Revelations from an Agile and DevSecOps Transformation in a Large Organization: An Experiential Case Study

Thomas, P, Scanlon
Software Engineering Institute, Carnegie Mellon University
scanlon@cert.org

Jose, Morales
Software Engineering Institute, Carnegie Mellon University
jamorales@sei.cmu.edu

ABSTRACT

This paper presents the lessons learned, and the observations and insights gained, from observing a software development effort for a large, well-funded, and highly regulated program that adopted Agile and DevSecOps principles during a 12-month period of iterative software development. The program was originally set up to use the waterfall software development approach with a traditional earned value (EV) scheme. It completed several iterations of development using this structure. The program then shifted to using a combination of Agile and DevSecOps. In this paper, we describe challenges encountered during this transition that inhibited realization of some of the benefits associated with Agile and DevSecOps. Largely, these challenges were a result of poor planning, engineering, and communication. We present this advisory account to others undertaking similar DevSecOps and Agile transitions, particularly in large organizations, so that they may better strategize and prepare methods to diminish similar shortcomings and increase the odds of a successful transition.

CCS CONCEPTS

• **Software and its engineering**; • **Software creation and management**; • **Agile software development**;

KEYWORDS

Agile, DevSecOps, software development, software engineering

ACM Reference Format:
Thomas, P, Scanlon and Jose, Morales. 2022. Revelations from an Agile and DevSecOps Transformation in a Large Organization: An Experiential Case Study. In *Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering (ICSSP'22), May 20–22, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3529320.3529329

1 INTRODUCTION

One of the benefits many organizations expect from adopting Agile principles and DevSecOps software development processes is a leaner, more efficient, and more secure development operation. Agile refers to a set of principles for software development, while DevSecOps refers to the engineering of a software development and delivery platform that incorporates security at every stage. While Agile and DevSecOps can be adopted independently, a symbiotic relationship exists between them that allows for significant impact when implemented together. However, if not implemented properly, the result can be a software pipeline that increases resource consumption (time, money, effort), rather than reducing it. This paper presents an experiential case study from such an occurrence, including an analysis of what wrong, and recommendations for how these pitfalls can be avoided.

The events and activities presented here came from observations of software development efforts for a large, well-funded, and highly regulated program that adopted Agile and DevSecOps principles during a 12-month period of iterative software development. During this period, five program increments were completed, resulting in five major releases of the software. Each program increment included multiple development sprints. The program was structured with a prime contractor (referred to henceforth as the "prime") for software development that employed and managed subcontractors. The program was originally set up to use waterfall software development methods and an earned value (EV) tracking approach. The transition to Agile and DevSecOps was expected to offer significant improvements for the organization, but instead resulted in delivery delays and increased costs.

The observations presented in this paper were captured from first-hand experiences. Members of the research team were often onsite with the development team during the 12-month timeframe. Researchers did not have preconceived research questions in mind during this experience, preferring a more ethnographic approach. They were not empowered to directly alter behaviors and development practices. The research team was present only to evaluate the successes and challenges of the program during their Agile and DevSecOps transition and to deliver guidance to the program office. The researchers delivered regular reports to the project's program office that included observations on the project's Agile and DevSecOps transition activities, review of development processes and practices, and recommendations for improvements. This paper will present five of the largest issues that impacted this project's efforts to adopt Agile principles and DevSecOps practices. After a discussion of each issue, recommendations are offered on how these issues can be addressed. These accounts are offered as an advisory tale so others can avoid these shortcomings in their own Agile and DevSecOps transitions.

2 WORK STRUCTURE INCENTIVIZES TECHNICAL DEBT

The way work priorities were determined after the project's transition to Agile and DevSecOps ended up incentivizing short-sighted

ICSSP'22, May 20–22, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9674-5/22/05…$15.00
https://doi.org/10.1145/3529320.3529329

decisions and the creation of technical debt. The subcontractors focused extensively on exit criteria to qualify development tasks for completion designations. The completion designations were based on percent of requirements satisfied and percent of tests passed. This approach resulted in seeking completion qualification for easier work while more challenging development tasks were "shifted to the right" for completion at a later date. This type of behavior defines technical debt: the extra work you have to do as a result of choosing an easy solution to achieve short-term benefits, which increases cost over time [1].

As the subcontractors performed work, the code delivery process was governed by the prime. Critical to the completion of work was validating that requirements associated with a block of code were fulfilled. This validation was done via functional tests. The prime set a threshold of 80% pass rate for functional tests related to a cluster of code blocks. The 80% threshold left a 20% gap of tests that did not need to pass to claim completion. The subcontractors regularly discussed the current passing percentage for a cluster of code blocks and what needed to be done to minimally surpass the threshold. They used the 20% gap to shift harder coding tasks to the right, incurring technical debt. Every failed test was added to a backlog of pending code fixes to be carried out at a later date. After reaching the required development threshold, remaining failed tasks were converted to defects to be solved in a different program phase. This process created a defect backlog consisting mostly of the hardest work that the subcontractors could not accomplish during the scheduled development time.

By instituting a metric that 80% of test cases must pass for a chunk of code to pass testing, the program set a standard that all test cases are equal. In practice, that is rarely the case. Some tests are always more important than others. From the onset, the team should have used a weighted-value approach to testing. For example, the analytic hierarchy process (AHP) is a subjective weighting method that places more weight on some test cases than others based on the subjective experience and judgment of subject matter experts [2]. Similarly, another test case prioritization method places more weight on some test cases than others based on time factors, defect factors, requirement factors, and complexity factors [3]. Even if these formal approaches to weighted testing were not implemented, the program should have at least incorporated some type of basic weighting into their test process to ensure that the most important test cases for a user request passed testing.

Further, stricter adherence to Agile concepts would have curtailed these activities. Establishing a Definition of Ready for user requests would have included laying out exactly how these requests would be satisfied and would have eliminated opportunities to manipulate test results. Likewise, establishing a Definition of Done would have explicitly mapped out everything needed for the subcontractor to qualify a request as complete. Furthermore, the development teams alone should not have been selecting the tasks to complete. All scheduling should have been done in PI planning events and sprint planning meetings. Decisions on when and how to take on tough tasks should have been made at these events and meetings. There are several meeting techniques, such as Stack Ranking, the Kano Model, the MoSCoW Method, and Cost of Delay [4], that can be used to prioritize requirements and make informed decisions to assume technical debt.

3 ABSENCE OF TEST ENVIRONMENT PARITY MASKS SOFTWARE DEFECTS

Automated testing has frequently been shown to be a critical factor in increasing software quality in DevSecOps environments [5–7]. The subcontractors for this program had a well-defined DevSecOps process in place for automated testing in various stages of development. This process included unit tests, functional tests, and integration tests. These pipelines were virtualized and were meant to be used as a staging environment, loosely emulating the targeted production environment's architecture. However, the virtualized pipeline environment diverged from production in key areas that affected the effectiveness of testing in that environment. There were differences in network topology, storage devices, and user groups and permissions, among other areas. Testing in an environment that was so different from the actual production environment allowed some software defects to go unnoticed during initial testing.

The testing process only validated that the code ran correctly in the test pipeline, which was a simplified replica of the production environment. This virtualized environment did not have the required emulated components to fully test the software components end to end. The testing process in the pipeline did not ensure the code would run in the actual production environment. Many issues arose when significant portions of code were delivered from the program into actual production, often weeks or months after software development was complete. The resources needed to correct many issues were significantly higher after this time elapsed, serving as a penalty for initially testing in an environment that deviated substantially from production. Longer-term quality concerns also arose when some issues discovered during deployment were allowed to remain in the system for future remediation.

To prevent this kind of environment from obscuring and masking software defects, a properly established DevSecOps testing process details the iterative testing of small units of code in isolation (unit test), satisfying requirements (functional test), and properly executing in a production-like environment with previously delivered components (integration test). A pipeline test environment should be provisioned with the same exact Infrastructure-as-Code (IaC) as the production environment. IaC is a method for provisioning DevSecOps environments via code and machine-readable definition files rather than by manually configuring physical hardware. The entire production environment should be reproducible in the pipeline, enabling the repeated execution of automated tests to guarantee all code will function properly in the actual production environment. Additionally, the program could have used container technology to promote greater parity between environments. Containers are a packaged bundle that includes the following in a single, self-contained unit: executable application software code, the software dependencies for the application, and the hardware requirements needed to run the application.

While using containers would greatly reduce the variability in testing, it would not have eliminated all testing discrepancies between production and development environments. The containers would stabilize the foundation that the software is running on, but they would not account for differences in things like data sets and user-activity sequences. In large enterprises that want to test the impact that changes will have on the production environment,

model-based systems engineering (MBSE) should be practiced from the beginning. MBSE is a formal application of modeling to support system requirements, design, analysis, and verification and validation during all software lifecycle phases. A basic precept of MBSE is the use of digital twins. A digital twin is traditionally thought of as a digital representation or virtualized prototype of a physical system, but the digital twin concept is also useful for testing large software systems. A non-production digital twin that is an exact replica of production can be leveraged for performing test cases in a setting that is as real as possible. However, MBSE approaches are not ready-made solutions. It takes a lot of resources to build and maintain a digital twin system, so this solution is advisable only for large programs. The program observed in this study had the resources for such an approach.

4 IMPROPERLY ENGINEERED AND IMPLEMENTED DEVSECOPS PIPELINE YIELDS COSTLY DELAYS

This program was leveraging traditional software development methods for several years before switching to a DevSecOps pipeline. As such, a strategic decision was made to accept some switching costs immediately and then build the complete DevSecOps pipeline over a longer period of time. However, this decision resulted in numerous complications that significantly increased the costs of switching. Continuous integration has been shown to aid in reducing costs [8, 9], but this benefit can be negated by an unreliable pipeline where frequent outages disrupt the entire development process.

In this case, the introduction of a virtualized development environment, provisioned and controlled by the prime, was disruptive for the subcontractors. Prior to introducing this environment, the subcontractors utilized a suite of internal tools and environments for testing. This suite allowed them to quickly make changes to the test environment. When a failure occurred, the developers would go directly to the hardware to investigate and even reset it if needed. Switching to the virtualized environment enabled the prime to manage one development environment for all project participants, which should have yielded savings. However, this new, virtualized environment resulted in negative impacts in the form of delays in its availability and often outright unavailability.

The frequent disruptions in DevSecOps pipeline availability were caused by the pipeline being built on the fly, and new tools and configurations constantly being introduced. This resulted in both planned outages for changes to be made, and unplanned outages due to repeated misconfiguration issues. Subcontractors constantly struggled with sub-optimal development pipelines. Often, the prime team tasked with building and provisioning these environments was focused on other tasking. The need to provide all the necessary tools in the development environments was pushed back months, inhibiting the developers' progress.

Additionally, the pipelines were virtual and accessed via remote servers. The prime failed to maintain the pipelines as highly available services. The primary reason for many development failures and delays was the unavailability of environment pipelines. It was clear that the prime did not fully understand the infrastructure and tooling needed to implement an operational DevSecOps pipeline. The deployment of services needed for the pipeline to run automated tests caused numerous failures. The pipelines appeared to be a complex weave of several commercial products that needed to execute in a particular manner to properly run tests that required hours to complete. The prime did not fully comprehend the detailed intricacies of the intercommunications between these various products. The outcome was that developers had to routinely repeat and delay tests. The DevSecOps pipeline itself turned into a "cost-increasing factory" because developing any software incurred added costs and scheduling delays due to unreliable tooling.

The program office attempted to build the DevSecOps platform in what it considered to be an Agile manner, but, in reality, it was figuring it out as the project went along. In an initiative like this, the DevSecOps platform is the backbone that enables all software development to occur, and it should be well engineered from the onset. Software can be developed in an Agile manner, but the DevSecOps pipeline needs to be planned and engineered. Environment parity, infrastructure as code (IaC), and automated testing are principal components of DevSecOps. It was clear that even though these three concepts are meant to improve software development, for this program, they added layers of complexity and violated a core tenet of DevSecOps: continuous availability. What the program should have done from the beginning was dedicate resources to an architecture spike to properly engineer the DevSecOps platform. There are numerous resources available that present reference architectures and basic guidance for engineering a DevSecOps platform, including free offerings from non-commercial parties such as the GSA DevSecOps Guide [10] or the DoD Enterprise DevSecOps Reference Design [11].

Reference designs provide a menu of components that should be considered for a DevSecOps platform. An evaluation should be done at the beginning of the project to determine which components are absolutely needed to begin work, establishing the minimal viable product (MVP) for the DevSecOps platform itself. Once this MVP DevSecOps platform is established, it should be built and rigorously tested before software development begins. In short, what should have been done to prevent the fragility and unavailability of this DevSecOps platform is to have properly engineered and architected it from the beginning.

5 LACK OF INTEGRATED SECURITY TESTS CREATES RISK

Throughout the engagement, no security-specific testing was conducted during development, functional testing, or integration testing. Moreover, project leadership indicated that using secure coding practices was not a requirement. The main focus of development was to always adhere to the scheduled dates and meet the minimum required exit criteria for various stages of development, test, and delivery. Given the multiple scheduling slips, failures, and delays, it was clear that secure development practices would only add more work and potentially delay the schedule. Before production deployment, some minimal application security testing was performed by the prime. When the prime discovered a security fault, it was usually long after the code weakness was first introduced. This late discovery would require a considerable amount of time and resources to trace it back to the sub, analyze the error, fix it, test it,

redeploy it, and validate it. The lack of integrated security testing throughout the development process, a staple of DevSecOps, was creating risk for the entire organization. Further, the cost to correct software weaknesses that were discovered was significantly higher than it should have been due to the late detection of security issues. The overall lack of comprehensive security testing means that countless security vulnerabilities have been released into the production system, a collection of looming risks that will need to be addressed when a security audit eventually occurs or, worse, when a security incident occurs.

As the name suggests, DevSecOps mandates that security should be a key component of the application pipeline. Security flaws that make it undetected into production are embodiments of "invisible" technical debt which can be considered the most pressing kind of debt [12]. Essentially, the risk of a costly security incident is lurking in the production codebase, unbeknownst to anyone. There are many application security testing tools available, such as static code analyzers, dynamic code analyzers, origin analysis tools, correlation tools, and more [13]. These tools come in both open source and commercial offerings, and most are designed to be fully automated and integrated into DevSecOps pipelines. At the beginning of the project, an evaluation should be performed to determine which application security testing tools are appropriate for this project, and they should then be integrated into the development process.

Further, sufficient time for triaging and remediating security-tool findings should be incorporated into the project schedule. Security scanning tools often generate a lot of output, especially in initial runs before tools are tuned, so the development teams need time to evaluate the findings and take corrective actions. For this reason, these testing tools should be used earlier in the software development process, and preferably integrated throughout the process. Waiting to run security tools until a security-testing stage near the end of the process often yields copious amounts of findings that are costly and time consuming to fix. Performing software security testing at the end of the process, as this program did, rather than integrated it throughout, is an improper, expensive, and risky implementation of DevSecOps.

6 COMMUNICATION ISSUES GENERATE CONFIGURATION-MANAGEMENT ISSUES

In parallel with software development occurring in one environment, changes were made to the configurations used to provision test environments. Once development testing in a sprint was completed in the "old" version of the environment, the software changes would be automatically deployed and tested in a newly provisioned test environment, which had often received dozens of modifications compared to the "old" version. More often than not, this approach led to a break-fix cycle after a software development sprint was declared complete. In addition, overall suboptimal dissemination of environment changes and updates to all related teams resulted in lost time and resources because uninformed staff members would schedule tests on modified environments, which, in most cases, resulted in failure.

Subcontractors would submit issues to the prime's DevSecOps team in an issue tracker for each of these failures. After some investigating by both the prime and the development teams, they typically found the culprit was a change in the environment: either missing files, dormant services, or absent permissions. In these cases, the developers claimed that they did not know about the change, or they were informed that the change would occur at some later date. Since these tests took many hours to run, they were always scheduled to run overnight or over the weekend. A failure typically cost at least a day of productivity. The prime often made changes in the form of custom modifications, updates, upgrades, patches, and the addition or removal of resources. The prime maintained multiple versions of the environments running on pipelines. The changes were almost always implemented on the latest available version of an environment. Developers rarely developed on the latest DevSecOps pipeline version available, because it often was not present when a development pipeline instance was created. Modifications to pipeline versions were poorly communicated, documented, and managed. This resulted in unnecessary wasting of time and money every time a configuration change was introduced.

When incrementally developing both a software product and its corresponding DevSecOps pipeline, a balance must be struck between stability and change. Successful DevSecOps is built on a working collaboration between developers and operations staff [14]. Software development teams must not be permitted to use an outdated version of an integration environment for an extended period of time. At the same time, their work should not be disrupted by constantly shifting sands in the environments used to integrate and test their work products. To empower teams to quickly switch between environments and versions, the program should have adopted the use of container technology from the beginning of the project. Application containers address the problem of getting software to run reliably when moved from one environment to another. Application containers also abstract away the impact of hardware differences in environments. This approach would have reduced, and possibly eliminated, discrepancies between environments in configuration settings, software versions, and dependencies. Containers are a common mechanism used to govern environments in DevSecOps platforms, and a mitigation for incurring costs and delays from configuration issues.

7 CONCLUSION

Agile and DevSecOps offer great promise for making software development faster, cheaper, and more efficient, along with possibly reducing software complexity. However, care should be taken when switching to Agile and DevSecOps to ensure that the new DevSecOps pipeline does not increase resource consumption rather than reduce it. The program in this study had significant resources available to enable a successful transition to Agile and DevSecOps, yet it failed in many ways due to poor planning, engineering, and communication. In short, the program failed to invest in the upfront planning and design needed to make a successful transition. Collectively, all the issues presented here influenced the program's lack of success. Alternative actions they could have taken to increase the likelihood of success are presented throughout the paper. The experiences described here are intended to help others achieve better outcomes by learning from the tribulations of this program.

ACKNOWLEDGMENTS

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM22-007.

REFERENCES

[1] B. A. Lunde and R. Colomo-Palacios, 2020. "Continuous practices and technical debt: a systematic literature review," *2020 20th International Conference on Computational Science and Its Applications (ICCSA)*, pp. 40-44, 2020.

[2] R.V. Vargas, 2010. "Using the analytic hierarchy process (AHP) to select and prioritize projects in a portfolio," *PMI®Global Congress 2010*, North America, Washington, DC. Newtown Square, PA: Project Management Institute.

[3] T. Muthusamy, 2014. "A Test Case Prioritization Method with Weight Factors in Regression Testing Based on Measurement Metrics" [Online]. Available: https://www.semanticscholar.org/paper/A-Test-Case-Prioritization-Method-with-Weight-in-on-Muthusamy/9d76eda59c02cb58198e77c1eb799b550d49f1f2. [Accessed April 14, 2021].

[4] J. Karlsson, 2018. "Four Product Backlog Prioritization Techniques That Work" [Online]. Available: https://www.perforce.com/blog/hns/4-product-backlog-prioritization-techniques-work. [Accessed April 14, 2021].

[5] E. Kula, A. Rastogi, H. Huijgens and G. Gousios, 2019. "Releasing fast and slow: an exploratory case study" In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations* of Software Engineering, 785-795, Aug. 2019. . https://doi.org/10.1145/3338906.3338978

[6] S. M. Agren, E. Knauss, R. Heldal, P. Pelliccione, G. Malmqvist and J. Boden, 2019. "The impact of requirements on systems development speed: a multiple-case study in automotive," *Requir. Eng.*, vol. 24, no. 3 (Sept 2019), 315-340.

[7] Z. Li, P. Avgeriou and P. Liang, 2015. "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, (March 2015), 193-220.

[8] J. Holvitie *et al.*, "Technical debt and agile software development practices and processes: An industry practitioner survey", *Inf. Softw. Technol.*, vol. 96, pp. 141-160, Apr. 2018.

[9] B. Adams and S. McIntosh, 2016. "Modern Release Engineering in a Nutshell - Why Researchers Should Care" *IEEE 23rd International Conference on Software Analysis Evolution and Reengineering (SANER)*, vol. 5, (March 2016),78-90.

[10] GSA, U.S. General Services Administration, "DevSecOps Guide," 2020. [Online]. Available: https://tech.gsa.gov/guides/dev_sec_ops_guide/. [Accessed April 9, 2020].

[11] DoD CIO, 2019. "DoD Enterprise DevSecOps Reference Design".

[12] [12] P. Kruchten, R. L. Nord and I. Ozkaya, 2-12. "Technical Debt: From Metaphor to Theory and Practice" *IEEE Softw.*, vol. 29, no. 6, (Nov 2012) 18-21.

[13] [13] T. Scanlon, 2018. "Types of Application Security Testing Tools: When and How to Use Them" *SEI Insights*, Software Engineering Institute. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2018/07/10-types-of-application-security-testing-tools-when-and-how-to-use-them.html. [Accessed April 14, 2021].

[14] C. Ebert, G. Gallardo, J. Hernantes and N. Serrano, 2016. "DevOps" *IEEE Softw.*, vol. 33, no. 3, (May 2016) 94-100.