K*. This row represents the availability of capability C1, which, as seen in the Functional Matrix, only depends on one technology (T1). Therefore, the matrix multiplication and the application of the K* function produce the same results. The yellow cells in the final matrix changed their values to zero (0). This result is due to the multiplication of probabilities instead of the summation of probabilities. Both capability C2 and capability C3 depend on technology T2. Technology T2 has a zero probability of being developed in time period P1. Therefore, when the probabilities are multiplied, there is a zero probability of completing C2 and C3 in time period P1. The traditional matrix math summed the probabilities of the supporting technologies, thereby producing incorrect results.

4.2.2.1.4.2 Including Probabilities in the LEAP Process

Having demonstrated the usage of the K* function to combine probabilities in matrix multiplication, the LEAP equations presented in [160] can be updated to account for the

probabilistic Development Matrix. As a result of these updates the Availability and Delivery Matrices, which are the outputs of the Evaluation phase of the LEAP process, will both produce probabilistic values for capability availability and delivery. The probabilistic Delivery Matrix defines the likelihood of meeting the stakeholders' needs on time and therefore becomes a decision aid for the system developer.

4.2.2.1.4.2.1 Availability Matrix

The Availability Matrix determines if a capability will be available in a specific time period [160]. The K* function computes this probability when applied to the Functional and Development Matrices. Therefore, calculating the probabilistic Availability Matrix requires using the K* function as shown in (4-25).

$$A = (K^*(F, V, 1))^T \qquad (4\text{-}25)$$

Within the K* function, z is set to one (1) such that a zero (0) value in the Functional Matrix translates to one (1) in the multiplication instead of the value in the Development Matrix. This choice effectively eliminates the zero entry in the Functional Matrix from the probability computation. This behavior is desired since that capability is not dependent on the technology. The transpose of the K* function is taken to produce an Availability Matrix with the same dimensions as the Need Matrix in [160].

4.2.2.1.4.2.2 Delivery Matrix

While the Availability Matrix specifies when capabilities are available, the Delivery Matrix (D) defines whether or not the capabilities are delivered in time to meet the stakeholders' needs. In the qualitative LEAP process, the Delivery Matrix is calculated by subtracting the Availability Matrix from the Need Matrix [160]. Applying the same calculation here would result in the Delivery

Matrix specifying the probability of *not* delivering the capability on time. Logically, it makes more sense to have the Delivery Matrix indicate the probability of delivering on time instead. Since the Availability Matrix contains probability values, it is necessary to distinguish between capabilities that have zero probability of being delivered on time and those that are not needed in a time period. Therefore, the Delivery Matrix is calculated using the K function on the Need and Availability Matrices as shown in (4-26). The z value in the K function is set to negative one (-1) to identify the time periods where a capability is not needed.

$$D = K(N,\ A, -1) \qquad (4\text{-}26)$$

The values in the Delivery Matrix take on different meaning than those in the qualitative LEAP process. A value that is greater than or equal to zero (0) indicates the probability of delivering a capability in the time period. A negative value indicates that the capability is not needed in that time period.

4.2.2.1.4.2.3 Investment Matrix

The final matrix produced by the LEAP process is the Investment Matrix (I). The Investment Matrix identifies those technologies that have the greatest contributions to the late satisfaction of stakeholder needs. In the qualitative LEAP formulation, the values in the Investment Matrix represent the number of late capabilities contributed to by each technology [160]. Using the probabilistic formulation of the Development Matrix, the values in the Investment Matrix become a score – the higher the value, the larger the impact of the technology. The updated Investment Matrix equation is shown in (4-27).

$$I = (NF)^T \circ (J - V) \qquad (4\text{-}27)$$

The Need and the Functional Matrices both contain binary values, so standard matrix multiplication is used. This product gives the number of needed capabilities affected by a specific technology. J is the Hadamard identity matrix, which is a matrix of all ones (1) [166]. Subtracting the Development Matrix, V, from J, produces a matrix of probabilities of *not* delivering technologies. The two resulting matrices are then combined element-wise using the Hadamard product (∘) [166], producing an Investment Matrix where each value is the number of affected capabilities times the probability of late delivery.

Larger scores in the Investment Matrix represent a greater contribution of that technology to the late delivery of the system in the specified time period. The score is the number of affected late capabilities times the probability of late delivery of the technology. Table 4-6 shows examples of Investment Matrix scores including the number of impacted capabilities and the probability of late delivery.

Table 4-6. Examples of investment matrix scores

	Number of Late Capabilities Impacted	**Probability of Late Delivery**	**Score**
Formula	$(NF)^T$	$(J - V)$	$(NF)^T \circ (J - V)$
Technology 1	1	0.1	0.1
Technology 2	4	0.1	0.4
Technology 3	1	0.7	0.7
Technology 4	4	0.7	2.8

From these examples, it can be clearly seen that the score provides additional insight into the importance of a technology. Larger scores indicate a larger potential return-on-investment (ROI) if the likelihood of delivering the technology on time can be increased. Consider a situation where a choice is made to invest in either Technology 2 or Technology 3. The qualitative LEAP model would imply that Technology 2 provides the bigger ROI as it impacts more capabilities than Technology 3. The quantitative LEAP process, on the other hand, indicates that Technology 3

provides the bigger ROI. Although it only affects one capability, it has a much higher likelihood of delivering late and therefore a correspondingly larger score.

4.2.2.1.4.2.4 Adjustments for Dependent Technologies

The above process relies on an assumption of independence between the technologies. In situations where technologies depend upon each other, the model defined above will incorrectly calculate the probabilities. This restriction is remedied by redefining the Functional Matrix. The Functional Matrix maps the capabilities to the supporting technologies. When the technologies are independent, then all technologies should be included in each row of the Functional Matrix. However, if technologies are dependent upon each other, then only the latest technology should be included in the row in the Functional Matrix. For example, consider the Functional Matrix in Figure 4-15. If Technology 2 is dependent upon Technology 1, then the Functional Matrix would be rewritten as shown in Figure 4-16, with Capability 2 only showing Technology 2 as a supporting technology. The highlighted cell indicates the change in the matrix. With this redefinition of the Functional Matrix, the capabilities are still composed of independent technologies and the rest of the analysis process is valid.

	T1	T2	T3
C1	1	0	0
C2	0	1	0
C3	0	1	1

Functional Matrix, F

Figure 4-16. Functional Matrix accounting for technology dependencies

4.2.2.1.4.3 Example Application of the Quantitative LEAP Process

The updates to the LEAP process are best understood through an example application. Kleinwaks et al. [167] applied the qualitative LEAP process to the development of optical terminals at the Space Development Agency. As an example of the quantitative LEAP process,

this work is modified to use notional probabilistic values in the Development Matrix. The left side of Figure 4-17 shows the initial qualitative Development Matrix from [167] after notional investments were made to increase the likelihood of meeting the stakeholder capabilities. To estimate the probabilities, a normal distribution was applied to each technology, with a mean set to the first time period identified in [167] minus two years and the standard deviation set to two years. This distribution was chosen to give an 84% probability of delivering at the times identified in the qualitative analysis, which is based on the expert opinions used to establish the delivery timelines in [167]. The probability of delivering each technology in each time period is computed from the distribution. The resulting probabilistic Development Matrix is shown on the right side of Figure 4-17, where the colors go from red (low probability of delivering) to green (high probability of delivering).

	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032
T1				1	1	1	1	1	1	1	1	1
T15	1	1	1	1	1	1	1	1	1	1	1	1
T2		1	1	1	1	1	1	1	1	1	1	1
T4			1	1	1	1	1	1	1	1	1	1
T6				1	1	1	1	1	1	1	1	1
T3				1	1	1	1	1	1	1	1	1
T8										1	1	1
T10					1	1	1	1	1	1	1	1
T11		1	1	1	1	1	1	1	1	1	1	1
T12		1	1	1	1	1	1	1	1	1	1	1
T7												1
T13	1	1	1	1	1	1	1	1	1	1	1	1
T5		1	1	1	1	1	1	1	1	1	1	1
T9			1	1	1	1	1	1	1	1	1	1
T14						1	1	1	1	1	1	1
T22		1	1	1	1	1	1	1	1	1	1	1
T21		1	1	1	1	1	1	1	1	1	1	1
T16		1	1	1	1	1	1	1	1	1	1	1
T17			1	1	1	1	1	1	1	1	1	1
T18		1	1	1	1	1	1	1	1	1	1	1
T19			1	1	1	1	1	1	1	1	1	1
T20					1	1	1	1	1	1	1	1
T23		1	1	1	1	1	1	1	1	1	1	1
T24				1	1	1	1	1	1	1	1	1
T25		1	1	1	1	1	1	1	1	1	1	1
T26						1	1	1	1	1	1	1

	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032
T1	0.31	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00
T15	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T2	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T4	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
T6	0.31	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00
T3	0.31	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00
T8	0.00	0.00	0.01	0.02	0.07	0.16	0.31	0.50	0.69	0.84	0.93	0.98
T10	0.16	0.31	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00
T11	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T12	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T7	0.00	0.00	0.00	0.00	0.01	0.02	0.07	0.16	0.31	0.50	0.69	0.84
T13	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T5	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T9	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
T14	0.07	0.16	0.31	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00
T22	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T21	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T16	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T17	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
T18	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T19	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
T20	0.16	0.31	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00
T23	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T24	0.31	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00
T25	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
T26	0.07	0.16	0.31	0.50	0.69	0.84	0.93	0.98	0.99	1.00	1.00	1.00

Figure 4-17. Qualitative (left) and quantitative (right) Development Matrices, based on [167]

The probabilistic Development Matrix is used in the quantitative LEAP process to determine the likelihood of delivering the capabilities in each time period. Figure 4-18 shows the Delivery

Matrix from [167] on the left and the probabilistic Delivery Matrix on the right. The qualitative LEAP Delivery Matrix uses zero (0) to indicate that the capability is either on time or not needed in a specific time period and one (1) to indicate that the capability is late. In Figure 4-18, late capabilities are highlighted in red in the qualitative Delivery Matrix. The quantitative LEAP Delivery Matrix gives the probability of the capability being ready in a time period it is needed, or a negative one (-1) if the capability is not needed. In the quantitative Delivery Matrix in Figure 4-18, the color scale goes from low likelihood of delivering a needed capability (red) to a high likelihood of delivering the needed capability (green). White cells indicate when the capability is not needed.

	C1	C2	C3	C4	C5	C6	C7
2021	0	0	0	0	0	0	0
2022	-1	0	0	0	0	0	0
2023	0	0	0	0	0	0	0
2024	0	0	0	0	0	0	0
2025	0	0	0	0	-1	0	0
2026	0	0	0	-1	-1	0	0
2027	0	0	0	-1	-1	0	0
2028	0	0	0	0	-1	0	1
2029	0	0	0	0	-1	0	1
2030	0	0	0	0	-1	0	1
2031	0	0	0	0	-1	0	1
2032	0	0	0	0	-1	0	0

	C1	C2	C3	C4	C5	C6	C7
2021	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000
2022	-1.000	-1.000	-1.000	-1.000	0.501	-1.000	-1.000
2023	0.724	0.417	-1.000	-1.000	0.758	-1.000	-1.000
2024	0.901	0.732	-1.000	-1.000	0.912	0.507	-1.000
2025	0.973	0.912	0.790	-1.000	-1.000	0.775	-1.000
2026	0.994	0.978	0.919	-1.000	-1.000	0.922	-1.000
2027	0.999	0.996	0.975	-1.000	-1.000	0.979	-1.000
2028	1.000	0.999	0.993	0.976	-1.000	0.996	0.153
2029	1.000	1.000	0.999	0.994	-1.000	0.999	0.306
2030	1.000	1.000	1.000	0.999	-1.000	1.000	0.499
2031	1.000	1.000	1.000	1.000	-1.000	1.000	0.691
2032	1.000	1.000	1.000	1.000	-1.000	1.000	0.841

Figure 4-18. Qualitative (left) and quantitative (right) Delivery Matrices, based on [167]

In the qualitative Delivery Matrix, capability C7 is late marked as late to need in 2028 (a red 1). However, in the quantitative matrix, the probability of delivering C7 in 2028 is 0.153. While this probability is small, there is still a chance of delivering the capability on time. There is a greater than 50% chance that C7 is delivered in 2031, while the qualitative matrix says that it still will not be ready. Other capabilities, such as C5, may be late (a 50% chance of delivering in 2022) at their first needed time period, a factor which is missed in the qualitative LEAP matrix. The

movement away from the binary nature of the qualitative LEAP model increases the fidelity and the realism of the Delivery Matrix. The quantitative model clearly distinguishes between time periods where the capability is delivered on time and when it is not required. For example, the quantitative model identifies that capability C1 is required in 2023 and that there is a 72% probability of it being delivered on time. The qualitative model shows a zero (0) in the entry for C1 in 2023, which is interpreted as either delivering on time or not being needed. The increased fidelity of the model makes the quantitative LEAP model more effective in predicting outcomes for the stakeholders.

The final calculation in the LEAP model is to determine the Investment Matrix, which highlights which technologies are contributing to the late delivery of capabilities in each time period. In the qualitative model, shown on the left of Figure 4-19, technologies T8 and T7 are identified as each contributing to a late capability starting in 2028 (shown as red boxes). The quantitative model, shown on the right side of Figure 4-19, shows the investment 'score' for each of the technologies in each time period with low values in green and high values in red.

	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032
T1	0	0	0	0	0	0	0	0	0	0	0	0
T15	0	0	0	0	0	0	0	0	0	0	0	0
T2	0	0	0	0	0	0	0	0	0	0	0	0
T4	0	0	0	0	0	0	0	0	0	0	0	0
T6	0	0	0	0	0	0	0	0	0	0	0	0
T3	0	0	0	0	0	0	0	0	0	0	0	0
T8	0	0	0	0	0	0	0	1	1	0	0	0
T10	0	0	0	0	0	0	0	0	0	0	0	0
T11	0	0	0	0	0	0	0	0	0	0	0	0
T12	0	0	0	0	0	0	0	0	0	0	0	0
T7	0	0	0	0	0	0	0	1	1	1	1	0
T13	0	0	0	0	0	0	0	0	0	0	0	0
T5	0	0	0	0	0	0	0	0	0	0	0	0
T9	0	0	0	0	0	0	0	0	0	0	0	0
T14	0	0	0	0	0	0	0	0	0	0	0	0
T22	0	0	0	0	0	0	0	0	0	0	0	0
T21	0	0	0	0	0	0	0	0	0	0	0	0
T16	0	0	0	0	0	0	0	0	0	0	0	0
T17	0	0	0	0	0	0	0	0	0	0	0	0
T18	0	0	0	0	0	0	0	0	0	0	0	0
T19	0	0	0	0	0	0	0	0	0	0	0	0
T20	0	0	0	0	0	0	0	0	0	0	0	0
T23	0	0	0	0	0	0	0	0	0	0	0	0
T24	0	0	0	0	0	0	0	0	0	0	0	0
T25	0	0	0	0	0	0	0	0	0	0	0	0
T26	0	0	0	0	0	0	0	0	0	0	0	0

	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032
T1	0.000	0.000	0.000	0.159	0.067	0.023	0.006	0.003	0.000	0.000	0.000	0.000
T15	0.000	0.000	0.317	0.134	0.068	0.019	0.004	0.001	0.000	0.000	0.000	0.000
T2	0.000	0.309	0.159	0.134	0.023	0.006	0.001	0.000	0.000	0.000	0.000	0.000
T4	0.000	0.000	0.000	0.067	0.023	0.006	0.001	0.000	0.000	0.000	0.000	0.000
T6	0.000	0.000	0.000	0.159	0.067	0.023	0.006	0.003	0.000	0.000	0.000	0.000
T3	0.000	0.000	0.000	0.159	0.067	0.023	0.006	0.003	0.000	0.000	0.000	0.000
T8	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.500	0.309	0.159	0.067	0.023
T10	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.006	0.001	0.000	0.000	0.000
T11	0.000	0.000	0.317	0.134	0.068	0.019	0.004	0.001	0.000	0.000	0.000	0.000
T12	0.000	0.309	0.476	0.267	0.091	0.025	0.005	0.001	0.000	0.000	0.000	0.000
T7	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.841	0.691	0.500	0.309	0.159
T13	0.000	0.000	0.317	0.134	0.068	0.019	0.004	0.001	0.000	0.000	0.000	0.000
T5	0.000	0.309	0.317	0.200	0.068	0.019	0.004	0.001	0.000	0.000	0.000	0.000
T9	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
T14	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.023	0.006	0.001	0.000	0.000
T22	0.000	0.000	0.317	0.134	0.068	0.019	0.004	0.001	0.000	0.000	0.000	0.000
T21	0.000	0.000	0.317	0.134	0.068	0.019	0.004	0.001	0.000	0.000	0.000	0.000
T16	0.000	0.000	0.159	0.067	0.046	0.012	0.003	0.000	0.000	0.000	0.000	0.000
T17	0.000	0.000	0.159	0.067	0.046	0.012	0.003	0.000	0.000	0.000	0.000	0.000
T18	0.000	0.000	0.159	0.067	0.023	0.006	0.001	0.000	0.000	0.000	0.000	0.000
T19	0.000	0.000	0.159	0.067	0.023	0.006	0.001	0.000	0.000	0.000	0.000	0.000
T20	0.000	0.000	0.000	0.000	0.159	0.067	0.023	0.006	0.001	0.000	0.000	0.000
T23	0.000	0.309	0.159	0.134	0.023	0.006	0.001	0.000	0.000	0.000	0.000	0.000
T24	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000
T25	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
T26	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.023	0.006	0.001	0.000	0.000

Figure 4-19. Qualitative (left) and quantitative (right) Investment Matrices, based on [167]

The values in the quantitative Investment Matrix take on a slightly different meaning from those in the qualitative model. In the qualitative model, the Investment Matrix values indicate the count of the late capabilities that depend on the technology [160]. In the quantitative model, the value is a score that represents how important the technology is in driving late capability deliveries in the time period. For example, consider technologies T8 and T7 in 2028. In the qualitative model, they both have the same value (1) in the Investment Matrix since they each contribute to the late delivery of a single capability. In the quantitative model, the value for T8 is 0.5 and the value for T7 is 0.841 in 2028. The higher score for T7 indicates a higher potential ROI if the probability of delivering the technology on time could be increased. The cost to increase the delivery probability would need to be accounted for in any ROI calculation, but that is out of the scope of this paper.

4.2.2.1.4.4 Accounting for Technical Debt in the Quantitative LEAP Process

Including probabilities in the LEAP process enhances its ability to identify the technologies that can be potential sources of technical debt within a system development. Kleinwaks, Batchelor, and

Bradley [19] use the technical debt metaphor to reflect the long-term system impacts of short-term decisions. The LEAP framework enables a system developer to rapidly assess the potential for long-term impacts of short-term decisions that impact the development of critical technologies and capabilities. For example, system developers are often faced with choices on the sequencing of technology development due to cost, schedule, and performance limitations. Often, a particular technology is delayed because it is viewed as less valuable, even though it may be necessary for later development tasks.

In the LEAP model, technologies with the high potential for technical debt manifest themselves in the Investment Matrix. Higher scores in the Investment Matrix indicate increased dependencies on on-time delivery and therefore the potential for impacts due to the presence of technical debt. With the probabilistic nature of the quantitative LEAP model, these relationships become clearer as the potential late delivery of a technology can be assessed, including its cascading impacts on the delivery of capabilities.

4.2.2.1.5 Conclusions and Future Work

Including probabilities within the LEAP framework enables a more realistic assessment of the ability of the system to deliver in time to meet the stakeholders' needs. This research updates the LEAP process defined in [160] to account for a probabilistic Development Matrix and to propagate those probabilities through the system. This update is critical to better align the LEAP process to real-world systems. Real systems do not guarantee system delivery in a specific time period and the ability to estimate the likelihood of delivery allows for higher fidelity modeling.

The user of the quantitative LEAP process can more accurately assess the potential for achieving technology development by determining the increase in the likelihood of delivering a capability. An achievement initiative may speed up technology development, but does not

guarantee that the technology will be achieved in a specific time frame. The quantitative LEAP process enables the modeling of this change in probability, instead of an assumption of complete success. This change in probability can also be mapped more directly to an implementation cost, enabling a calculation of the ROI for these decisions. The score in the Investment Matrix provides a more refined estimate of the impact of a technology's late delivery, highlighting the potential for higher ROI.

These updates to the process represent the second step in defining a full process for accounting for technical debt within system development planning, as defined in [160]. Similar to the qualitative model, the quantitative LEAP framework highlights technologies with the potential for introducing technical debt into the system. Combining the quantitative LEAP framework with a scheduling model that accounts for technical debt will highlight the downstream impacts of the technical debt on the delivery of system capabilities. By modeling the impact of the technical debt of one technology on its successor technologies in the development cycle, the probabilities of delivering each technology in a defined time period can be estimated. These estimates, when included in the quantitative LEAP process, will provide insight to system stakeholders to enable proper investment decisions to limit the risk of late deliveries. Further verification and validation of the process includes implementing the quantitative LEAP process on additional real-world systems to identify insights provided by the process to assist users in delivering capabilities on time.

4.2.3 Summary of LEAP Quantitative Updates

With the updates to the mathematical processes defined in this section and the ability to determine a probabilistic development matrix discussed earlier, the LEAP process is ready to use within a release planning context, as described in the following section.

4.3 Including Proactive Technical Debt Assessments in Release Planning

Several authors [28] [91] [94] have discussed the inclusion of technical debt in release planning. However, the focus of most of their work has been on deciding when to repay the technical debt and when to let the technical debt remain in the system. For example, Schmid [94] provides a quantitative assessment of the value of repaying technical debt compared to the risk of leaving it in the system. This assessment can be used to assist in deciding when to repay technical debt by performing a value trade. While accounting for technical debt repayment is required to maintain a healthy system, it is a reactive mechanism for dealing with technical debt.

A proactive mechanism for considering technical debt in release planning would identify the potential long-term impacts due to the inclusion or exclusion of features in each release of the system. Running these tools in conjunction with a traditional release planning algorithm would enable the incorporation of technical debt forecasts and risk assessments into the planning cycle.

In release planning activities, the features are selected and prioritized based on their value, either to the stakeholders, the developers, or a combination of both [93]. Traditional release planning methods do not have a method for assessing the potential technical debt introduced into the system due to the selected order of feature development. Taking the selected development order and running it through the LEAP process provides an assessment of the potential for technical debt. The following section describes how to use the LEAP process as a decision support system for release planning.

4.3.1 LEAP as a Decision Support System for Release Planning

The complete LEAP process involves accounting for technical debt to create probabilistic technology development timelines and then using those probabilities to assess the ability to deliver capabilities that meet the stakeholders' temporal needs. The process identifies sources of outside

investments to achieve earlier technology development. The final LEAP phase, the Procure phase, involves the execution of a system iteration or release to develop and deliver a set of capability. The accomplishments of that delivery are then fed back into the need and capability decomposition in an iterative manner to inform future releases.

The LEAP process is a decision support system that can be used to support release planning while accounting for technical debt. The technology development timelines created in the Evaluate phase represent the release plan – the order in which technologies and features will be developed. The process then evaluates this plan to determine if capabilities are delivered on time to the stakeholders. The Investment Matrix indicates which technologies are likely to produce the largest return on investment if their development timelines are changed.

By explicitly and mathematically tying the development of technologies to the temporal satisfaction of stakeholder needs, the LEAP process becomes a tool for evaluating release plans and project schedules. It enables an assessment of the specified release plan and its projection into the future. The LEAP process accounts for technical debt by identifying the impacts of technical concessions based on the temporal ability to satisfy stakeholder needs.

4.3.1.1 Example of LEAP as a Decision Support System in Incremental Development

While the LEAP process was originally designed for use within iterative system development, it can also be used within a single iteration to identify critical technologies and the impacts of technical debt. Williams [173] presents a schedule for the development of an aircraft which was used as the baseline for the evaluation of technical debt impacts using Monte Carlo schedule analysis techniques [169]. The next sections demonstrate how the LEAP process can be used with this example project.

In the LEAP process, the first step is the List step, which identifies the stakeholder needs and decomposes them into tactical capabilities and their component technologies. To demonstrate the use of the quantitative LEAP mode, the aircraft development project is assumed to be an incremental development project with the following increments:

1. Design completion;
2. Manufacturing of d/b (development batch, or test) components;
3. Assembly of d/b aircraft;
4. Flight trials; and,
5. Production of components leading to the ready to assemble state.

Therefore, the overall stakeholder need – the assembly of the aircraft, can be decomposed into a capability representing each increment in the above list. These capabilities are further decomposed into the technologies identified as the tasks in the schedule in [173]. The interdependencies of the technologies are identified through the schedule as well. Figure 4-20 shows the relationship between the tasks, color coded by increments. The numbers represent the identifier for each of the tasks.

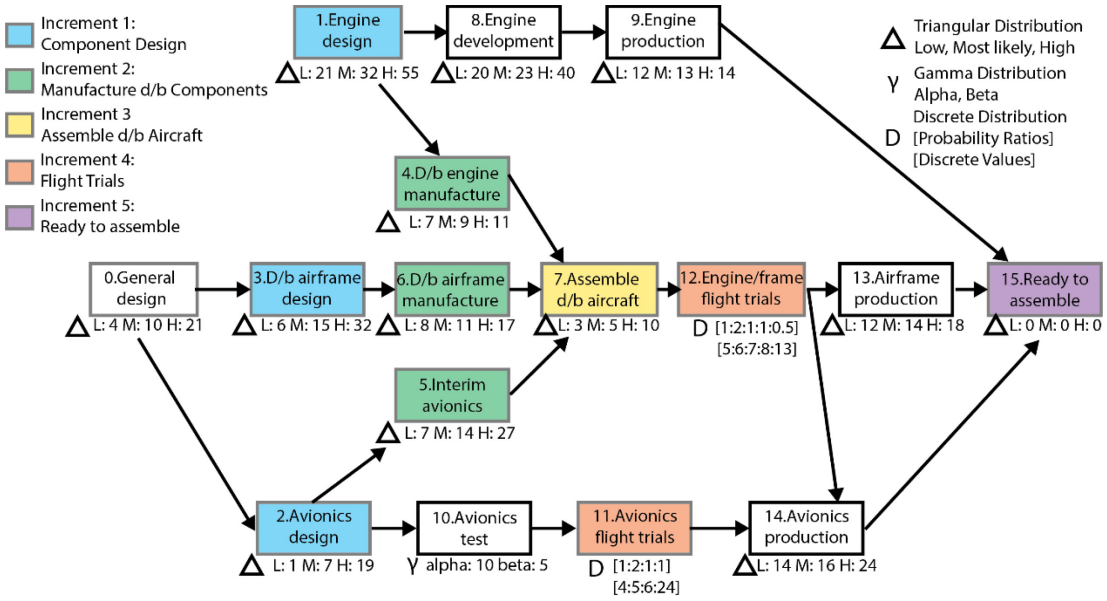

Figure 4-20. Aircraft example task relationships to increments

The resulting Technology Matrix is shown in Figure 4-21 and the resulting Functional Matrix is shown in Figure 4-22, which completes the List phase of the LEAP process. Note that the project represented in this analysis consists of dependent technologies, and therefore the mapping of capability to technology only includes the last technologies in the dependency tree, as identified in the Technology Matrix. This restriction is required to enable proper calculation of probabilities [186]. For the technical debt analysis, the T, α, r, τ, and U parameters must also be set during this phase. α, T, and U are set corresponding to the compounding interest case in [169]. r and τ will be varied to demonstrate the impacts of technical debt on the ability to deliver the system on time.

Technology Matrix

ID	Technology	0 General Design	1 Engine Design	2 Avionics Design	3 d/b airframe design	4 d/b engine manufacture	5 interim avionics	6 d/b airframe manufacture	7 assemble d/b aircraft	8 engine development	9 engine production	10 avionics test	11 avionics flight trials	12 engine/frame flight trials	13 airframe production	14 avionics production	15 ready to assemble
0	General Design																
1	Engine Design																
2	Avionics Design	1															
3	d/b airframe design	1															
4	d/b engine manufacture		1														
5	interim avionics	1		1													
6	d/b airframe manufacture	1			1												
7	assemble d/b aircraft	1	1	1	1	1	1	1									
8	engine development		1														
9	engine production		1							1							
10	avionics test	1		1													
11	avionics flight trials	1		1								1					
12	engine/frame flight trials	1	1	1	1	1	1	1	1								
13	airframe production	1	1	1	1	1	1	1	1					1			
14	avionics production	1	1	1	1	1	1	1	1			1	1	1			
15	ready to assemble	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Figure 4-21. Technology Matrix for aircraft example. Based on [173]

Functional Matrix

		Technologies															
		T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
Capabilities	C0: Design Completion	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	C1: Manufacture d/b Components	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
	C2: Assemble d/b Aircraft	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	C3: Flight Trials	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
	C4:Ready to Assemble	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 4-22. Functional Matrix for aircraft example

The next phase in the LEAP process is the Evaluate phase, where the timelines required for capability delivery and the development timelines of the technologies are calculated and then combined into the Delivery Matrix [160]. For this analysis, it is assumed that the stakeholders were presented with the "no technical debt" schedule from [169], where the mean duration is 90.5 months and the 90% point (the time at which there is a 90% probability that the system will be complete) is 101 months. It is further assumed that the stakeholders have agreed to adjust their need dates to accept delivery no later than 101 months after project start. Therefore, the need date for capability 4, having the aircraft ready to assemble, is set to 102 months after project start,

allowing for delivery throughout the 101st month. Using the maximum durations for predecessor tasks from [173], the need dates for the other capabilities can be set as shown at the top of Figure 4-23. These dates result in the Need Matrix shown in Figure 4-23, where the time periods are listed as months after program start.

	Capabilities				
	C0	C1	C2	C3	C4
Need Date (months from project start)	43	54	64	77	102

Need Matrix

Time Period (months from project start)	Capabilities						Capabilities					
	C0	C1	C2	C3	C4		C0	C1	C2	C3	C4
35	0	0	0	0	0	81	1	1	1	1	0
36	0	0	0	0	0	82	1	1	1	1	0
37	0	0	0	0	0	83	1	1	1	1	0
38	0	0	0	0	0	84	1	1	1	1	0
39	0	0	0	0	0	85	1	1	1	1	0
40	0	0	0	0	0	86	1	1	1	1	0
41	0	0	0	0	0	87	1	1	1	1	0
42	0	0	0	0	0	88	1	1	1	1	0
43	1	0	0	0	0	89	1	1	1	1	0
44	1	0	0	0	0	90	1	1	1	1	0
45	1	0	0	0	0	91	1	1	1	1	0
46	1	0	0	0	0	92	1	1	1	1	0
47	1	0	0	0	0	93	1	1	1	1	0
48	1	0	0	0	0	94	1	1	1	1	0
49	1	0	0	0	0	95	1	1	1	1	0
50	1	0	0	0	0	96	1	1	1	1	0
51	1	0	0	0	0	97	1	1	1	1	0
52	1	0	0	0	0	98	1	1	1	1	0
53	1	0	0	0	0	99	1	1	1	1	0
54	1	1	0	0	0	100	1	1	1	1	0
55	1	1	0	0	0	101	1	1	1	1	0
56	1	1	0	0	0	102	1	1	1	1	1
57	1	1	0	0	0	103	1	1	1	1	1
58	1	1	0	0	0	104	1	1	1	1	1
59	1	1	0	0	0	105	1	1	1	1	1
60	1	1	0	0	0	106	1	1	1	1	1
61	1	1	0	0	0	107	1	1	1	1	1
62	1	1	0	0	0	108	1	1	1	1	1
63	1	1	0	0	0	109	1	1	1	1	1
64	1	1	1	0	0	110	1	1	1	1	1
65	1	1	1	0	0	111	1	1	1	1	1
66	1	1	1	0	0	112	1	1	1	1	1
67	1	1	1	0	0	113	1	1	1	1	1
68	1	1	1	0	0	114	1	1	1	1	1
69	1	1	1	0	0	115	1	1	1	1	1
70	1	1	1	0	0	116	1	1	1	1	1
71	1	1	1	0	0	117	1	1	1	1	1
72	1	1	1	0	0	118	1	1	1	1	1
73	1	1	1	0	0	119	1	1	1	1	1
74	1	1	1	0	0	120	1	1	1	1	1
75	1	1	1	0	0	121	1	1	1	1	1
76	1	1	1	0	0	122	1	1	1	1	1
77	1	1	1	1	0	123	1	1	1	1	1
78	1	1	1	1	0	124	1	1	1	1	1
79	1	1	1	1	0	125	1	1	1	1	1
80	1	1	1	1	0						

Figure 4-23. Need Matrix for aircraft example.

Using the schedule analysis techniques defined in [169], a Monte Carlo analysis was run to produce the estimated development timelines for each technology. Figure 4-24 shows the Development Matrix created as a result of the schedule analysis.

Development Matrix

		Time Period																															
		35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66
Technologies	T0	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T1	0.27	0.33	0.38	0.43	0.48	0.52	0.56	0.59	0.63	0.66	0.70	0.74	0.78	0.80	0.82	0.85	0.88	0.91	0.93	0.94	0.96	0.97	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T2	0.97	0.98	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T3	0.60	0.65	0.70	0.74	0.77	0.81	0.85	0.87	0.90	0.93	0.95	0.96	0.97	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T4	0.01	0.02	0.03	0.04	0.05	0.08	0.10	0.14	0.17	0.21	0.26	0.32	0.37	0.42	0.46	0.50	0.54	0.59	0.62	0.65	0.70	0.73	0.76	0.79	0.82	0.85	0.88	0.90	0.91	0.94	0.96	0.97
	T5	0.20	0.25	0.31	0.37	0.40	0.45	0.49	0.54	0.60	0.66	0.70	0.74	0.77	0.81	0.85	0.87	0.89	0.92	0.93	0.95	0.96	0.97	0.98	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
	T6	0.06	0.09	0.11	0.14	0.18	0.21	0.26	0.30	0.33	0.37	0.42	0.47	0.52	0.57	0.62	0.65	0.70	0.74	0.78	0.82	0.86	0.89	0.91	0.93	0.95	0.96	0.97	0.98	0.99	0.99	1.00	1.00
	T7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.04	0.06	0.08	0.11	0.14	0.18	0.22	0.26	0.32	0.38	0.43	0.49	0.55	0.60	0.66	0.70	0.74	0.78
	T8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.03	0.05	0.06	0.07	0.10	0.13	0.16	0.19	0.22	0.26	0.30
	T9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	T10	0.92	0.95	0.96	0.97	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T11	0.56	0.58	0.61	0.65	0.67	0.70	0.73	0.74	0.76	0.77	0.78	0.79	0.80	0.82	0.83	0.84	0.86	0.87	0.89	0.90	0.92	0.94	0.95	0.96	0.98	0.98	0.99	0.99	0.99	1.00	1.00	1.00
	T12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.04	0.05	0.07	0.09	0.12	0.16	0.20	0.24	0.28	0.32	0.38
	T13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	T14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	T15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

		Time Period																															
		67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98
Technologies	T0	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T4	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T7	0.81	0.85	0.88	0.91	0.93	0.95	0.97	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T8	0.34	0.38	0.43	0.46	0.51	0.54	0.58	0.61	0.65	0.69	0.73	0.76	0.78	0.81	0.83	0.86	0.87	0.88	0.90	0.92	0.94	0.95	0.95	0.96	0.97	0.98	0.98	0.99	0.99	0.99	1.00	1.00
	T9	0.00	0.01	0.02	0.03	0.04	0.06	0.06	0.09	0.11	0.14	0.17	0.20	0.23	0.28	0.33	0.36	0.40	0.44	0.48	0.52	0.55	0.59	0.63	0.67	0.70	0.74	0.77	0.79	0.81	0.85	0.86	0.87
	T10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T11	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T12	0.43	0.50	0.55	0.60	0.64	0.69	0.73	0.77	0.80	0.84	0.88	0.90	0.92	0.95	0.96	0.98	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T13	0.00	0.01	0.01	0.02	0.03	0.03	0.04	0.06	0.08	0.10	0.14	0.17	0.22	0.27	0.31	0.36	0.42	0.47	0.52	0.58	0.62	0.67	0.72	0.75	0.79	0.83	0.86	0.88	0.91	0.93	0.95	0.97
	T14	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.03	0.04	0.05	0.08	0.09	0.11	0.15	0.19	0.23	0.27	0.33	0.36	0.41	0.46	0.51	0.57	0.62	0.66	0.72	0.76	0.79	0.82	0.86	0.88
	T15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.03	0.04	0.05	0.07	0.09	0.12	0.17	0.21	0.26	0.29	0.33	0.38	0.43	0.49	0.54	0.58	0.63	0.67	0.71	0.76	0.79	0.81

		Time Period																										
		99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
Technologies	T0	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T8	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T9	0.89	0.91	0.93	0.94	0.95	0.96	0.96	0.97	0.98	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T11	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T12	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T13	0.98	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T14	0.90	0.93	0.94	0.96	0.97	0.98	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	T15	0.83	0.87	0.89	0.91	0.93	0.94	0.96	0.97	0.98	0.98	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 4-24. Development Matrix for aircraft example

This probabilistic matrix is used as the input to the quantitative LEAP process defined in [186] to produce the Delivery Matrix shown in Figure 4-25. If need satisfaction is assessed as the probability of completing capability 4 when needed, then there is 91% chance of delivering the system on time at month 102 without the presence of technical debt. This result is shown highlighted in the red box in Figure 4-25. While the overall project has a relatively high probability of delivering on time, several of the increments, represented by the capabilities, are likely to be late to need. Capability 0, the component design, only has a 57% change of completing by the

stated need date and capability 1, the manufacturing of the test articles, only has a 51% chance of being complete. However, these delays are compensated for by the overall schedule margin.

Delivery Matrix

Time Period (months from project start)	Capabilities				
	C0	C1	C2	C3	C4
35	-1	-1	-1	-1	-1
36	-1	-1	-1	-1	-1
37	-1	-1	-1	-1	-1
38	-1	-1	-1	-1	-1
39	-1	-1	-1	-1	-1
40	-1	-1	-1	-1	-1
41	-1	-1	-1	-1	-1
42	-1	-1	-1	-1	-1
43	0.57	-1	-1	-1	-1
44	0.61	-1	-1	-1	-1
45	0.66	-1	-1	-1	-1
46	0.71	-1	-1	-1	-1
47	0.75	-1	-1	-1	-1
48	0.78	-1	-1	-1	-1
49	0.81	-1	-1	-1	-1
50	0.84	-1	-1	-1	-1
51	0.87	-1	-1	-1	-1
52	0.90	-1	-1	-1	-1
53	0.93	-1	-1	-1	-1
54	0.94	0.51	-1	-1	-1
55	0.96	0.57	-1	-1	-1
56	0.97	0.63	-1	-1	-1
57	0.97	0.67	-1	-1	-1
58	0.988	0.72	-1	-1	-1
59	0.99	0.77	-1	-1	-1
60	0.99	0.81	-1	-1	-1
61	1	0.85	-1	-1	-1
62	1	0.87	-1	-1	-1
63	1	0.90	-1	-1	-1
64	1	0.93	0.70	-1	-1
65	1	0.95	0.74	-1	-1
66	1	0.97	0.78	-1	-1
67	1	0.98	0.81	-1	-1
68	1	0.99	0.85	-1	-1
69	1	0.99	0.88	-1	-1
70	1	0.99	0.91	-1	-1
71	1	1	0.93	-1	-1
72	1	1	0.95	-1	-1
73	1	1	0.97	-1	-1
74	1	1	0.98	-1	-1
75	1	1	0.99	-1	-1
76	1	1	0.99	-1	-1
77	1	1	1	0.88	-1
78	1	1	1	0.90	-1
79	1	1	1	0.92	-1
80	1	1	1	0.95	-1

Time Period (months from project start)	Capabilities				
	C0	C1	C2	C3	C4
81	1	1	1	0.96	-1
82	1	1	1	0.98	-1
83	1	1	1	0.99	-1
84	1	1	1	0.99	-1
85	1	1	1	0.999	-1
86	1	1	1	1	-1
87	1	1	1	1	-1
88	1	1	1	1	-1
89	1	1	1	1	-1
90	1	1	1	1	-1
91	1	1	1	1	-1
92	1	1	1	1	-1
93	1	1	1	1	-1
94	1	1	1	1	-1
95	1	1	1	1	-1
96	1	1	1	1	-1
97	1	1	1	1	-1
98	1	1	1	1	-1
99	1	1	1	1	-1
100	1	1	1	1	-1
101	1	1	1	1	-1
102	1	1	1	1	0.91
103	1	1	1	1	0.93
104	1	1	1	1	0.94
105	1	1	1	1	0.96
106	1	1	1	1	0.97
107	1	1	1	1	0.98
108	1	1	1	1	0.98
109	1	1	1	1	0.99
110	1	1	1	1	0.99
111	1	1	1	1	0.99
112	1	1	1	1	1
113	1	1	1	1	1
114	1	1	1	1	1
115	1	1	1	1	1
116	1	1	1	1	1
117	1	1	1	1	1
118	1	1	1	1	1
119	1	1	1	1	1
120	1	1	1	1	1
121	1	1	1	1	1
122	1	1	1	1	1
123	1	1	1	1	1
124	1	1	1	1	1
125	1	1	1	1	1

Figure 4-25. Delivery Matrix for aircraft example

The Investment Matrix shows the technologies that have direct impact on the late delivery of capabilities [160]. The calculated Investment Matrix for the aircraft example is shown in Figure 4-26. The larger values in the Investment Matrix indicate which technologies would most benefit from investments to increase the speed of delivery. In this instance, technologies 1, 4, and 7 have the largest values in the Investment Matrix, indicating that they have the largest overall impact. These technologies drive the development of their respective capabilities, which can be determined by examining the Functional Matrix. In this case, however, there is not likely a large return on

investment available by investing in these technologies, as the overall project has a 91% chance of delivering on time.

Investment Matrix

Technologies	Time Period 35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66
T0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.37	0.34	0.30	0.26	0.22	0.20	0.18	0.15	0.12	0.09	0.07	0.06	0.04	0.03	0.02	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.07	0.05	0.04	0.03	0.02	0.02	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.35	0.30	0.27	0.24	0.21	0.18	0.15	0.12	0.10	0.09	0.06	0.04	0.03
T5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.04	0.03	0.02	0.02	0.01	0.01	0.01	0.00	0.00	0.00	0.00	0.00
T6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.18	0.14	0.11	0.09	0.07	0.05	0.04	0.03	0.02	0.01	0.01	0.01	0.00
T7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.30	0.26	0.22
T8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Technologies	Time Period 67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98
T0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T4	0.02	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T7	0.19	0.15	0.12	0.09	0.07	0.05	0.03	0.02	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.10	0.08	0.05	0.04	0.02	0.02	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Technologies	Time Period 99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
T0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T15	0.00	0.00	0.00	0.09	0.07	0.06	0.04	0.03	0.03	0.02	0.02	0.01	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Figure 4-26. Investment Matrix for aircraft example

In this analysis, the component designs, represented as Capability 0, are only 57% likely to be delivered at the need date specified by the stakeholders (month 43). The Investment matrix shows that it is primarily Technology 1, the engine design, that is driving this late delivery. During project execution, stakeholders may push for an earlier release of the engine design with the perception that the entire aircraft project will be late. Since schedule pressure is a primary cause of technical debt [18], it is reasonable to assume that this pressure could create technical debt in the engine

design, which could then percolate through the system. Technical debt could easily be added to other technologies within the system in similar manners.

To isolate the impacts of technical debt on a particular technology, variations in r and τ were input for each technology. Using the above example, and analysis was performed iterating through values of r and τ from 0.1 to 1 in steps of 0.1 for each technology while leaving the other tasks unchanged. The technical debt was assumed to compound with time and r and τ were assumed to have the same value for each successor technology. For example, if Technology 1 creates the technical debt, r and τ for Technology 1 are left unchanged, but r and τ for its successor tasks (Technologies 4, 7-9, and 12-15) are all set to the same values, which are varied through the analysis. After setting the parameters for each run, the Development Matrix is created through a Monte Carlo analysis. The Development Matrix is then input to the quantitative LEAP process to determine the probability of satisfying the stakeholder needs. Figure 4-27 summarizes this process.

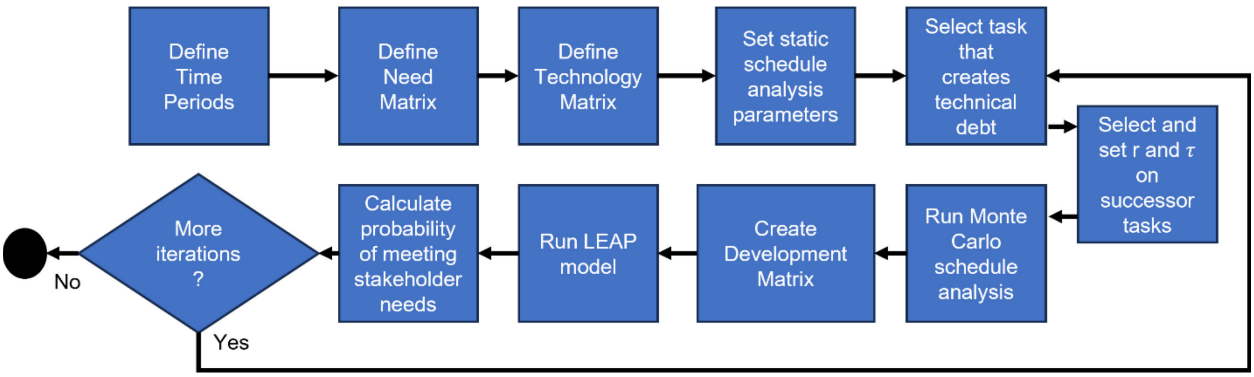

Figure 4-27. Technical debt analysis process for individual tasks with the LEAP process

Figure 4-28 shows the results of the analysis. In this figure, the probability of overall satisfaction, defined as delivering Capability 4 on time, is plotted against the id number of each case, where each case represents a different combination of r and τ values. The color coding indicates which technology was modeled as having technical debt in each case. Lower values on the y-axis indicate combinations of r and τ values for a particular technology that create a reduced

likelihood of delivering the entire project on time, represented as P(satisfaction). In the figure, it can be seen that Technologies 0 (blue circles), 1 (navy squares), 2 (orange exes), 3 (purple triangles), and 8 (green exes) have multiple combinations of r and τ values which produce substantially lower probabilities of delivering the system on time. Therefore, technical debt incurred on these technologies have a greater chance of longer-term impacts on the system development.

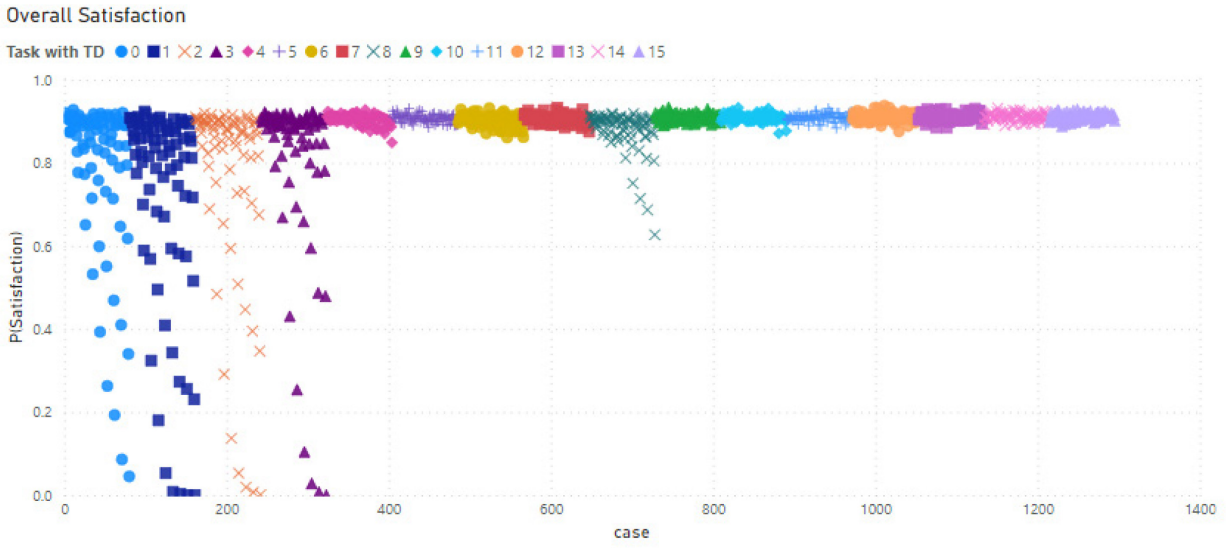

Figure 4-28. Overall probability of meeting stakeholders' needs for the aircraft example

Figure 4-29 shows the probability of delivering the remaining four capabilities by their stated need dates for each case that was run. It can be seen that Technology 0, the general design, has a large potential impact on the ability to satisfy each capability if it incurs technical debt. Technical debt on Technologies 1, 2, and 3 also has substantial impacts on the delivery of Capabilities 1, 2, and 3. These tasks are the initial design tasks, and therefore their outsized impact on the delivery of system capabilities is not unexpected. However, the impact of Technology 0 was not highlighted in the initial Investment Matrix as a potential source of late delivery. It is only when technical debt is assessed that the impacts of the general design are revealed. Therefore, it can be concluded that

it is critical to minimize the technical debt incurred by the general design phase, which occurs early in the lifecycle.

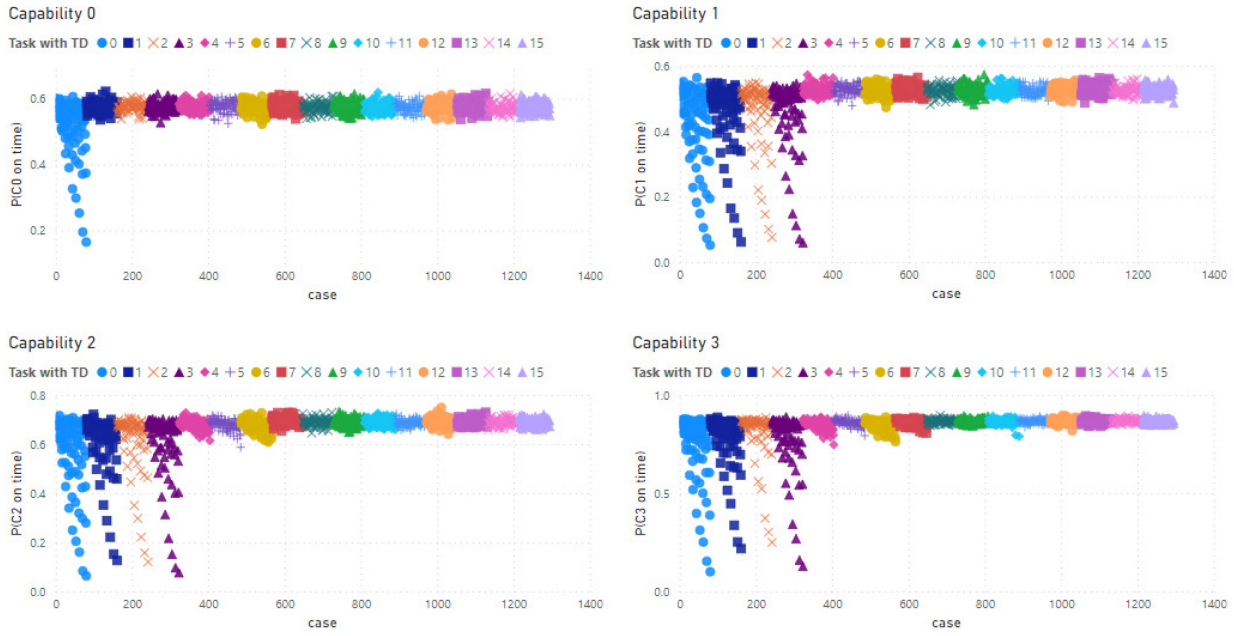

Figure 4-29. Satisfaction of each capability in the aircraft example

The initial Investment Matrix (Figure 4-26) highlighted that Technology 1 had a large impact on the late delivery of Capability 0. However, this is due to the duration estimates of Technology 1 and not to its technical debt. The technical debt incurred on Technology 1 does not delay Capability 0, but it will impact the delivery timelines of the completed aircraft. The reason for this impact can be explored in more detail. Figure 4-30 shows the average percent increase in task duration due to technical debt on specified predecessor tasks. The top plot shows the increase due to technical debt created by task 1 and the bottom plot shows technical debt created by task 11, which is independent from task 1 and occurs in a later increment. Technical debt from task 1 increases the duration of subsequent tasks by up to 14%. Even tasks that are several steps away, such as task 12, experience duration increases of over 6%. Task 11, on the other hand, does not have a significant impact on future tasks, increasing durations by less than 2%, which is within the

error bounds of the Monte Carlo analysis. Note that the decrease in task 10 duration is also less than 2% and therefore within the same error bounds of the Monte Carlo analysis.

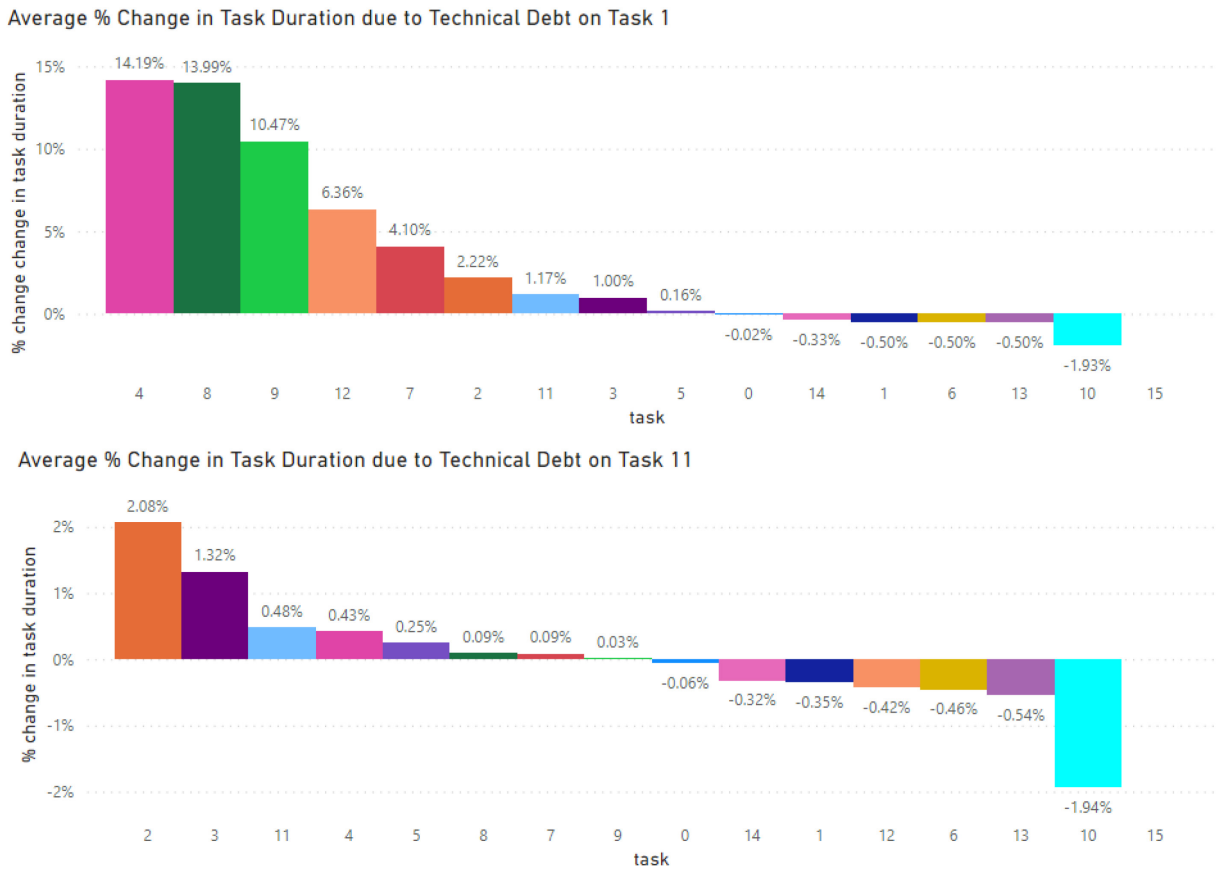

Figure 4-30. Average percent increase in task duration due to technical debt on tasks 1 and 11

Figure 4-31 examines the direct impact on development timelines of the technologies and delivery timelines of the capabilities due to the creation of technical debt by Technology 1. The top plot shows the average change in the development probability (the value in the Development Matrix) for each of the successors of Technology 1 across all the technical debt cases examined. The dashed line represents the overall probability of delivering the project on time. The technical debt from Technology 1 significantly decreases the probability of developing Technology 9 (engine production) on time. In these cases, the probability of delivering the entire project doesn't

reach the 90% threshold until month 114, twelve months later than the baseline case without technical debt.

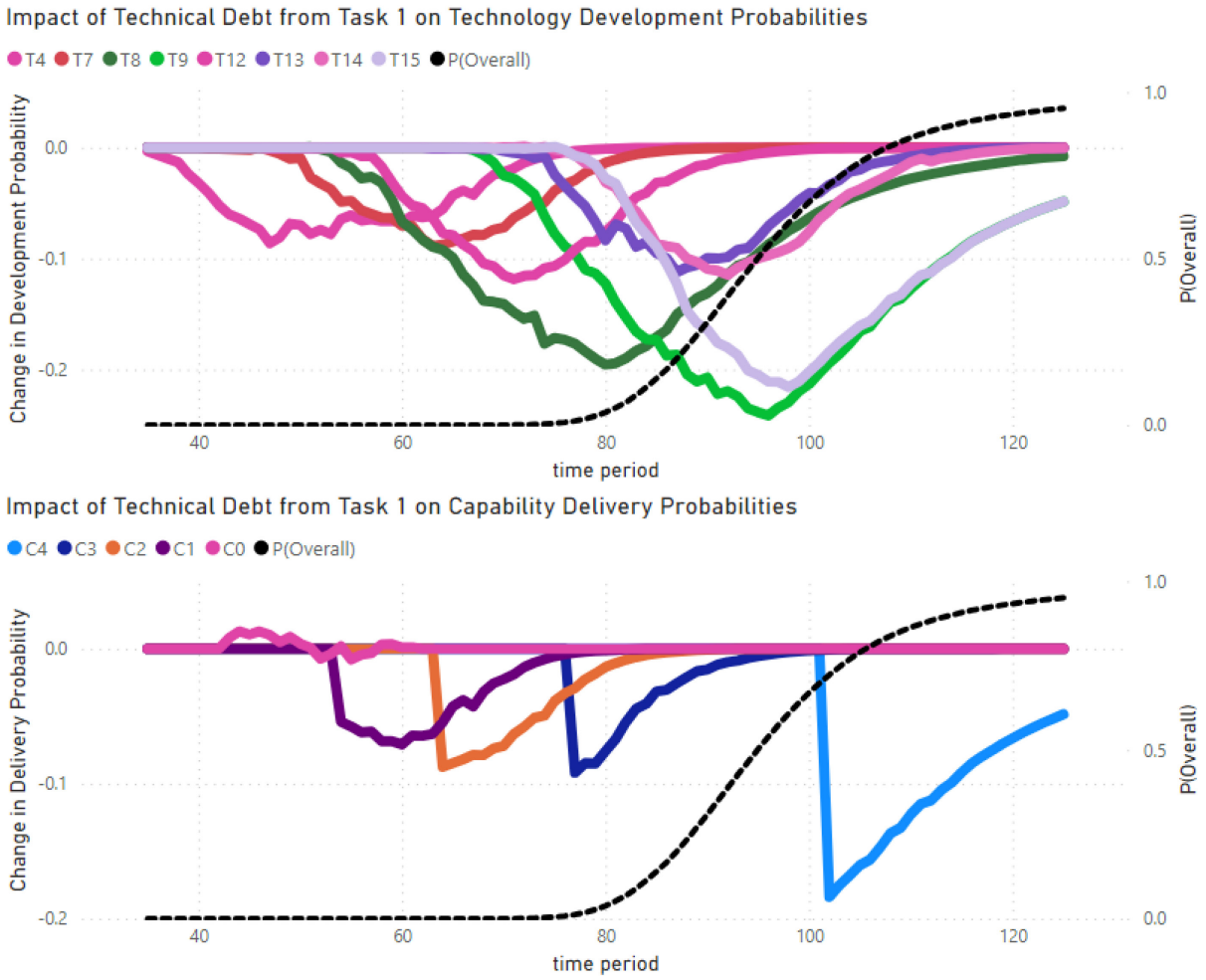

Figure 4-31. Impact of technical debt from task 1 on technology development and capability delivery

The bottom plot of Figure 4-31 shows the average change in the delivery probability for the capabilities based on the technical debt induced by Technology 1. The sharp changes in the plot show the change at the specified need date for each of the capabilities, where probabilities significantly reduce for all capabilities except Capability 0. Capability 0 depends directly on Technology 1 but on none of its successors, and therefore the changes are due to the expected variations within the Monte Carlo analysis process.

Figure 4-31 averages the impacts of technical debt induced by Technology 1 across all the different variations of r and τ used in the analysis. However, understanding the separate contributions of r and τ can yield insight into how the technical debt impacts can be minimized: is it more important to limit the proportion of the successor task subject to the technical debt (lower values of r) or to minimize the delays associated with the technical debt (lower values of τ)?

Figure 4-32 shows the overall probability of meeting the stakeholder needs (completing the project on time) as a function of the changing values of r and τ for Technology 1. In this figure, it is clearly seen that τ has a greater individual impact than r. If τ is kept at or below 0.5, then the probability of meeting the need stays above 80%, regardless of the value of r. These results show that it is more important to limit the amount of delay caused by the technical debt compared to how much of the successor task is affected by the technical debt – in this example, small amounts of technical debt in more places cause smaller impacts than large amounts of technical debt in one place.

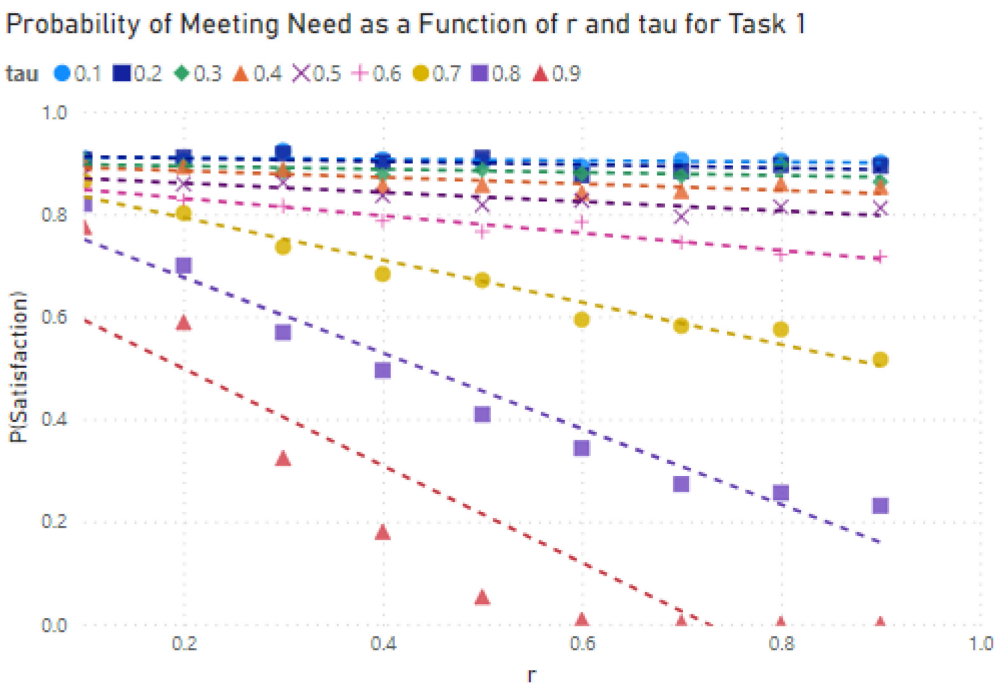

Figure 4-32. Probability of meeting stakeholders' needs based on r and τ for technology 1

This analysis shows the ability of the LEAP process to identify the technologies that can cause program delays based on their technical debt parameters. Using the Investment Matrix, the key late technologies are identified, which can then be traced through the Technology Matrix to identify responsible predecessors. The analysis shows that the downstream impacts of technical debt, modeled as increased durations for successor tasks, can be responsible for delays in the delivery of capabilities. For example, technical debt associated with Technology 1 is more likely to prevent the on-time delivery than technical debt associated with Technology 9, the last task in the engine development sequence. This conclusion reinforces the understanding that early errors have a greater change to propagate through the system. Additional cases could be run combining technical debt parameters for each task to evaluate additional scenarios.

4.4 Conclusion

This chapter addresses RQ3: *How can technical debt be used as a guide in release planning?* In the release planning process, it is critical to select features that not only deliver value to the stakeholders but that also do not incur significant technical debt. Within fixed-time iterations, it is easy for system developers and stakeholders to focus on the delivery of value to the detriment of the future health of the system. Proactive technical debt assessment can guide release planning by identifying the features within the system that are most likely to cause future problems if not addressed early. Within the release planning cycle, these features must be prioritized and also additional resources must be allocated to support additional evaluation and control of the development of these features. The occurrence of technical debt on a task does not necessarily impact that specific task's development timelines, but could impact the overall ability to deliver the system on time. Therefore, proactive identification of tasks with large impacts on the rest of the system enables their development to be prioritized and the controls to be put in place to limit the potential for technical debt. For example, if a release is behind schedule due to two tasks, the system developer can assess which task has a higher potential downstream impact and divert the resources to ensure that task completes successfully. The developer can also assess and prioritize the repayment of technical debt in the next release given that they are aware that it has been introduced into the system.

The quantitative LEAP process presented in this chapter can be used as a probabilistic decision support system to investigate the impacts of design choices on the ability to deliver in accordance with the stakeholder needs. By modeling technical debt through earned value analysis, the technical debt impacts associated with each potential decision can be quantified. Using these development timelines within the LEAP process enables a system developer to identify which

stakeholder needs and system capabilities are likely to be affected based on the decisions made during release planning. The system developer can identify which technologies need to have improved process controls to limit the impact of incurred technical debt. These capabilities enable the system developer to minimize the risks associated with technical debt, including technical bankruptcy. The next chapter defines technical bankruptcy in the context of the LEAP process and provides example usages of the LEAP process in industry.

CHAPTER 5 – AVOIDING TECHNICAL BANKRUPTCY

5.1 Introduction

The previous chapters introduced the concept of technical debt and the LEAP process, which is a proactive method to identify potential technical debt in system development. This chapter takes the process one step further by answering RQ4: *How can the process and model be used to avoid technical bankruptcy?* This research question is broken into three tasks, each of which are addressed in this chapter:

- Task 4.1: Create a definition of technical bankruptcy within the context of the process and model outputs;
- Task 4.2: Utilize the developed process and model at the Space Development Agency and report on the results; and;
- Task 4.3: Create a simplified way of presenting and communicating the process and model.

First, Task 4.1 is addressed by defining technical bankruptcy in the context of the LEAP process and demonstrating how the process can be used to proactively identify systems that are at risk of technical bankruptcy. Next, Task 4.2 is addressed through an example use of the LEAP process, demonstrating how the process has been used in real-world situations to identify potential sources of technical bankruptcy. Finally, a simplified presentation of the LEAP process is provided to address Task 4.3.

5.2 Technical Bankruptcy in the Context of the LEAP Process

The technical debt ontology introduced in Chapter 2 defines technical bankruptcy as "the state where the system can no longer proceed with its lifecycle until some, or all, of the system technical

debt is repaid" [21]. A system may not be able to proceed with its lifecycle due to considerations in any of the three system dimensions: budget, schedule, or performance. The term "technical bankruptcy" does not imply that a system is only limited in the performance dimension. Rather, it means that the buildup of technical debt has reached the point where the compromises made in the performance dimension can no longer be covered by margins held in the other dimensions.

To determine the utility of the LEAP process to prevent technical bankruptcy, it is first necessary to define technical bankruptcy within the context of LEAP. The LEAP process utilizes technology and capability delivery timelines to indicate the health of the system. A healthy system has a sufficient probability of delivering capabilities on time to satisfy the stakeholders. Therefore, an unhealthy system is one which has an insufficient probability of delivering at least one capability on time, resulting in unsatisfied stakeholders. Definition 2 of the technical debt ontology states "Technical debt is the quantitative impact on the long-term health of the system accrued as the result of a technical compromise made to achieve a short-term benefit" [21]. Therefore, an unhealthy system can result from the accrual of technical debt. Within the LEAP process, the quantitative impact, and therefore the technical debt, can be measured as the difference in the delivery probability of capabilities as a result of a technical compromise.

5.2.1 Quantifying Technical Debt in the LEAP Process

Within the LEAP process, the quantitative impact of technical debt appears as changes in the probability of delivering a system that meets the stakeholder needs. The Delivery Matrix can be calculated prior to the implementation of a technical compromise. Then, the technical debt associated with that compromise can be estimated and a new Delivery Matrix calculated. The change in the delivery probabilities of each capability represent the quantitative impact of technical debt. Figure 4-31 shows this measurement by plotting the change in capability delivery

probabilities due to technical debt from a single task. In this figure, the technical debt is modeled as a random variable and its impacts are averaged. However, the same calculation could easily be applied to a single case.

Using the same aircraft scenario used in Chapter 4, an example scenario can be developed and examined. Assume that due to the anticipated late delivery of the engine design task, which is delaying the entire design increment, stakeholders pressure the development team to release the design sooner. The system developers determine that there are two ways that they can speed up the design release within the allocated budget. They can either reduce the amount of performance modeling of the engine design, incurring modeling and simulation debt, or they can reduce the design documentation, incurring documentation debt. If the performance modeling of the engine is reduced, then the maximum duration of the engine design task is reduced from 55 months to 45 months and the most likely duration is reduced from 32 to 27 months. If the documentation associated with the engine design is reduced, then the maximum duration of the engine design task is reduced from 55 months to 50 months and the most likely duration is reduced from 32 to 30 months. Figure 5-1 shows the resulting availability of the design capability, as calculated by the LEAP process. For this scenario, reducing the performance modeling, and thereby incurring modeling and simulation debt, increases the likelihood of delivering the design at the need date of month 43 by approximately 0.13, while reducing the documentation increases the delivery probability by approximately 0.09. Therefore, reducing the performance modeling appears to provide the highest probability of accelerating the release.

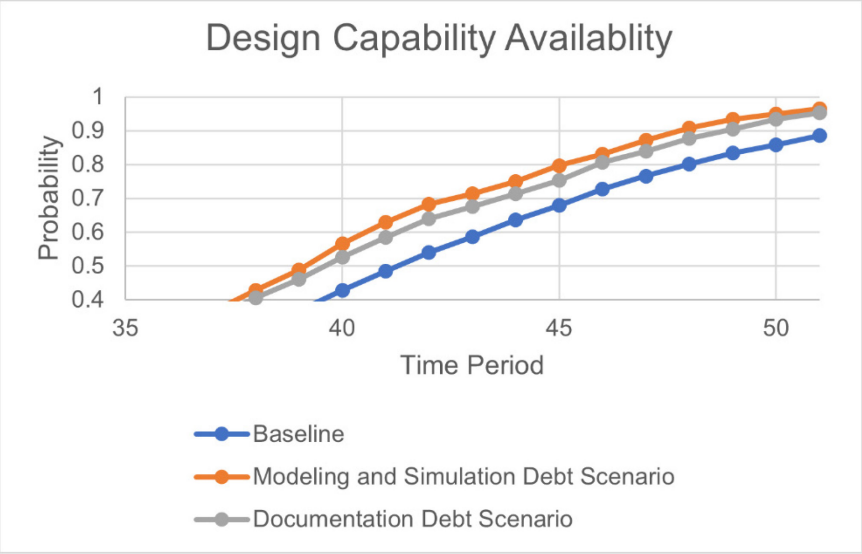

Figure 5-1. Design capability availability

To understand the technical debt impacts, the technical debt parameters (r and τ) are enumerated for each case as shown in Table 5-1. The modeling and simulation debt has the potential to significantly impact the engine and frame flight trials, since incomplete modeling may misrepresent the capabilities of the design. Modeling and simulation debt may also impact engine development, as the actual engine test cases may fail due to improper modeling assumptions. The modeling and simulation debt will have limited impact on the production of the engine and integration of the aircraft. This impact is captured through larger values of τ when r is specified. The documentation debt affects a larger number of successor tasks due to incomplete documentation. However, for most tasks, the individual impacts are smaller since it is expected that personnel will be available to provide information in support of later tasks, even if that information is not written down. The documentation debt impacts are assumed to be larger on the immediate successor tasks, engine development and d/b engine manufacture. These impacts are captured by larger values of r with smaller values of τ. In all cases, r and τ are modeled as normally distributed variables. Table 5-1 defines the mean (μ) and standard deviation (σ) used for each parameter in each analysis case.

Table 5-1. Technical debt parameters for engine design choice

Successor Task	Modeling and Simulation Debt	Documentation Debt
Engine Development	$r: \mu = 0.5, \sigma = 0.2$ $\tau: \mu = 0.5, \sigma = 0.3$	$r: \mu = 0.5, \sigma = 0.2$ $\tau: \mu = 0.5, \sigma = 0.2$
Engine Production	$r: \mu = 0, \sigma = 0$ $\tau: \mu = 0, \sigma = 0$	$r: \mu = 0.2, \sigma = 0.1$ $\tau: \mu = 0.2, \sigma = 0.05$
D/b Engine Manufacture	$r: \mu = 0, \sigma = 0$ $\tau: \mu = 0, \sigma = 0$	$r: \mu = 0.5, \sigma = 0.2$ $\tau: \mu = 0.5, \sigma = 0.2$
Assemble d/b Aircraft	$r: \mu = 0, \sigma = 0$ $\tau: \mu = 0, \sigma = 0$	$r: \mu = 0.2, \sigma = 0.1$ $\tau: \mu = 0.2, \sigma = 0.05$
Engine/frame Flight Trials	$r: \mu = 0.9, \sigma = 0.2$ $\tau: \mu = 0.8, \sigma = 0.2$	$r: \mu = 0.2, \sigma = 0.1$ $\tau: \mu = 0.2, \sigma = 0.05$
Avionics Production	$r: \mu = 0, \sigma = 0$ $\tau: \mu = 0, \sigma = 0$	$r: \mu = 0.2, \sigma = 0.1$ $\tau: \mu = 0.2, \sigma = 0.05$
Airframe Production	$r: \mu = 0, \sigma = 0$ $\tau: \mu = 0, \sigma = 0$	$r: \mu = 0.2, \sigma = 0.1$ $\tau: \mu = 0.2, \sigma = 0.05$
Ready to Assemble	$r: \mu = 0, \sigma = 0$ $\tau: \mu = 0, \sigma = 0$	$r: \mu = 0.2, \sigma = 0.1$ $\tau: \mu = 0.2, \sigma = 0.05$

The values in Table 5-1 were input into the LEAP process, starting with the generation of a new Monte Carlo schedule analysis. Figure 5-2 shows the outcome of the process, with the availability timelines of the ready to assemble task on the left and the delivery timelines of the same task on the right. These figures clearly show that the modeling and simulation debt has a larger impact on the ability to deliver the entire system, reducing the probability of delivery to 0.81 at the specified need date of 102 months. If the design capability was accelerated by reducing the documentation instead, then the probability of delivering the final system at month 102 would be 0.87, which is closer to the baseline value of 0.91. This analysis provides a quantitative measurement of the technical debt impact by measuring the likelihood of capability delivery. In this case, the analysis demonstrated that a choice that may appear more advantageous initially introduces additional technical debt into the system, resulting in a greater long-term impact. This result highlights the need to model technical debt throughout all phases of the system to ensure that technical concessions, if required, are chosen such that the lowest risk is imparted to the system development.

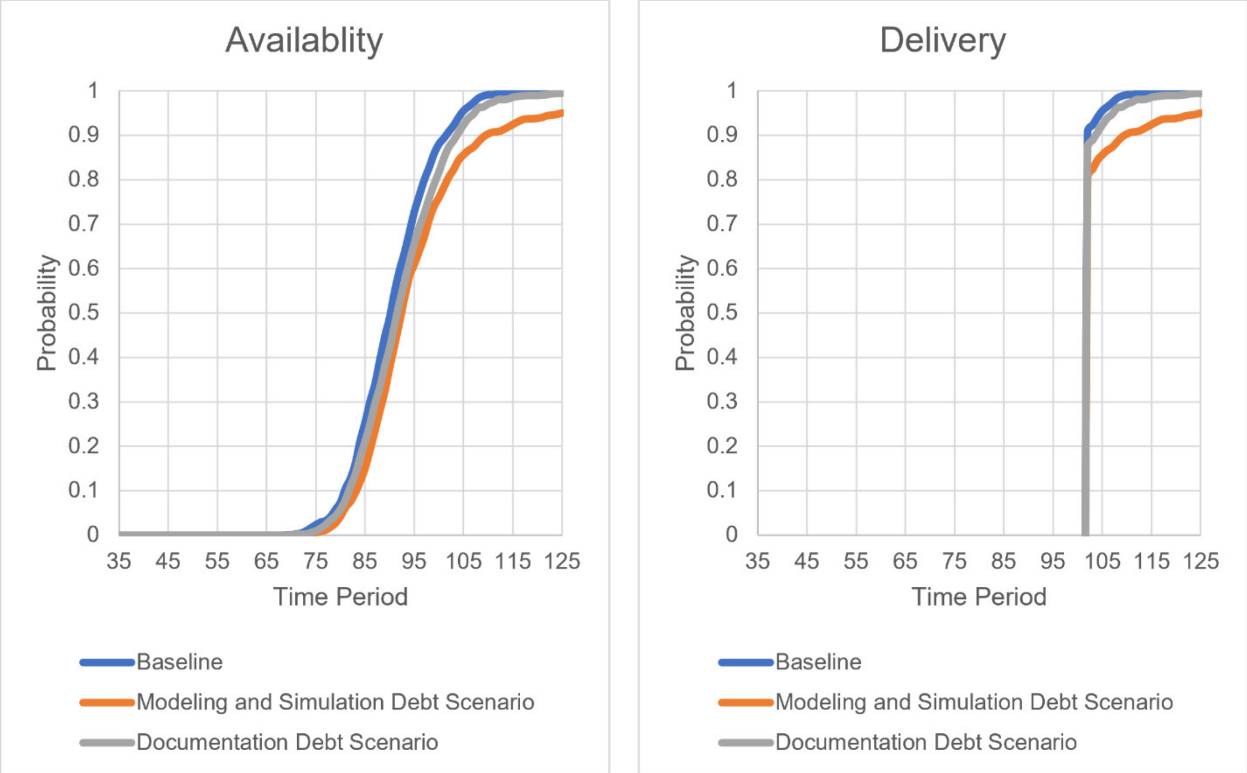

Figure 5-2. Comparison of technical debt impact on 'ready to assemble task' completion probability

5.2.2 Assessing Technical Bankruptcy with the LEAP Process

Having demonstrated the ability to assess technical debt within the LEAP process, it is necessary to determine when the system can no longer proceed with its lifecycle. A system which is proceeding with its lifecycle is continuing its development according to plan; there are no increases to schedule or cost outside of previously allocated system margins and the performance is within the required bounds. Therefore, if a system cannot proceed with its lifecycle, then it must acquire extra funds, increase its development timeline, or alter requirements to accept reduced performance.

Within the LEAP process, technical bankruptcy manifests as the inability to develop technologies on the timelines required to satisfy the stakeholder needs. The timeline, cost, and performance of a system are related through the triple constraint, which defines the interconnected nature of project time, scope, and cost [193]. Therefore, a change in schedule can be related to a

change in cost or to a change in performance (or both). For example, a longer schedule may be required due to rework required to reach desired performance levels and an increase in funding may enable a shortening of that schedule. By modeling the technology development using time, the LEAP process enables a stakeholder to 'buy back' time through increasing funding or decreasing performance requirements.

Hence, if the required technology development timelines cannot be adjusted within the available margins of funding or performance, then the system cannot proceed with its lifecycle: it is technically bankrupt. The severity of the bankruptcy and its mitigations will be project-dependent and based on the prioritization and importance of the individual stakeholder needs. A system that is likely to be technically bankrupt can be identified from the Delivery Matrix by the presence of cells where the probability of capability delivery is less than a threshold value specified by the system stakeholders. For example, Section 4.3.1.1 defined a large number of Monte Carlo simulations. Figure 5-3 shows a section of the Delivery Matrix from one of these simulations. In this Delivery Matrix, Capability 4 is unlikely to be delivered in time period 102 (only 18% likely), which is the time that it is needed by the stakeholder. The stakeholder's threshold is 90% probability of delivery. In this particular case, technical debt on the engine design task is driving the late delivery of the capability which results in the potential for technical bankruptcy.

time period	Capability 0	Capability 1	Capability 2	Capability 3	Capability 4
97	1	1	1	0.994	-1
98	1	1	1	0.995	-1
99	1	1	1	0.998	-1
100	1	1	1	0.998	-1
101	1	1	1	0.999	-1
102	1	1	1	0.999	0.18
103	1	1	1	0.999	0.209
104	1	1	1	1	0.247
105	1	1	1	1	0.279
106	1	1	1	1	0.299
107	1	1	1	1	0.325
108	1	1	1	1	0.347
109	1	1	1	1	0.383
110	1	1	1	1	0.412
111	1	1	1	1	0.438
112	1	1	1	1	0.473
113	1	1	1	1	0.497
114	1	1	1	1	0.523
115	1	1	1	1	0.559
116	1	1	1	1	0.588

Figure 5-3. Delivery Matrix showing potential technical bankruptcy

Although the Delivery Matrix identifies the potential for technical bankruptcy, it does not declare a system bankrupt. This distinction is necessary because a system developer may still be able to take actions to adjust the values in the Delivery Matrix to increase the likelihood of meeting the stakeholder needs. For example, the developer could follow the Achieve phase of the LEAP process to develop long-lead technologies outside of system procurement. Within an iteration, the system developer could increase resources to decrease the development time of a specific technology or to decrease the likelihood of a technology introducing technical debt to the system. The sequence of tasks could also be rearranged to develop more impactful technologies earlier in the iterative development cycle, even if they provide less initial value.

Therefore, the LEAP process does not provide an absolute determination of a technically bankrupt system. Instead, it provides leading indicators that a system is heading towards technical bankruptcy, in a timeframe where the system developer may still take steps to avoid reaching bankruptcy.

5.3 Using LEAP to Avoid Technical Bankruptcy

The LEAP process is a proactive process, and therefore can be used to avoid technical bankruptcy. The LEAP Delivery Matrix identifies which capabilities are not likely to be developed in time to meet the stakeholders' needs. Modelling different sequences of technology development, investments to achieve earlier capability development, and different procurement strategies can all be accounted for in the LEAP process to investigate the ability to satisfy the stakeholder needs.

The utility of the LEAP process in assessing technical bankruptcy can be evaluated by using the aircraft assembly example from Chapter 4. In this example, the system design capability is likely to be late to need, even though the overall system delivers on time. The lateness of the first design tasks could induce stakeholders to push for early delivery, resulting in technical compromises and creating technical debt. As seen in Figure 4-28, if that technical debt is limited and controlled, then the system still has a high likelihood of satisfying the stakeholder needs. However, if the technical debt is not carefully managed, then it may drive the system to technical bankruptcy.

The ability to quantify these technical debt aspects leads to an evaluation of potential technical bankruptcy within the LEAP process. Figure 5-4 shows the activity diagram by which the LEAP process is used to assess the potential for technical bankruptcy. The process starts with List phase of the LEAP process, shown in light blue in Figure 5-4. This phase decomposes the stakeholder needs into capabilities and creates the Functional Matrix. The creation of the Technology Matrix is updated to include the identification of technical debt parameters (r, τ, and α) between dependent technologies. Once all the dependencies are defined, the Evaluate phase of the LEAP process is executed, shown in dark blue. This phase runs the schedule analysis, accounting for technical debt, to create the Development Matrix and evaluates the stakeholder timelines to

produce the Need Matrix. These matrices are combined to produce the Delivery Matrix. If the Delivery Matrix outcomes are acceptable, then the system moves directly to the Procure phase, shown in yellow. In the Procure phase, the release plan is set. With stakeholder concurrence, the system is procured and the next iteration of the process begins with the List phase for the next release. If the stakeholders do not concur with the release plan, then there is the potential for technical bankruptcy, highlighted in orange.

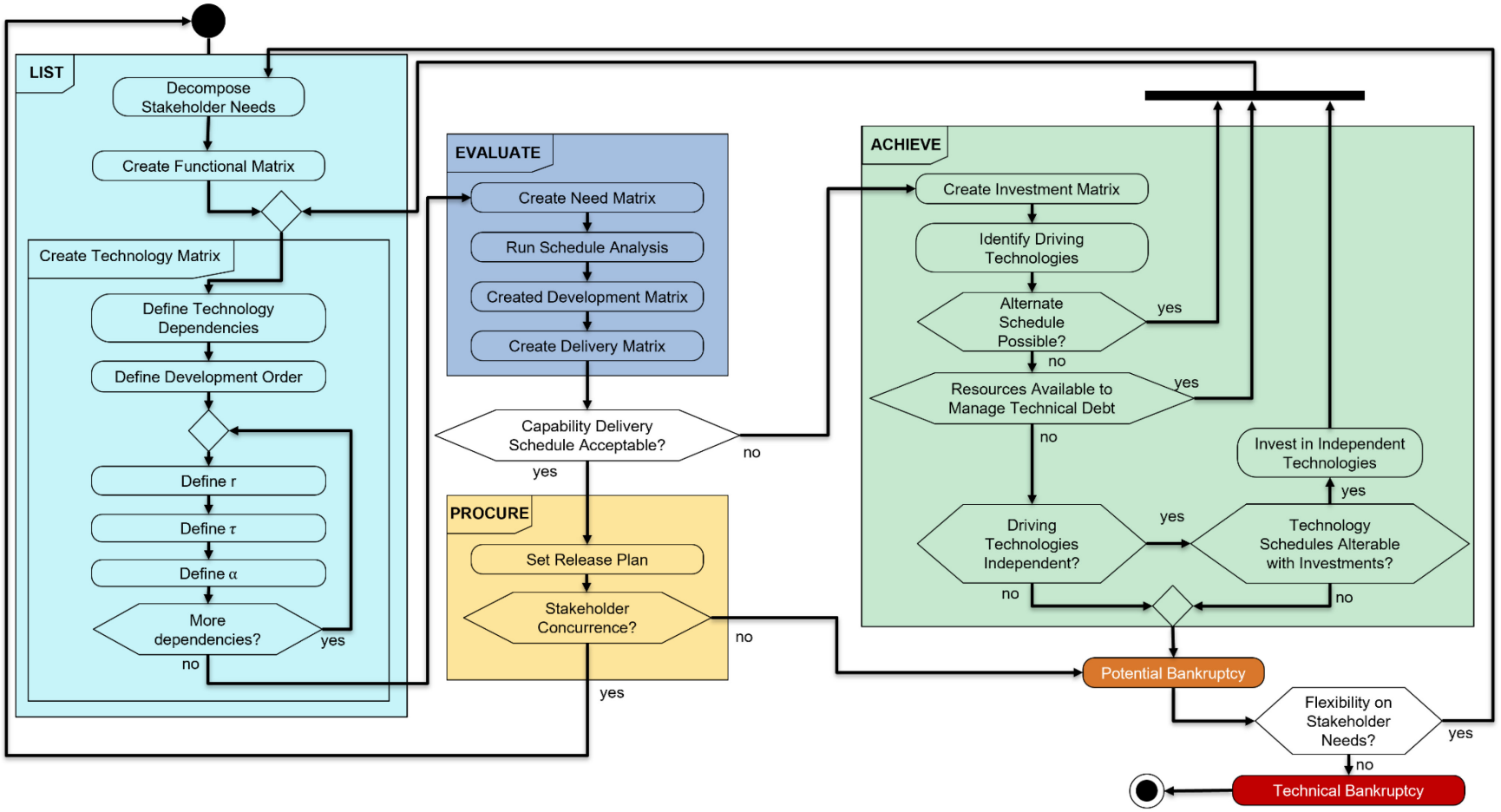

Figure 5-4. Activity diagram for assessing technical bankruptcy with LEAP process

If the Delivery Matrix is unacceptable, then the system moves into the Achieve phase, shown in green. Within this phase, the Investment Matrix is created enabling the identification of driving technologies. At this point, the system developer has several choices to adjust the delivery timelines. The development schedule can be adjusted to prioritize the driving technologies, additional resources can be applied to manage and control the potential technical debt, or investments can be made in independent technologies to accelerate their development. If any of these outcomes are possible, then the development order and dependencies will have changed and the process repeats, starting with the creation of the Technology Matrix. If none of the outcomes are possible, then the system has the potential to be technically bankrupt, highlighted as the dark orange activity.

If the system reaches the potential bankruptcy state, then the system developer must work with the stakeholders to determine if there is flexibility in the set of stakeholder needs in any of the system dimensions. If there is flexibility, then the process resets back to the start of the List phase with a redefinition of the stakeholder needs. If there is no flexibility, then the system is technically bankrupt (shown in red in Figure 5-4): it cannot satisfy the stakeholder needs and therefore will not be able to continue with its lifecycle until some other change is executed, such as acquiring additional resources to adjust development timelines.

The choices made within the Achieve phase of the process show the utility of the LEAP process as a decision support system. If the system developer has a choice of when to develop internal technologies as part of their release planning cycle, they can adjust the development sequence. By adjusting the development sequence and evaluating the resulting Delivery Matrix, the developer can identify technologies that are more likely to induce large delays in the overall system development. These technologies can then be prioritized in the release plans, even if they do not

deliver high levels of perceived value to the stakeholder. The LEAP process enables rapid assessment of the development order and release plans to highlight the outcomes of different delivery cadences to support decisions.

Technical debt mitigation is another important step that can be applied in the achieve phase. Capability delivery may be delayed due to technical debt induced by one of the technologies on the rest of the system. By tracing the late capabilities to the driving technologies and identifying the predecessors of those technologies, the source of technical debt can be found within the LEAP process. Resources can then be applied to either mitigate the occurrence of technical debt in the originating technology or to mitigate the effects of the technical debt on successor tasks. By enabling the identification of technical debt, the LEAP process allows the system developer to put controls in place to mitigate the effects.

Finally, the achieve phase also highlights the ability to invest in independent technologies, where an independent technology is defined as one that is developed outside the scope of the current system. By adding resources to the independent technologies, the development timelines of those technologies may be accelerated, thereby reducing risk to the system under development. This step is especially important for iteratively developed systems. In these systems it is critical to understand the reliance on outside technologies and to limit the risk on a specific procurement by only including mature technologies [162]**.** The LEAP process proactively identifies these technologies and therefore can provide early indications to the system stakeholders on the risk associated with including the technologies in a given release.

Iterating on the LEAP process with various assumptions of resource allocation, technical debt controls, and development schedules enables rapid assessments of release plans and timelines. These assessments can quantify the risks associated with each option. Therefore, the risk of

technical bankruptcy can be quantified and presented to the stakeholders. The following section provides an example application of the LEAP process and discusses the assessment of technical bankruptcy within each application.

5.3.1 Example Applications

The LEAP process described in [160] enables a decision maker to rapidly assess the impact of their decisions. The achieve phase of the process allows the decision maker to investigate different potential investments to determine which one best increases the chances of delivering capability on time to the stakeholders. This section provides two example applications of the LEAP process at the Space Development Agency (SDA). First, an application of the qualitative LEAP process to a notional selection of optical communications terminal investments is presented. Second, an example of the application of the quantitative LEAP process to identify and prevent potential technical debt within the iterative ground system development is presented.

The qualitative LEAP application at SDA was presented at the *2023 INCOSE International Symposium* [167] and the paper is reprinted here.

5.3.1.1 LEAPing Ahead – The Space Development Agency's Method for Planning for the Future

5.3.1.1.1 Abstract

The Space Development Agency (SDA) is a constructive disruptor within the Department of Defense, tasked with rapidly acquiring and delivering space-based capabilities. SDA delivers dozens of satellites on two-year cycles and this pace requires defined processes to ensure that the technology exists to support the required capabilities. SDA has developed a process called List, Evaluate, Achieve, Procure (LEAP) which is used to identify technologies that require additional development resources to meet both current Warfighter needs and those that will occur in the

future. This paper provides an illustrative example of SDA applying the LEAP process to the development of SDA's optical communications terminals, demonstrating how it is used to identify critical technologies to be supported through investment opportunities that augment the system acquisitions.

5.3.1.1.2 Introduction

The Department of Defense (DoD) develops and delivers capabilities into the hands of the warfighters. In recent years, the DoD has emphasized rapid acquisition timelines - less than five years from program approval to delivery, with less than two years for urgent needs [32]. Achieving these objectives requires reducing the cycle time for a system, defined as the time from program start until the system is declared to have initial operational capability [33]. The actual cycle times for Major Defense Acquisition Programs (MDAPs) from 1997 to 2015 averaged 6.9 years with a median time of 7.6 years, both of which are above the planned values [194]. The Space Development Agency (SDA) was created in 2019 as a constructive disruptor within the Office of the Secretary of Defense to deliver space-based capabilities within this rapid acquisition environment.

SDA will "deliver a minimum viable product – on time, every two years – by employing spiral development methods, adding capabilities to future generations as the threat evolves" [46]. SDA is building the Proliferated Warfighter Space Architecture (PWSA), a proliferated low Earth orbit (pLEO) constellation of hundreds of satellites providing missile warning, low-latency communications direct to the warfighter, alternate position navigation and timing services, and in-space computation nodes. SDA is driven by the need to provide "good-enough" capability to the warfighter on a regular schedule, ensuring that the capability exists in time to meet the need. The

SDA delivery cycle, shown in Figure 5-5, provides a new release of capability, called a Tranche, on a two-year cadence [46].

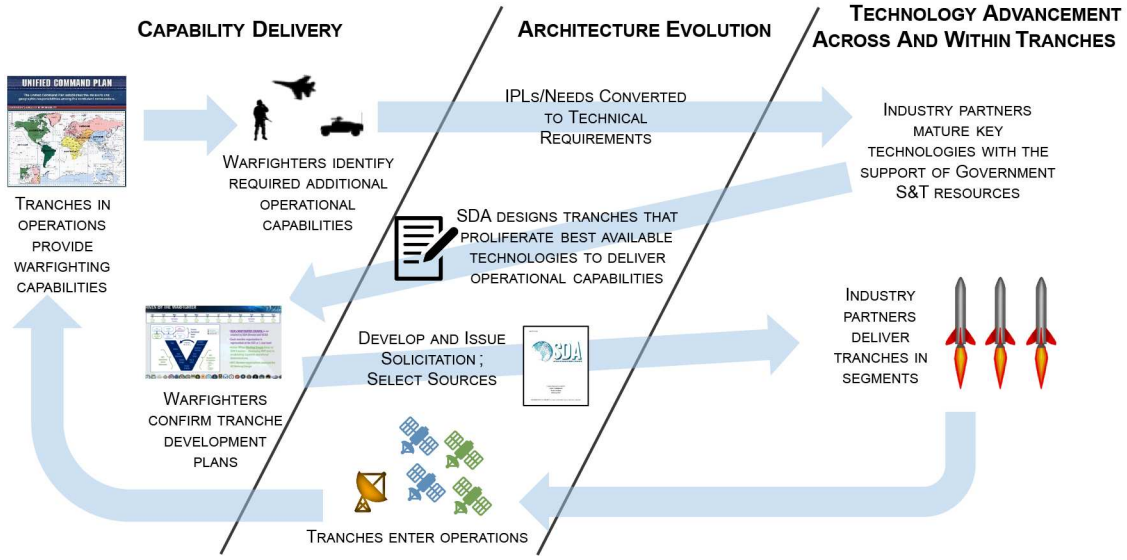

Figure 5-5. SDA capability delivery lifecycle

Maintaining a two-year delivery cycle for a complex system eventually consisting of hundreds of satellites requires extensive planning and trade-offs. In the rapid acquisition environment, the development timelines cannot delay delivery of capability to the field (i.e., satellites on orbit providing support to tactical operations). Any changes to the planned capabilities delivered by the system have the potential to impact the future state of the system. When making any such changes, SDA must be careful not to introduce too much technical debt into the system. Technical debt, the negative long-term impacts of short-term decisions, can accumulate such that a system can no longer meet its requirements without first addressing the technical debt [60]. For example, a system developer may forgo completing interface documentation in order to release a component on schedule. The lack of that documentation complicates future development involving those interfaces and increases the cost and schedule of capability upgrades in the long-term. SDA developed the List, Evaluate, Achieve, and Procure (LEAP) process [160] to identify and assess

the current state of technology and identify investments required to deliver the capability required by the warfighter on schedule.

This paper introduces the concepts that led to the creation of the LEAP process and presents an example of the use of the LEAP process within SDA. First, related work in the field of rapid acquisition development is reviewed. Next, the LEAP process is introduced and an overview of its use at SDA is provided. The next section provides an example of using the LEAP process at SDA, specifically focusing on the technology development of the optical communications terminals. Finally, the paper is concluded and recommendations for future work are presented.

5.3.1.1.3 Related Work

Traditional DoD programs emphasize meeting performance requirements at the risk of cost growth and/or schedule delays. Both the DoD and the Government Accountability Office (GAO) have identified best practices to mitigate these negative results. The DoD recommends that systems achieve technical maturity prior to Milestone B [195]. Milestone B is the transition point between the technology maturation and risk reduction (TMRR) phase and the engineering and manufacturing development phase (EMD) in a traditional acquisition program. Milestone B occurs after the system preliminary design review (PDR) and prior to the system critical design review (CDR) [196]. The GAO also recommends that systems achieve technical maturity [197] by Milestone B. In both cases, technical maturity is defined by the technology readiness level (TRL). The DoD recommends that systems achieve TRL 6, where a model or prototype of the system or subsystem is tested in a relevant environment, while the GAO recommends that systems achieve TRL 7, where the system prototype is tested in its operational environment [197]. In addition, the GAO recommends that systems achieve design maturity, where 90 percent of the design drawings have been released, by CDR [195].

Katz et al. [195] demonstrated that the technical maturity of a system is correlated with the schedule change of the system – a less technically mature system is more likely to experience schedule slips and budget increases. Schedule slips in defense systems, especially those designed to counter immediate adversary threats, can significantly impact the Warfighter. In these systems, a partial solution delivered now is more valuable than an optimal solution delivered late to need [162]. Tate [162] identifies seven different types of systems that can be delivered and fielded quickly:

1. Commercially purchased products: existing products that require no additional development
2. Upgrades to existing systems to add existing products and systems: adding mature technology and components onto an already mature system, where the addition does not put undue stress on the host system capabilities (size, weight, power, computational load)
3. Integrating existing systems: combing already existing mature systems to produce a new system
4. New systems developed through direct collaboration with the users to identify the required capabilities that should be fielded: Agile methods focusing on user feedback can evolve requirements instead of driving towards preset objective and threshold requirements
5. New systems with limited requirements: systems that provide one or two critical capabilities while other capabilities may be at or below the current level in other systems
6. New systems developed external to the program: leveraging a system developed by an external agency, such as corporate internal research, and demonstrated to high TRL. The system must only be modified to work within the operational environment

7. Modular subsystems that replace legacy subsystems: requires a modular architecture with sufficient design margin to accommodate the new upgrades

With the exception of items (4) and (5), each of these types of systems relies on the reuse of existing technology and capability.

Knoll, Fortin, and Golkar [198] identify that complex systems often involve concurrent design and therefore changes in one discipline must be propagated to the other disciplines. Decisions made in one discipline are not isolated events; they impact every other part of the system. Issues such as technical debt present in one discipline or one part of a system can aggregate and grow as the complete system is developed [80]. In iterative development systems, Sangwan et al. [91] identify the need to consider dependencies for release planning, which exist between the requirements and features, between features and architectural elements, and between all architectural elements.

These processes and techniques identify the types of systems amenable to rapid development and methods for considering dependencies and impacts during system development. However, they do not provide guidelines for how to manage the dependencies across iterations to minimize increases in complexity during system development. To address this gap, Kleinwaks et al. [160] developed a process called List, Evaluate, Achieve, Procure (LEAP). The LEAP process provides mathematical processes to identify which technologies drive the development of capabilities required by the system users in both the functional and temporal dimensions. The process consists of four phases:

- List – the identification of dependencies between the required capabilities and the supporting technologies

- Evaluate – the application of temporal needs to the capabilities and the determination of the ability to deliver capabilities on time
- Achieve – the identification of technologies which require investments to accelerate their development, and
- Procure – the acquisition of the system that will meet the user's needs.

5.3.1.1.4 SDA and the LEAP Process

The PWSA is an interconnected set of ground and space-based systems with a priority on delivering each Tranche on schedule. Each satellite consists of optical communications terminals, radio frequency (RF) communications systems, on-board battle management systems, tactical data links, and mission payloads. Each Tranche is scheduled for launch approximately 30 months after contract award. To enable this schedule, SDA has adopted several of the principles from Tate [162], namely:

- Leveraging commercial products where possible to reduce non-recurring engineering activities
- Integrating new Tranches into existing Tranches to augment the delivered capability and to add new capability
- Frequently collaborating with the users and the warfighters to ensure that the right capabilities are developed
- Encouraging and leveraging the development of critical technologies outside of the SDA Tranches

In developing each Tranche, SDA identifies which technologies are mature enough to include in the Tranche, such that non-recurring engineering (NRE) is limited. SDA developed the LEAP process [160] to identify the functional and temporal dependencies between the capabilities needed

by the Warfighter and the technologies that support those capabilities. This process provides a direct linkage between technology development timelines, system iterations, and the satisfaction of stakeholder needs not found in other similar processes. The LEAP process also identifies technologies that require investment such that they can support future Tranches. The ability of the LEAP process to identify both the functional and temporal dependencies is critical to SDA maintaining its rapid development schedule.

SDA applies the four steps of the LEAP process as follows:

1. List: identify the strategic (long-term) and tactical (short-term) capabilities required by the Warfighter and identify the technologies required to achieve these capabilities
2. Evaluate: determine the current development timelines for the technologies and evaluate those timelines against the Warfighter's need dates
3. Achieve: identify the technologies that will be late to need and the create investments to accelerate the development of those technologies through programs and partnerships with other Government agencies and industry
4. Procure: produce a solicitation to for a Tranche to proliferate the technologies that provide the capabilities to the Warfighter

Each step in the LEAP process produces matrices that map capabilities to technologies and time periods. By following the steps in Kleinwaks et al. [160], it is possible to mathematically identify the technologies that will be developed late to need and the impact of each late technology on SDA's ability to meet the Warfighter's needs.

5.3.1.1.4.1 Iterations within the LEAP process

Figure 5-6 shows the technical timeline targeted by SDA starting with its first application of LEAP during the first quarter of Government Fiscal Year 2021 and broken into the various phases

of LEAP. The List, Evaluate, and Achieve phases occur for each Tranche as part of the road mapping exercise for the Tranche. A full cycle is performed approximately 18 months prior to the start of the system definition for the Tranche, allowing short-term achievements to positively affect the technology development timelines. A short LEAP cycle begins approximately one year prior to the release of the acquisition documentation. In this cycle, the List and Evaluate phases lead to the definition of the minimum viable product (MVP) for the Tranche, which represents the planned acquisition for the minimum viable capability (MVC) that the system will deliver. The MVP definition is socialized with industry through the Request for Information (RFI) process, and any last changes are made prior to the technology freeze (TF). The TF marks the beginning of the Procurement phase. At this point, the decision is made on which capabilities will make it into the solicitation including which, if any, will require development during the Tranche. Finally, the acquisition documentation is drafted (the draft solicitation (DS)) and released (the final solicitation (FS)) and the contract is awarded following standard Government acquisition processes. At award, the selected contractors are given authority to proceed (ATP) to begin developing the system. Within 30 months from ATP, the system is scheduled to achieve initial launch capability (ILC) consisting of the launch of the first set of newly developed satellites. Subsequent launches occur on a one-month cadence to complete the on-orbit population of the Tranche.

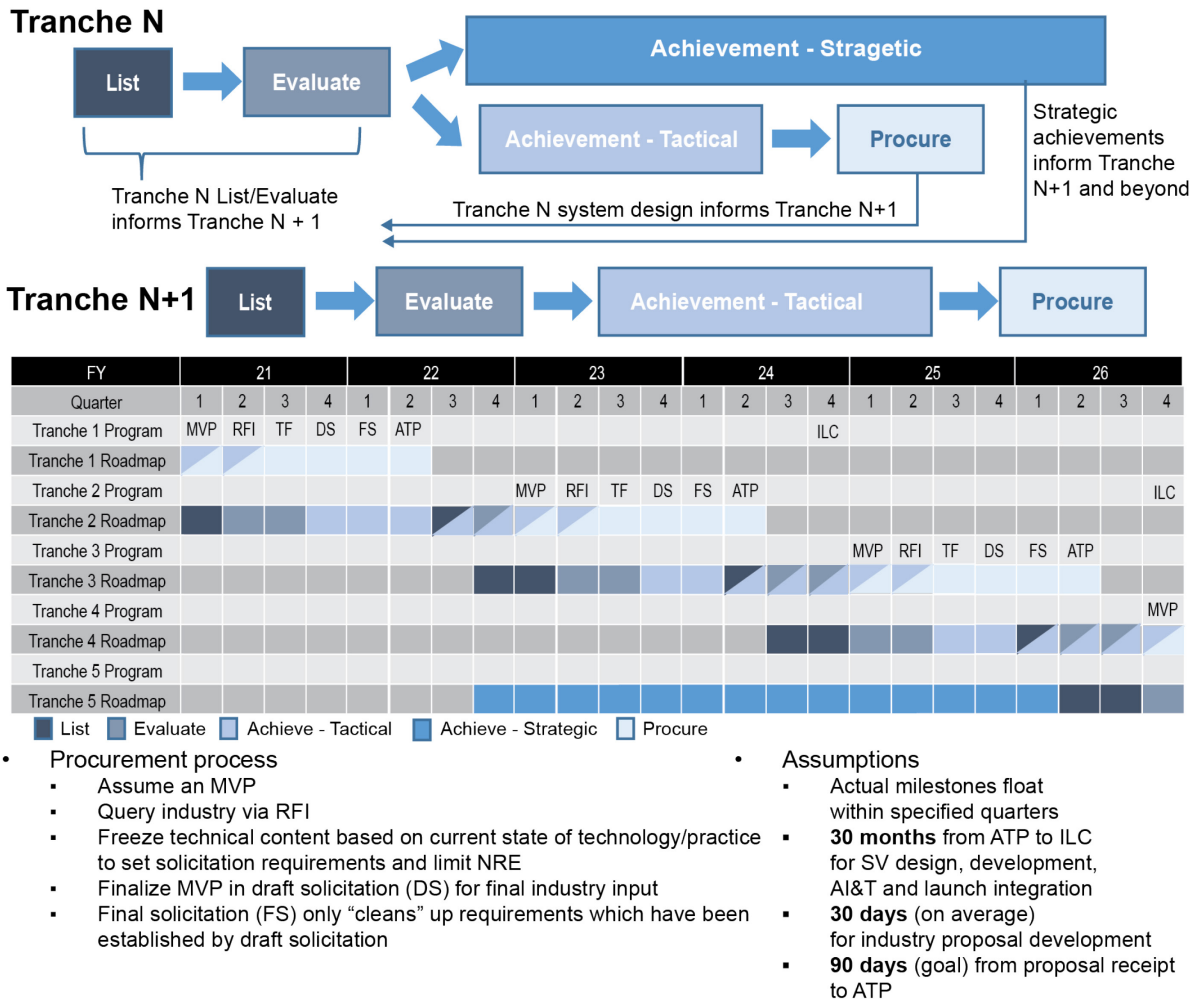

Figure 5-6. Example SDA timeline implementing the LEAP process in iterative fashion across multiple tranches

Critical to the process is the inclusion of strategic achievements, indicated in the royal-blue sections of Figure 5-6. The Delivery and the Investment Matrices created as part of the Evaluate and Achieve steps clearly indicate which technologies and capabilities will be late to need not only for the next Tranche, but also for future Tranches. Early investments in developing these long-term technologies reduce the risk in those future Tranches. Figure 5-6 shows the application of these long-term technologies through Tranche 5.

5.3.1.1.5 Application of LEAP to SDA's Optical Communications Terminal Development

A key pillar of the PWSA is the development of global beyond line-of-sight (BLOS), low-latency, high data rate communications. This section provides a notional example of using the LEAP process on a subset of the development of this capability. The mathematical equations used to perform the matrix transformations are defined in Kleinwaks et al. [160].

5.3.1.1.5.1 LEAP Phase 1: List

The List phase decomposes the stakeholder need into strategic and tactical capabilities and identifies the technologies required to support the tactical capabilities. Table 5-2 shows the breakdown of the need for BLOS, low-latency, high data rate communications into strategic and tactical capabilities. This need decomposes into two strategic capabilities: an optical communications network providing communications, and optical global operations - the ability to connect to multiple users anywhere on the globe at any point in time. These strategic capabilities are decomposed into tactical capabilities and then the enabling technologies are identified. The technologies may support one or more capabilities. In these cases, the technologies are listed in one row by their identifier only.

Table 5-2. Decomposition of stakeholder needs to strategic capabilities, tactical capabilities, and technologies

Strategic Capabilities	Tactical Capabilities	Technologies
Optical Communications Network	C1. Space-to-ground (S2G) optical communications	T15. Communications standards
		T11. Multiple vendor interoperability
		T12. Tasking and scheduling algorithms
		T13. Common network protocols and routing mechanisms
		T21. Pointing, acquisition, and tracking algorithms
		T22. Bus stability
	C2. Space-to-space (S2S) communications in the same orbital shell (same altitude and inclination)	T11, T12, T13, T15, T22, T21
		T5. Space-based mesh network
		T16. Space-to-space same vendor communications
		T17. Space-to-space different vendor communications
		T18. Space-to-space in-plane communications
		T19. Space-to-space out-of-plane communications
		T5, T11, T12, T13, T15, T16, T17, T21, T22

Strategic Capabilities	Tactical Capabilities	Technologies
	C3. Space-to-space (S2S) communications in different orbital shells	T20. Space-to-space out-of-shell communications
	C4. High data rates	T24. 1 Gbps data rates
		T25. 10 Gbps data rates
		T26. 100 Gbps data rates
Optical Global Operations	C5. Regional Access (the ability of the constellation to communicate optically with at least one user within a specified region for a specified period of time)	T5, T12
		T2. Small satellite compliant size, weight, and power (SWAP) for bus and payloads
		T23. Ranges up to 6500 km
	C6. Global Access (the ability of the constellation to communicate optically with at least one user anywhere on the globe for a specified period of time)	T2, T5, T12, T23
		T1. Commoditization of satellite bus and payloads
		T3. Satellite proliferation
		T4. Manufacturing at scale
		T6. Fleet-based environmental testing
	C7. Global Operations (the ability of the constellation to communicate optically with multiple users anywhere on the globe at any time)	T1, T2, T3, T4, T5, T6, T12, T23
		T7. Multiple terrestrial users per space-based communications terminal
		T8. Mobile/transportable ground terminals
		T9. Orbit-aware network routing protocols
		T10. Operation in all lighting conditions
		T14. All weather communications

The data in the table is translated into the Functional Matrix, shown in Figure 5-7, where only the capability and technology identifiers (e.g., C1 and T1) are used. The highlighted cells indicate the dependencies between the capabilities and technologies. The Functional Matrix shows that the identified technologies support multiple capabilities.

Functional Matrix, F

	T1	T15	T2	T4	T6	T3	T8	T10	T11	T12	T7	T13	T5	T9	T14	T22	T21	T16	T17	T18	T19	T20	T23	T24	T25	T26
C1		1							1	1		1				1	1									
C2		1							1	1		1	1			1	1	1	1	1	1					
C3		1							1	1		1	1			1	1	1	1			1				
C4																								1	1	1
C5			1							1			1										1			
C6	1		1	1	1	1				1			1										1			
C7	1		1	1	1	1	1	1		1	1		1	1	1								1			

Figure 5-7. Functional Matrix

In addition to the dependence of capabilities on technologies, there are also interdependencies between the technologies themselves. These interdependencies are captured in a design structure matrix (DSM) called the Technology Matrix. The Technology Matrix shown in Figure 5-8 has undergone a preliminary partitioning to make it a lower triangular matrix, which results in reshuffling of the technologies compared to the order in Table 5-2. Further ordering and partitioning can be performed to have the rows in the matrix correspond to the expected development order as well. The Technology Matrix is used to inform the technology development sequence included in the Development Matrix, which is discussed in the next section.

Technology Matrix, T

	T1	T15	T2	T4	T6	T3	T8	T10	T11	T12	T7	T13	T5	T9	T14	T22	T21	T16	T17	T18	T19	T20	T23	T24	T25	T26
T1																										
T15																										
T2																										
T4	1																									
T6				1																						
T3	1			1	1																					
T8																										
T10																										
T11		1																								
T12				1																						
T7							1			1																
T13		1																								
T5									1			1														
T9										1		1	1													
T14																										
T22																										
T21		1														1										
T16		1															1									
T17		1															1									
T18		1														1	1	1								
T19		1														1	1	1	1							
T20		1														1	1									
T23																										
T24		1														1	1	1	1	1	1	1				
T25		1														1	1	1	1	1	1	1	1			
T26		1														1	1	1	1	1	1	1	1	1		

Figure 5-8. Technology Matrix

5.3.1.1.5.2 LEAP Phase 2: Evaluate

The Evaluate phase starts with the identification of the capability need dates and the expected development timelines for each technology. SDA delivers its capabilities in two-year cycles which would imply a two-year time period for the LEAP analysis. However, as shown in Figure 5-6, the

procurements (FS) are released in the off years from the launches (ILC). To fully establish the technological availability, a maximum of a one-year spacing for the time periods is required. The Need Matrix for the tactical capabilities associated with delivering global BLOS high data rate communications is shown in the upper center of Figure 5-9, with one-year time periods through the notional Tranche 5 timeline. A value of one (1) in the Need Matrix indicates that a capability is needed in a time period and a blank cell indicates that the capability is not needed in the time period. Note that tactical capability C5, regional access, is only needed until tactical capability C6, global access, is delivered.

The next step is to evaluate the technologies to determine their development timelines. The development timelines indicate when a technology is expected to be ready to incorporate into a larger capability development. The Development Matrix, shown on the left of Figure 5-9, includes the expected development timelines for the SDA technologies as estimated in 2020, prior to any investments or procurements. In this matrix, a one (1) indicates that a technology is expected to be fully developed in the time period and a blank cell indicates that the technology is not expected to be complete in the time period.

With the Need and Development Matrices evaluated, the Availability Matrix is computed using the methods from Kleinwaks et al. [160]. The Availability Matrix shows when each capability is expected to be available based on the technical development timelines and the functional dependencies. A value of one (1) indicates that the capability is expected to be available in the time period and a value of zero (0) indicates that the capability is not expected to be available. The result, shown in the lower center of Figure 5-9, shows that tactical capability C1, S2G optical communications, and tactical capability C5, regional access, are expected to be available in 2022. Tactical capability C3, S2S optical communications in the same orbital shell, is expected to be

available in 2023. Tactical capability C7, global operations, is not expected to be available in the considered time frame, based on the current development path of its supporting technologies.

The final step in the evaluation phase is to calculate the Delivery Matrix to determine which capabilities, if any, will be available late to need. This Matrix is shown on the right side of Figure 5-9, with late capabilities highlighted in red. In the Delivery Matrix, a value of one (1) indicates that a capability is late to need: it is needed but not available. A value of zero (0) in the Delivery Matrix indicates that either the capability is available and not needed or the capability is not available but is not needed. A value of negative one (-1) in the Delivery Matrix indicates that the capability is available and not needed, either due to early delivery or the removal of the need (Kleinwaks et al. 2023). In this example, tactical capability C4, high data rates, is late by two years. It is needed in 2028, but not available until 2030. Tactical capability C7, global operations, is needed in 2028, but is not expected to be available in the specified time frame. Tactical capability C5, regional access, remains available after 2025 even though it is no longer needed.

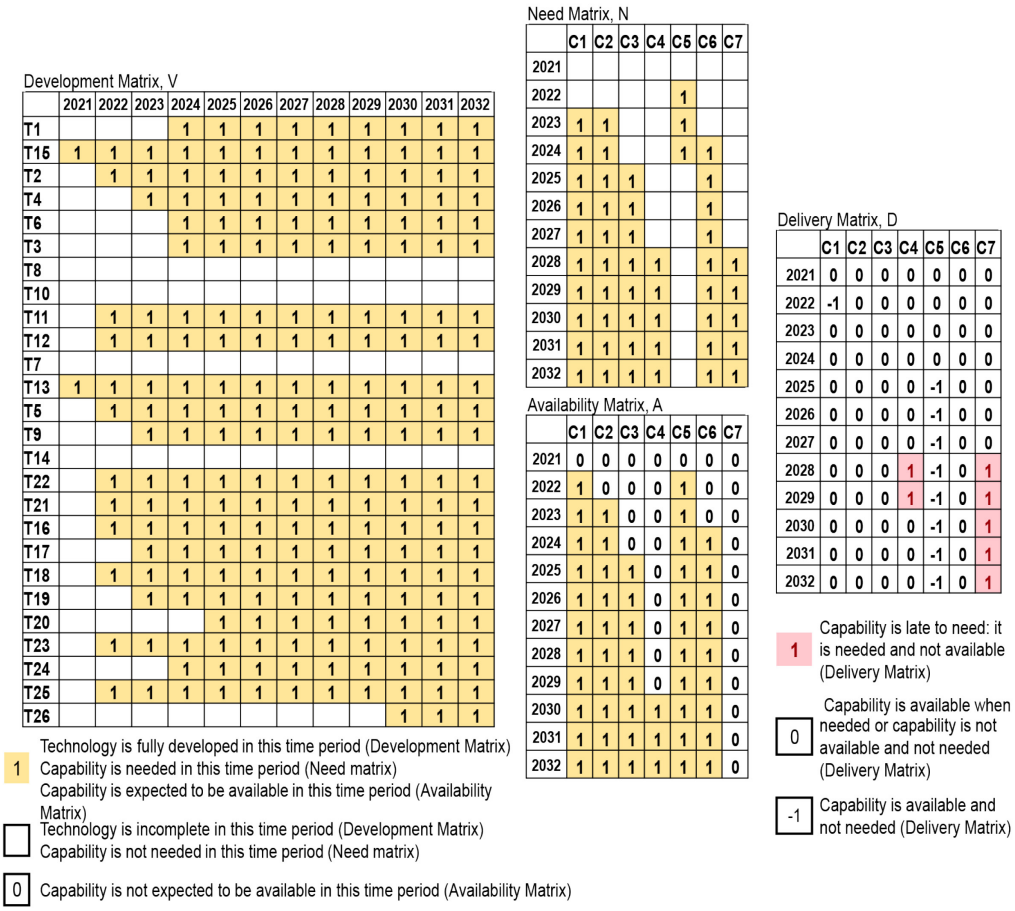

Figure 5-9. Matrices used in the evaluation phase of the LEAP process

5.3.1.1.5.3 LEAP Phase 3: Achieve

The Delivery Matrix in Figure 5-9 clearly shows tactical capabilities that will be delivered late to need. The Achieve phase analyzes the technologies that contribute to the late delivery to determine investments that could accelerate the development of those key technologies. Within SDA, investment strategies include investing in technology development programs, partnering with other U.S. Government agencies, and encouraging industry growth and investment. Parallel investments may be made to reduce the risk associated with a single technology development cycle. Using the methods in Kleinwaks et al. [160], the Investment Matrix is calculated. This calculation determines the technologies that drive the late arrival of capabilities. The Investment Matrix is shown in Figure 5-10. Technologies that contribute to the late delivery of tactical

capabilities are highlighted in red. The number in the matrix indicates how many late tactical capabilities are contributed to by the technology in that time period [160].

Investment Matrix, I

	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032
T1	0	0	0	0	0	0	0	0	0	0	0	0
T15	0	0	0	0	0	0	0	0	0	0	0	0
T2	0	0	0	0	0	0	0	0	0	0	0	0
T4	0	0	0	0	0	0	0	0	0	0	0	0
T6	0	0	0	0	0	0	0	0	0	0	0	0
T3	0	0	0	0	0	0	0	0	0	0	0	0
T8	0	0	0	0	0	0	0	1	1	1	1	1
T10	0	0	0	0	0	0	0	1	1	1	1	1
T11	0	0	0	0	0	0	0	0	0	0	0	0
T12	0	0	0	0	0	0	0	0	0	0	0	0
T7	0	0	0	0	0	0	0	1	1	1	1	1
T13	0	0	0	0	0	0	0	0	0	0	0	0
T5	0	0	0	0	0	0	0	0	0	0	0	0
T9	0	0	0	0	0	0	0	0	0	0	0	0
T14	0	0	0	0	0	0	0	1	1	1	1	1
T22	0	0	0	0	0	0	0	0	0	0	0	0
T21	0	0	0	0	0	0	0	0	0	0	0	0
T16	0	0	0	0	0	0	0	0	0	0	0	0
T17	0	0	0	0	0	0	0	0	0	0	0	0
T18	0	0	0	0	0	0	0	0	0	0	0	0
T19	0	0	0	0	0	0	0	0	0	0	0	0
T20	0	0	0	0	0	0	0	0	0	0	0	0
T23	0	0	0	0	0	0	0	0	0	0	0	0
T24	0	0	0	0	0	0	0	0	0	0	0	0
T25	0	0	0	0	0	0	0	0	0	0	0	0
T26	0	0	0	0	0	0	0	1	1	0	0	0

Figure 5-10. Investment Matrix

The Investment Matrix clearly shows that the following technologies contribute late delivery:

- T7. Multiple terrestrial users per space-based communications terminal
- T8. Mobile/transportable ground terminals
- T10. Operation in all lighting conditions
- T14. All weather communications
- T26: 100 Gbps data rates

Technologies T7, T8, T10, and T14 directly impact the ability to deliver the global operations tactical capability. Technology T26, 100 Gbps data rates, delays the high data rate tactical capability by two years. Of note is that SDA's practice of using existing technology where possible

to minimize NRE on contracts limits the number of non-zero entries in the Investment Matrix. Technology T10 is a traditional non-functional requirement, specifying that the optical communications terminal must minimize the impact of communications outages due to solar impingement on the device. These types of technologies typically are lower priority during development, but the Investment Matrix makes clear that developing this technology, although potentially not seen as adding as much value as the other technologies, impacts the satisfaction of the stakeholders' needs.

The Investment Matrix identifies technologies which can benefit from additional investments to shorten the development timeline. SDA invests by encouraging industry development of new technologies. In 2021, SDA funded efforts shown in Table 5-3. Table 5-3 also identifies which technology is supported by each topic and the expected change in the development timeline of that technology.

Table 5-3. SDA investments mapped to the enabling technologies

Investment	Technology Addressed	Projected Change in Development Timeline
Reduction of SWAP-C per bit	T2. Small satellite compliant size, weight, and power (SWaP) for bus and payloads	None. However, smaller terminals with less power are still beneficial
Design for manufacturing considerations to support high-rate production	T4. Manufacturing at scale	None, but improvements to manufacturing processes buy down system risk
Demonstration of a path to 100 Gbps for S2S comms	T26. 100 Gbps OCTs	The investment is expected to accelerate the development timeline from 2030 to 2026
Development of low-cost, mobile, or fixed optical ground terminals	T8. Mobile/transportable ground terminals	Without the investment, there was no known timeline for creating these terminals. With the investment, the delivery timeline is expected to be 2030
Demonstration of enhanced S2G and space-to-air (S2A) links	T14. All weather communications	Without the investment, there was no known timeline for this technology. With the investment, the technology is expected to be ready for all-weather optical communications by 2026

Investment	Technology Addressed	Projected Change in Development Timeline
Development of compact systems capable of supporting coherent (e.g., QPSK) and non-coherent (e.g., OOK) links	T13. Common network protocols and routing mechanisms T17. Space-to-space different vendor communications	Developing these technologies enables better S2S communication links and a more robust mesh network. With the investment, the technology is expected to be ready for inclusion in a procurement by 2027
Demonstration of one-to-many optical terminal links	T7. Multiple terrestrial users per communications terminal	The investment is expected to accelerate the development timeline from 2035 to 2032

The benefit of the investments made by SDA is seen by updating the Development Matrix with the new timelines. Then, the rest of the matrices are recalculated, which produces the updated Delivery and Investment matrices shown in Figure 5-11 with changes highlighted in green. Compared to the Delivery Matrix in Figure 5-9, it is immediately apparent that capability C4, high data rates, is achieved ahead of the user's needs (values of negative one (-1) in 2026 and 2027). Capability C7, global operations, is achieved in 2032, which is still not in time to meet the stakeholder's needs. The Investment Matrix (right side of Figure 5-11) shows that technologies T8 and T7 are the sources for the delay in capability delivery.

Delivery Matrix, D

	C1	C2	C3	C4	C5	C6	C7
2021	0	0	0	0	0	0	0
2022	-1	0	0	0	0	0	0
2023	0	0	0	0	0	0	0
2024	0	0	0	0	0	0	0
2025	0	0	0	0	-1	0	0
2026	0	0	0	-1	-1	0	0
2027	0	0	0	-1	-1	0	0
2028	0	0	0	0	-1	0	1
2029	0	0	0	0	-1	0	1
2030	0	0	0	0	-1	0	1
2031	0	0	0	0	-1	0	1
2032	0	0	0	0	-1	0	0

Investment Matrix, I

	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032
T1	0	0	0	0	0	0	0	0	0	0	0	0
T15	0	0	0	0	0	0	0	0	0	0	0	0
T2	0	0	0	0	0	0	0	0	0	0	0	0
T4	0	0	0	0	0	0	0	0	0	0	0	0
T6	0	0	0	0	0	0	0	0	0	0	0	0
T3	0	0	0	0	0	0	0	0	0	0	0	0
T8	0	0	0	0	0	0	0	1	1	0	0	0
T10	0	0	0	0	0	0	0	0	0	0	0	0
T11	0	0	0	0	0	0	0	0	0	0	0	0
T12	0	0	0	0	0	0	0	0	0	0	0	0
T7	0	0	0	0	0	0	0	1	1	1	1	0
T13	0	0	0	0	0	0	0	0	0	0	0	0
T5	0	0	0	0	0	0	0	0	0	0	0	0
T9	0	0	0	0	0	0	0	0	0	0	0	0
T14	0	0	0	0	0	0	0	0	0	0	0	0
T22	0	0	0	0	0	0	0	0	0	0	0	0
T21	0	0	0	0	0	0	0	0	0	0	0	0
T16	0	0	0	0	0	0	0	0	0	0	0	0
T17	0	0	0	0	0	0	0	0	0	0	0	0
T18	0	0	0	0	0	0	0	0	0	0	0	0
T19	0	0	0	0	0	0	0	0	0	0	0	0
T20	0	0	0	0	0	0	0	0	0	0	0	0
T23	0	0	0	0	0	0	0	0	0	0	0	0
T24	0	0	0	0	0	0	0	0	0	0	0	0
T25	0	0	0	0	0	0	0	0	0	0	0	0
T26	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5-11. Updated Delivery and Investment Matrices based on investments

Technology T7 is the development of multiple terrestrial users per space-based optical terminal and the investment accelerates the development schedule to 2032. Technology T8 is the development of mobile and transportable optical ground terminals. Both of these technologies represent novel technologies that offer significant improvements in optical communications between space and ground. Even with the updated development timelines, the technologies may not be ready in time for inclusion in a procurement in 2028. Therefore, SDA will iterate on the LEAP process as a way to monitor the development progress of these technologies. SDA may choose to commit additional investments or to utilize the investments of partners to reduce the development timelines of these technologies. There may be other technologies that can achieve the same capability, such as including additional optical communications terminals on each satellite, and the results of the LEAP process indicate that these technologies should also be explored. The iterative nature of the LEAP process allows SDA to reconsider these investments and others in light of the changing needs of the Warfighter.

Technology T10 is the ability to operate optical communications terminals in all lighting conditions, including when the sun is in the field of regard. SDA's investments do not directly address technology T10. Instead, data collected during the execution of the SDA Tranches that will establish if there is truly a need for a technological improvement to the optical communications terminals or if there is a planning solution that can provide the required network connectivity while working around solar exclusion angles. Therefore, this technology is expected to be developed in 2025, after of SDA's Tranche 1 enters operations. As this example shows, technology can be advanced through materiel solutions or through non-materiel solutions, such as tactics, techniques, and procedures (TTP). LEAP is designed to aid technology advancement through materiel solutions and to help identify where TTP development may be a more appropriate solution.

5.3.1.1.5.4 LEAP Phase 4: Procure

SDA will monitor the progress of its investments to inform the capabilities that will be included in its Tranche 2 procurement and other future development efforts. SDA's goal is to utilize as much proven technology as possible in each Tranche. The use of proven technology reduces the complexity of the satellites. Satellite complexity correlates with larger costs and development timelines [199]; therefore, a less complex system allows faster development timelines and reduced risk in large program acquisitions. Multiple iterations of the LEAP process will occur prior to the procurement release for each SDA Tranche to ensure that technology is developed in parallel with Warfighter needs. The iterative nature of the SDA development cycle allows for the adjustment of the requirements for a specific iteration to account for cases where technologies are not delivered on schedule.

5.3.1.1.6 Conclusions and Future Work

In a rapid development program, like that used by SDA, it is necessary to ensure that technology is developed on a pace commensurate with the system delivery to reduce the need for NRE efforts as part of the development contract. However, when the system needs to incorporate new capabilities, it is often difficult to understand and track where and when outside development must happen to enable the rapid deployment of production systems.

The LEAP process provides a mechanism to identify the functional and temporal dependencies between the required capabilities of a system and the technologies that enable those capabilities. SDA has applied the LEAP process to identify critical technologies that require investments to accelerate their development schedules. By using this process, SDA is ensuring that the technological landscape is ready to support the Tranche procurements that delivers the capabilities required by the Warfighter when they need them.

The LEAP process will continue to be refined through use at SDA as it builds out the PWSA. The usage of LEAP for the selection of optical investments represents the first use of the newly developed process. Future developments on the process itself are defined in Kleinwaks et al. [160] and include implementing probabilistic estimations of the values in the Development Matrix. As it is currently defined, the Development Matrix assumes that a technology either is or is not developed in a particular time period. Modeling the probability of the technology being developed will produce a more usable process for the stakeholders. Additional future work will include investigating the scalability of the LEAP process to requirement sets beyond the specific set defined herein. The LEAP process is designed to be scalable, relying on matrix math to enable the rapid processing of large numbers of tactical capabilities and supporting technologies. Application beyond a small subset of capability will demonstrate this scalability and potential for widespread

use. SDA's rapid development schedule provides an excellent testbed for the process and will guide its future refinement.

5.3.1.2 Application of Quantitative LEAP to Iterative Ground System Development

SDA is a schedule focused organization [46], producing and launching significant numbers of satellites in short time frames. As stated in Section 1.1.1.2, SDA delivers two-year tranches of satellites on two-year cycles, using a spiral development cycle where each tranche improves upon the capability of the previous tranche. This development cycle enables rapid fielding of critical capabilities [46]. In addition to the satellites, SDA also develops the corresponding ground system for each tranche. The ground systems are cyber-physical systems, where there is a large software component and a large hardware component, both of which must be managed and delivered on time.

Within Tranche 1 of the SDA system, the ground system is developed using an incremental development approach. The first several increments focus on capability that is delivered on the ground and is gated by the ground readiness review (GRR) and the first satellite launch. On-orbit capability is increased through three increments, called Crawl, Walk, and Run. Within each on-orbit increment, additional capabilities are brought into the ground system and additional levels of interoperability are implemented in the on-orbit constellation. Although the SDA satellites are the primary mechanism for delivering capability to the user, the ground system is a critical enabler of the satellites. The ground system must be developed on similar timelines and the resulting schedule pressure could easily create technical debt if it is not carefully managed. Therefore, this ground system is a logical choice of a system that could benefit from implementation of the LEAP process.

The SDA ground system developer implements the Scale Agile Framework (SAFe) methodology [200], using 3-month program increments (PI) for planning and executing

development cycles. As part of the SAFe methodology, each feature is reviewed at each PI planning event and assigned a business value. The features with the highest business value are selected for implementation in that PI. This method leads to a value-based development cycle, which is susceptible to technical debt, as discussed in Section 3.3.3.1.3. The LEAP process was applied in the middle of the ground system development cycle, starting at PI-3. It was first used to qualitatively assess the state of delivery of the system. Next, the quantitative LEAP process was applied to understand the driving technologies behind any late capability deliveries and the risk of technical bankruptcy. The results of these applications are described in the rest of this section.

The application of the LEAP process began with the List phase and an independent assessment of the system needs and a decomposition into capabilities by the system stakeholders. At the time of evaluation, the ground system developer had decomposed their requirements into their own set of capabilities and features. The set of capabilities identified by the stakeholders largely aligned with those in use by the ground system developer after adjusting for differences in nomenclature. These capabilities were mapped to the technologies provided by the ground system developer. This mapping resulted in the creation of the Functional Matrix and the developer's PI schedule produced the Development Matrix.

Since the ground system development had already begun, the stakeholders desired to get a quick understanding of the potential for technical bankruptcy of the system – was it on track to deliver on time or was it behind? Therefore, the qualitative LEAP process was applied. The qualitative LEAP process can be used to provide a rapid assessment of state of system delivery and a starting point for investigating which technologies may be driving late capability delivery. These results can then be used as a starting point for explorations into why the capabilities are late. The Evaluate phase began with the creation of the Need Matrix using the major increments (GRR, Launch,

Crawl, Walk, and Run) as the time periods. The ground system developer program increments were mapped to these major increments by associating the end date of the PI with the need date for the major increment. The resulting Delivery Matrix, shown in Figure 5-12, was calculated through the process. In the qualitative LEAP process, a value of one (1) indicates that a capability is late to need. From this matrix, it appeared that very few capabilities were on track to deliver on time, as indicated by the red cells.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26	C27	C28
GRR	1	1	1	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Launch	1	1	1	1	1	0	1	1	1	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Checkout	1	1	1	1	1	0	1	1	1	0	1	1	-1	1	1	0	0	0	1	1	0	1	1	1	1	1	1	0
Crawl	1	1	1	1	1	1	1	1	1	1	1	0	-1	1	1	-1	-1	-1	1	1	0	1	1	1	1	1	1	0
Walk	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	-1
Run	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	C29	C30	C31	C32	C33	C34	C35	C36	C37	C38	C39	C40	C41	C42	C43	C44	C45	C46	C47	C48	C49	C50	C51	C52	C53	C54	C55	C56
GRR	0	0	-1	-1	0	0	0	0	0	0	0	0	-1	1	1	1	0	1	1	1	1	0	0	0	1	1	0	0
Launch	0	0	0	0	1	1	0	0	0	0	0	0	-1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	0
Checkout	1	0	0	0	1	1	1	0	0	1	1	1	-1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1
Crawl	1	1	0	0	1	1	1	0	0	1	1	1	-1	1	1	1	1	1	1	1	1	0	0	1	0	0	0	1
Walk	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
Run	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5-12. Initial qualitative LEAP Delivery Matrix

The problems identified with the system delivery by the Delivery Matrix prompted discussions with the ground system developer. The developer's original plan included executing testing and evaluation of new software capabilities for a full year prior to declaring them complete. However, SDA's expectations were that some of this testing would occur with the on-orbit assets. Therefore, the stakeholders expected that the technologies would be considered complete once they are integrated into the deployed system, and not when the testing with the on-orbit assets is completed. This early conversation prevented a form of domain technical debt – where miscommunications about the needs of the stakeholders result in an improperly implemented system. Without this analysis, the system would have appeared to be late, potentially driving additional pressure from the stakeholders.

With the new definitions of task completeness in place, the quantitative LEAP process was executed. Delivery probabilities were assigned based on the priority of the features being developed in each PI as shown in Table 5-4. Low priority features assigned to PIs are still planned to be completed by the ground system developer, but they will be out-prioritized by other work if required. Therefore, even low priority features have a relatively high probability of completion. If a feature was not planned to be worked on within a PI, then it was assigned a probability of zero.

Table 5-4. Probabilities of completing features based on feature priority

Priority in PI	Probability of Completing within the PI
Low	75%
Medium	85%
High	100%

The Delivery Matrix resulting from the application of the quantitative LEAP process is shown in Figure 5-13. The delivery probability value is in the cell (from zero (0) to one (1)) are highlighted from red (lowest probability) to green (highest probability). If a capability is not needed by the stakeholder in a particular increment, then the cell is white and the value is negative one (-1). Examining this figure, it can be seen that some capabilities are now delivered on time, indicated by the dark green cell with a value of one (1) below a white cell in a column. This shift in results from the qualitative LEAP analysis is due to the recharacterization of the delivery definition following the initial analysis. However, even though the end state of the system is likely to deliver on time, several capabilities still have low probabilities of delivering by the stakeholder need date, indicated by red and orange cells.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26	C27	C28
GRR	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1
Launch	0	0	0	0	0.20	-1	0.36	0.36	0.27	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0.75	-1	-1	-1	-1
Checkout	0	0	0	0	0.24	-1	0.42	0.42	0.32	-1	0.56	1	-1	0.75	0	-1	-1	-1	0.56	0	-1	0.56	0	0.75	0.75	0.42	0.42	-1
Crawl	0.18	0.18	0.18	0.18	0.24	0.42	0.42	0.42	0.32	0.13	0.56	1	-1	0.75	0.42	-1	-1	-1	0.56	0.32	-1	0.56	0.32	0.75	0.7-15	0.42	0.42	-1
Walk	0.33	0.33	0.33	0.33	0.44	0.54	0.61	0.61	0.52	0.28	0.72	1	1	0.85	0.54	1	1	1	0.72	0.46	0.85	0.72	0.46	0.85	0.85	0.61	0.61	-1
Run	0.85	0.85	0.85	0.85	1	0.85	1	1	1	0.85	1	1	1	1	0.85	1	1	1	1	0.85	1	1	1	1	1	1	1	1

	C29	C30	C31	C32	C33	C34	C35	C36	C37	C38	C39	C40	C41	C42	C43	C44	C45	C46	C47	C48	C49	C50	C51	C52	C53	C54	C55	C56
GRR	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	-1	0	0	0	0	1	-1	-1	0.85	0.85	-1	-1
Launch	-1	-1	1	1	0.64	0.64	-1	-1	-1	-1	-1	-1	-1	0.36	0.36	0.36	0.64	0	0	0	0	1	1	-1	1	1	-1	-1
Checkout	0.75	-1	1	1	0.75	0.75	0.75	-1	-1	0.75	0.75	0.75	-1	0.42	0.42	0.42	0.75	0	0	0	0	1	1	-1	1	1	-1	0.56
Crawl	0.75	0.75	1	1	0.75	0.75	0.75	-1	-1	0.75	0.75	0.75	-1	0.42	0.42	0.42	0.75	0.42	0.75	0.75	0.75	1	1	0.56	1	1	-1	0.56
Walk	0.85	0.85	1	1	0.85	0.85	0.85	0.85	0.72	0.85	0.85	0.85	-1	0.61	0.61	0.61	0.85	0.54	0.75	0.75	0.75	1	1	0.72	1	1	-1	0.72
Run	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0.85	0.85	0.85	0.85	1	1	1	1	1	0.85	1

Figure 5-13. Delivery Matrix from quantitative LEAP application

To determine the driving technologies behind the late deliveries, the Investment Matrix shown in Figure 5-14 was calculated. In the Investment Matrix, higher values are less desirable – they indicate that the technology has a higher likelihood of causing late delivery of one or more capabilities in each time period. Therefore, the cells with values greater than one (1) in Figure 5-14 are highlighted in red. The highest scoring cells in the matrix are shown in red. Clearly, Technologies 11 and 24 are driving factors for the late delivery of the capabilities required at GRR as they had the highest scores in the Investment Matrix.

	GRR	Launch	Checkout	Crawl	Walk	Run
T1	0	0	0	0	0	0
T2	0	0	0	0	0	0
T3	0	0	0	0	0	0
T4	0	0	0	0	0	0
T5	1.05	1.05	0	0	0	0
T6	0.15	0	0	0	0	0
T7	0	0	0	0	0	0
T8	5	1	1	1	0.75	0
T9	0	0	0	0	0	0
T10	0	0	0	0	0	0
T11	7	1.75	1.75	1.5	1.2	0
T12	0	0.25	0	0	0.15	0
T13	0	1	0	0	0.25	0
T14	0	0	0	0	0	0
T15	0.6	0	0	0	0	0
T16	0	0	0	0	0	0
T17	0	0	0	0	0	0
T18	3	0.75	0.5	0	0	0
T19	3	0.5	0.5	0	0	0
T20	3	0.5	0.25	0	0	0
T21	3	0.5	0.5	0.5	0.45	0
T22	3	0.75	0.25	0	0.15	0
T23	0	0	0	0	0	0
T24	7	6	4	0.5	0.5	0.3
T25	0	0	0	0	0	0

Figure 5-14. Investment Matrix from quantitative LEAP application

Using the results from the Investment Matrix, the stakeholders came into the next program increment planning meeting with new priorities and assessments of business value. Technology 24 involved the creation of a user interface for external users to interact with the system. After discussing this specific technology with the system developer, potential requirements technical debt was identified – the developer's interpretation of the requirement did not match the stakeholder's interpretation of the requirement and would not have satisfied the user's needs. This requirements debt was the reason for the late delivery – the system developer had not planned on adding features to the user interface that the stakeholder deemed critical and, from the stakeholder's view, could not complete the technology, which forced the associated capabilities to be considered late. Therefore, an improved definition of the user interface requirements was created, and the appropriate scope was added to the contract, including the prioritization of the activities in the next program increment. While removing this potential technical debt required the use of additional funds, the early identification mitigated the potential long-term consequences.

In this particular case, the misunderstanding of the stakeholder needs could be, and was, identified in parallel with the LEAP process. However, the LEAP process provides a structured approach for assessing potential technical debt within the development cycle, enabling repeated and objective application. Having this process ensures that all features will be considered for their ability to introduce technical debt into the system without relying on the intuition of a few capable engineers. Additionally, the LEAP process enables an estimate of the return-on-investment made by accelerating the development of this technology. By shifting the prioritization of Technology 24 development to High in each time period, the change in delivery probability for each of the capabilities dependent upon Technology 24 can be calculated. Figure 5-15 shows the change in delivery probability for Capability 46 across each time period due to the changes in Technology

24 development timelines. In this case, the largest increase in delivery probability is seen in the earlier phases. Additional technologies would need to be accelerated to ensure delivery by the launch phase, but the investments in Technology 24 have significantly increased the likelihood of existing capability in this time phase. Similar values could be calculated for each affected capability and aggregated for the entire system.

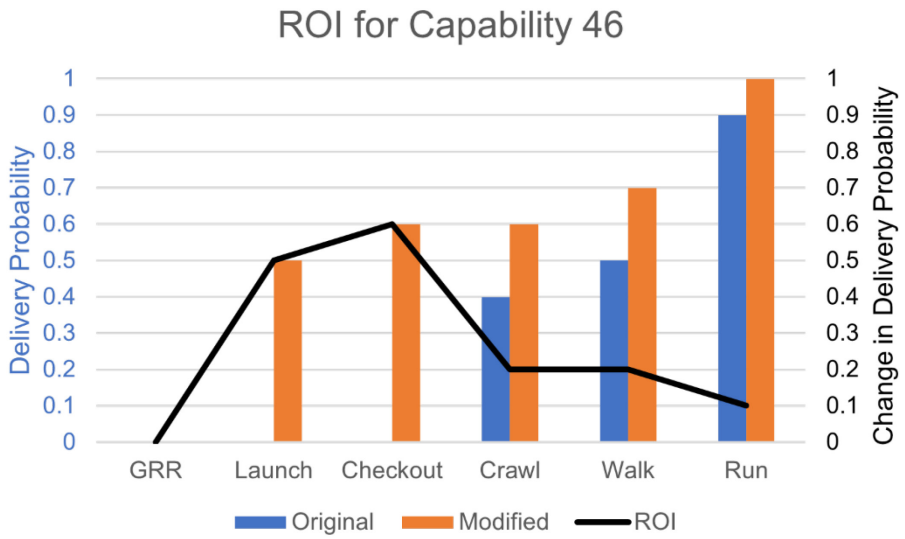

Figure 5-15. Return-on-Investment calculated with LEAP

Technology 11, which also has a high score in the Investment Matrix, provides an example of where the process revealed a potential problem that was not uncovered by the system developers and stakeholders prior to the application of the LEAP process. Technology 11 involves functions that were well understood but planned for later program increments that would not likely complete prior to GRR. However, the stakeholders stated that these functions were necessary to support capabilities that they deemed required for GRR. The LEAP process identified the disconnect in the timelines and that Technology 11 was the driving factor. Using this information, the stakeholders and the system developer were able to have productive conversations about the capabilities that were really required at GRR. In this case, the stakeholders desired to have this capability at GRR, however it would not be used until after launch. Therefore, the stakeholders

were convinced to shift their need date to align with the developer's schedule to avoid additional cost increases.

This example shows the full path through the LEAP process as follows.

1. The stakeholders identified their capability need dates and supporting technologies (List)
2. The stakeholders, in collaboration with the ground system developer, evaluated the probability of the system, as planned, to deliver the capabilities in accordance with the need dates (Evaluate)
3. The stakeholders identified which technologies needed to have need dates adjusted and decided on a plan (Achieve)
4. The stakeholders adjusted their procurement requirements to minimize cost (Procure)

Through the use of the LEAP process, the ground system developer and the stakeholders were able to identify disconnects in the delivery cadence and come to a mutually agreed-upon path to system delivery. In doing so, the risk of introducing technical debt was reduced. The change in the procurement strategy reduces the schedule pressure on the ground system developer to deliver capabilities ahead of plan and therefore the risk of taking shortcuts and introducing technical debt is reduced.

Similar processes were used to discuss all of the potentially late technologies identified in the delivery matrix at the PI planning meeting. Having the LEAP Delivery and Investment Matrices available enabled more fruitful discussions between the stakeholders and ground system developer, and resulted in reprioritization of other key technologies in the development cycle.

5.3.1.3 Review of the Example Applications

The above applications show how the LEAP process can be used to identify investments that are critical to meeting future capability needs. The initial Evaluation phase identified technologies that would be late to need. Utilizing the Achieve phase of the process, it is possible to determine which capabilities would benefit from investments in these technologies, following the "Driving Technologies Independent" pathway identified in Figure 5-4. By making investments to develop critical technologies, the decision maker can prevent technical debt from building up within their system and can therefore avoid technical bankruptcy.

The LEAP process can also be used to evaluate the consequences of an investment decision between different technologies. For example, decision makers will often have to make choices in a budget constrained environment. Such environments prohibit the ability to invest in all technologies and often require compromises. By clearly associating the stakeholder needs with the technology development timelines, the LEAP process enables an assessment of the consequences of those decisions.

Finally, the application of the LEAP process to the ground system development demonstrates how it can be used to proactively identify potential technical debt. By using the LEAP process to identify disconnects between the system developers and the stakeholders, technical debt was prevented from entering the system, reducing the risk of future problems within the system development.

5.4 Presentation of the Process in simplified terms

Task 4.3 is to produce a simplified way of presenting and communicating the LEAP process. Expressing a complicated process through simplified graphics and concepts enhances its utility as a communication device. The technical debt metaphor was initially created for this exact reason,

as Cunningham sought to find a method to communicate needs to refactor software to his management [17]. The LEAP process attempts to bridge the divide between the technical staff and management and stakeholders and therefore needs to present a simplified way of discussing the technical concepts contained in the process.

The presentation of the LEAP process starts with the intentional choice of the acronym – the names List, Evaluate, Achieve, and Procure convey the intent of each step in easy-to-understand terminology. These names help to understand what the process is intended to do. The List phase lists out the capabilities and technologies. The Evaluate phase evaluates the current state of the system and the stakeholder needs. The Achieve phase accelerates development as required, and the Procure phase procures a new release.

Graphically, the LEAP process has been depicted as a set of interconnected matrices, as shown in Figure 3-4. While this depiction helps to understand the technical details of the process, it is not a simple, easy-to-grasp graphical representation of the process. Figure 5-16, originally presented at the *2023 INCOSE International Symposium* [201], shows the LEAP process as an inherently iterative process, but does not provide any details of the events associated with each step.

Figure 5-16. Simplified LEAP process description

Figure 5-17 provides an alternative depiction of the process. It removes the mathematical formulas and the matrices from the graphics and instead shows simple information flow through the steps in the process. In this graphic, the steps occur from bottom to top and color coding is used to match the step to the graphic. First, the List step establishes the capabilities and their supporting technologies, creating the x-axis of the graph. Next, the Evaluate step determines the availability timelines for each capability, locating the orange circles on the graph. The Achieve step accelerates timelines for selected capabilities by investing in technology development, converting the orange circles into blue squares. Finally, the Procure step selects the set of capabilities to include in each release for a given time period, shown as the circles and squares included in the gray ellipse.

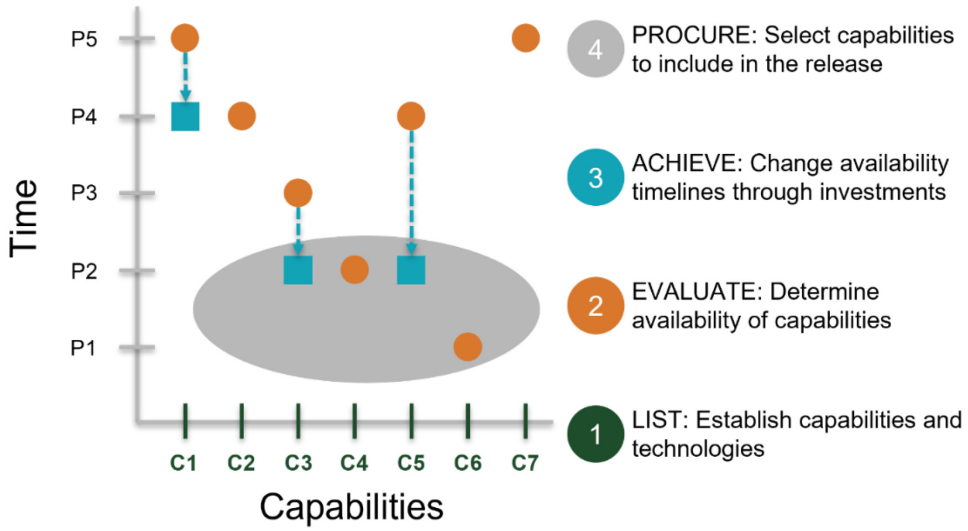

Figure 5-17. Simplified LEAP process model

Figure 5-17 provides a simpler conceptual version of the LEAP process, using colors and steps to indicate the passage of time through each step of the process. This representation conveys the overall goals of the process without using the matrices, which makes the rationale for the process easier to discuss.

The construction of the Delivery Matrix enables multiple viewing options to convey the likelihood of delivering capabilities on time to the stakeholders. Time-based graphs, such as those shown in Figure 5-18, can easily be created from the matrix outputs. The Delivery Matrix only contains positive values for when the capability is needed, and therefore the graph provides an instantaneous view of the initial and time-based likelihood of satisfying stakeholder needs.

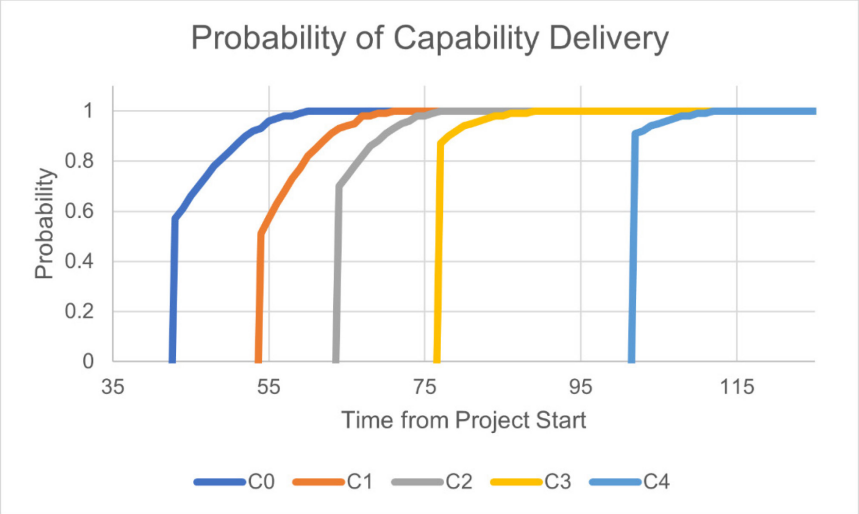

Figure 5-18. Plot of delivery timelines

Similarly, the Availability Matrix can also be plotted, as shown in Figure 5-19. The Availability Matrix contains information on when each capability is likely to be available, independent of the stakeholder needs. This graph shows the likelihood of delivering each capability over time, regardless of when it is needed.

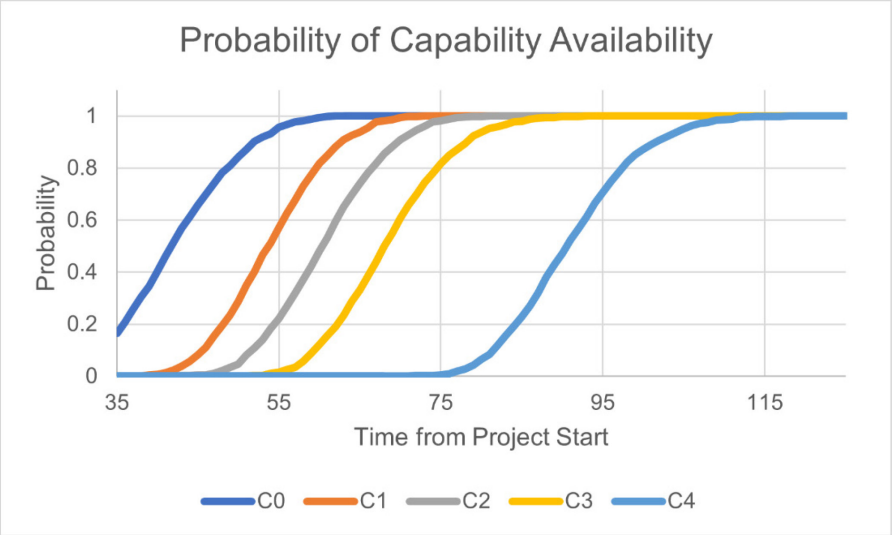

Figure 5-19. Plot of availability timelines

These visualizations quickly show the state of stakeholder satisfaction to both technical and non-technical stakeholders. Simple outputs like these increase the ability of the LEAP process to convey critical information to stakeholders to enable investments that minimize the risk of technical bankruptcy.

For the system developer, the matrix-based nature of the process enables simple implementation. Appendix A includes Python scripts implementing both the qualitative and quantitative LEAP processes.

5.5 Conclusion

This chapter focuses on technical bankruptcy, presenting a definition of technical bankruptcy in the context of the LEAP process and providing example usages of the LEAP process in real-world scenarios. These examples identify how the LEAP process can be used to avoid technical bankruptcy through use of both the qualitative and quantitative versions of the process, thereby addressing RQ4: *How can the process and model be used to avoid technical bankruptcy?* Figure 5-4 shows a process by which the potential for technical debt is identified through iterative use of the LEAP process during system development. Tasks which have a larger potential for inducing

technical debt are identified and therefore can be controlled. Tasks which are late to need are highlighted through the investment matrix, allowing the diversion of resources or alteration of release plans to prevent the occurrence of the associated technical debt. Finally, the process can be used to evaluate release plans and to present results to the stakeholders showing the rationale behind the construction of the release plans.

The applications of the LEAP process at SDA provide real-world examples of proactive identification of technical debt. Advantageous technology investments were identified such that their development timelines could be accelerated to support larger-scale procurements. Sources of technical debt were identified prior to their introduction into the system, reducing the cost of correcting these issues in later iterations. These applications reduced the risk of accumulated technical debt and demonstrate the utility of the LEAP process.

CHAPTER 6 – CONCLUSIONS AND FUTURE WORK

6.1 Research Contributions

This dissertation contributed to the state of the art of systems engineering by developing the following items: 1) an understanding of the prevalence of technical debt within the field, 2) a recommended ontology for furthering technical debt discussions, and 3) a defined process for determining system dependencies that may be susceptible to technical debt in order to limit the potential for technical bankruptcy.

Technical debt, while present in systems engineering [18], was found to be not-well researched in the field [19]. Existing research on technical debt was found to focus on specific areas of systems engineering, such as automated production systems [108]. This research contributed novel empirical data on the prevalence of systems engineering by surveying systems engineers across multiple disciplines [18]. Using the results from the systematic literature review and the empirical survey, a new ontology for discussing technical debt within the context of systems engineering was created [21]. Published discussions of technical debt tend to focus on creating taxonomies for classifying technical debt. This ontology is the first that the author is aware of to focus on producing concise definitions of technical debt terms with specific applications to systems engineering. Adoption of this ontology will enable practitioners to share methods for technical debt management and mitigation by using concise and unambiguous terminology.

Managing technical debt, and avoiding technical bankruptcy is found to be a real problem in systems engineering. Systems fail due to the accumulation of technical debt, especially when using stakeholder value-driven development methodologies that force technical compromises to be made at the expense of preferred development schedules. Through the definition of the LEAP process,

this research has contributed a novel method of combining the temporal and functional dependencies of a system to identify the time-phased capability to satisfy stakeholder needs [160]. The quantitative LEAP method discussed in Chapter 4 provides a probabilistic approach, including a new method of estimating project schedules based on technical debt [169]. This process, when used within iterative development, identifies which technologies will require additional investment ahead of the need to incorporate those technologies into the system development. These results enable a system developer to minimize the non-recurring engineering on their system by scoping the release to include components that meet specified readiness threshold. By investing in technology development outside the release cycle, the system developer can ensure that the needed components are ready in time to include in larger system procurements. The LEAP process provided mathematical identification and integration of these components.

The LEAP process is developed as a decision support system for release planning by identifying the probability of delivering capabilities on time, including technical debt estimates. While traditional schedule analysis can provide predictions of completion dates, the inclusion of technical debt's impacts on successor task duration in the schedule process is a new contribution to schedule analysis. Additionally, traditional schedule analysis can be used to define when a system is complete, but does not directly associate system completion with the schedule of stakeholder needs. The LEAP process provides a mathematical relationship between the capability delivery and the stakeholder needs. This linkage enables the LEAP process to be used as a decision support system for iterative development – different release plans can be modeled in the LEAP process to determine the different outcomes which can lead to an evaluation of the return-on-investment for release planning decisions.

The LEAP process was applied at the Space Development Agency to assess potential investments in optical communications terminal technology and to evaluate and prioritize tasks within the iterative development of the ground segment for the satellite constellation. This translational research task was able to measure the costs and benefits of LEAP as a program of quantifying, managing, and planning for technology development. Both the qualitative and quantitative LEAP processes were applied, enabling rapid identification of technologies whose development required acceleration to meet the stakeholder needs. The quantitative LEAP application revealed instances of technical debt early in the development process, enabling the technical debt to be addressed before it compounded into a larger problem. These results are novel in that similar on-site studies of proactive technical debt identification have not been performed to date. The application of the LEAP process to the rapid development programs at SDA identified issues that could have produced undesirable outcomes.

This research program has produced the following peer-reviewed publications:

1. H. Kleinwaks, A. Batchelor and T. H. Bradley, "Technical Debt in Systems Engineering - A Systematic Literature Review," *Systems Engineering,* vol. 26, no. 5, pp. 675-687, 2023. [19]
2. H. Kleinwaks, A. Batchelor and T. H. Bradley, "An Empirical Survey on the Prevalence of Technical Debt in Systems Engineering," *INCOSE International Symposium,* vol. 33, no. 1, pp. 1640-1658, 2023. [18]
3. H. Kleinwaks, A. Batchelor, T. H. Bradley, M. Rich and J. F. Turner, "LEAP - A process for identifying potential technical debt in iterative system development," *INCOSE International Symposium,* vol. 33, no. 1, pp. 535-553, 2023. [160]

4. H. Kleinwaks, M. Rich, M. C. Butterfield and J. F. Turner, "LEAPing Ahead - The Space Development Agency's Method for Planning for the Future," *INCOSE International Symposium,* vol. 33, no. 1, pp. 925-942, 2023. [167]
5. H. Kleinwaks, A. Batchelor and T. H. Bradley, "An Ontology for Technical Debt in Systems Engineering," *IEEE Open Journal of Systems Engineering,* vol. 1, pp. 111-122, September 2023. [21]

At the time of submission of this dissertation, the following manuscripts have been submitted for publication:

1. H. Kleinwaks, A. Batchelor and T. H. Bradley, "Predicting the Dynamics of Earned Value Creation in the Presence of Technical Debt," *Submitted to IEEE Access*, 29 July 2023. [169]
2. H. Kleinwaks, A. Batchelor and T. H. Bradley, "Probabilistic Enhancement to the LEAP Process for Identifying Technical Debt in Iterative System Development," *Submitted to IEEE Access,* 9 Sep 2023. [186]

6.2 Future Work

This dissertation introduced a new technical debt ontology and defined and developed the LEAP process. Additional work in each of these areas can further advance the state of the art in the field.

The technical debt ontology should be evaluated for its costs and benefits through practical applications in systems engineering domains. The ontology can be introduced to systems engineering practitioners and their usage of it can be evaluated. Research questions such as “Does the use of the technical debt ontology impact the occurrence of technical debt?” can be evaluated

and new technical debt identification and mitigation techniques identified as a result. Based on the usage of the ontology, the definitions contained within should be updated and additional definitions added as required.

The LEAP process should also be further evaluated through practical applications. Empirical evidence of its utility can be gathered through multiple applications in various systems engineering contexts. Different applications of the LEAP process can be explored, such as the application to additional industries and fields and the utility of the LEAP process in return-on-investment studies.

The LEAP process can be improved by adding prioritization matrices and optimizing the process. Prioritization matrices were explicitly excluded from this dissertation since prioritization has the potential to skew the results based on the supplied priority values. However, with the baseline process defined, the Need Matrix could be augmented by prioritization of the needs and any adjustments to the ensuing equations identified. The result could then be used to optimize the development order of technologies within the LEAP process to minimize the impact of the late delivery of capabilities, which would be associated with the priority of the needs. This optimization would convert the LEAP process from a decision support system for release planning to a release planning tool.

6.3 Conclusion

In the volatile and uncertain market, system developers often face pressure from stakeholders to release high-performance systems faster and cheaper. Such pressures can result in the system developer making technical compromises, thereby introducing technical debt into the system. If the technical debt remains in the system, then it can accumulate, eventually resulting in technical bankruptcy. To limit this risk, technical debt must be defined, predicted, and managed in both the

temporal and functional dimensions, and there must be the knowledge, processes, and tools to do so.

This dissertation addressed this problem by first defining technical debt in the context of systems engineering. Common definitions and terminology enable systems engineers to leverage work done by other systems engineers to develop processes and techniques for identifying and managing technical debt. This dissertation also introduced the LEAP process as a decision support system for accounting for technical debt within release planning. Proactive methods that identify technical debt at the time that technical compromises are made are critical to avoid technical bankruptcy. It is by planning ahead that the impacts of decisions can be estimated and mitigated through the application of additional resources or through the selection of alternative choices.

Application of the LEAP process, as seen through the examples presented in this dissertation, enables both a proactive assessment of the readiness of a system to begin development as well as an assessment of potential release plans and design choices. These assessments can be used to simply communicate with stakeholders on the repercussions of their decisions, identify where technical debt may occur, and identify when a system may be on the verge of bankruptcy. Armed with these data sets, system developers can work with the stakeholders to determine the right technical compromises to make or not to make to ensure that the system development continues on plan.

REFERENCES

[1] M. A. G. Darrin and W. S. Devereux, "The Agile Manifesto, design thinking, and systems engineering," in *2017 Annual IEEE International Systems Conference (SysCon)*, 2017.

[2] S. Koolmanojwong and J. A. Lane, "Enablers and inhibitors of expediting systems engineering," *Procedia Computer Science,* vol. 16, pp. 483-491, 2013.

[3] T. S. Schmidt, S. Weiss and K. Paetzold, "Expected vs. real effects of agile development of physical products: Apportioning the hype," in *DS 92: Proceedings of the DESIGN 2018 15th International Design Conference*, 2018.

[4] W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proceedings of the 9th international conference on Software Engineering*, 1987.

[5] R. F. Bordley and J. M. Keisler, "Managing Systems Engineering Projects with Uncertain Requirements," in *INCOSE International Symposium*, Orlando, FL, 2019.

[6] H. Dunlap and D. Chesebrough, "Transforming Our Systems Engineering Approach Using Digital Technology," National Defense Industrial Association, 4 October 2021. [Online]. Available: https://www.nationaldefensemagazine.org/articles/2021/10/4/transforming-our-systems-engineering-approach-using-digital-technology. [Accessed 10 April 2022].

[7] J. A. Lane, S. Koolmanojwong and B. Boehm, "4.6.3 Affordable Systems: Balancing the Capability, Schedule, Flexibility, and Technical Debt Tradespace," in *INCOSE International Symposium*, 2013.

[8] R. Haberfellner and O. de Weck, "Agile SYSTEMS ENGINEERING versus AGILE SYSTEMS engineering," in *Fifteenth Annual International Symposium of the International Council On Systems Engineering (INCOSE)*, 2005.

[9] A. P. Schulz and E. Fricke, "Incorporating flexibility, agility, robustness, and adaptability wihtin the design of integrated systems - key to success?," in *Gateway to the New Millenium. 18th Digial Avionics System Conference Proceedings*, 1999.

[10] A. M. Ross, D. H. Rhodes and D. E. Hastings, "Defining Changeability: Reconciling Flexibility, Adaptability, Scalability, Modifiability, and Robustness for Maintaining System Lifecycle Value," *Systems Engineering,* vol. 11, no. 3, pp. 246-262, 2008.

[11] United States Government Accountability Office, "DoD Space Acquisitions: Including Users Early and Often in Software Development Could Benefit Programs," United States Government Accountability Office, Washington, D.C., 2019.

[12] C. Larman and V. R. Basili, "Iterative and Incremental Development: A Brief History," *Computer,* vol. 36, no. 6, pp. 47-56, 2003.

[13] B. P. Douglass, Agile Systems Engineering, Waltham: Morgan Kaufmanm, 2016.

[14] A. Atzberger, C. Gerling, J. Schrof, T. S. Schmidt, S. Weiss and K. Paetzold, "Evolution of the Hype around Agile Hardware Development," in *2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, 2019.

[15] G. Robiolo, E. Scott, S. Matalonga and M. Felderer, "Technical Debt and Waste in Non-functional Requirements Documentation: An Exploratory Study," in *International Conference on Product-Focused Software Process Improvement*, 2019.

[16] R. L. Nord, I. Ozkaya, P. Kruchten and M. Gonzalez-Rojas, "In Search of a Metric for Managing Architectural Technical Debt," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2012.

[17] W. Cunningham, "The WyCash Portfolio Management System," 26 March 1992. [Online]. Available: http://c2.com/doc/oopsla92.html. [Accessed 29 January 2022].

[18] H. Kleinwaks, A. Batchelor and T. H. Bradley, "An Empirical Survey on the Prevalence of Technical Debt in Systems Engineering," *INCOSE International Symposium,* vol. 33, no. 1, pp. 1640-1658, 2023.

[19] H. Kleinwaks, A. Batchelor and T. H. Bradley, "Technical Debt in Systems Engineering - A Systematic Literature Review," *Systems Engineering,* vol. 26, no. 5, pp. 675-687, 2023.

[20] S. Malakuti and J. Heuschkel, "The Need for Holistic Technical Debt Management across the Value Stream: Lessons Learnt and Open Challenges," in *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2021.

[21] H. Kleinwaks, A. Batchelor and T. H. Bradley, "An Ontology for Technical Debt in Systems Engineering," *IEEE Open Journal of Systems Engineering,* vol. 1, pp. 111-122, September 2023.

[22] B. W. Boehm, "A Spiral Model of Software Devleopment and Enhancement," *Computer,* pp. 61-72, May 1988.

[23] H. Mooz and K. Forsberg, "Clearing the Confusion About Spiral/Evolutionary Development," *INCOSE International Symposium,* vol. 14, no. 1, pp. 1675-1688, June 2004.

[24] M. Cohn, Succeeding with Agile: Software Development Using Scrum, Upper Saddle River, NJ: Addison-Wesley, 2010.

[25] B. W. Boehm, J. A. Lane, S. Koolmanojwong and R. Turner, The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software, Upper Saddle River, NJ: Addison-Wesley, 2014.

[26] K. Forsberg and H. Mooz, "Application of the 'Vee' to Incremental and Evolutionary Development," *INCOSE International Symposium,* vol. 5, no. 1, pp. 848-855, 1995.

[27] P. Laplante, What Every Engineer Should Know about Software Engineering, Boca Raton, FL: CRC Press, 2007.

[28] O. Oni and E. Letier, "Analyzing Uncertainty in Release Planning: A Method and Experiment for Fixed-Date Release Cycles," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 31, no. 2, pp. 1-39, 2021.

[29] D. Dalcher, O. Benediktsson and H. Thorbergsson, "Development Life Cycle Management: A Multiproject Experiment," in *12th IEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, 2005.

[30] Scaled Agile, Inc., "Weighted Shortest Job First," 10 February 2021. [Online]. Available: https://www.scaledagileframework.com/wsjf/. [Accessed 2 November 2022].

[31] M. Roy, N. Deb, A. Cortesi, R. Chaki and N. Chaki, "Requirement-oriented risk management for incremental software development," *Innovations in Systems and Software Engineering,* vol. 17, no. 3, pp. 187-204, 2021.

[32] A. Etemadi and J. Kamp, "Acquisition strategy factors related to faster defense acquisitions," *Systems Engineering,* vol. 25, no. 2, pp. 144-156, 2022.

[33] A. Etemadi and J. Kamp, "Market and contractor factors affecting rapid acquisition strategies," *Systems Engineering,* vol. 24, no. 4, pp. 250-265, 2021.

[34] United States Department of Defense, Office of the Undersecretary of Defense for Research and Engineering, Office of the Deputy Directory for Engineering, "Systems Engineering Guidebook," United States Department of Defense, Washington, D.C., 2022.

[35] United States Government Accountability Office, "DOD Cost Overruns: Trends in Nunn-McCurdy Breaches and Tools to Manage Weapon Systems Acquisition Costs," United States Government Accountability Office, Washington, D.C., 2011.

[36] J. Ferrara, "DoD's 5000 Documents: Evolution and Change in Defense Acquisition Policy," *Acquisition Review Quarterly,* pp. 109-130, 1996.

[37] United States Department of Defense, Office of the Undersecretary of Defense (Acquisition, Technology, and Logistics), "Department of Defense Instruction Number 5000.2," United States Department of Defense, 2003.

[38] United States Department of Defense, Office of the Undersecretary of Defense for Acquisition, Technology, and Logistics, "Department of Defense Instruction Number 5000.02," United States Department of Defense, 2008.

[39] United States Department of Defense, Office of the Undersecretary of Defense for Acqusition, Technology, and Logistics, "Department of Defense Instruction Interim Number 5000.02," United States Department of Defense, 2013.

[40] United States Department of Defense, Office of the Undersecretary of Defense for Acquisition and Sustainment, "DoD Directive 5000.01 The Defense Acquisition System," United States Department of Deferense, Washington, D.C., 2020.

[41] J. S. Gansler, W. Lucyshyn and A. Spiers, "Using Spiral Development to Reduce Acqusition Cycle Times," Center for Public Policy and Private Enterprise. University of Maryland School of Public Policy, 2008.

[42] United States Department of Defense, "Military Standard Software Development and Documentation," United States Department of Defense, 1994.

[43] United States Department of Defense, Office of the Undersecretary of Defense (Acquisition, Technology, and Logistics), "DoD 5000.2-R: Mandatory Procedures for Major Defense Acquisition Programs (MDAPS) and Major Automated Information System (MAIS) Acquisition Programs," United States Department of Defense, 2002.

[44] C. H. Spenny and D. R. Jacques, "A Perspective on System Engineering Under the New US Department of Defense Acquisition Policy," in *INCOSE International Symposium*, 2004.

[45] United States Department of Defense, Office of the Undersecretary of Defense for Acquisition and Sustainment, "Depatment of Defense Instruction 5000.02 Operation of the Adaptive Acquisition Framework," United States Department of Defense, 2020.

[46] Space Development Agency, "SDA About Us 2020 to 2021," Space Development Agency, 2022.

[47] Space Development Agency, "Space Development Agency," [Online]. Available: https://www.sda.mil/home/about-us/faq/. [Accessed 11 June 2022].

[48] V. Lenarduzzi, D. Fucci and D. Mendez, "On the Perceived Harmfulness of Requirement Smells: An Empirical Study," in *Joint 26th Internation Conference on Requirements*

Engineering: Foundation for Software Quality Workshops, Doctoral Symposium, Live Studies Track, and Poster Track, Pisa, Italy, 2020.

[49] B. Curtis, J. Sappidi and A. Szynkarski, "Estimating the Size, Cost, and Types of Technical Debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[50] I. Ozkaya, "Managing Technical Debt throughout the Software Development Lifecycle," Carnegie-Mellon University, Pittsburgh, PA, 2022.

[51] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull and A. Vetro, "Using Technical Debt Data in Decision Making: Potential Decision Approaches," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[52] E. Tom, A. Aurum and R. Vidgen, "An exploration of technical debt," *The Journal of Systems and Software,* vol. 86, pp. 1498-1516, 2013.

[53] A. Melo, R. Fagundes, V. Lenarduzzi and W. Santos, "Indentification and Measurement of Technical Debt Requirements in Software Development: a Systematic Literature Review," *arXiv preprint arXiv:2105.14232.*

[54] T. Besker, A. Martini and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," *The Journal of Systems and Software,* vol. 135, pp. 1-16, 2017.

[55] Z. Li, P. Liang and P. Avgeriou, "Architectural debt management in value-oriented architecting," in *Economics-Driven Software Architecture*, Morgan Kaufmann, 2014, pp. 183-204.

[56] P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya and C. Seaman, "Reducing friction in software development," *IEEE Software,* vol. 33, no. 1, pp. 66-73, 2015.

[57] M. Fowler, "TechnicalDebtQuadrant," 14 October 2009. [Online]. Available: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html. [Accessed 27 January 2022].

[58] S. McConnell, "Managing Technical Debt," Construx Software Builders, 2008.

[59] A. Martini, J. Bosch and M. Chaudron, "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple case study," *Information and Software Technology,* vol. 67, pp. 237-253, 2015.

[60] Z. Li, P. Avgeriou and P. Liang, "A systematic mapping study on technical debt and its management," *The Journal of Systems and Software,* vol. 101, pp. 193-220, 2015.

[61] C. Izurieta, G. Rojas and I. Griffith, "Preemptive Management of Model Driven Technical Debt for Improving Software Quality," in *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, 2015.

[62] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology,* vol. 64, pp. 52-73, 2015.

[63] M. Ciolkowski, V. Lenarduzzi and A. Martini, "10 Years of Technical Debt Research and Practice: Past, Present, and Future," *IEEE Software,* vol. 38, no. 6, pp. 24-29, 2021.

[64] R. E. Fairley, "Assessing, Analyzing, and Controlling Technical Work," in *Systems Engineering of Software-Enabled Systems*, John Wiley & Sons, Inc, 2019, pp. 291-328.

[65] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan and N. Zazworka, "Managing

Technical Debt in Software-Reliant Systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010.

[66] Z. S. H. Abad and G. Ruhe, "Using Real Options to Manage Technical Debt in Requirements Engineering," in *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, Ottawa, 2015.

[67] C. Seaman and Y. Guo, "Measuring and Monitoring Technical Debt," *Advances in Computers,* vol. 82, pp. 25-46, 2011.

[68] A. Ampatzoglou, N. Mittas, A.-A. Tsintzira, A. Ampatzoglou, E.-M. Arvanitou, A. Chatzigeorgiou, P. Avgeriou and L. Angelis, "Exploring the Relation between Technical Debt Principal and Interest: An Empirical Approach," *Information and Software Technology,* vol. 128, 2020.

[69] V. Lenarduzzi, T. Besker, D. Taibi and A. Martini, "A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools," *The Journal of Systems and Software,* vol. 171, 2021.

[70] L. A. Rosser and Z. Ouzzif, "Technical Debt in Hardware Systems and Elements," in *IEEE Aerospace Conference (50100)*, 2021.

[71] C. Fernandez-Sanchez, J. Garbajosa and A. Yague, "A Framework to Aid in Decision Making for Technical Debt Management," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015.

[72] United States Government Accountability Office, "What GAO Does," [Online]. Available: https://www.gao.gov/about/what-gao-does. [Accessed 24 May 2022].

[73] United States Government Accountability Office, "NASA: Lessons from Ongoing Major Projects Could Improve Future Outcomes," United States Government Accountability Office, Washington, D.C., 2022.

[74] United States Government Accountability Office, "Space Acquisitions: Changing Environment Presents Continuing Challenges and Opportunities for DOD," United States Government Accountability Office, Washington, D.C., 2022.

[75] United States Government Accountability Office, "F-35 Joint Strike Fighter: Cost Growth and Schedule Delays Continue," United States Government Accountability Office, Washington, D.C., 2022.

[76] S. Erwin, "'Agile software' to replace troubled JMS," 8 May 2019. [Online]. Available: https://spacenews.com/agile-software-to-replace-troubled-jms/.

[77] L. Wheatcraft, T. Katz, M. Ryan and R. B. Wolfgang, "Needs, Requirements, Verification, Validation Lifecycle Manual," INCOSE, 2022.

[78] R. S. Carson, P. J. Frenz and E. O'Donnell, "Project Manager's Guide to Systems Engineering Measurement for Project Success: A Basic Introduction to Systems Engineering Measures for Use by Project Managers (Version 1.0)," INCOSE, San Diego, CA, 2015.

[79] L. A. Rosser and J. H. Norton, "A Systems Perspective on Technical Debt," in *IEEE Aerospace Conference (50100)*, 2021.

[80] J. Y. Monteith, J. D. McGregor and J. Zhang, "Technical Debt Aggregation in Ecosystems," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[81] United States Government Accountability Office, "Space Command and Control: Opportunities Exist to Enhance Annual Reporting," United States Government Accountability Office, Washington, D.C., 2021.

[82] J. M. Gilmore, "Director, Operational Test and Evaluation FY 2016 Annual Report," The Office of the Director, Operational Test and Evaluation, 2016.

[83] R. F. Behler, "Director, Operational Test and Evaluation FY 2017 Annual Report," The Office of the Director, Operational Test and Evaluation, 2017.

[84] R. F. Behler, "Director, Operational Test and Evaluation FY 2018 Annual Report," The Office of the Director, Operational Test and Evaluation, 2018.

[85] United States Government Accountability Office, "Space Acquisitions: Development and Oversight Challenges in Delivering Improved Space Situational Awareness Capabilities," United States Government Accountability Office, Washington, D.C., 2011.

[86] Y. Yang, R. Michel, J. Wade, D. Verma, M. Torngren and T. Alelyani, "Towards a taxonomy of technical debt for COTS-intensive cyber physical systems," *Procedia Computer Science,* vol. 153, pp. 108-117, 2019.

[87] United States Government Accountability Office, "Defense Major Automated Information Systems: Cost and Schedule Commitments Need to Be Established Earlier," United States Government Accountability Office, Washington, D.C., 2015.

[88] Air Force Operational Test and Evaluation Center, "Air Force Operational Test and Evaluation Center," [Online]. Available: https://www.afotec.af.mil/About-Us/Fact-Sheets/Display/Article/872935/air-force-operational-test-and-evaluation-center/. [Accessed 9 July 2022].

[89] H. Storrle and M. Ciolkowski, "Stepping Away from the Lamppost: Domain-Level Technical Debt," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019.

[90] J. L. Homer, "The role of project control systems in facilitating and measuring project success," in *PMI(R) Global Congress 2004 - North America*, Anaheim, CA. Newton Square, PA, 2004.

[91] R. S. Sangwan, A. Negahban, R. L. Nord and I. Ozkaya, "Optimization of Software Release Planning Considering Architectural Dependencies, Cost, and Value," *IEEE Transactions on Software Engineering,* 2020.

[92] A. Martini and J. Bosch, "The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, 2015.

[93] D. Ameller, C. Farre, X. Franch and G. Rufian, "A survey on software release planning models," in *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, Proceedings 17*, Trondheim, Norway, 2016.

[94] K. Schmid, "A formal approach to technical debt decision making," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, 2013.

[95] L. Delligatti, SysML Distilled: A Brief guide to the Systems Modeling Language, Upper Saddle River, NJ: Addison-Wesley, 2014.

[96] T. R. Browning, "Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions," *IEEE Transactions on Engineering Management,* vol. 48, no. 3, pp. 292-306, 2001.

[97] R. Verdecchia, I. Malavolta and P. Lago, "Architectural Technical Debt Identification: the Research Landscape," in *2018 ACM/IEEE International Conference on Technical Debt*, 2018.

[98] A. T. Bahill, "Diogenes, a Process for Identifying Unintended Consequences," *Systems Engineering,* vol. 15, no. 3, pp. 287-306, 2012.

[99] M. P. De Lessio, M.-A. Cardin, A. Astaman and V. Djie, "A process to analyze strategic design and management decisions under uncertainty in complex entrepreneurial systems," *Systems Engineering,* vol. 18, no. 6, pp. 604-624, 2015.

[100] T. F. Bowlds, J. M. Fossaceca and R. Iammartino, "software obsolescence risk assessment approach using multicriteria decision-making," *Systems Engineering,* vol. 21, no. 5, pp. 455-465, 2018.

[101] B. Boehm and P. Behnamghader, "Anticipatory development processes for reducing total ownership costs and schedules," *Systems Engineering,* vol. 22, no. 5, pp. 401-410, 2019.

[102] A. Sharon, O. L. de Weck and D. Dori, "Project Management vs. Systems Engineering Management: A Practitioners' View on Integrating the Project and Product Domains," *Systems Engineering,* vol. 14, no. 4, pp. 427-440, 2011.

[103] B. Kitchenham, "Procedures for Performing Systematic Reviews," Keele University, Keele, UK, 2004.

[104] S. Cha, Q. H. Dong and B. Vogel-Heuser, "Preventing Technical Debt for Automated Production System Maintenance using Systematic Change Effort Estimation with Considering Contingent Cost," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 2018.

[105] Q. H. Dong and B. Vogel-Heuser, "Cross-disciplinary and cross-life-cycle-phase Technical Debt in automated Production Systems: two industrial case studies and a survey," *IFAC-PapersOnLine,* vol. 51, no. 11, pp. 1192-1199, 2018.

[106] Q. H. Dong and B. Vogel-Heuser, "Modelling Industrial Technical Compromises in Production Systems with Causal Loop Diagrams," *IFAC-PapersOnline,* vol. 54, no. 4, pp. 212-219, 2021.

[107] Q. H. Dong, F. Ocker and B. Vogel-Heuser, "Technical Debt as indicator for weaknesses in engineering of automated production systems," *Production Engineering,* vol. 13, no. 3, pp. 273-282, 2019.

[108] B. Vogel-Heuser and E.-M. Neumann, "Adapting the concept of technical debt to software of automated Production Systems focusing on fault handling, mode of operation, and safety aspects," *IFAC-PapersOnLine,* vol. 50, no. 1, pp. 5887-5894, 2017.

[109] B. Vogel-Heuser, S. Rosch, A. Martini and M. Tichy, "Technical Debt in Automated Production Systems," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015.

[110] F. Ocker, M. Seitz, M. Oligschlager, M. Zou and B. Vogel-Heuser, "Increasing Awareness for Potential Technical Debt in the Engineering of Production Systems," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, 2019.

[111] B. Vogel-Heuser and S. Rosch, "Applicability of Technical Debt as a Concept to Understand Obstacles for Evolution of Automated Production Systems," in *2015 IEEE International COnference on Systems, Man, and Cybernetics*, 2015.

[112] S. Biffl, F. Ekaptura, A. Luder, J. L. Pauly, F. Rinker, L. Waltersdorfer and D. Winkler, "Technical Debt Analysis in Parallel Multi-Disciplinary Systems Engineering," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019.

[113] B. Vogel-Heuser and F. Bi, "Interdisciplinary effects of technical debt in companies with mechatronic products - a qualitative study," *Journal of Systems and Software,* vol. 171, 2021.

[114] R. E. Fairley and M. J. Willshire, "Better Now Than Later: Managing Technical Debt in Systems Development," *Computer,* vol. 50, no. 5, pp. 80-87, 2017.

[115] P. Kruchten, R. L. Nord and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software,* vol. 29, no. 6, pp. 18-21, 2012.

[116] P. Kruchten, R. L. Nord, I. Ozkaya and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes,* vol. 38, no. 5, pp. 51-54, 2013.

[117] P. S. Callister and J. Andersson, "Evaluation of System Integration and Qualification Strategies using the Technical Debt metaphor; a case study in Subsea System Development," in *26th Annual INCOSE International Symposium*, Edinburgh, Scotland, 2016.

[118] N. A. Ernst, "On the Role of Requirements in Understanding and Managing Technical Debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[119] B. Brenner, E. Weippl and A. Ekelhart, "Security Related Technical Debt in the Cyber-Physical Production Systems Engineering Process," in *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, 2019.

[120] J. Schutz and M. Uslar, "Introducing the Concept of Technical Debt to Smart Grids: A System Engineering Perspective," in *25th International Conference on Electricity Distribution*, Madrid, 2019.

[121] D. D. Walden, G. J. Roedler, K. J. Forsberg, R. D. Hamelin and T. M. Shortell, Eds., Systems Engineering Handbook, 4th ed., Hoboken: John Wiley and Sons, 2015.

[122] A. Martini, J. Bosch and M. Chaudron, "Architecture Technical Debt: Understanding Causes and a Qualitative Model," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2014.

[123] M. D. Guenov and S. G. Barker, "Application of axiomatic design and design structure matrix to the decomposition of engineering systems," *Systems Engineering,* vol. 8, no. 1, pp. 29-40, 2005.

[124] B. Boehm, R. Valerdi and E. Honour, "The ROI of systems engineering: Some quantitative results for software-intensive systems," *Systems Engineering,* vol. 11, no. 3, pp. 221-234, 2008.

[125] D. A. Broniatowski and J. Moses, "Measuring flexibility, descriptive complexity, and rework potential in generic system architectures," *Systems Engineering,* vol. 19, no. 3, pp. 207-221, 2016.

[126] R. Raman and M. D'Souza, "Decision learning framework for architecture design decisions of complex systems and system-of-systems," *Systems Engineering,* vol. 22, no. 6, pp. 538-560, 2019.

[127] N. Shallcross, G. S. Parnell, E. Pohl and E. Specking, "Set-based design: The state-of-practice and research opportunities," *Systems Engineering,* vol. 23, no. 5, pp. 557-578, 2020.

[128] G. I. Siyam, D. C. Wynn and P. J. Clarkson, "Review of value and lean in complex product development," *Systems Engineering,* vol. 18, no. 2, pp. 192-207, 2015.

[129] M. Uschold and R. Jasper, "A framework for understanding and classifying ontology applications," in *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods (KRR5)*, Stockholm, Sweden, 1999.

[130] L. M. Crawford, "Qualitative Research Designs," in *Research Design and Methods: An Applied Guide for the Scholar-Practitioner*, United States, SAGE Publications, 2019, pp. 81-98.

[131] B. Boehm, "Some future trends and implications for systems and software engineering processes," *Systems Engineering,* vol. 9, no. 1, pp. 1-19, 2006.

[132] A. Pyster, D. H. Olwell, T. L. Ferris, N. Hutchison, S. Enck, J. F. Anthony, D. Henry and A. Squires (eds), "Graduate Reference Curriculum for Systems Engineering (GRCSE™) V1.0," The Trustees of Stevens Institute of Technology, Hoboken, NJ, USA, 2012.

[133] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes and R. O. Spinola, "Towards an Ontology of Terms on Technical Debt," in *2014 Sixth International Workshop on Managing Technical Debt*, 2014.

[134] H. Kleinwaks, "An Empirical Survey on the Prevalence of Technical Debt in Systems Engineering [Poster Presentation]," in *INCOSE International Symposium*, Honoulu, HI, United States, 2023.

[135] D. Allaverdi and T. R. Browning, "A methodology for identifying flexible design opportunities in large-scale systems," *Systems Engineering,* vol. 23, no. 5, pp. 534-556, 2020.

[136] S. Dullen, D. Verma and M. Blackburn, "Review of Research into the Nature of Engineering and Development Rework: Need for a Systems Engineering Framework for Enabling Rapid Prototyping and Rapid Fielding," *Procedia Computer Science,* vol. 153, pp. 118-125, 2019.

[137] N. Rios, M. G. de Mendonca Neto and R. O. Spinola, "A tertiary study on technical debt: Types, management strategies, research," *Information and Software Technology,* vol. 102, pp. 117-145, 2018.

[138] D. Pina, A. Goldman and G. Tonin, "Technical Debt Prioritization: Taxonomy, Methods Results, and Practical Characteristics," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2021.

[139] Project Management Institute, "Technical Debt," February 2022. [Online]. Available: https://www.pmi.org/disciplined-agile/agile/technicaldebt. [Accessed 11 March 2023].

[140] V. Dalal, K. Krishnakanthan, B. Munstermann and R. Patenge, "Tech debt: Reclaiming tech equity," McKinsey Digital, 2020.

[141] Y. Yang, D. Verma and P. S. Anton, "Technical debt in the engineering of complex systems," *Systems Engineering,* 7 April 2023.

[142] N. S. Alves, T. S. Mendes, M. G. de Mendonca, R. O. Spinola, F. Shull and C. Seaman, "Identification and Management of Technical Debt: A Systematic Mapping Study," *Information and Software Technology,* vol. 70, pp. 100-121, 2016.

[143] S. Blumberg, R. Das, J. Lansing, N. Motsch, B. Munstermann and R. Patenge, "Demystifying digital dark matter: A new standard to tame technical debt," McKinsey Digital, 2022.

[144] C. Izurieta, I. Ozkaya, C. Seaman, P. Kruchten, R. Nord, W. Snipes and P. Avgeriou, "Perspectives on Managing Technical Debt: A transition point and roadmap from Dagstuhl," in *Joint of the 4th International Workshop on Quantiative Approaches to Software Quality, QuASoQ 2016 and 1st International Workshop on Technical Detb Analytics, TDA 2016*, 2016.

[145] J. W. Boswell, F. T. Anbari and J. W. Via III, "Systems Engineering and Project Management: Points of Intersection, Overlaps, and Tensions," in *2017 Portland International Conference on Management of Engineering and Technology*, Portland, 2017.

[146] A. Kossiakoff, S. J. Seymour, D. A. Flanigan and S. M. Biemer, Systems Engineering: Principles and Practice, 3rd ed., Hoboken, NJ: John Wiley and Sons, 2020.

[147] R. Verdecchia, P. Kruchten and P. Lago, "Architectural Technical Debt: A Grounded Theory," in *European Conference on Software Architecture*, 2020.

[148] K. Borowa, A. Zalewski and A. Saczko, "Living With Technical Debt – A Perspective from the Video Game Industry," *IEEE Software,* vol. 38, no. 6, pp. 65-70, 2021.

[149] V. Lenarduzzi and D. Fucci, "Towards a Holistic Definition of Requirements Debt," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019.

[150] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary and M. Young, "Machine Learning: The High-Interest Credit Card of Technical Debt," Google, Inc, 2014.

[151] C. L. Jones, G. Draper, B. Golaz and P. Januz, "Practical Software and Systems Measurement Continuous Iterative Development Measurement Framework. Part 3: Software Assurance and Technical Debt Version 2.1," Practical Software and Systems Measurement, National Defense Industrial Association, and International Council on Systems Engineering, 2021.

[152] R. Kothamasu, S. H. Huang and W. H. VerDuin, "System health monitoring and prognostics - a review of current paradigms and practices," *The International Journal of Advanced Manufacturing Technology,* vol. 28, no. 9, pp. 1012-1024, 2006.

[153] D. H. Collins, C. M. Anderson-Cook and A. V. Huzurbazar, "System health assessment," *Quality Engineering,* vol. 23, no. 2, pp. 142-151, 2011.

[154] G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou and O. Matel, "The Risk of Generating Technical Debt Interest: A Case Study," *SN Computer Science,* vol. 2, no. 1, pp. 1-12, 2020.

[155] M. Soliman, P. Avgeriou and Y. Li, "Architectural design decisions that incur technical debt – An industrial case study," *Information and Software Technology,* vol. 139, 2021.

[156] R. O. Spinola, N. Zazworka, A. Vetro, F. Shull and C. Seaman, "Understanding automated and human-based technical debt identfication approaches - a two-phase study," *Journal of the Brazilian Computer Society,* vol. 25, no. 5, pp. 1-21, 2019.

[157] N. Zazworka, A. Vetro', C. Izurieta, S. Wong, Y. Cai, C. Seaman and F. Shull, "Comparing four approaches for technical debt identification," *Software Quality Journal,* vol. 22, no. 3, pp. 403-426, 2014.

[158] G. Sierra, E. Shihab and Y. Kamei, "A survey of self-admitted technical debt," *The Journal of Systems and Software,* vol. 152, pp. 70-82, 2019.

[159] INCOSE, "INCOSE-TP-2010-006-03 Guide for Writing Requirements," INCOSE, San Diego, CA, 2019.

[160] H. Kleinwaks, A. Batchelor, T. Bradley, M. Rich and J. F. Turner, "LEAP - A Process for Identifying Potential Technical Debt in Iterative System Development," *INCOSE International Symposium,* vol. 33, no. 1, pp. 535-553, 2023.

[161] Scaled Agile, Inc., "Program Increment," 6 September 2022. [Online]. Available: https://www.scaledagileframework.com/program-increment/. [Accessed 2 November 2022].

[162] D. M. Tate, "Acquisition Cycle Time: Defining the Problem (Revised)," Institute for Defense Analyses, Alexandria, United States, 2016.

[163] T. R. Browning, "Use of Dependency Structure Matrices for Product Development Cycle Time Reduction," in *Proceedings of the Fifth ISPE International Conference on Concurrent Engineering: Research and Applications*, Tokyo, Japan, 1998.

[164] A. A. Yassine, D. E. Whitney and T. Zambito, "Assessment of rework probabilities for simulating product development processes using the design structure matrix (DSM)," in *Proceedings of DETC '01 ASME 2001 International Design Engineering Technical Conferences as Computers and Information in Engineering Conference*, Pittsburgh, Pennsylvania, 2011.

[165] E. W. Weisstein, "Heaviside Step Function," From Mathworld--A Wolfram Web Resource, [Online]. Available: https://mathworld.wolfram.com/HeavisideStepFunction.html. [Accessed 22 August 2022].

[166] E. Million, "The Hadamard Product," 2007. [Online]. Available: http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf. [Accessed 22 August 2022].

[167] H. Kleinwaks, M. Rich, M. C. Butterfield and J. F. Turner, "LEAPing Ahead - The Space Development Agency's Method for Planning for the Future," *INCOSE International Symposium,* vol. 33, no. 1, pp. 925-942, 2023.

[168] M. Svahnberg, T. Gorschek, R. Feldt, R. Torkar and S. Bin Saleem, "A systematic review on strategic release planning models," *Information and Software Technology,* vol. 52, pp. 237-248, 2010.

[169] H. Kleinwaks, A. Batchelor and T. H. Bradley, "Predicting the Dynamics of Earned Value Creation in the Presence of Technical Debt," *Submitted to IEEE Access,* 2023.

[170] P. E. D. Love, F. Ackermann, J. Smith, Z. Irani and D. J. Edwards, "Making Sense of Rework Causation in Offshore Hydrocarbon Projects," *Project Management Journal,* vol. 47, no. 4, pp. 16-28, 2016.

[171] P. E. D. Love, "Influence of Project Type and Procurement Method on Rework Costs in Building Construction Projects," *Journal of construction engineering and management,* vol. 128, no. 1, pp. 18-29, 2002.

[172] R. L. Yim, J. M. Castaneda, T. L. Doolen, I. Y. Tumer and R. Malak, "Exploring the Relationship Between Rework Projects and Risk Indicators," *Project Management Journal,* vol. 46, no. 4, pp. 63-75, 2015.

[173] T. Williams, "Why Monte Carlo Simulations of Project Networks Can Mislead," *Project Management Journal,* vol. 35, no. 3, pp. 53-61, 2004.

[174] G. Ma, J. Jia, T. Zhu and S. Jiang, "A Critical Design Structure Method for Project Schedule Development under Rework Risks," *Sustainability,* vol. 11, no. 24, p. 7229, 2019.

[175] V. Krishnan, S. D. Eppinger and D. E. Whitney, "A Model-Based Framework to Overlap Product Development Activities," *Management Science,* vol. 43, no. 4, pp. 437-451, 1997.

[176] R. P. Smith and S. D. Eppinger, "A Predictive Model of Sequential Iteration in Engineering Design," *Management Science,* vol. 43, no. 8, pp. 1104-1120, 1997.

[177] J. U. Maheswari and K. Varghese, "Project Scheduling using Dependency Structure Matrix," *International Journal of Project Management,* vol. 23, no. 3, pp. 223-230, 2005.

[178] Project Management Institute, Inc, A Guide to the Project Management Body of Knowledge (PMBOK Guide), 6th ed., Newtown Square, Pennsylvania, USA: Project Management Institute, Inc, 2017.

[179] H. Khamooshi and H. Golafshani, "EDM: Earned Duration Management, a new approach to schedule performance management and measurement," *International Journal of Project Management,* vol. 32, no. 6, pp. 1019-1041, 2014.

[180] W. Lipke, "Schedule is different," *The measurable news,* vol. 31, no. 4, 2003.

[181] S. Vandevoorde and M. Vanhoucke, "A comparison of different project duration forecasting methods using earned value metrics," *International Journal of Project Management,* vol. 24, no. 4, pp. 289-302, 2006.

[182] R. D. Warburton, "A time-dependent earned value model for software projects," *International Journal of Project Management,* vol. 29, no. 8, pp. 1082-1090, 2011.

[183] T. R. Browning, "Planning, Tracking, and Reducing a Complex Project's Value at Risk," *Project Management Journal,* vol. 50, no. 1, pp. 71-85, 2019.

[184] K. Fordyce, "Some Basics on the Value of S Curves and Market Adoption of a New Product," Arkieva, 1 April 2020. [Online]. Available: https://blog.arkieva.com/basics-on-s-curves/. [Accessed 26 March 2023].

[185] E. W. Weisstein, "Inflection Point," From MathWorld - A Wolfram Web Resource, [Online]. Available: https://mathworld.wolfram.com/InflectionPoint.html. [Accessed 26 March 2023].

[186] H. Kleinwaks, A. Batchelor and T. H. Bradley, "Probabilistic Enhancement to the LEAP Process for Identifying Technical Debt in Iterative System Development," *Submitted to IEEE Access,* 2023.

[187] F. T. Anbari, "Earned Value Project Management Method and Extensions," *Project Management Journal,* vol. 34, no. 4, pp. 12-23, 2003.

[188] D. T. Hulett, "Advanced Quantitative Schedule Risk Analysis," Hulett & Associates, LLC., 2004.

[189] T. Walworth, M. Yearworth, J. Davis and P. Davies, "Early estimation of project performance: the application of a system dynamics rework model," in *2013 IEEE international systems conference (SysCon)*, 2013.

[190] R. R. de Almeida, U. Kulesza, C. Treude, D. C. Feitosa and A. H. G. Lima, "Aligning Technical Debt Prioritization with Business Objectives: A Multiple-Case Study," in *2018 IEEE Interantional Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[191] R. P. Smith and S. D. Eppinger, "Identifying Controlling Features of Engineering Design Iteration," *Management Science,* vol. 43, no. 3, pp. 276-293, 1997.

[192] J. Kim, C. Kang and I. Hwang, "A practical approach to project scheduling: considering the potential quality loss cost in the time-cost tradeoff problem," *Internation Journal of Project Management,* vol. 30, no. 2, pp. 264-272, 2012.

[193] C. J. Van Wyngaard, J.-H. C. Pretorius and L. Pretorius, "Theory of the Triple Constraint - a Conceptual Review," in *2012 IEEE International Conference on Industrial Engineering and Engineering Management*, Hong Kong, China, 2012.

[194] F. Kendall, "Performance of the Defense Acquisition System 2016 Annual Report," 2016.

[195] D. R. Katz, S. Sarkani, T. Mazzuchi and E. H. Conrow, "The relationship of technology and design maturity to DoD weapon system cost change and schedule change during engineering and manufacturing development," *Systems Engineering,* vol. 18, no. 1, pp. 1-15, 2015.

[196] United States Department of Defense, Office of the Undersecretary of Defense for Acquisition and Sustainment, "DoD Instruction 5000.85: Major Capability Acquisition," United States Department of Defense, Washington, D.C., 2021.

[197] United States Government Accountability Office, "Technology Readiness Assessment Guide," United States Government Accountability Office, 2020.

[198] D. Knoll, C. Fortin and A. Golkar, "A process model for concurrent conceptual design of space systems," *Systems Engineering,* vol. 24, no. 4, pp. 234-49, 2021.

[199] D. A. Bearden, "A complexity-based risk assessment of low-cost planetary missions: when is a mission too fast and too cheap?," *Acta Astronautica,* vol. 52, pp. 371-379, 2003.

[200] Scaled Agile, Inc., "SAFe 6.0," [Online]. Available: https://scaledagileframework.com/. [Accessed 7 September 2023].

[201] H. Kleinwaks, "LEAP - A Process for Identifying Potential Technical Debt in Iterative Systems Engineering [Conference Presentation]," in *INCOSE International Symposium*, Honolulu, HI, United States, 2023.

APPENDIX A: EXAMPLE PYTHON CODE FOR LEAP IMPLEMENTATION

A.1 Probability Distribution Classes

```
# -*- coding: utf-8 -*-
import math
import numpy as np
import random
import matplotlib.pyplot as plt
import datetime
from multiprocessing import Pool
import sys

class distribution():
 """
 Base class for different distribution types
 """
 NORMAL_DISTRIBUTION = 0
 TRIANGULAR_DISTRIBUTION = 1
 GAMMA_DISTRIBUTION = 2
 DISCRETE_DISTRIBUTION = 3
 UNIFORM_DISTRIBUTION = 4
 CUSTOM_DISTRIBUTION = 5
 CONSTANT_DISTRIBUTION = 6

 def __init__(self):
  """
  default constructor, sets to normal distribution
  Returns
  -------
  None.
  """
  self.distributionType = distribution.NORMAL_DISTRIBUTION

 def getRandomVariable(self):
  """
  base class function for getting a random variable from the distribution
  Returns
  -------
  int always returns 0
  """
  pass

 def getMean(self, n=10000):
  """
  returns the mean value of the distribution
  Parameters
  ----------
  n: int
   number of trials to run (default = 10000)
  Returns
  -------
  None.
  """
  x = np.zeros(n)
  for i in range(0, len(x)):
   x[i] = self.getRandomVariable()
  return np.average(x)

 def decrement(self, percentage):
  """
```

```
    decrement the values in the distribution settings by the specified percentage
    value of 0.1 means that the new value will be 90% of the original (1-0.1)*value
    Parameters
    ----------
    percentage : float
     percent by which the value is decremented
    Returns
    -------
    None.
    """
    pass

  def clone(self):
    """
    return a new distribution that clones this one
    Returns
    -------
    clone of the distribution
    """
    pass

  def __str__(self):
    """
    print out the parameters of the distribution
    Returns
    -------
    string description of distribution
    """
    return str(self.distributionType)

class ConstantDistribution(distribution):
  """
  constant distribution: always returns the same value
  """

  def __init__(self, value):
    """
    Parameterized constructor
    Parameters
    ----------
    value : float
     the value to return
    Returns
    -------
    None.
    """
    self.value = value

  def getRandomVariable(self):
    """
    returns the constant value
    Returns
    -------
    float the random number
    """
    return self.value

  def decrement(self, percentage):
    """
    decrement the values in the distribution settings by the specified percentage
    value of 0.1 means that the new value will be 90% of the original (1-0.1)*value
    Parameters
    ----------
    percentage : float
     percent by which the value is decremented
```

```
        Returns
        -------
        None.
        """
        self.value = (1-percentage)*self.value

    def clone(self):
        """
        return a new distribution that clones this one
        Returns
        -------
        clone of the distribution
        """
        return ConstantDistribution(self.value)

    def __str__(self):
        """
        print out the parameters of the distribution
        Returns
        -------
        string description of distribution
        """
        return "Constant Value {}".format(self.value)

class CustomTriangularDistribution(distribution):
    """
    custom distribution to model the distribution used in Williams:
     triangular distribution that reduces the time by 2/3 if the value is over a specified limit
    """
    def __init__(self, low, mode, high, threshold, reduction):
        """
        Parameterized constructor
        Parameters
        ----------
        low : float
         the low end of the triangular estimate
        mode : float
         the most likely value in the estimate
        high : float
         the high end of the triangular estimate
        threshold : float
         the value at which the reduction is applied
        reduction : float
         reduction to apply (from 0 to 1), multiplied by the difference in random value and threshold
        Returns
        -------
        None.
        """
        self.distributionType = distribution.CUSTOM_DISTRIBUTION
        self.low = low
        self.mode = mode
        self.high = high
        self.threshold = threshold
        self.reduction = reduction

    def getRandomVariable(self):
        """
        returns a random number selected from the distribution using the random.triangular function
        and then applies the reduction based on the threshold
        Returns
        -------
        float the random number
        """
        #note that input order into np random function is low, high, mode,
        value = random.triangular(self.low, self.high, self.mode)
```

```
        if (value > self.threshold):
            value = self.threshold + self.reduction*(value-self.threshold)
        return value

    def decrement(self, percentage):
        """
        decrement the values in the distribution settings by the specified percentage
        value of 0.1 means that the new value will be 90% of the original (1-0.1)*value
        Note that decrement only applies to the triangular distribution parameters and not the threshold or
reduction
        Parameters
        ----------
        percentage : float
            percent by which the value is decremented
        Returns
        -------
        None.
        """
        self.low = (1-percentage)*self.low
        self.mode = (1-percentage)*self.mode
        self.high = (1-percentage)*self.high

    def clone(self):
        """
        return a new distribution that clones this one
        Returns
        -------
        clone of the distribution
        """
        return CustomTriangularDistribution(self.low, self.mode, self.high, self.threshold, self.reduction)

    def __str__(self):
        """
        print out the parameters of the distribution
        Returns
        -------
        string description of distribution
        """
        return "CustomTriangular Low {} Mode {} High {} Threshold {} Reduction {}".format(self.low,
self.mode, self.high, self.threshold, self.reduction)

class NormalDistribution(distribution):
    """
    Normal Distribution
    """
    def __init__(self, mean, standardDeviation):
        """
        Parameterized constructor
        Parameters
        ----------
        mean : float
            mean of the distribution
        standardDeviation : float
            standard deviation of the distribution
        Returns
        -------
        None.
        """
        self.distributionType = distribution.NORMAL_DISTRIBUTION
        self.mean = mean
        self.standardDeviation = standardDeviation

    def getRandomVariable(self):
        """
        returns a random number selected from the distribution using the random.gauss function
```

```
        Returns
        -------
        float the random number
        """
        return random.gauss(self.mean, self.standardDeviation)

    def decrement(self, percentage):
        """
        decrement the values in the distribution settings by the specified percentage
        value of 0.1 means that the new value will be 90% of the original (1-0.1)*value
        Parameters
        ----------
        percentage : float
         percent by which the value is decremented
        Returns
        -------
        None.
        """
        self.mean = (1-percentage)*self.mean
        self.standardDeviation = (1-percentage)*self.standardDeviation

    def clone(self):
        """
        return a new distribution that clones this one
        Returns
        -------
        clone of the distribution
        """
        return NormalDistribution(self.mean, self.standardDeviation)

    def __str__(self):
        """
        print out the parameters of the distribution
        Returns
        -------
        string description of distribution
        """
        return "Normal Mean {} Std Dev {}".format(self.mean, self.standardDeviation)

class TriangularDistribution(distribution):
    """
    Triangular Distribution
    """
    def __init__(self, low, mode, high):
        """
        Parameterized constructor
        Parameters
        ----------
        low : float
         the low end of the triangular estimate
        mode : float
         the most likely value in the estimate
        high : float
         the high end of the triangular estimate
        Returns
        -------
        None.
        """
        self.distributionType = distribution.TRIANGULAR_DISTRIBUTION
        self.low = low
        self.mode = mode
        self.high = high

    def getRandomVariable(self):
        """
```

```
        returns a random number selected from the distribution using the random.triangular function
        Returns
        -------
        float the random number
        """
        #note that input order into numpy random function is low, high, mode
        return random.triangular(self.low, self.high, self.mode)

    def decrement(self, percentage):
        """
        decrement the values in the distribution settings by the specified percentage
        value of 0.1 means that the new value will be 90% of the original (1-0.1)*value
        Parameters
        ----------
        percentage : float
         percent by which the value is decremented
        Returns
        -------
        None.
        """
        self.low = (1-percentage)*self.low
        self.mode = (1-percentage)*self.mode
        self.high = (1-percentage)*self.high

    def clone(self):
        """
        return a new distribution that clones this one

        Returns
        -------
        clone of the distribution
        """
        return TriangularDistribution(self.low, self.mode, self.high)

    def __str__(self):
        """
        print out the parameters of the distribution
        Returns
        -------
        string description of distribution
        """
        return "Triangular Low {} Mode {} High {}".format(self.low, self.mode, self.high)

class GammaDistribution(distribution):

    def __init__(self, alpha, beta):
        """
        Parameterized constructor
        Parameters
        ----------
        alpha : float
         alpha parameter for the gamma distribution
        beta : float
         beta parameter for the gamma distribution
        Returns
        -------
        None.
        """
        self.distributionType = distribution.GAMMA_DISTRIBUTION
        self.alpha = alpha
        self.beta = beta

    def getRandomVariable(self):
        """
        returns a random number selected from the distribution using the random.gammavariate function
        Returns
```

```
        -------
        float the random number
        """
        return random.gammavariate(self.alpha, self.beta)

    def decrement(self, percentage):
        """
        decrement the values in the distribution settings by the specified percentage
        value of 0.1 means that the new value will be 90% of the original (1-0.1)*value
        Parameters
        ----------
        percentage : float
         percent by which the value is decremented
        Returns
        -------
        None.
        """
        self.alpha = (1-percentage)*self.alpha
        self.beta = (1-percentage)*self.beta

    def clone(self):
        """
        return a new distribution that clones this one
        Returns
        -------
        clone of the distribution
        """
        return GammaDistribution(self.alpha, self.beta)

    def __str__(self):
        """
        print out the parameters of the distribution
        Returns
        -------
        string description of distribution
        """
        return "Gamma {} {}".format(self.alpha, self.beta)

class DiscreteDistribution(distribution):
    def __init__(self, values, probabilities):
        """
        Parameterized constructor
        Parameters
        ----------
        values : array of float
         array of possible values for the discrete distribution, must be same length as probabilities
        probabilities : array of float
         array of probabilities for each of the values, must be the same length as values
        Returns
        -------
        None.
        """
        self.distributionType = distribution.DISCRETE_DISTRIBUTION
        self.values = values
        self.probabilities = probabilities

    def getRandomVariable(self):
        """
        returns a random number selected from the distribution using the random.choices function
        Returns
        -------
        float the random number
        """
        return random.choices(self.values, self.probabilities)[0]
```

```
 def decrement(self, percentage):
  """
  decrement the values in the distribution settings by the specified percentage
  value of 0.1 means that the new value will be 90% of the original (1-0.1)*value
  Note: only changes the values not the probabilities
  Parameters
  ----------
  percentage : float
   percent by which the value is decremented
  Returns
  -------
  None.
  """
  vals = []
  for i in range(0, len(self.values)):
   vals.append((1-percentage)*self.values[i])
  self.values = vals

 def clone(self):
  """
  return a new distribution that clones this one
  Returns
  -------
  clone of the distribution
  """
  return DiscreteDistribution(self.values, self.probabilities)

 def __str__(self):
  """
  print out the parameters of the distribution
  Returns
  -------
  string description of distribution

  """
  return "Discrete Values {} Probabilities {}".format(self.values, self.probabilities)

class UniformDistribution(distribution):
 """
 Uniform distribution
 """
 def __init__(self, low, high):
  """
  Parameterized constructor
  Parameters
  ----------
  low : float
   low end of the value range for the distribution
  high : float
   high end of the value range for the distribution
  Returns
  -------
  None.
  """
  self.distributionType = distribution.UNIFORM_DISTRIBUTION
  self.low = low
  self.high = high

 def getRandomVariable(self):
  """
  returns a random number selected from the distribution using the random.uniform function
  Returns
  -------
  float the random number
  """
```

```
        return random.uniform(self.low, self.high)

    def decrement(self, percentage):
        """
        decrement the values in the distribution settings by the specified percentage
        value of 0.1 means that the new value will be 90% of the original (1-0.1)*value
        Parameters
        ----------
        percentage : float
         percent by which the value is decremented
        Returns
        -------
        None.
        """
        self.low = (1-percentage)*self.low
        self.high = (1-percentage)*self.high

    def clone(self):
        """
        return a new distribution that clones this one
        Returns
        -------
        clone of the distribution
        """
        return UniformDistribution(self.low, self.high)

    def __str__(self):
        """
        print out the parameters of the distribution
        Returns
        -------
        string description of distribution.
        """
        return "Uniform Low {} High {}".format(self.low, self.high)
```

A.2 Implementation of the LEAP Process

```
class leap():
    #Supporting functions
    #K and K* functions
    g1Param = -(math.sqrt(6)-3)
    g2Param = math.sqrt(3)
    g3Param = math.sqrt(6)

    def kFunction(x,y,z):
        """
        executes the k function:
         k (x,y,z) = y, if x != 0 and z if x = 0
        Parameters
        ----------
        x : float or int
         Value checked to see if it is 0
        y : any type
         Value returned if x != 0
        z : any type
         Value returned if x = 0
        Returns
        -------
        y or z depending on the value of x
        """
        if (x == 0):
         return z
        else:
         return y
```

```
    def KFunction(U,V,z):
     """
     matrix version of the k function, returns a matrix where each cell is the result of the k function of
the input matrices
     Matrices U and V must be the same dimensions
     Parameters
     ----------
     U : 2d numpy array
      first matrix
     V : 2d numpy array
      second matrix
     z : float
      value to use when U[i,j] = 0
     Returns
     -------
     matrix of k(U[i,j], V[i,j], z)
     """
     K = np.zeros((U.shape[0], U.shape[1]))
     for i in range(0, U.shape[0]):
      for j in range(0, U.shape[1]):
       K[i,j] = leap.kFunction(U[i,j], V[i,j], z)
     return K

    def kStarFunction(u, v, z):
     """
     returns the product of the k function of each element of u and v
     Parameters
     ----------
     u : 1d numpy array
      first array
     v : 1d numpy array
      second array
     z : float
      value to use if u[i] == 0
     Returns
     -------
     product of the k function of each element of u and v
     """
     prod = 1
     for i in range(0, len(u)):
      prod *= leap.kFunction(u[i],v[i],z)
     return prod

    def KStarFunction(U,V,z):
     """
     matrix version of the kStarFunction
     the value in each cell of the resultant matrix is the kStarFunction of the row of U and column of V
     traditional matrix multiplication rules about dimensions apply (# columns of U = # rows of V)
     Parameters
     ----------
     U : 2d numpy array
      first matrix
     V : 2d numpy array
      second array
     z : float
      value to use if U[i,j] == 0
     Returns
     -------
     Matrix containing kStar results of each row/column pair
     """
     K = np.zeros((U.shape[0], V.shape[1]))
     for i in range(0, U.shape[0]):
      for j in range(0, V.shape[1]):
       K[i,j] = leap.kStarFunction(U[i,:], V[:,j], z)
     return K
    #End K and K* functions
```

```
    #Monte Carlo supporting functions
    def getGrowthInflectionPoints(T):
     """
     determines the roots of the second and third derivatives of the planned value curve
     which give the inflection points
     G1: boundary between initial slow growth and rapid growth
     G2: boundary between rapid growth and concluding slow growth
     G3: boundary between concluding slow growth and limited growth
     Parameters
     ----------
     T : time of peak instantaneous planned value, expressed as a percentage of duration (0 to 1)
     Returns
     -------
     tuple of (G1, G2, G3)
     """
     #ensure all values are between zero and 1
     t = np.clip(T, 0, 1)
     t2 = t*t

     g1 = np.sqrt(leap.g1Param*(t2))
     g2 = leap.g2Param*t
     g3 = np.sqrt(leap.g3Param*t2+3*t2)
     return (g1,g2,g3)

    def getRandomVariables( techParameters, techTechParameters, distributionList):
     """
     determine random variables based on the input parameters
     Distributions are selected from the distribution classes
     Currently available distributions are:
      NORMAL_DISTRIBUTION: two parameters - mean and standard deviation
      TRIANGULAR_DISTRIBUTION: three parameters - low, most likely, and high estimatesf
      GAMMA_DISTRIBUTION: two parameters - alpha and beta
      DISCRETE_DISTRIBUTION: two parameters: list of relative weights, list of values
     Parameters
     ----------
     techParameters : 2d numpy array
      2d array where the number rows is equal to the number of technologies
      Columns are:
       [0] T - time of peak instantaneous planned value, expressed from 0 to 1
       [1] earliest start time, expressed in time units
       [2] distribution index into the the distributionList

     techTechParameters : 2d numpy array
      2d array where the number rows is equal to a maximum of the number of technologies squared (one for
each combination)
      Columns are:
       [0] predecessor technology index
       [1] successor technology index
       [2] alpha - impact on successor, from 0 to 1
       [3] U - utility threshold, from 0 to 1
       [4] r distribution index into the the distributionList
       [5] tau distribution index into the the distributionList

     distributionList : array of distribution
      array of distribution classes containing the distribution information
     Returns
     -------
     tuple of (randomDuration, randomTechTechParameters)
     randomDuration is a 1d array of the duration for each technology
     randomTechTechParameters is a 2d numpy array with the following columns:
      [0] predecessor technology index
      [1] successor technology index
      [2] alpha - impact on successor, from 0 to 1
      [3] U - utility threshold, from 0 to 1
      [4] r for the predecessor-successor combination
      [5] tau for the predecessor-successor combination
```

```
      """

      #start with the tech paramenters
      randomDuration = np.zeros(len(techParameters))
      for i in range(0, len(techParameters)):
       dist = distributionList[int(techParameters[i,2])]
       randomDuration[i] = dist.getRandomVariable()
       #randomDuration[i] = leap.randomVariable(techParameters[i,2], techParameters[i,3:])
       if (randomDuration[i] < 0):
        randomDuration[i] = 0
      #get the tech-tech parameters
      randomTechTechParameters = np.zeros((len(techTechParameters), 6))
      for i in range(0, len(techTechParameters)):
       randomTechTechParameters[i,0] = techTechParameters[i,0]
       randomTechTechParameters[i,1] = techTechParameters[i,1]
       randomTechTechParameters[i,2] = techTechParameters[i,2]
       randomTechTechParameters[i,3] = techTechParameters[i,3]
       #get the r value
       rDist = distributionList[int(techTechParameters[i,4])]
       randomTechTechParameters[i,4] = np.clip(rDist.getRandomVariable(), 0, 1)
       tDist = distributionList[int(techTechParameters[i,5])]
       randomTechTechParameters[i,5] = np.clip(tDist.getRandomVariable(),0, 1)
      return (randomDuration, randomTechTechParameters)

     def getEarnedValue(t, r, tau, alpha, T):
      """
      return the cumulative earned value at time T
      Parameters
      ----------
      t : float
       time at which cumulative earned value is calculated, from 0 to 1
      r : float
       portion of alpha subject to delays, from 0 to 1
      tau : float
       delay introduced by predecessor task, from 0 to 1
      alpha : float
       % of successor task impacted by predecessor task
      T : float
       time (from 0 to 1) of maximum instantaneous planned value
      Returns
      -------
      Cumulative Earned Value at t
      """
      N = 1 #N always equals 1 in this implementation
      if (t <= tau):
       return (1-r)*alpha*N*(1-math.exp((-t**2)/(2*(T**2))))
      else:
        return alpha*N-alpha*N*(1-r)*(math.exp((-t**2)/(2*T**2)))-r*alpha*N*math.exp(-((t-
tau)**2)/(2*(T**2)))

     def getEarnedValueAtTransitionPoints(techParameters, randomTechTechParameters):
      """
      return the earned value at each of the transition points for each technology
      Parameters
      ----------
      techParameters : 2d numpy array
       2d numpy array where the number rows is equal to the number of technologies
       Columns are:
        [0] T - time of peak instantaneous planned value, expressed from 0 to 1
        [1] earliest start time, expressed in time units
        [2] distribution type (as a enumerated type from the LEAP class) for the duration
        columns 3 and higher are the parameters used for the distribution of the duration
      randomTechTechParameters : 2d numpy array
       2d numpy array where the number rows is equal to a maximum of the number of technologies squared
(one for each combination)
       Columns are:
        [0] predecessor technology index
```

```
        [1] successor technology index
        [2] alpha - impact on successor, from 0 to 1
        [3] U - utility threshold, from 0 to 1
        [4] r randomly determined r value
        [5] tau randomly determined tau value
      Returns
      -------
      2d numpy array where each row is a technology and the columns are G1, G2, G3, EV(G1), EV(G2), EV(G3),
EV(1) for that technology
      """
      res = np.zeros((len(techParameters),7))
      for i in range(0, len(techParameters)):
       T = techParameters[i][0]
       (G1, G2, G3) = leap.getGrowthInflectionPoints(T)
       #get the predecessor indices
       rows = randomTechTechParameters[np.where(randomTechTechParameters[:,0] == i)[0]]
       EV1 = 0
       EV2 = 0
       EV3 = 0
       EV4 = 0
       for j in range(0, len(rows)):
        ri = rows[j][4]
        ai = rows[j][2]
        taui = rows[j][5]
        EV1 += leap.getEarnedValue(G1, ri, taui, ai, T)
        EV2 += leap.getEarnedValue(G2, ri, taui, ai, T)
        EV3 += leap.getEarnedValue(G3, ri, taui, ai, T)
        EV4 += leap.getEarnedValue(1, ri, taui, ai, T)

       res[i]= [G1, G2, G3, EV1, EV2, EV3, EV4]
      return res

    def getTimeFromEarnedValue(V, G1, G2, G3, EV1, EV2, EV3, EV4):
      """
      return the time based on the specified earned value V
      Uses a piece-wise linear approximation to the earned value to calculate
      Parameters
      ----------
      V : float
       the earned value of interest
      G1 : float
       first inflection point - transition point between initial slow growth and rapid growth
      G2 : float
       second inflection point - transition point between rapid growth and concluding slow growth
      G3 : float
       third infleciton point - transition point between concluding slow growth and limited growth
      EV1 : flaot
       earned value at G1
      EV2 : float
       earned value at G2
      EV3 : float
       earned value at G3
      EV4: float
       earned value at 1
      Returns
      -------
      the time corresponding to the input earned value
      """
      #values less than or equal to zero always returns zero
      if (V <= 0):
       return 0
      if (V <= EV1):
       return V*G1/EV1
      elif (V <= EV2):
       m2 = (EV2-EV1)/(G2-G1)
       return (V-EV1)/m2+G1
      elif (V <= EV3):
       m3 = (EV3-EV1)/(G3-G2)
```

```
        return (V-EV2)/m3+G2
       else:
        m4 = (EV4-EV3)/(1-G3)
        if (m4 == 0):
         return 0
        return (V-EV3)/m4+G3
      #End Monte Carlo supporting functions

      def getLinearizedEarnedValue(t, G1, G2, G3, EV1, EV2, EV3, EV4):
       """
       return the linearized earned value for the specified time
       Parameters
       ----------
       t : float
        the time of interest
       G1 : float
        first inflection point - transition point between initial slow growth and rapid growth
       G2 : float
        second inflection point - transition point between rapid growth and concluding slow growth
       G3 : float
        third infleciton point - transition point between concluding slow growth and limited growth
       EV1 : flaot
        earned value at G1
       EV2 : float
        earned value at G2
       EV3 : float
        earned value at G3
       EV4: float
        earned value at 1
       Returns
       -------
       the time corresponding to the input earned value
       """
       if (t <= G1):
        m = EV1/G1
        return t*m
       elif (t <= G2):
        m = (EV2-EV1)/(G2-G1)
        return m*(t-G1)+EV1
       elif (t <= G3):
        m = (EV3-EV2)/(G3-G2)
        return m*(t-G2)+EV2
       else:
        m = (EV4-EV3)/(1-G3)
        return m*(t-G3)+EV3

      def runErrorAnalysis(TArray,rArray,tauArray, tStep):
       """
       run an error analysis for the input combinations of T, r, and tau
       Parameters
       ----------
       TArray : list (of float)
        values of T to consider, from 0 to 1
       rArray : list (of float)
        values of r to consider, from 0 to 1
       tauArray : list (of float)
        values of tau to consider, from 0 to 1
       tSetp : float
        step to use for t, will evaluate from 0 to 1 at tStep
       Returns
       -------
       array of [[T, r, tau, t1, max e1, max % e1, t2, max e2, max % e2, t3, max e3, max % e3, t4, max e4,
max %e4]]
       e1-4 refer to the sections of the linearized ev plot
       e1: t <= G1
       e2: t > G1 and <= G2
       e3: t > G2 and <= G3
```

```
 e4: t > G3 and <= G4
 """
 res = []
 for i in range(0, len(TArray)):
  T = TArray[i]
  #get the inflection points, these are functions of T only
  (G1, G2, G3) = leap.getGrowthInflectionPoints(T)
  print("G1", G1, "G2", G2, "G3", G3)
  for j in range(0, len(rArray)):
   r = rArray[j]
   for k in range(0, len(tauArray)):
    tau = tauArray[k]
    t = tStep
    #get the earned value at the inflection points
    EV1 = leap.getEarnedValue(G1, r, tau, 1, T)
    EV2 = leap.getEarnedValue(G2, r, tau, 1, T)
    EV3 = leap.getEarnedValue(G3, r, tau, 1, T)
    EV4 = leap.getEarnedValue(1, r, tau, 1, T)
    maxErr = [0,0,0,0]
    maxPercentError = [0,0,0,0]
    maxTime = [0,0,0,0]

    answer = [T,r,tau,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
    while t <= 1:
     ev = leap.getEarnedValue(t, r, tau, 1, T) #actual earned value
     lev = leap.getLinearizedEarnedValue(t, G1, G2, G3, EV1, EV2, EV3, EV4) #linearized ev
     err = abs(ev-lev)
     percentErr = err/ev*100
     if (t <= G1):
      if (err >= maxErr[0]):
       maxErr[0] = err
       maxPercentError[0] = percentErr
       maxTime[0] = t
     elif (t <= G2):
      if (err >= maxErr[1]):
       maxErr[1] = err
       maxPercentError[1] = percentErr
       maxTime[1] = t
     elif (t <= G3):
      if (err >= maxErr[2]):
       maxErr[2] = err
       maxPercentError[2] = percentErr
       maxTime[2] = t
     else:
      if (err >= maxErr[3]):
       maxErr[3] = err
       maxPercentError[3] = percentErr
       maxTime[3] = t

     t += tStep
    for n in range(0, len(maxErr)):
     answer[3+n*3] = maxTime[n]
     answer[4+n*3] = maxErr[n]
     answer[5+n*3] = maxPercentError[n]

    res.append(answer)
 return res
#End supporting functions
#####################################################

def qualitativeLeap(functionalMatrix, developmentMatrix, needMatrix, debug=False):
 """
 runs the qualitative leap algorithm.
  t = number of technologies
  c = number of capabilities
  p = number of time periods
 Parameters
 ----------
```

```
    functionalMatrix : c x t numpy array
     matrix defining the mapping of capabilities to the technologies that support them.
     Technologies are the columns (t) and the capabilities are the rows (c).
     A one (1) is entered in cell (c, t) if the technology t supports the capability c
    developmentMatrix : t x p numpy array
     matrix defining the development timelines of the technology.
     Technologies are the rows (t) and the time periods are the columns (p)
     A one (1) is entered in cell (t, p) if the technology t will be developed in the time period p
    needMatrix : p x c numpy array
     matrix defining the mapping of capabilities to the time periods in which they are needed
     time periods are the rows (p) and the capabilities are the columns (c)
     A one (1) is entered in cell (p, c) if the capability c is needed in time period p
    debug: boolean
     If true, prints debug information, such as the values of the matrices
    Returns
    -------
    tuple of (numpy array, numpy array, numpy array)
    First element of the tuple is the availability matrix, a (p x c) numpy array that defines if a
capability c will be available in time period p
     A one in the cell indicates that the capability will be available
    Second element of the tuple is the delivery matrix, a (p x c) numpy array that defines if a
capability will be delivered late
     A one (1) in the cell indicates that the capability will be late to need
     A zero (0) in the cell indicates that the capability either is ready when needed, or is not ready
and not needed
     A negative one (-1) in the cell indicates that the capability is ready ahead of need
    Third element of the tuple is the investment matrix, a (t x p) matrix that shows how many late
capabilities that the technology impacts in the time period
     Positive values indicate the number of late capabilities in that time period that depend on the
technology
     Negative values indicate the number of early capabilities in that time period that are impacted by
the technology
    """
    #create the summation matrix
    summationMatrix = np.zeros(needMatrix.shape)
    for i in range(0, numCapabilities):
     summationMatrix[:,i] = np.sum(functionalMatrix[i,:])

    #calculate the availability matrix
    #A=H((FV)^T-S+0.5J)
    #J = hadamard identity matrix
    J = np.ones(needMatrix.shape)
    #determine transpose of Functional Matrix * development matrix
    FVT = np.transpose(np.matmul(functionalMatrix, developmentMatrix))

    #calculate the temporary availability matrix and then apply the heaviside function
    #apply heaviside function
    # note that the numpy implementation allows for specification of the x2 term in the heaviside
    # function.
    availabilityMatrix = np.heaviside(FVT-summationMatrix,1)

    #calculate delivery matrix
    #D = N-A
    deliveryMatrix = needMatrix-availabilityMatrix

    #investment matrix
    #I = transpose(D*F).(J-V)
    #in numpy, the multiplication operator (*) for matrices provides the hadamard product
    #reset J to be size of the development matrix
    J = np.ones(developmentMatrix.shape)
    investmentMatrix = np.transpose(np.matmul(deliveryMatrix, functionalMatrix))*(J-developmentMatrix)


    #print the results
    if (debug):
     #availability results
     print("Availability Matrix\n", availabilityMatrix)
     #determine when a capability is first available
     availableCapabilityIndices = np.where(np.transpose(availabilityMatrix) == 1)
```

```
        #print out so that the output is [capability index, time period index], ordered by capability index
        availableCapabilities = np.transpose(np.array([availableCapabilityIndices[0],
availableCapabilityIndices[1]]))
        print("capability availability times\n", availableCapabilities)

        #delivery results
        print("Delivery Matrix\n", deliveryMatrix)
        #determine what capabilities are late
        deliveryCapabilityIndices = np.where(np.transpose(deliveryMatrix) == 1)
        #print out so that the output is [capability index, time period index], ordered by capability index
        deliveryCapabilities = np.transpose(np.array([deliveryCapabilityIndices[0],
deliveryCapabilityIndices[1]]))
        print("late capability deliveries\n", deliveryCapabilities)

        #investment results
        print("Investment Matrix\n", investmentMatrix)
        #determine which technologies need to be invested in
        investmentIndices = np.where(investmentMatrix > 0)
        #print out so that the output is [technology, time period, value]
        print("investments by technology")
        for i in range(0, len(investmentIndices[0])):
         printString = ""
         if (i == 0):
          printString = "["
         printString += "[{} {} {}]".format(investmentIndices[0][i], investmentIndices[1][i],
investmentMatrix[investmentIndices[0][i],investmentIndices[1][i]])
         if i == len(investmentIndices[0])-1:
          printString += "]"
         print(printString)
      return (availabilityMatrix, deliveryMatrix, investmentMatrix)

    def quantitativeLeap(functionalMatrix, developmentMatrix, needMatrix, debug = False,
includeInvestmentMatrix = False):
      """
      runs the quantitative leap algorithm.
       t = number of technologies
       c = number of capabilities
       p = number of time periods
      Parameters
      ----------
      functionalMatrix : c x t numpy array
       matrix defining the mapping of capabilities to the technologies that support them.
       Technologies are the columns (t) and the capabilities are the rows (c).
       A one (1) is entered in cell (c, t) if the technology t supports the capability c
      developmentMatrix : t x p numpy array
       matrix defining the development timelines of the technology.
       Technologies are the rows (t) and the time periods are the columns (p)
       Each cell contains the probability (0 to 1) that t will be developed in the time period p
      needMatrix : p x c numpy array
       matrix defining the mapping of capabilities to the time periods in which they are needed
       time periods are the rows (p) and the capabilities are the columns (c)
       A one (1) is entered in cell (p, c) if the capability c is needed in time period p
      debug: boolean
       If true, prints debug information, such as the values of the matrices
      includeInvestmentMatrix: boolean
       If true, calculates the investment matrix. If False, returns an empty array
      Returns
      -------
      tuple of (numpy array, numpy array, numpy array)
      First element of the tuple is the availability matrix, a (p x c) numpy array that defines if a
capability c will be available in time period p
       The value in the cell indicates the probability that the capability will be available in the time
period p
      Second element of the tuple is the delivery matrix, a (p x c) numpy array that defines if a
capability will be delivered late
       The value in the cell indicates the probability (0 to 1) that the capbility will be available when
it is needed
```

```
    Third element of the tuple is the investment matrix, a (t x p) matrix that shows how many late
capabilities that the technology impacts in the time period
     The value in the cell indicates the score of the technology in that time period. Higher scores
indicate a larger impact - larger number of capabilities and/or higher probabilities
    """

    #calculate the availability matrix
    #A=K*(F,V,1)
    availabilityMatrix = np.transpose(leap.KStarFunction(functionalMatrix, developmentMatrix, 1))

    #calculate delivery matrix
    #D = K(N,A,-1)
    deliveryMatrix = leap.KFunction(needMatrix, availabilityMatrix, -1)

    #investment matrix
    #I = transpose(D*F).(J-V)
    #in numpy, the multiplication operator (*) for matrices provides the hadamard product
    #reset J to be size of the development matrix
    if (includeInvestmentMatrix):
     J = np.ones(developmentMatrix.shape)
     investmentMatrix = np.transpose(np.matmul(needMatrix, functionalMatrix))*(J-developmentMatrix)
    else:
     investmentMatrix = []


    #print the results
    if (debug):
     #availability results
     print("Availability Matrix\n", availabilityMatrix)

     #delivery results
     print("Delivery Matrix\n", deliveryMatrix)

     #investment results
     print("Investment Matrix\n", investmentMatrix)

    return (availabilityMatrix, deliveryMatrix, investmentMatrix)

def findDevelopmentProbabilitySingleTrial(predecessorKeyDictionary, schedule, techParameters,
techTechParameters, distributionList, timePeriods,debug = False):
    """
    runs a single instance of the earned value calculatiosn to find the develpoment timelines
    Parameters
    ----------
    schedule : TYPE
     DESCRIPTION.
    techParameters : TYPE
     DESCRIPTION.
    techTechParameters : TYPE
     DESCRIPTION.
    distributionList : TYPE
     DESCRIPTION.
    timePeriods : TYPE
     DESCRIPTION.
    plotTechs : TYPE, optional
     DESCRIPTION. The default is [].
    debug : TYPE, optional
     DESCRIPTION. The default is False.
    outputFile : TYPE, optional
     DESCRIPTION. The default is "".
    maxAllowableValue : TYPE, optional
     DESCRIPTION. The default is 1e20.
    Returns
    -------
    None.
    """
    startTimes = np.zeros(len(techParameters)) #start time of each technology
    endTimes = np.zeros(len(techParameters)) #end time of each technology
```

```
     taskDurations = np.zeros(len(techParameters))
     pens = np.zeros(len(techParameters))
     #set up monte carlo parameters
     (randomDuration, randomTechTechParameters) = leap.getRandomVariables(techParameters,
techTechParameters, distributionList)
     if (debug):
      print("randomTechTechParameters")
      print(randomTechTechParameters)
      print("randomDuration", randomDuration)
     #get the earned value at the transition points for each technology
     ev = leap.getEarnedValueAtTransitionPoints(techParameters, randomTechTechParameters)
     #go through the schedule and determine the start and finish date of each task
     # step 1. get the predecessor tasks
     for j in range(0, len(schedule)):
      predecessorKeys = predecessorKeyDictionary[j]
      predecessorIndices = techTechParameters[predecessorKeys]
      # step 2. for each predecessor, find the utility time. Start time is the maximum utility time + the
start time of the previous task
      startTime = 0
      earliestStartTime = techParameters[schedule[j]][1]
      for k in range(0, len(predecessorIndices)):

       kp = int(predecessorIndices[k][0])
       #do not adjust start time if we are looking at self-inflicted technical debt
       if (kp != int(predecessorIndices[k][1])):
        #find the index into the techTech array - need to have
        p = int(techTechParameters[predecessorKeys][k][1])
        u = techTechParameters[predecessorKeys][k][3]
        tu = leap.getTimeFromEarnedValue(u, ev[p][0], ev[p][1], ev[p][2], ev[p][3], ev[p][4], ev[p][5],
ev[p][6])
        #multiply tu by the duration of the task and add to the start time of the prior task
        pStart = startTimes[int(predecessorIndices[k][1])]
        utilityTime = randomDuration[p]*tu
        startTime = max(earliestStartTime, startTime, utilityTime+pStart)
        if (debug):
         print("***********")
         print("kp", kp,"k", k, "u", u, "p", p)
         print("schedule[j]", schedule[j])
         print("randomDuration", randomDuration[p])
         print("ev[p]", ev[p])
         print('tu', tu)
         print("utilityTime", utilityTime)
         print('pstart', pStart)
         print('tu=1', leap.getTimeFromEarnedValue(1, ev[p][0], ev[p][1], ev[p][2], ev[p][3], ev[p][4],
ev[p][5], ev[p][6]))

      startTimes[schedule[j]] = startTime
      #Step 3. add the duration to the start time for each technology
      # impact of self-inflicted technical debt is in the earned value calculations for utility as seen by
successors
      # need to calculate the time at which the earned value = 1, this is the td penalty - this is the
accumulated td interest
      tdPenalty = leap.getTimeFromEarnedValue(1, ev[j][0], ev[j][1], ev[j][2], ev[j][3], ev[j][4],
ev[j][5], ev[j][6])
      #tdPenalty = 1
      endTimes[schedule[j]] = startTime + randomDuration[schedule[j]]*tdPenalty

      taskDurations[schedule[j]] = randomDuration[schedule[j]]*tdPenalty
      pens[schedule[j]] = (tdPenalty-1)
     return (startTimes, endTimes, taskDurations, pens)

    def threadedFindDevProbabilities(input_index):

     res = leap.findDevelopmentProbabilitySingleTrial(input_index[0], input_index[1], input_index[2],
input_index[3], input_index[4], input_index[5], False)

     return res
```

```
    def findDevelopmentProbabilities(numTrials, schedule, techParameters, techTechParameters,
distributionList, timePeriods, debug = False, maxAllowableValue=1e20, outputDurations = False):
     """
     Determine the probability of each technology completing in each time period by doing a Monte Carlo
analysis
     Parameters
     ----------
     numTrials : int
      number of Monte Carlo trials to run
     schedule : 1d numpy array
      list of technologies, in notional schedule order. This is the order in which the durations will be
evaluated

     techParameters : 2d array
      2d numpy array where the number rows is equal to the number of technologies
      Columns are:
       [0] T - time of peak instantaneous planned value, expressed from 0 to 1
       [1] earliest start time, expressed in time units
       [2] distribution inded for the duration
     techTechParameters : 2d numpy array
      2d numpy array where the number rows is equal to a maximum of the number of technologies squared
(one for each combination)
      Columns are:
       [0] predecessor technology index
       [1] successor technology index
       [2] alpha - impact on successor, from 0 to 1
       [3] U - utility threshold, from 0 to 1
       [4] r distribution index
       [5] tau distribution index
     distributionList : array
      array of distrbution, which are indexed by tech parameters and techtechparameters
     timePeriods : 1d numpy array
      array of time periods, expressed as time units from start for which the development probabilities
will be calculated
     debug : boolean
      if true, prints debug information to the console
     maxAllowableValue: float
      any iterations that return end times for the last schedule item greater that this value will be
ignored
     Returns
     -------
     Development matrix a (t x p) matrix where the rows represent each technology and the columns
represent each time period.
      The value in the cell is the probability of the technology being ready in the time period
     """
     #output array stores each trial in a row and the development time for each technology in the columns
     results = np.zeros((numTrials, len(techParameters)))
     penalties = np.zeros((numTrials, len(techParameters)))
     durations = np.zeros((numTrials, len(techParameters)))
     validTrials = numTrials
     predecessorKeyDictionary = {}
     for i in range(0, len(schedule)):
      predecessorKeyDictionary[i] = np.where(techTechParameters[:,0] == schedule[i])[0]

     inputs = []
     for i in range(0, numTrials):
      inputs.append((predecessorKeyDictionary, schedule, techParameters, techTechParameters,
distributionList, timePeriods))

     with Pool(8) as pool:
      count = 0
      mapResults = pool.map(leap.threadedFindDevProbabilities, inputs)
      for res in mapResults:
       results[count] = res[1]
       durations[count] = res[2]
       penalties[count] = res[3]
       count += 1
```

```
     #remove the invlaid rows
     if (validTrials < numTrials):
      if (debug):
       print("removing ", numTrials-validTrials, "invalid rows")
      results = np.delete(results,slice(validTrials,numTrials,1), 0)
     if (debug):
      print("results")
      print(results)
     # step 5. create the development matrix with the probabilities
     devMatrix = np.zeros((len(techParameters), len(timePeriods)))
     for i in range(0, len(devMatrix)):
      for j in range(0, len(devMatrix[i])):
       timePeriod = timePeriods[j]
       #for each technology (columns of results), count the number of times that it finishes before the
specified time period
       #dev matrix is the cumulative probability
       devMatrix[i,j] = len(np.where(results[:,i]<=timePeriod)[0])/numTrials
     if (outputDurations):
      return (devMatrix, durations)
     else:
      return devMatrix
```

A.3 Example Application from Section 4.3.1.1

```
   from leap_forAppendix import leap, NormalDistribution, ConstantDistribution, TriangularDistribution,
GammaDistribution, DiscreteDistribution, UniformDistribution, CustomTriangularDistribution
   import numpy as np
   import datetime
   from multiprocessing import set_start_method

   if __name__ == "__main__":

    #data import from Williams, Why Monte Carlo Simulations of Project Networks can Mislead
    #tasks
    #ID, Name, Distribution, Triangular Parameters (Min, Most Likely, Max)
    # 0, General Design, Triangular, 4,10,21
    # 1, Engine Design, Triangular, 21,32,55
    # 2, Avionics Design, Triangular, 1, 7, 19
    # 3, D/b airframe design, Triangular, 6,15,32
    # 4, D/b engine manufacture, Triangular, 7,9,11
    # 5, Interim avionics, Triangular, 7,14,27
    # 6, D/b airframe manufacture, Triangular, 8,11,17
    # 7, Assemble d/b aircraft, Triangular, 3,5,10
    # 8, Engine development, Triangular, 20,23,40
    # 9, Engine production, Triangular, 12,13,14
    # 10, Avionics Test, Gamma mean 10 mode 5
    # 11, Avionics flight trials, discrete, relative probability 1:2;1:1 of 4,5,6,24
    # 12, Engine/frame flight trials, discrete, relative probability of 1:2;2:1:0.5 of 5,6,7,8,13
    # 13, Airframe production, Triangular, 12,14,18
    # 14, Avionics Production, Triangular, 14, 16, 24
    # 15, ready to assemble

    #Duration parameters are:
     #[0] T - time of peak instantaneous planned value, expressed from 0 to 1
     #[1] earliest start time, expressed in time units
     #[2] duration distribution type, as an enumerated type from the LEAP class
     #[3] duration mean - mean duration of the technology development in time units
     #[4] duration standard deviation - standard deviation of the time units

    #times are in months
    numTechnologies = 16
    distributionList = []
    durationDistributions = []
    #corresponding data T = 0.395, alpha = 0, alpha0 = 1, U = 1, r = 0, tau = 0 for ALL
    numTrials = 1000
    T = 0.395
    alpha = 0
```

```
runComparativeAnalysis = False
runTDAnalysis = not runComparativeAnalysis
plotTechs = np.array([15])
maxvalue = 200

debug = False
useCompound = False
useLowQualityEngine = False
techParameters = np.zeros((numTechnologies, 3))
 #General design
d0 = TriangularDistribution(4,10,21)
durationDistributions.append(d0)
techParameters[0] = [T, 0, 0]
 #Engine design
d1 = TriangularDistribution(21,32,55)
durationDistributions.append(d1)
engineDesignDistributionIndex = len(durationDistributions)-1
techParameters[1] = [T, 0, 1]
 #Avionics design
d2 = TriangularDistribution(1, 7, 19)
durationDistributions.append(d2)
techParameters[2] = [T, 0, 2 ]
 #D/b airframe design
d3 = TriangularDistribution(6,15,32)
durationDistributions.append(d3)
techParameters[3] = [T, 0, 3 ]
#D/b engine manufacture
d4 = TriangularDistribution(7,9,11)
durationDistributions.append(d4)
techParameters[4] = [T, 0, 4 ]
#interim avionics
d5 = TriangularDistribution(7,14,27)
durationDistributions.append(d5)
techParameters[5] = [T, 0, 5 ]
#d/b airframe manufacture
d6 = TriangularDistribution(8,11,17)
durationDistributions.append(d6)
techParameters[6] = [T, 0, 6]
#assemble d/b aircraft
d7 = TriangularDistribution(3,5,10)
durationDistributions.append(d7)
techParameters[7] = [T, 0, 7]
#engine development
d8 = TriangularDistribution(20,23,40)
durationDistributions.append(d8)
techParameters[8] = [T, 0, 8 ]
#engine production
d9 = TriangularDistribution(12,13,14)
durationDistributions.append(d9)
techParameters[9] = [T, 0, 9 ]

#avionics test
gamma_alpha = 5
gamma_beta = 0.5
d10 = GammaDistribution(gamma_alpha, gamma_beta)
durationDistributions.append(d10)
techParameters[10] = [T, 0, 10]
#avionics flight trials
d11 = DiscreteDistribution([4,5,6,24],[1,2,1,1])
durationDistributions.append(d11)
techParameters[11] = [T, 0, 11]
#engine/frame flight trials
d12 = DiscreteDistribution([5,6,7,8,13],[1,2,2,1,0.5])
durationDistributions.append(d12)
techParameters[12] = [T, 0, 12]
#airframe production
d13 = TriangularDistribution(12, 14, 18)
durationDistributions.append(d13)
techParameters[13] = [T, 0, 13]
```

```
#avionics production
d14 = TriangularDistribution(14,16,24)
durationDistributions.append(d14)
techParameters[14] = [T, 0, 14]
#ready to assemble
d15 = TriangularDistribution(0,0,0)
durationDistributions.append(d15)
techParameters[15] = [T, 0, 15]

for i in range(0, len(durationDistributions)):
 distributionList.append(durationDistributions[i])

#predecessors
#ID, predecessor IDs
# 0, N/A
# 1, N/A
# 2, 0
# 3, 0
# 4, 1
# 5, 2
# 6, 3
# 7, 4,5,6
# 8, 1
# 9, 8
# 10, 2
# 11, 10
# 12, 7
# 13, 12
# 14, 11,12
# 15, 9,13,14

schedule = np.array([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
techTechParameters = np.zeros((35, 6))
#include the dependencies on other technologies
U = 1
rMean = 0
rSD = 0
tauMean=500
tauSD =0
rDist = ConstantDistribution(rMean)
tauDist = ConstantDistribution(tauMean)
distributionList.append(rDist)
rIndex = len(distributionList)-1
baseRIndex = rIndex
distributionList.append(tauDist)
tauIndex = len(distributionList)-1
baseTauIndex = tauIndex


techTechParameters = np.zeros((80, 6))
#include the dependencies on other technologies
U = 1
rIndex = baseRIndex#len(distributionList)-1
tauIndex = baseTauIndex#len(distributionList)-1
rConDist = NormalDistribution(0.5,0.2)
tauConDist = NormalDistribution(0.8,0.1)
distributionList.append(rConDist)
rConIndex = len(distributionList)-1
distributionList.append(tauConDist)
tauConIndex = len(distributionList)-1
if (useCompound):
 edsTauIndex = tauConIndex
 edsRIndex = rConIndex
else:
 edsTauIndex = tauIndex
 edsRIndex = rIndex
engineTauIndex= tauIndex
engineRIndex = rIndex
if (useLowQualityEngine):
```

```
     engineTauIndex = tauConIndex
     engineRIndex = rConIndex

    techTechParameters[0] = [2,0,0.4,U,rIndex, tauIndex] #td in general design impact avionics design
    techTechParameters[1] = [3,0,0.4,U,rIndex, tauIndex] #td in general design impact d/b airframe design
    techTechParameters[2] = [4,1,0.7,U,engineRIndex, engineTauIndex] #td in engine design impact d/b
engine manufacture
    techTechParameters[3] = [5,0,0.1,U,rIndex, tauIndex] #td in general design impact interim avionics
    techTechParameters[4] = [5,2,0.7,U,rIndex, tauIndex] #td in avionics design impact interim avionics
    techTechParameters[5] = [6,0,0.1,U,rIndex, tauIndex] #td in general design impact d/b airframe
manufacture
    techTechParameters[6] = [6,3,0.7,U,rIndex, tauIndex]#td in d/b airframe design impact d/b airframe
manufacture
    techTechParameters[7] = [7,0,0.2,U,rIndex, tauIndex] #td in general design impact assemble d/b
aircraft
    techTechParameters[8] = [7,1,0.1,U,edsRIndex, edsTauIndex]#td in engine design impact assemble d/b
aircraft
    techTechParameters[9] = [7,2,0.1,U,rIndex, tauIndex]#td in avionics design impact assemble d/b
aircraft
    techTechParameters[10] = [7,3,0.2,U,rIndex, tauIndex]#td in d/b airframe design impact assemble d/b
aircraft
    techTechParameters[11] = [7,4,0.1,U,rIndex, tauIndex]#td in d/b engine manufacture impact assemble d/b
aircraft
    techTechParameters[12] = [7,5,0.1,U,rIndex, tauIndex]#td in interim avionics impact assemble d/b
aircraft
    techTechParameters[13] = [7,6,0.1,U,rIndex, tauIndex]#td in d/b airframe manufacture impact assemble
d/b aircraft
    techTechParameters[14] = [8,1,0.7,U,engineRIndex, engineTauIndex]#td in engine design impact engine
development
    techTechParameters[15] = [9,1,0.4,U,edsRIndex, edsTauIndex]#td in engine design impact engine
production
    techTechParameters[16] = [9,8,0.4,U,rIndex, tauIndex]#td in engine development impact engine
production
    techTechParameters[17] = [10,0,0,U,rIndex, tauIndex]#td in general design impact avionics test
    techTechParameters[18] = [10,2,0.7,U,rIndex, tauIndex]#td in avionics design impact avionics test
    techTechParameters[19] = [11,0,0,U,rIndex, tauIndex]#td in general design impact avionics flight
trials
    techTechParameters[20] = [11,2,0.4,U,rIndex, tauIndex]#td in avionics design impact avionics flight
trials
    techTechParameters[21] = [11,10,0.4,U,rIndex, tauIndex]#td in avionics test impact avionics flight
trials
    techTechParameters[22] = [12,0,0.2,U,rIndex, tauIndex]#td in general design impact engine flight
trials
    techTechParameters[23] = [12,1,0.2,U,edsRIndex, edsTauIndex]#td in engine design impact engine flight
trials
    techTechParameters[24] = [12,2,0,U,rIndex, tauIndex]#td in avionics design impact engine flight trials
    techTechParameters[25] = [12,3,0.2,U,rIndex, tauIndex]#td in d/b airframe design impact engine flight
trials
    techTechParameters[26] = [12,4,0.1,U,rIndex, tauIndex]#td in d/b engine manufacture impact engine
flight trials
    techTechParameters[27] = [12,5,0,U,rIndex, tauIndex]#td in interim avionics impact engine flight
trials
    techTechParameters[28] = [12,6,0.1,U,rIndex, tauIndex]#td in d/b airframe manufacture impact engine
flight trials
    techTechParameters[29] = [12,7,0.1,U,rIndex, tauIndex]#td in assemble d/b aircraft impact engine
flight trials
    techTechParameters[30] = [13,0,0.2,U,rIndex, tauIndex]#td in general design impact airframe production
    techTechParameters[31] = [13,1,0,U,edsRIndex, edsTauIndex]#td in engine design impact airframe
production
    techTechParameters[32] = [13,2,0,U,rIndex, tauIndex]#td in avionics design impact airframe production
    techTechParameters[33] = [13,3,0.7,U,rIndex, tauIndex]#td in d/b airframe design impact airframe
production
    techTechParameters[34] = [13,4,0,U,rIndex, tauIndex]#td in d/b engine manufacture impact airframe
production
    techTechParameters[35] = [13,5,0,U,rIndex, tauIndex]#td in interim avionics impact airframe production
    techTechParameters[36] = [13,6,0,U,rIndex, tauIndex]#td in d/b airframe manufacture impact airframe
production
    techTechParameters[37] = [13,7,0,U,rIndex, tauIndex]#td in assemble d/b aircraft impact airframe
production
```

```
    techTechParameters[38] = [13,12,0,U,rIndex, tauIndex]#td in engine flight trials impact airframe
production
    techTechParameters[39] = [14,0,0.2,U,rIndex, tauIndex]#td in general design impact avionics production
    techTechParameters[40] = [14,1,0,U,edsRIndex, edsTauIndex]#td in engine design impact avionics
production
    techTechParameters[41] = [14,2,0.7,U,rIndex, tauIndex]#td in avionics design impact avionics
production
    techTechParameters[42] = [14,3,0,U,rIndex, tauIndex]#td in d/b airframe design impact avionics
production
    techTechParameters[43] = [14,4,0,U,rIndex, tauIndex]#td in d/b engine manufacture impact avionics
production
    techTechParameters[44] = [14,5,0,U,rIndex, tauIndex]#td in interim avionics impact avionics production
    techTechParameters[45] = [14,6,0,U,rIndex, tauIndex]#td in d/b airframe manufacture impact avionics
production
    techTechParameters[46] = [14,7,0,U,rIndex, tauIndex]#td in assemble d/b aircraft impact avionics
production
    techTechParameters[47] = [14,10,0,U,rIndex, tauIndex]#td in avionics test impact avionics production
    techTechParameters[48] = [14,11,0,U,rIndex, tauIndex]#td in avionics flight trials impact avionics
production
    techTechParameters[49] = [14,12,0,U,rIndex, tauIndex]#td in engine flight trials impact avionics
production
    #impacts on ready to assemble
    techTechParameters[50] = [15,0,0.1,1,rIndex, tauIndex]
    techTechParameters[51] = [15,1,0.1,1,edsRIndex, edsTauIndex]
    techTechParameters[52] = [15,2,0.1,1,rIndex, tauIndex]
    techTechParameters[53] = [15,3,0,1,rIndex, tauIndex]
    techTechParameters[54] = [15,4,0,1,rIndex, tauIndex]
    techTechParameters[55] = [15,6,0,1,rIndex, tauIndex]
    techTechParameters[56] = [15,7,0,1,rIndex, tauIndex]
    techTechParameters[57] = [15,8,0,1,rIndex, tauIndex]
    techTechParameters[58] = [15,9,0.1,1,rIndex, tauIndex]
    techTechParameters[59] = [15,10,0,1,rIndex, tauIndex]
    techTechParameters[60] = [15,11,0,1,rIndex, tauIndex]
    techTechParameters[61] = [15,12,0,1,rIndex, tauIndex]
    techTechParameters[62] = [15,13,0.1,1,rIndex, tauIndex]
    techTechParameters[63] = [15,14,0.1,1,rIndex, tauIndex]

    #include the self-inflicted technical debt
    U = 1
    #tech 0 and tech 1 always have alpha 0 = 1 since they have no dependencies
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 0)[0]][:,2])
    techTechParameters[64]= [0,0,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 1)[0]][:,2])
    techTechParameters[65] = [1,1,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 2)[0]][:,2])
    techTechParameters[66] = [2,2,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 3)[0]][:,2])
    techTechParameters[67] = [3,3,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 4)[0]][:,2])
    techTechParameters[68] = [4,4,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 5)[0]][:,2])
    techTechParameters[69] = [5,5,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 6)[0]][:,2])
    techTechParameters[70] = [6,6,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 7)[0]][:,2])
    techTechParameters[71] = [7,7,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 8)[0]][:,2])
    techTechParameters[72] = [8,8,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 9)[0]][:,2])
    techTechParameters[73] = [9,9,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 10)[0]][:,2])
    techTechParameters[74] = [10,10,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 11)[0]][:,2])
    techTechParameters[75] = [11,11,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 12)[0]][:,2])
    techTechParameters[76] = [12,12,alpha0,U,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 13)[0]][:,2])
    techTechParameters[77] = [13,13,alpha0,1,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 14)[0]][:,2])
```

```
    techTechParameters[78] = [14,14,alpha0,1,rIndex, tauIndex]
    alpha0 = 1-np.sum(techTechParameters[np.where(techTechParameters[:,0] == 15)[0]][:,2])
    techTechParameters[79] = [15,15,alpha0,1,rIndex, tauIndex]

    #increments:
     #increment 1: complete design (C0, complete by 43 months)
     #increment 2: manufacture components (C1, complete by 54 months)
     #increment 3: assemble d/b aircraft (C2, complete by 64 months)
     #increment 4: flight trials (C3) complete by 77 months
     #increment 5: final production/ready to assemble (C4): complete by 102 months
    numCapabilities = 5
    satisfactionIndices=[4]
    functionalMatrix = np.zeros((numCapabilities, numTechnologies))
    #design completion
    functionalMatrix[0]= [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    #manufacturing of d/b components
    functionalMatrix[1] = [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    #assemble d/b aircraft
    functionalMatrix[2] = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
    #flight trials
    functionalMatrix[3] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0]
    #ready to assemble
    functionalMatrix[4] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    #time periods to cover the need dates of interest
    timePeriods = np.arange(35,126,1)
    numPeriods = len(timePeriods)
    needMatrix = np.zeros((numPeriods, numCapabilities))
    satisfactionIndex = -1
    for i in range(0, len(timePeriods)):
     if (timePeriods[i]<43):
      needMatrix[i] = [0,0,0,0,0]
     elif (timePeriods[i] < 54):
      needMatrix[i] = [1,0,0,0,0]
     elif (timePeriods[i] < 64):
      needMatrix[i] = [1,1,0,0,0]
     elif (timePeriods[i] < 77):
      needMatrix[i] = [1,1,1,0,0]
     elif (timePeriods[i] < 102):
      needMatrix[i] = [1,1,1,1,0]
     else:
      if (satisfactionIndex == -1):
       satisfactionIndex = i
      needMatrix[i] = [1,1,1,1,1]
    replace = True #if true, changes the r/tau parameters to investigate the impact. if false, decrements
the values instead
    # run cases where we modify the different parameters
    techTechIndicesToModify = np.arange(0, len(techParameters),1)
    rPercentReductions = np.arange(0.1,1,0.1)
    tauPercentReductions = np.arange(0.1,1,0.1)
    metaNumTrials = len(techTechIndicesToModify)*(len(rPercentReductions)*len(tauPercentReductions))
    trials = []
    (baselineDevMatrix, baselineDurations) = leap.findDevelopmentProbabilities(1, schedule,
techParameters, techTechParameters, distributionList, timePeriods, False, maxvalue, outputDurations=True)
    (baselineAvailabilityMatrix, baselineDeliveryMatrix, baselineInvestmentMatrix) =
leap.quantitativeLeap(functionalMatrix, baselineDevMatrix, needMatrix, debug=False,
includeInvestmentMatrix=True)
     #find all successor dependencies in the tech parameters list based on the indices to modify
    for i in range(0, len(techTechIndicesToModify)):
     techTechKeys = np.where(techTechParameters[:,1] == techTechIndicesToModify[i])[0] #index 1 is the
predecessor
     for j in range(0, len(rPercentReductions)):
      for k in range(0, len(tauPercentReductions)):
       trials.append([techTechKeys, rPercentReductions[j], tauPercentReductions[k],
techTechIndicesToModify[i]])
    for i in range(0, len(trials)):
     trial = trials[i]
     print("trial", (i+1), "of", metaNumTrials)
     #reset the distribution for each key in the list
     origValueDictionary ={}
```

```
      for j in range(0, len(trial[0])):
       key = trial[0][j]
       if (techTechParameters[key][0] != techTechParameters[key][1]):
        #ignore the self-inflicted TD
        origValueDictionary[key] = (techTechParameters[key][4],techTechParameters[key][5])
        #need to clone the distribution and add it to the list and decrement the values
        deltaR = trial[1]
        deltaTau = trial[2]
        newRDistribution = distributionList[int(techTechParameters[key][4])].clone()
        newTauDistribution = distributionList[int(techTechParameters[key][5])].clone()
        if (not replace):
         newRDistribution.decrement(deltaR)
         newTauDistribution.decrement(deltaTau)
        else:
         newRDistribution = ConstantDistribution(deltaR)
         newTauDistribution = ConstantDistribution(deltaTau)
         distributionList.append(newRDistribution)
         techTechParameters[key][4] = len(distributionList)-1
         distributionList.append(newTauDistribution)
         techTechParameters[key][5] = len(distributionList)-1

      #run the monte carlo for the new distribution list
      durations = np.zeros((1000, len(techParameters)))
      (devMatrix, durations) = leap.findDevelopmentProbabilities(1000, schedule, techParameters,
techTechParameters, distributionList, timePeriods, False,maxvalue, True)
      (availabilityMatrix, deliveryMatrix, investmentMatrix) = leap.quantitativeLeap(functionalMatrix,
devMatrix, needMatrix, debug=False, includeInvestmentMatrix=True)
      investmentMatrixSum += investmentMatrix
      pMeetingNeed =1
      for p in range(0, len(satisfactionIndices)):
       pMeetingNeed = pMeetingNeed*deliveryMatrix[satisfactionIndex][satisfactionIndices[p]]
       print("p Meeting need",pMeetingNeed)
       for key in origValueDictionary:
        techTechParameters[key][4] = origValueDictionary[key][0]
        techTechParameters[key][5] = origValueDictionary[key][1]
```