# Quantifying the Impact of Staged Rollout Policies on Software Process and Product Metrics

Kenan Chen, University of Massachusetts Dartmouth, USA
Zakaria Faddi, University of Massachusetts Dartmouth, USA
Vidhyashree Nagaraju, PhD, University of Tulsa, USA
Lance Fiondella, PhD, University of Massachusetts Dartmouth, USA

Key Words: availability, DevSecOps, process performance, reliability, security

*SUMMARY & CONCLUSIONS*

Software processes define specific sequences of activities performed to effectively produce software, whereas tools provide concrete computational artifacts by which these processes are carried out. Tool independent modeling of processes and related practices enable quantitative assessment of software and competing approaches. This paper presents a framework to assess an approach employed in modern software development known as staged rollout, which releases new or updated software features to a fraction of the user base in order to accelerate defect discovery without imposing the possibility of failure on all users. The framework quantifies process metrics such as delivery time and product metrics, including reliability, availability, security, and safety, enabling tradeoff analysis to objectively assess the quality of software produced by vendors, establish baselines, and guide process and product improvement. Failure data collected during software testing is employed to emulate the approach as if the project were ongoing. The underlying problem is to identify a policy that decides when to perform various stages of rollout based on the software's failure intensity. The illustrations demonstrate how alternative policies impose tradeoffs between two or more of the process and product metrics.

## *1 INTRODUCTION*

Software has transformed modern society in many ways enabling a spectrum of products and services. Software engineering [1] is a large and vibrant field driving innovations, and the processes and tools employed have experienced substantial change over a period of decades. An undesirable side effect of this evolution has been the proliferation of buzzwords that attempt to brand new techniques but often confuse many outside of the discipline, impeding organizations' from producing high quality software. Terms like agile methods [2], DevOps (Development and IT Operations) [3], DevSecOps (Development, Security, and IT Operations) [4], site reliability engineering (SRE) [5], and continuous integration and continuous delivery/deployment (CI/CD) [6] are examples of modern software practices that have emerged. The CI/CD pipeline forms the backbone of modern day DevOps operations, providing a framework for automation that promotes problem identification and resolution. DevOps is an emerging set of practices, including agile methods, which combines software development and operations. Despite these advances, academic research has not kept pace with the software industry. For example, software reliability growth models [7] were developed in the 1970's when the Waterfall Model [8] was widely employed. Models of modern software practices are needed to capture multiple process and product metrics such as reliability, availability, security, and safety as well as tradeoffs among these attributes.

The most prevalent body of research on modern software development practices focuses on process related issues [9], [10] and case studies [11], [12], which offer empirical evidence of DevOps' efficacy. The promises of DevOps are often echoed in such studies, and while there is an established body of literature on enabling technologies such as cloud computing infrastructure [12], [14], few efforts to formally model DevOps specific activities [15]-[17] have been carried out.

This paper develops a framework to quantify the tradeoffs among alternative staged rollout policies, which deploy new software features to progressively larger fractions of the user base. The framework quantifies process metrics such as delivery time and product metrics, including reliability, availability, security, and safety. Greater modeling rigor is achieved by deconstructing the popular definition of DevOps, which Bass et al. [3] define as, *"a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality."* Reducing the time between committing a change and placing it into production is a performance engineering problem, while quality implies reliability engineering and related measures. A software failure dataset consisting of multiple severities is employed to illustrate the assessments enabled by the framework. The results indicate that the approach promotes objective comparison of alternative staged rollout policies, which can be used to establish baselines, assess the maturity of an organization, and inform process improvement efforts.

The remainder of this paper is organized as follows: Section 2 describes the modeling framework and defines metrics for the tradeoffs considered. Section 3 illustrates the approach, examining tradeoffs for a range of policies. Section 4 concludes prioritizing goals for future research.

## *2 MODELING FRAMEWORK*

This section develops a model of staged rollout to characterize the tradeoff between (i) the time to deliver new functionality and (ii) the downtime incurred. This model explicitly specifies the parameters for the underlying decisions, enabling reasoning about process improvement and formulation of optimal staged rollout policies.

### 2.1 *Staged rollout of software*

Staged rollout of software [5] has been praised as a strategy to field new functionality on an ongoing basis without incurring failures that induce system outages, widespread unavailability of services, economic losses, and user dissatisfaction. The rationale for staged rollout is to publish an updated software possessing new functionality for use by a subset of the user base to avoid the major problems described above, but also to accelerate the discovery of defects. The development team then attempts to correct the source of the problem and begins the process of staged rollout anew.

Figure 1 shows a simple state diagram of staged rollout, which represents a concrete instance of the framework.

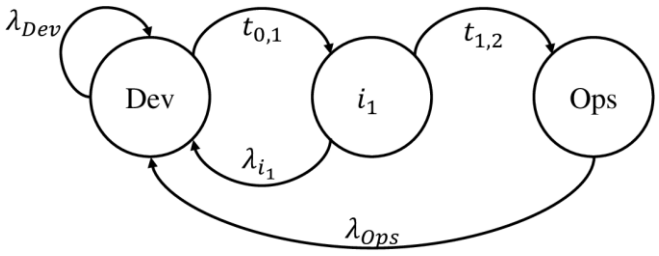


Figure 1: State diagram of staged rollout

State *Dev* represents the development state, where software is tested by an internal team. An elementary model of the traditional approach to software updates simply transitions to the *Ops* state, once the software is deemed satisfactory with respect to functional requirements, reliability, and other desired attributes, where the *Ops* state exposes the software to the entire base of $n_{Ops}$ users. Staged rollout, instead, transitions from the *Dev* state to state $i_1$, which represents the first stage of staged rollout, where the software is published for use by $p_{i_1}$ percent or $n_{i_1} = p_{i_1} \times n_{Ops}$ of the user base. In general, multiple stages of staged rollout between development and full deployment are possible. This more general case possesses $m$ intermediate staged rollout states, it is reasonable to assume that each transition from $i_j$ to $i_{j+1}$ increases the fraction of the user base exposed to the software such that $p_{i_j} < p_{i_{j+1}}$ and $n_{i_j} < n_{i_{j+1}}$. Failure in any state transitions to the *Dev* state, where root cause analysis and defect removal are attempted.

The state model described in Figure 1 enables explicit consideration of the tradeoffs between downtime and delivery time. Downtime is determined by the state in which the failure occurs and is proportional to the fraction of the user base $n_{i_j}$ and mean time to repair (MTTR). Thus, failure in the $j^{th}$ state of staged rollout ($i_j$) contributes less to downtime than failure in the state $i_{j+1}$. However, the defect exposure rate in the $j^{th}$ state of staged rollout ($\lambda(i_j)$) is also less than $\lambda(i_{j+1})$, meaning that the downtime experienced, and the time required to discover and remove all defects are competing constraints. Thus, minimizing downtime by remaining in the *Dev* state until all defects have been detected and removed will likely delay delivery time. Similarly, unrestrained transition to the *Ops* state immediately after each defect is repaired is likely to exacerbate downtime. Therefore, it may not be possible to simultaneously minimize downtime and delivery time, posing a multi-objective problem. Moreover, organizations implementing staged rollout, or their customers will express different levels of tolerance for these undesirable outcomes. Subsequently, it is unlikely that a single optimal policy or "one size fits all" approach to staged rollout exists. Instead, it is necessary to select transition times $t_{j,j+1}$ that balance downtime and delivery time in a manner that is satisfactory to the customer. Intuitively, a high failure rate in the *Dev* state is likely to indicate that additional failures will occur. Hence, staged rollout should not be performed because it would risk greater downtime. Therefore, the problem is to select numerical values of transition times $t_{j,j+1}$ that achieve the desired balance between downtime and delivery time.

### 2.2 *Software failure data*

This section describes how times series of software failure data collected during testing can be used to drive the staged rollout model described in Section 3 in order to explicitly consider the impact of failures on downtime and delivery time.

Figure 2 shows the timeline of the SYS1 dataset [7], which includes 136 unique defects discovered over 88,682 seconds or approximately 24.63 hours of testing.


Figure 2: Timeline of failures during testing in the SYS1 dataset

Figure 2 indicates that a majority of the failures (nearly 60%) occurred during the first 20,000 seconds and less than 10% were discovered in the last third of testing after 60,000, suggesting that optimal transition times $t_{j,j+1}$ may be time-varying.

#### 2.2.1 *Delivery time*

To model the impact of staged rollout on delivery time, we assume that one unit of time in the *Dev* state advances the SYS1 timeline by one unit, whereas time in the staged rollout and *Ops* state accelerate the rate at which time advances on the SYS1 timeline proportional to the percentage of the user base. Therefore, staged rollout may be regarded as a modern form of accelerated life testing (ALT) [18] for software. For example,

if the complete user base is composed of $n_{Ops} = 10{,}000$ users and staged rollout exposes new functionality to $p_{i_1} = 0.1$ or 10% of the user base, then $n_{i_1} = 1{,}000$. Similarly, if $n_{Dev} = 50$, then a simple method to compute the acceleration factor in each state of staged rollout is the ratio between the number of users in a state over the baseline in the *Dev* state such that the acceleration factor in the staged rollout and *Ops* states are $a_{i_1} = \frac{n_{i_1}}{n_{Dev}} = 20$ and $a_{Ops} = \frac{n_{Ops}}{n_{Dev}} = 200$ respectively. This simplifying assumption can be improved, since testers are familiar with the functionality and intentionally stress the program to expose defects. Modeling these ratios is a research question that requires staged rollout data. Nevertheless, the simplifying assumptions made here enable a quantitative framework upon which to improve.

The preliminary assumption of linear acceleration factors described above provides a concrete starting point to measure the cumulative time required to reach the 136th failure. Specifically, *delivery time* may be defined as the time to reach this final failure plus the time to transition from the *Dev* to *Ops* state or $t_{Dev,i_1} + t_{i_1,Ops}$ under the simplifying assumption that the final defect is repaired immediately. Modeling advances that explicitly consider the time between defect discovery and resolution [19] can further enhance the realism of the staged rollout deployment model.

#### 2.2.2 *Downtime*

To model the impact of staged rollout on downtime, we assume that failure in the *Dev* state does not incur downtime, since only internal testing is performed at this stage. However, downtime incurred in the staged rollout and *Ops* states are proportional to the fraction of the user base multiplied by the mean time to repair such that the accumulated downtime increases by $p_{i_{1_1}} \times MTTR$ or $p_{Ops} \times MTTR$. This simplifying assumption may be conservative, since not all users exposed to the functionality will necessarily experience the failure. Similar to delivery time, the downtime experienced must be modeled from staged rollout data and has important implications for identifying an optimal deployment policy for transition times $t_{j,j+1}$, since conservative assumptions may unnecessarily increase delivery times. Thus, our preliminary model expresses *downtime* as the weighted sum $MTTR \times \sum_{i=1}^{n} p_{s(i)}$, where $s(i)$ denotes the state in which the $i^{th}$ failure occurs.

#### 2.2.3 *Safety and Security*

In some cases, failures may produce consequences of varying severity. Safety and security related failures are two examples. In either case, failures of higher severity correspond to greater economic damage or other undesirable outcomes. As an example, one class of security related failures of widespread concern is information loss, whether intellectual property produced by private industry or government secrets related to national security. In either case, the consequences of information loss will be proportional to the severity of the loss. For example, historical documents on the principles for classification of information [20], assume that Confidential, Secret, and Top Secret data differ in severity by an order of magnitude. Thus, if the loss of Confidential information is assigned unit cost $\$_c = 1.0$, then the corresponding loss of Secret and Top Secret information are $\$_s = 10$ and $\$_{ts} = 100$ respectively. Assuming that failure in the *Dev* state does not lead to information loss, but that all other states do, the *data loss* is $\sum_{i \notin Dev} \$_{sev(i)}$, where $sev(i)$ is the severity of the $i^{th}$ failure occurring in a non *Dev* state.

## 3 ILLUSTRATIONS

This section illustrates the proposed approach to assess alternative staged rollout policies. Section 3.1 provides a detailed walkthrough for a single policy, clarifying the logic. Section 3.2 subsequently illustrates tradeoffs between delivery time and downtime in the context of the SYS1 data set [7] and examines the impact of policies on delivery time and downtime in isolation. Section 3.3 illustrates tradeoffs between delivery time, downtime, and safety through a NASA data set [19].

### 3.1 *Policy evaluation*

For the sake of exposition, this section assumes that, for the staged rollout model described in Figure 1, $MTTR = 10$ and that $t_{Dev,i_1} = 35$ and $t_{i_1,Ops} = 350$, which are referred to as Policy 1.

Figure 3 illustrates the impact of the Policy 1 on the first 250 time units of the SYS1 dataset.

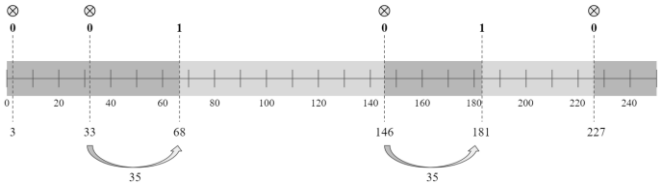


Figure 3: Timeline of failures in the first 250 seconds of testing

The staged rollout process begins in state *Dev* at $t = 0$. Since the first failure occurs at time $t_1 = 3$, the $\lambda_{Dev}$ transition is taken and the timer is reset to the 35 units of time that must elapse before transitioning to the staged rollout state $i_1$. The second failure at $t_2 = 33$ also triggers the $\lambda_{Dev}$ transition and resets the timer. However, after 35 additional time units without failure, transition to the staged rollout state occurs at time $t = 68$. The 20-fold acceleration factor achieved by the larger user base means that only $\frac{t_3 - 68}{a_{i_1}} = \frac{146-68}{20} = 3.9$ units of time are needed to reach the third failure at time $t_3 = 146$, but also incurs downtime $p_{i_1} \times MTTR = 0.1 \times 10 = 1.0$. Thus, only $68 + 3.9 = 71.9$ seconds are required to uncover the first three failures, but at the expense of some downtime, demonstrating how the quantitative tradeoffs are captured by the model. This process continues, transitioning to staged rollout at time 181 after no failures in the *Dev* state, failing at time $t_4 = 227$ after 2.3 time units and incurring an additional 1.0 unit of downtime. Continuing this process through all 136 failures shown in Figure 2 plus the time to transition from *Dev* to $i_1$ and $i_1$ to *Ops*, the delivery time and downtime are 6,031.59 and 599.08 respectively.

To illustrate the tradeoffs imposed by alternative policies on the SYS1 dataset, a similar experiment was performed with Policy 2, possessing parameters $t_{Dev,i_1} = 125$ and $t_{i_1,Ops} = 800$, which produced delivery time 15,808.43 and downtime 455.65. Thus, the larger and more conservative transition times of Policy 2 reduced the downtime by nearly 24%, but more than doubled the delivery time.

### 3.2 *Tradeoff analysis*

The previous example demonstrated how a staged rollout policy produces a tuple composed of downtime and delivery time and that no single policy is necessarily optimal with respect to both of these objectives. Therefore, the example in this section further explores the tradeoffs between downtime and delivery time by emulating various policies on the SYS1 data set. For the sake of illustration, two experiments were conducted. The first applied 100 pairwise combinations of policies with $t_{Dev,i_i} = \{10, 20 \ldots, 100\}$ and $t_{i_1,Ops} = \{10, 20, \ldots, 100\}$ and the downtime and delivery calculated. The second applied 100 pairwise combinations of policies with $t_{Dev,i_i} = \{100, 200 \ldots, 1{,}000\}$ and $t_{i_1,Ops} = \{100, 200, \ldots, 1{,}000\}$.

Figure 4 shows the results of the two experiments described above. The line indicates Pareto optimal policies.

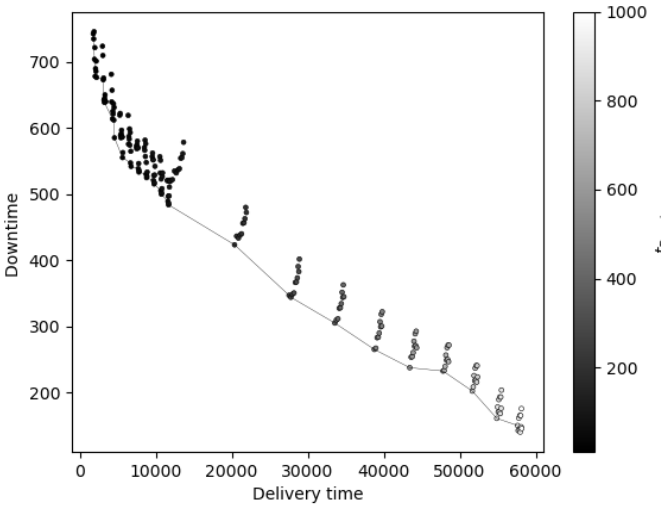


Figure 4: Tradeoff between downtime and delivery time for alternative staged rollout policies

The denser set of points in the upper left of Figure 4 correspond to the first experiment, while the remaining points were produced by the second experiment. Figure 4 indicates that policies with lower downtime possess higher delivery times and that lowering delivery time tended to increase downtime. Each point corresponds to a policy and the shading indicates the value of $t_{Dev,i_1}$. Values of $t_{Dev,i_1} = 10$ resulted in greater downtime, which agrees with intuition because more failures in the staged rollout and *Ops* state occurred. Shading the policies according to the value of $t_{i_1,Ops}$ produced similar results.

Figure 4 also shows that the range of delivery times $(1793.71, 58148.42)$ is wider than the range of downtimes $(140.35, 746.05)$. However, these ranges are a function of the numerical parameters chosen for the sake of illustration and assumptions. For example, delivery time is determined by the duration of the failure data and acceleration factors in staged rollout states, whereas downtime is influenced by MTTR. Since the time of the last failure in the SYS1 dataset is $t_{136} = 88{,}682$, the minimum delivery time compressed the testing schedule by a factor of 49.44 $(88{,}682/1{,}793.71)$ to only 2.02% $(1{,}793.71/88{,}682)$ of the original time, while the maximum delivery time only compressed the testing schedule by a factor of 1.53 to 65.6%. However, decreasing delivery time from the largest to smallest values also increased downtime by a factor of 5.32 $(746.05/140.35)$.

Ultimately, the selection of a policy depends on the subjective preference of the organization or their customer. Downtime may incur loss of sales and customers, fines by regulators, or other negative consequences, while late delivery also possesses negative consequences such as reduced market capture.

Figures 5 and 6 provide alternative perspectives on the impact of the state transition times of staged rollout policies on delivery and downtime respectively.

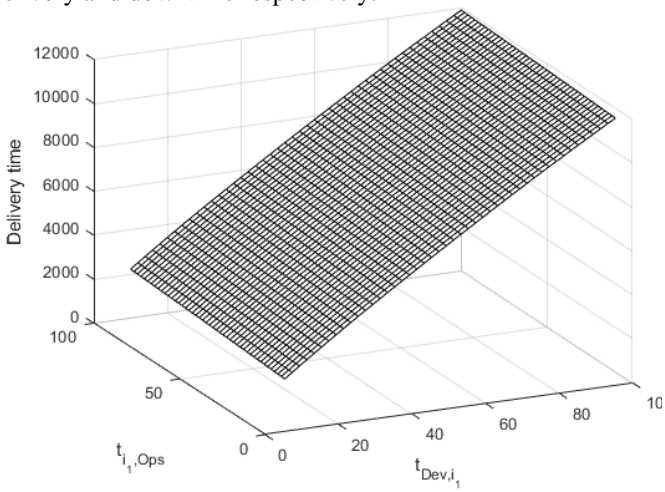


Figure 5: Impact of staged rollout policy on delivery time

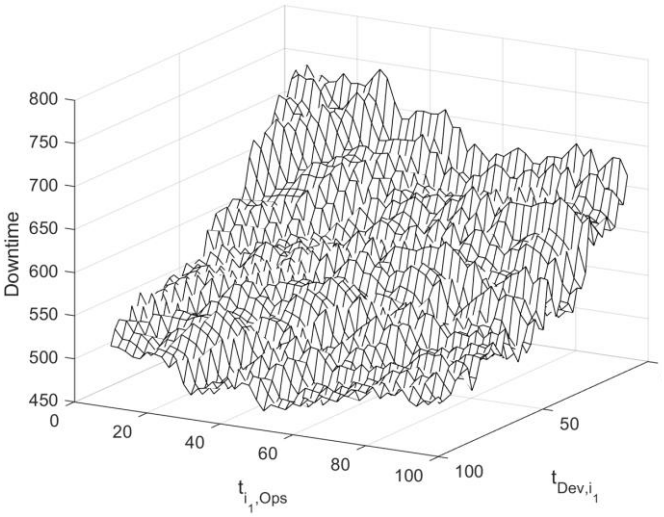


Figure 6: Impact of staged rollout policy on downtime

Figure 5 indicates that increasing $t_{Dev,i_1}$ and $t_{i_1,Ops}$ both

increase delivery time, but that increasing $t_{Dev,i_1}$ increases delivery time more quickly. This observation also agrees with intuition because progress on the testing timeline proceeds most slowly in the $Dev$ state. Similarly, Figure 6 indicates that increasing $t_{Dev,i_1}$ and $t_{i_1,Ops}$ both decrease downtime, but that increasing $t_{Dev,i_1}$ decreases downtime more quickly because failures in the $Dev$ state do not to incur downtime.

### 3.3 *Safety and Security*

Figure 7 shows tradeoffs between severity, downtime, and delivery time for alternative staged rollout policies in the context of a NASA data set [19], which is similar to the timeline given in Figure 2 but also assigned one of three levels of severity to each defect discovered.

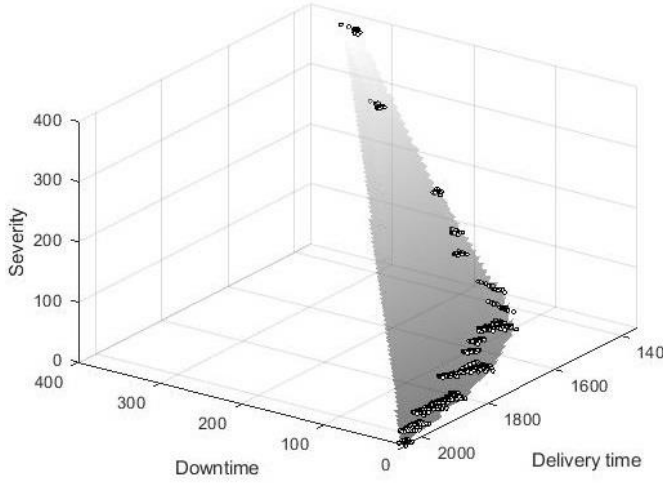


Figure 7: Tradeoff between severity, downtime, and delivery time for alternative staged rollout policies

Here, severity indicates the accumulated consequences of failures in the staged rollout or $Ops$ state of low, medium, and high severity incurring costs of $\$_l = 1.0$, $\$_m = 10$ and $\$_h = 100$ respectively. The primary observation is that lowering the accumulated severity of unsafe failures requires increasing the delivery time because policies that require more time between failure before transitioning to later states of staged rollout correspond to more conservative strategies that spend more time in earlier states with smaller acceleration factors. Figure 7 also indicates that lowering the accumulated severity of unsafe failures is positively correlated with decreased downtime, since downtime is associated with the staged rollout and *Ops* state, where failures also incur a penalty according to their severity.

Plots similar to Figures 5 and 6 were created to assess the impact of transition times $t_{Dev,i_1}$ and $t_{i_1,Ops}$ on delivery time, downtime, and severity. However, the results of this analysis were similar to the trends observed in Figures 5 and 6. Moreover, the impact of transition times on severity was similar to the trends observed in Figure 7, where increasing $t_{Dev,i_1}$ decreased severity more rapidly than $t_{i_1,Ops}$ because failures in the *Dev* state did not contribute to the cumulative severity experienced by the end of testing.

## *4 CONCLUSIONS AND FUTURE WORK*

This paper presented a framework to assess the staged rollout approach performed in agile methods, DevOps, DevSecOps, site reliability engineering, and continuous integration and continuous delivery/deployment (CI/CD). The approach enables objective assessment of process and product metrics for vendors to strive for improvement and customers to measure the quality of software produced by their suppliers. The approach was demonstrated with data traditionally employed by software defect and vulnerability discovery, tracking, and resolution models. The illustrations examined how alternative policies impose tradeoffs between two or more of the process and product metrics.

Future research will extend the models upon which policies are based to further formalize the security dimension of the tradespace as well as to determine if more than one intermediate staged rollout state can improve tradeoffs between process and product metrics. Techniques that automate the staged rollout decision-making process such as reinforcement learning will also be explored to solve constrained optimization problems such as minimizing downtime while achieving a specified delivery date or alternative combinations of primary objectives and constraints.

## *ACKNOWLEDGMENTS*

This material is based upon work supported by the National Science Foundation under Grant Number (#1749635). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## *REFERENCES*

1. R.S. Pressman, *Software Engineering: A Practitioner's Approach*. Palgrave Macmillan, 2005.
2. R. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.
3. L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.
4. K. Carter, "Francois Raynaud on DevSecOps," *IEEE Software*, vol. 34, no. 5, pp. 93-96, 2017.
5. B. Beyer, C. Jones, J. Petoff, and R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.
6. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.
7. M. Lyu, *Handbook of Software Reliability Engineering*, 2nd ed., New York, NY: McGraw-Hill, 1996.
8. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *International Conference on Software Engineering*, 1987.
9. C.-P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villaviencio, J. Walter et al., "How is performance

addressed in devops? A survey on industrial practices," arXiv preprint arXiv:1808.06915, 2018.
10. R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer, "Towards a benefits dependency network for devops based on a systematic literature review," *Journal of Software: Evolution and Process*, vol. 30, no. 11, p. e1957, 2018.
11. L. Riungu-Kalliosaari, S. Makinen, L. Lwakatare, J. Tiihonen, and T. Mannist, "DevOps adoption benefits and challenges in practice: A case study," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2016, pp. 590–597.
12. M. Senapathi, J. Buchan, and H. Osman, "Devops capabilities, practices, and challenges: Insights from a case study," in Proceedings of the *22nd International Conference on Evaluation and Assessment in Software Engineering* 2018, 2018, pp. 57–67.
13. W. Pourmajidi, A. Miranskyy, J. Steinbacher, T. Erwin, and D. Godwin, "Dogfooding: Use IBM cloud services to monitor IBM cloud infrastructure," *arXiv preprint* arXiv:1907.06094, 2019.
14. K. Hwang, X. Bai, Y. Shi, M. Li, W. Chen, and Y. Wu, "Cloud performance modeling with benchmark evaluation of elastic scaling strategies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 130–143, 2015.
15. F. Zhao, X. Niu, S. Huang, and L. Zhang, "Reproducing scientific experiment with cloud devops," *arXiv preprint* arXiv:1910.13397, 2019.
16. C. Bansal, S. Renganathan, A. Asudani, O. Midy, and M. Janakiraman, "Decaf: Diagnosing and triaging performance issues in large-scale cloud services," arXiv preprint arXiv:1910.05339, 2019.
17. M. Guerriero, M. Ciavotta, G. P. Gibilisco, and D. Ardagna, "A model driven devops framework for QoS-aware cloud applications," in IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2015, pp. 345–351.
18. W. Nelson, *Accelerated Testing: Statistical Models, Test Plans, and Data Analysis*. John Wiley & Sons, 2009, vol. 344.
19. M. Nafreen, V. Nagaraju, M. Luperon, Y. Shi, T. Wandji, and L. Fiondella, Connecting Software Reliability Growth Models to Software Defect Tracking, in *International Symposium on Software Reliability Engineering*, Coimbra, Portugal, pp. 138-147, Nov 2020.
20. A. Quist, "Security Classification of Information. volume 2, *Principles for Classification of Information*," Retrieved September, vol. 30, p. 2004, 1993.

*BIOGRAPHIES*

Kenan Chen
Department of Electrical and Computer Engineering
University of Massachusetts - Dartmouth
285 Old Westport Road
North Dartmouth, MA 02747, USA

e-mail: kchen3@umassd.edu

Kenan Chen is a Computer Engineering major at the University of Massachusetts Dartmouth (UMassD).

Zakaria Faddi
Department of Electrical and Computer Engineering
University of Massachusetts - Dartmouth
285 Old Westport Road
North Dartmouth, MA 02747, USA

e-mail: zfaddi@umassd.edu

Zakaria Faddi is a double major in Electrical and Computer Engineering at UMassD.

Vidhyashree Nagaraju, PhD
Tandy School of Computer Science
University of Tulsa
800 South Tucker Drive
Tulsa, OK, 74107, USA

e-mail: vidhyashree-nagaraju@utulsa.edu

Vidhyashree Nagaraju is an assistant professor in the Tandy School of Computer Science at the University of Tulsa. She received her PhD (2020) in Computer Engineering from UMassD.

Lance Fiondella, PhD
Department of Electrical and Computer Engineering
University of Massachusetts - Dartmouth
285 Old Westport Road
North Dartmouth, MA 02747, USA

e-mail: lfiondella@umassd.edu

Lance Fiondella (S'07-M'12-SM'20) received the Ph.D. degree in computer science & engineering from the University of Connecticut, Storrs, CT, USA, in 2012.

He is an associate professor of Electrical and Computer Engineering at the University of Massachusetts Dartmouth and the Director of the UMassD Cybersecurity Center, a NSA/DHS designated Center of Academic Excellence in Cyber Research (CAE-R). His peer-reviewed conference papers have been the recipient of 12 awards, including five as first author and seven with his students as first author. His research has been funded by the Department of Homeland Security, NASA, U.S. Department of Defense, and National Science Foundation, including a CAREER Award. He is an associate editor of the Military Operations Research Journal and the North American Regional Editor of the International Journal of Performability Engineering.

Dr. Fiondella served as the vice-chair of IEEE Standard 1633: Recommended Practice on Software Reliability from 2013-15 and a three-year term as a Member of the Administrative Committee of the IEEE Reliability Society from 2015-2017. He presently serves as a technical committee chair of the Annual IEEE Symposium on Technologies for Homeland Security.