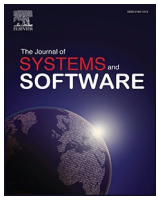
The Journal of Systems and Software 213 (2024) 112030

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss



Investigating effectiveness and compliance to DevOps policies and practices for managing productivity and quality variability

Dan Port [a,*], Bill Taber [b], Parisa Emkani [c]

[a] Shidler College of Business University of Hawaii at Manoa, Information Technology Management Department, United States
[b] Jet Propulsion Laboratory, California Institute of Technology Mission Design and Navigation, United States
[c] College of Natural Sciences, University of Hawaii at Manoa, Information and Computer Sciences Department, United States

ARTICLE INFO

Keywords:
DevOps
Software process
Software maintenance
Software reliability
Software quality
Software productivity

ABSTRACT

The Mission Design and Navigation Software (MDN) Group at the Jet Propulsion Laboratory (JPL) develops and continuously maintains software systems critical for NASA deep space missions. Due to limited resources and tight schedules, there is always the temptation to prioritize productivity over quality. However, such quality and productivity variations may be manageable during development, but they are too risky for operational use when there is a time-critical need for repair or enhancement and high reliability is essential.

As a result, our process must be both highly productive and maintain high quality (e.g., reliability, maintainability, usability). Inspired by the "quality is free" paradigm, we have instituted six policies established from DevOps principles and practices specifically to address maintenance risk caused by high variability in quality and productivity concerns we encountered in the development phase from quality-productivity tradeoffs.

CMMI level 3 requirements mandate that process improvements are monitored and adjusted to ensure that policies are adhered to, practices are performed, and that they are effective in producing the desired results. This article presents a process improvement investigation as to whether the DevOps practices, as currently implemented, are effective in achieving the expected effects and impacts for managing variability in quality and productivity. Additionally, we investigate practical methods to assure compliance to the six policies to determine if any adjustments to policy or practice are needed.

For this investigation we focus on our flagship system MONTE. We have over 15 years of reliable and accurate quality and productivity data for MONTE, a critical system currently in continual operation and maintenance. Time series cross-correlation analyses were used to compare process productivity and quality characteristics before and after the implementation of DevOps. From this, we found strong evidence that:

- There is a continual maintenance risk due to variability in productivity and quality.
- The majority of the DevOps policies and practices are being complied with.
- The policies and practices have been effective in managing this risk.

1. Introduction

The JPL Mission Design and Navigation Software Group (MDN) develops and maintains several software systems used for NASA's deep space missions. Enhancing system capabilities and repairing bugs encountered during operation in a predictable manner are critical to the success of those missions. It is simply too risky to have a critical system non-operational for an arbitrary or unknown period, or worse, operate with uncertain reliability or capability during a mission. Bugs and enhancement demands are inevitable, continuously introducing risk of system failure or insufficiency for operation. Therefore, we aim to manage our maintenance process so we can, with high confidence, always repair critical bugs and implement needed new features within the timeframe demanded by our users[1]. To obtain high confidence in our process, we must control the variability in the quality of our releases and the productivity in implementing needed enhancements. High variability decreases the predictability in our ability to maintain our critical systems. This is fundamentally different from the development stage where high variability is expected and predictability is less of a concern since risk from bugs or lacking features is low.

* Corresponding author.
E-mail address: dport@hawaii.edu (D. Port).

[1] The timeframe is dictated more by the mission constraints than desires from operators of the system.

https://doi.org/10.1016/j.jss.2024.112030
Received 31 January 2023; Received in revised form 23 February 2024; Accepted 17 March 2024
Available online 26 March 2024

In this article, we report the results of an empirical investigation on the use of DevOps practices to manage productivity and quality variability in the maintenance of our flagship system MONTE (Evans et al., al.,2018).

1.1. Objectives of the investigation

Our software systems are characterized by many options and operational components to meet a diverse set of user needs, and we simultaneously operate and maintain multiple versions, all of which can lead to an increased risk from variability. DevOps practices such as continuous integration, continuous delivery, and infrastructure as code claim to mitigate this risk by automating and controlling many of the processes that cause variability (Grande et al., al.,2024).

In our transition of MONTE from development to operations, we introduced six policies into our release process to instantiate key DevOps practices specifically targeted to control variability in quality and productivity. However, the effectiveness of these practices in managing variability in complex software systems is a topic of ongoing research and debate.

The introduction of our DevOps practices is viewed as a deviation from NASA and JPL software development standards. As a CMMI Level 3 (https://cmmiinstitute.com/) organization, to institutionalize these deviations as a process improvement, we must collect and analyze data on our processes and use this data to improve performance and justify process changes. MONTE was not specifically selected as a case to be studied and we are not reporting on an experiment. Rather, it is part of our CMMI Level continual process improvement effort. Both CMMI Level 3 and DevOps share common principles such as process improvement, data-driven decision making, risk management, continuous improvement, customer satisfaction, and operational efficiency. We expect Applying these principles in a DevOps context will help achieve some of our CMMI Level 3 objectives to enhance process maturity, optimize software delivery, improve collaboration, and ensure higher quality.

Data was collected from MONTE development before and after implementing DevOps practices. We analyzed the data to assess the impact of DevOps practices on variability management, as well as compliance and performance of the practices. We find that, for MONTE, the DevOps practices we have adopted are quite effective in managing variability in quality and productivity and valuable approach for achieving CMMI Level 3 objectives.

This investigation was conducted to justify institutionalizing the DevOps practices we introduced in the transition of MONTE from development to operations in late 2009. The policies that led to our DevOps practices were based on the early concepts of DevOps and as such do not encompass the scope of DevOps as it is currently viewed. Therefore, this article should not be taken as reporting a case-study or experiment with DevOps. Our findings may not generalize to other contexts or implementations of DevOps practices. However, by placing our practices in a contemporary DevOps context, our findings provide valuable insights into the benefits and challenges of using DevOps to manage variability in critical software systems and will be of interest to researchers, practitioners, and decision-makers in the software development industry who are looking for concrete examples of the use of DevOps to manage quality and productivity variability in the maintenance of critical software systems.

1.2. Related works

DevOps, a methodology combining development and operations teams, has gained significant attention in recent years. In the realm of DevOps adoption and implementation, numerous studies have been conducted to explore its benefits, challenges, and best practices. By examining a diverse range of studies, we obtained valuable insights into the benefits, challenges, and critical factors we can expect in the adoption of our DevOps practices we are investigating.

Firstly, Challenges in DevOps adoption and their potential solutions have been extensively discussed. Jayakody and Wijayanayake (2021) identified critical DevOps adoption challenges in software development firms related to changing organizational culture, finding experienced professionals, and lack of management support. The study highlighted that improving communication, procedures, and collaboration tools between development and operation teams was the most common strategy to overcome these challenges. Additionally, Anandya et al. (2021) identified key challenges hindering DevOps adoption, emphasizing the need to address cultural, capability, and technological criteria for successful implementation, including continuous delivery, deployment practices, and test automation.

Khan et al. (2022) also found ten critical challenges hindering DevOps culture adoption, proposing a DevOps Culture Challenges Model (DC2M) to improve team collaboration and understanding. Mamdouh et al. (2019) also found low DevOps awareness and limited implementation in the Saudi IT sector, recommending Continuous Delivery, Continuous Integration, improved communication, and high-quality product delivery as essential practices for enhancing skills and software delivery efficiency. In structured software-intensive organizations, Maroukia & R. Gulliver (2020) emphasized the importance of ITIL-based practices and fostering a collaborative culture with strong leadership in their exploration of DevOps adoption.

Moreover, Gwangwadza & Hanslo (2022) identified key success factors for a DevOps environment, including communication, sharing, organizational culture, automation, and continuous practices (CI/CD, monitoring, and experimentation). In discussing CD adoption challenges in DevOps, Afzal et al. (2023) stressed the significance of advanced tools, cultural change, and hiring experts as solutions. In addition, Sravan et al. (2023) found challenges in adopting DevOps culture, suggesting solutions such as CI/CD, infrastructure as code, and DevOps-as-a-Service to address issues like lack of understanding, cultural resistance, technical issues, and measurement/visibility.

Azad and Hyrynsalmi (2022) also researched DevOps implementation challenges, addressing team coordination, risky development, expertise levels, and integration issues for improved collaboration between development and operations teams. In addition, Tenzin et al. (2022) studied DevOps in software companies, showing its impact on continuous delivery, automated builds, and collaboration, while addressing challenges of longer software delivery time with conventional methods.

These papers contribute to the understanding of potential barriers and provide insights on strategies to overcome them. The challenges shed light on the various obstacles faced during the implementation and adoption of DevOps, such as cultural change, coordination issues, expertise levels, and integration challenges.

Numerous studies have examined the success factors and benefits of implementing DevOps practices in various contexts. For instance, Blüher et al. (2023) implemented DevOps practices in an industrial production environment, resulting in a significant 90% reduction in deployment time and improved software quality. Similarly, Wahaballa et al. (2015) introduced the Unified DevOps Model (UDOM), which emphasized collaboration, Continuous Testing, and Infrastructure Optimization to achieve faster, customer-focused releases. Addressing challenges specific to highly regulated environments (HREs), Morales et al. (2018) explored the implementation of DevOps and proposed a DevOps assessment method that involved stakeholder involvement, version control, and Infrastructure as Code (IaC) to enhance the Software Development Lifecycle in HREs. Ali et al. (2020) introduced a hybrid DevOps process with software reuse, reducing rework, boosting productivity, and enabling systematic component reuse for faster development and continuous delivery.

Cloud-based DevOps implementations have also demonstrated benefits. Khan et al. (2020) achieved faster software delivery, improved collaboration, and business benefits by implementing a DevOps pipeline on the cloud. They highlighted the significance of DevOps culture,

automated testing, and best practices for reliable software delivery and team collaboration. Moreover, Gokarna (2021) provides a comprehensive approach to implementing DevOps in the software development lifecycle, emphasizing cultural changes, essential practices, and leading tools for efficient application releases and improved service quality. Pedra et al. (2021) found that DevOps adoption, with key practices like Continuous Integration, Continuous Deployment, automation, learning, and knowledge propagation, led to significant improvements in speed, quality, reliability, collaboration, and innovation in Brazilian organizations. A collaborative culture supported by enablers such as automation, sharing, transparency, continuous measurement, quality assurance, and resilience was identified by Luz et al. (2019) as critical for successful DevOps adoption. Lazuardi et al. (2021) conducted a review of DevOps implementations and found common benefits in software deployment, team productivity, and software quality improvements, with continuous integration, continuous delivery, and automation as frequently adopted practices.

Faustino et al. (2022), in a systematic literature review on DevOps benefits, identified improved collaboration, faster software delivery, increased code quality, reduced manual work, enhanced customer satisfaction, and alignment with core DevOps principles through automation and cultural changes. Marques & Correia (2023) introduced four DevOps patterns (Version Control, CI, Deployment Automation, Monitoring) for Continuous Improvement, providing practitioners with valuable metrics.

Senapathi et al. (2018) found that implementing DevOps led to increased deployment frequency, improved communication, and highlighted the importance of engineering capabilities and technological enablers. João et al. (2023), in a literature review, identified automated deployment as the most useful DevOps practice for ITSM assistance, followed by incident and problem management.

Combining DevOps with Kubernetes was found to enable automation, collaboration, and continuous improvement, streamlining software delivery for faster and more efficient releases with zero downtime, as discovered by Hira et al. (2023). Pardo et al. (2023) provide structured approaches to enhance the implementation and evaluation of DevOps practices in software development, addressing confusion and improving productivity and quality. Amaro et al. (2023) conducted a Multivocal Literature Review, identifying crucial DevOps Practices and Capabilities that enable organizations to achieve faster, reliable software delivery, enhanced collaboration, and continuous improvement, leading to increased customer satisfaction and competitiveness.

These papers in the benefits category highlight the positive outcomes of adopting DevOps practices, including improved collaboration, faster software delivery, increased code quality, reduced manual work, enhanced customer satisfaction, and alignment with core DevOps principles. They emphasize the advantages organizations can gain by implementing DevOps and provide insights into various strategies, approaches, and enabling technologies.

Furthermore, several studies have examined the impact of DevOps practices and their adoption on software delivery, collaboration, and quality improvement. These studies collectively provide valuable insights into the positive effects of DevOps on software development. These papers provide guidance and recommendations on best practices to achieve the desired outcomes in DevOps adoption.

In a comprehensive systematic literature review (SLR) by Zulkarnain et al. (2022), core capabilities such as automated deployment, continuous integration, microservices, continuous delivery, monitoring automation, and test automation were identified. These findings offer valuable guidance for organizations considering DevOps adoption. In line with streamlined software development, Srivastav et al. (2023) emphasize the importance of automation and cloud security while highlighting the benefits of continuous integration and delivery, resource savings, real-time activity tracking, and improved software reliability. To facilitate successful DevOps implementation in small organizations, Muño & Rodríguez (2021) developed a structured guide and case study, focusing on cultural change, collaboration, and automation as key factors. In the context of requirements engineering in DevOps, Hernández et al. (2023) shed light on areas for improvement, such as reuse, communication, traceability, non-functional requirements, and tool integration. These aspects aim to enhance software development speed, efficiency, quality, and risk mitigation. Highlighting success factors, Gwangwadza (2022) conducted a systematic literature review, emphasizing collaboration, communication, training, automation, and continuous practices as drivers of efficiency and reliability in software organizations. Lwakatare et al. (2016) also explored DevOps and emphasized the significance of practices such as continuous integration (CI), continuous delivery (CD), continuous testing (CT), Infrastructure as Code (IaC), and fostering a DevOps culture for achieving DevOps goals. In a literature review on DevOps implementation, Krey et al. (2022) underscored the importance of culture, automation, measurement, and sharing (CAMM) while addressing challenges faced by small and medium-sized enterprises (SMEs). They advocate for a cultural shift and supportive management for successful adoption. Similarly, Rütz (2019) delved into DevOps practices that promote collaboration, automation, measurement, and knowledge sharing. These practices were found to increase productivity and customer satisfaction while addressing challenges in large enterprise implementation. Akbar et al. (2022) developed a decision-making framework for successful DevOps implementation, prioritizing critical practices under the CAMS model (culture, automation, measurement, and sharing) and emphasizing the importance of culture, automation, measurement, and sharing. Building on empirical research, Zarour et al. (2020) utilized the Bucena DevOps maturity model to assess DevOps adoption. They found overall progress but recommended improvements in culture, process, technology, and measurement to enhance the benefits and impact of DevOps adoption. Arvanitou et al. (2022) conducted a tertiary study on DevOps, highlighting the importance of a unified vocabulary and emphasizing essential continuous practices for successful implementation.

Lastly, Trigo et al. (2022) examined DevOps adoption in a telecommunications company, observing that key practices such as continuous delivery, continuous integration, planning, customer feedback, monitoring, collaborative development, and automated testing led to improved software quality, faster delivery, and cost reduction.

In summary, these studies collectively demonstrate the positive impact of DevOps practices on software development. The findings highlight the significance of continuous deployment, continuous integration, automation, collaboration, and cultural change. By considering the insights provided and investigating the effectiveness and compliance with DevOps policies and practices, organizations can enhance productivity, manage quality variability, and drive continuous improvement in their software development processes.

2. DEVOPS practices and managing maintenance variability

When our systems are in operation, there are two maintenance demands from our users: fixing reported bugs and enhancing system capability. This is in addition to the continual development and update of required features and identification and repair of bugs found by our developers independent of users. While we cannot predict exactly what maintenance demands our systems will need, we understand well the demand for this maintenance and our limited ability to address such demand. We must carefully manage variability in both the *quality and productivity* of our maintenance effort, or risk contributing to missions not achieving their primary goals or failure due to unreliable operation or missing a needed feature.

Our primary challenge is managing the quality-versus-productivity tradeoff our developers make. It has been widely reported that high productivity, if not managed well, may decrease quality (Boehm et al., 1978; Humphrey, 1995; Jones,1991) We have found that prior to implementing DevOps practices, that our developers will sacrifice

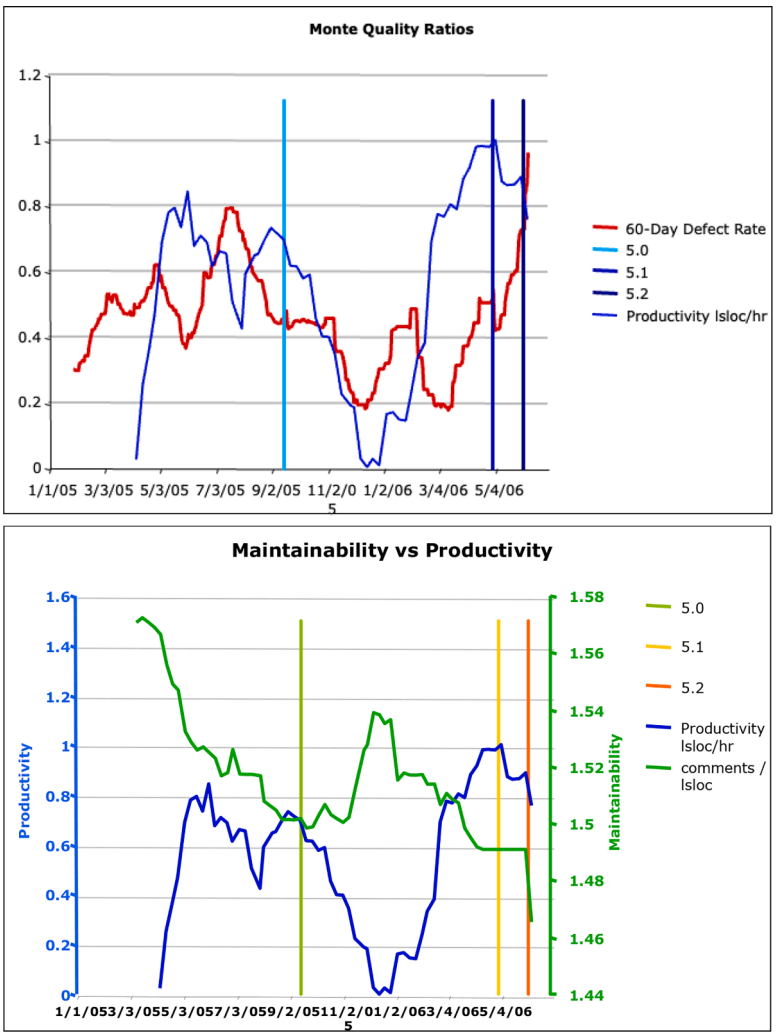


Fig. 1. MONTE productivity and quality, productivity and maintainability.

documentation and testing to meet release date goals and a subsequent increase in reported bugs. We have also found the converse, where our developers have focused on testing (i.e., developing test suites and tools) and extensive documentation to ensure reliability, but slipped a release date. That is, we have observed that a focus on quality decreases productivity.

More generally, we view this inverse quality-productivity tradeoff as an instance of the classic resources-scope-quality-time "choose any two" tradeoff phenomenon. We have clearly observed this phenomenon in our development process. Fig. 1 shows from 1/1/05 to 5/4/06 our productivity and bug rate (see (Jones,1991)), and productivity and comment density over 3 releases of MONTE versions 5.0, 5.1, and 5.2. To get an approximate “instantaneous” discovery rate for bugs and the rate the developer team produces code, we used the slope of the line fitted to a 60-day sliding window of reported bugs per day and Lsloc per day. The bug rate is an indicator of quality factors in the operation of the system and comment density is an indicator of quality factors in the revision of the system (maintainability) (FentonandPfleeger,1997). Together these two measures can provide insight for the quality of our release process. In particular, we can observe if developers are making quality – productivity tradeoffs that increase the variability in quality and productivity. Fig. 1 depicts an example observation of quality and productivity trade-off that commonly occurred during development. The time range was somewhat arbitrarily chosen based only on being well within the development period (pre-operations) and having at least a few releases to help explain periods of high productivity near a release that subsequently led to an increased defect rate. Consequently, the variability in quality is not controlled and the release is unreliable. This is observed in Fig. 1 as a lagged positive cross-correlation before the 5.0 release (the first productivity peak is followed by defect rate peak) and the productivity and bug rate run-up before release 5.1. This indicates that higher productivity, after a period of time, leads to higher bug rates. Productivity generally increases in a period leading up to a release, as developers are pressured to implement required features, and there is a reduced focus on quality. Conversely, in the middle of the release before 5.1 we observe lower productivity leads to lower bug rates.

The lag time appears inconstant due to the way productivity and bug rate are measured. The quality measure in Fig. 1 is the rate at which bugs are reported over a 60-day window. This is meant to get an “instantaneous” discovery rate for defects which we approximate by fitting a line to a 60-day sliding window of defects reported. We cannot know exactly the number of defects introduced in a given time period as defects may go undetected. However, the defect discovery rate is a useful proxy for the quality of the development (i.e., rate that new defects are introduced) in a given time period. Productivity is measured as the size of the deliverable code divided by the effort required to produce it in a given time period (in this case 60-days). Size can be quantified in several ways. We use logical source lines of code (Lsloc), but we just as easily could use the number of enhancements. Since we are only concerned with trends and associations and not particular magnitudes, Lsloc is a sufficient representative of size for this. Code that tests the deliverable does not count in productivity (but it does count in quality and reliability).

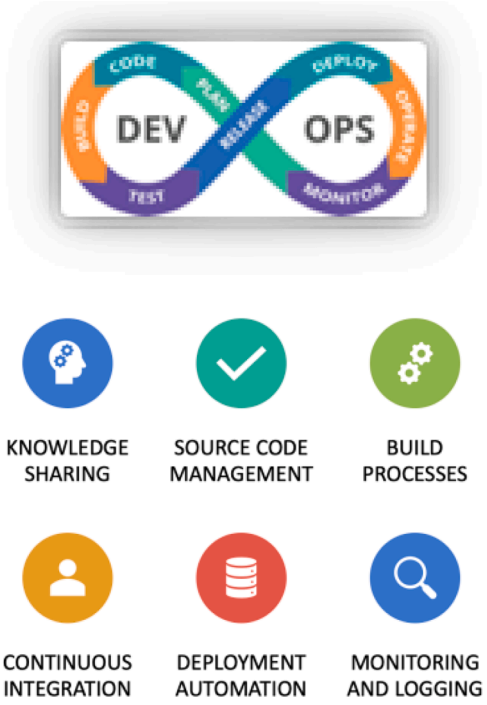


Fig. 2. Overview of DevOps process and practices.

In the bottom chart in Fig. 1 we observe quality and productivity trade-off before release 5.0 and 5.1 when developers sacrifice documentation (commenting code) for faster development at the expense of maintainability. We interpret it this way because in a related investigation of MONTE, we found higher comment density is associated with lower and more predictable bug repair effort [ref to ICSE-SEIP paper]. (Port et al., al.,2023). During the development phase such trade-offs may very well be an acceptable management strategy. For example, we may release a known buggy version for an integration test that must take place for another system at a given time. Or we may hold up a planned release to fix a non-critical bug in a feature required for that release. As the system is not in operation, the consequences for slipping a release may be negligible.

During operation of a critical system, we rarely have the luxury of making these kinds of quality-productivity trade-offs. Often, critical maintenance must be made within a given period. A spacecraft in route to a planet follows the laws of physics, not the schedule of engineers. The software used to track and control the spacecraft's trajectory must function reliably. Moreover, one of the few tools available to flight engineers in response to spacecraft hardware anomalies is the redesign of the mission trajectory. Such redesign may require upgrading mission design and navigation software. Not only must this avoid introducing new bugs, but we must also maintain functionality of existing capabilities.

So, how do we manage both high productivity and high quality? Indeed, as often espoused in the Agile community, productivity and quality are not inescapably inversely entangled (Jeffries,2009). In the book *Quality Is Free*, Philip B. Crosby (Crosby,1980) suggests that upfront investment in quality, or "build it right from the start," can pay for itself subsequently largely through reduced bug repair by avoiding bugs from the outset, and reduced enhancement effort through better documentation.

For maintenance effort, we cannot invest "upfront" since the system is already built and in use. We already invested in quality upfront. Also, the system is a moving target, not completely under our control, and we cannot avoid the time-sensitive issues that inevitably will arise. While we cannot perform upfront quality improvement, we can invest in an ongoing quality management maintenance process by instituting "mature" (in the CMMI sense) practices.

Inspired by the rise of Agile development and the burgeoning DevOps movement (KnowledgeHut,2022), since 2009 we have employed DevOps inspired policies and practices chiefly to manage the quality of our releases while maintaining high responsiveness to repairs and enhancements. The general DevOps process and practices are illustrated in Fig. 2 (as adapted from (Octopus Deploy,2022)). The MDN specific DevOps practices and process within the general DevOps framework is presented in Appendix A and elaborated in subsequent sections. The DevOps policies and practices we use are not currently part of NASA's coding or software development process guidelines.

The metrics and measures we use to monitor the performance of the DevOps practices are not typical KPI's or NASA-specific KPIs. Yet after nearly a decade of experience with the DevOps policy and practices, we have high confidence that they are effective and appropriate. Despite this, we regularly get push-back in compliance and questions as to the appropriateness and actual efficacy of the DevOps policies. Surprisingly, there is doubt, even from us, as to the extent our success in managing maintenance is due to DevOps or even whether DevOps is being practiced as we originally envisioned it.

To address this, we have undertaken this investigation to objectively evaluate DevOps policies as they are currently implemented. The main questions we wish to address are:

Q1. How do the DevOps policies affect productivity and quality?
Q2. How can we validate DevOps policy and process effectiveness?
Q3. How can we monitor compliance to DevOps policy and performance of DevOps process?
Q4. What latitude do we have in making adjustments? Can we improve the process?

This work presents our initial results from investigating these questions for a critical system developed and maintained by MDN called *MONTE* (**M**ission Analysis, **O**perations, and **N**avigation **T**oolkit **E**nvironment) (Evans et al., 2018). Since 2001 we have continuously collected, monitored, and analyzed quality and process data for MONTE. Using this data, we present the results of an empirical investigation into our DevOps policies and practices implemented to manage the productivity and quality for maintenance in the operations of MONTE. We present evidence of desired effects from these policies and that the practice is being performed in compliance to policy. To gain insight in the effect of the policies, we compare development and maintenance quality and productivity characteristics. Ultimately, we conclude that the DevOps practices initiated in the transition from development to operations are being performed as intended and continue to be effective in managing the quality and productivity in the ongoing maintenance of MONTE.

3. MONTE: from development to operations

DevOps practices were specified and introduced with the first official operational release of MONTE on Nov 17, 2009. While MONTE was already a mature system in operation prior to this official release, the risk profile substantially changed, and our development process had to be less variable and more predicable to address the increased demand for time sensitive bug repair and system enhancements. To explain how this came about, prior to MONTE's official operational release, the JPL MDN Group has two critical systems in operation – A legacy navigation system and its replacement, MONTE. The legacy system is composed of over one million logical lines of heritage FORTRAN code. The initial development of this system began in the late 1960s and continued until approximately 2006. During the period from 1999 until the present, an effort was undertaken to replace the legacy navigation system with a new system, MONTE, to be developed in modern languages using modern software development methodologies. As a result, in early 2006, the legacy system became a static (final release) system. Bugs in the

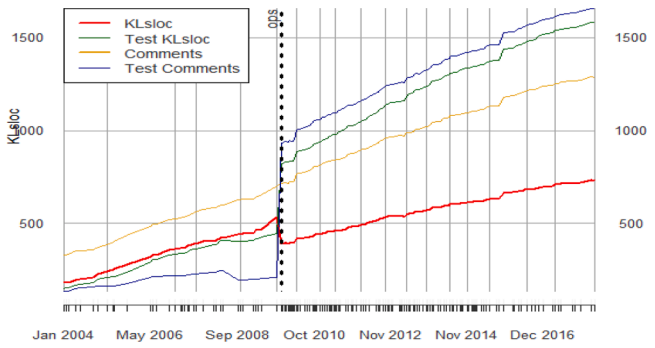


Fig. 3. MONTE size per release.

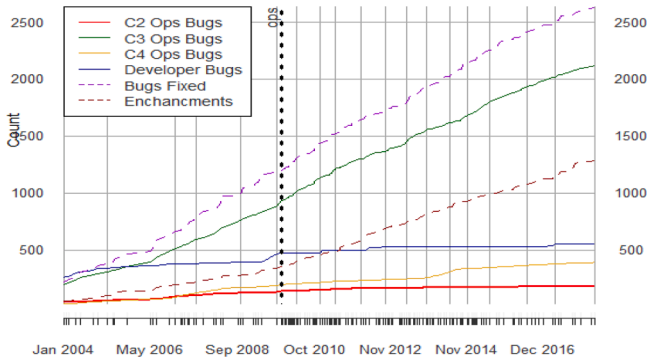


Fig. 4. MONTE cumulative bugs reported by type, bugs fixed, enhancements implemented per release.

system have been repaired, but aside from this no new code has been introduced into the system. In addition, the software has been in regular daily use by a nearly constant population of users during this time.

The adoption of MONTE by flight projects was a slow process and required continuous management pressure. Flight projects are reluctant to accept new software. They readily accept bug fixes and small additions, but entirely new systems are a major challenge. During the period in question, the navigation staff was very stable and nearly all were employed full time on flight projects: Cassini, Dawn, Deep Impact, Genesis, Hayabusa, MER, MRO, Odyssey, Spitzer, Stardust. The user base for MONTE was initially a different population of users. These were the early adopters and for the most part the new people who "grew up" with MONTE and never learned the legacy tools. The first mission to operate using MONTE was Phoenix which successfully landed near the Martian north pole in May of 2008. Following this successful first mission use, all flight projects gradually adopted the new system. MONTE has now successfully replaced the legacy product on all flight projects with the final transition away the legacy system with the Cassini mission in 2012.

Software maintenance is the ongoing activity performed on a system post-delivery ostensibly to keep a system in operation and of continued value to the users. It differs from pre-delivery development in several ways in so far as how bugs are discovered and removed and how requests for capability enhancements, updates and upgrades are handled. The exact maintenance needs are uncertain. However, what is certain is that maintenance issues will arise. As the system is used, the software will fail, new or revised capabilities will be demanded, and environments will change and become incompatible with the system. This latter issue is notably underappreciated as a fundamental maintenance management task. Even in the absence of bugs and enhancement demands, critical maintenance issues are *inevitable*.

We collect a great deal of empirical maintenance data and have found we can predict exceptionally well the demand for maintenance. For example, maintenance effort is needed when users report bugs from operating the system. Even though we cannot know exactly what these bugs will be, or when they will be reported, Figs. 3 and 4 illustrate that the evolution in size (code, test code, and documentation), the number of bug reports per release, the number of bugs fixed and enhancements implemented are quite predictable. The tick marks on the x-axis indicate release dates while the vertical grey grid lines are markers for a sample of release dates for reference. The dashed vertical line labeled "ops" indicates the release where MONTE officially transitioned from development to maintenance and the institution of DevOps practices.

From Figs. 3 and 4 we observe some notable differences between development and maintenance. In Fig. 3 we see the base code size (Ksloc) increases at a constant rate with the exception of a brief drop in size in the initial release to operations. We also note a very large increase in the size of test code and test comments in the initial release to operations. This is due to the test coverage and test documentation policies instituted at the start on maintenance. From Fig. 4 we note that the rate of enhancement requests increased in maintenance. This is expected as the primary system requirements had been implemented and the focus shifts to capabilities needed new users of the system. More notable is that while new development continued, the rate major operational bugs (C2 Ops Bugs) were reported essentially went to zero. While these are interesting observations, our concern is if we can attribute these differences (and more) in quality and productivity to the introduction of DevOps practices.

As was discussed earlier, the development of MONTE to replace the legacy system used for mission navigation began in 1999. MONTE has had frequent releases since it began so that developers could ascertain usability and gather feedback from users on needed features before its official release to operations and retirement of the legacy system. The system was first placed into a mission environment in May of 2006 after which the frequency of releases increased (note the higher density of release event lines). By this date, a stable base of users had been established. Moreover, since the software was used in a mission environment; its use was similar to the usage of the legacy navigation system. At the same time, new features continued to be added to the system to support the wide range of needs for the various trajectory design and navigation demands of NASA's deep space missions. The notable jump in size in Nov 2009 at the transition to maintenance. This was due to the final development push to satisfy remaining contractual requirements before official release of the system for operation (e.g. backward compatibility with the legacy system).

Prior to the release to operations, there was a deliberate trade of productivity for quality at this time. The goal was to complete all the development, drive the bug density to zero, and improve the quality of the testing and documentation to enable us to effectively focus on the subsequent issues that will arise as more users adopt MONTE and we no longer have the option of putting off a fix or enhancement requests. That is, we took our time to invest "upfront" in quality before releasing the system for operations. After this release, our DevOps practices aim to provide more frequent releases to respond more rapidly to users' needs which is evident from the notably higher density of release event lines after transition to operation (the ops event line).

Since May of 2006, there have been over 70 software releases of the software (on average a new release about every 4 weeks) and the system has experienced uniform usage by the user community. Hence, the development and usage of the system since 2006 is approximately a continuous release system.

We cannot know exactly when bugs will be discovered and how long it will take to fix them. To manage this uncertainty, we track the variability in bug discovery time (from both user and developer reporting) and bug repair effort. These follow a stable predictable pattern that is represented well with a Weibull distribution indicated in Fig. 5.

In the maintenance phase, we can plan and manage requests for capability enhancements as these go through a staged vetting and approval process. Here too, for management purposes, we track the

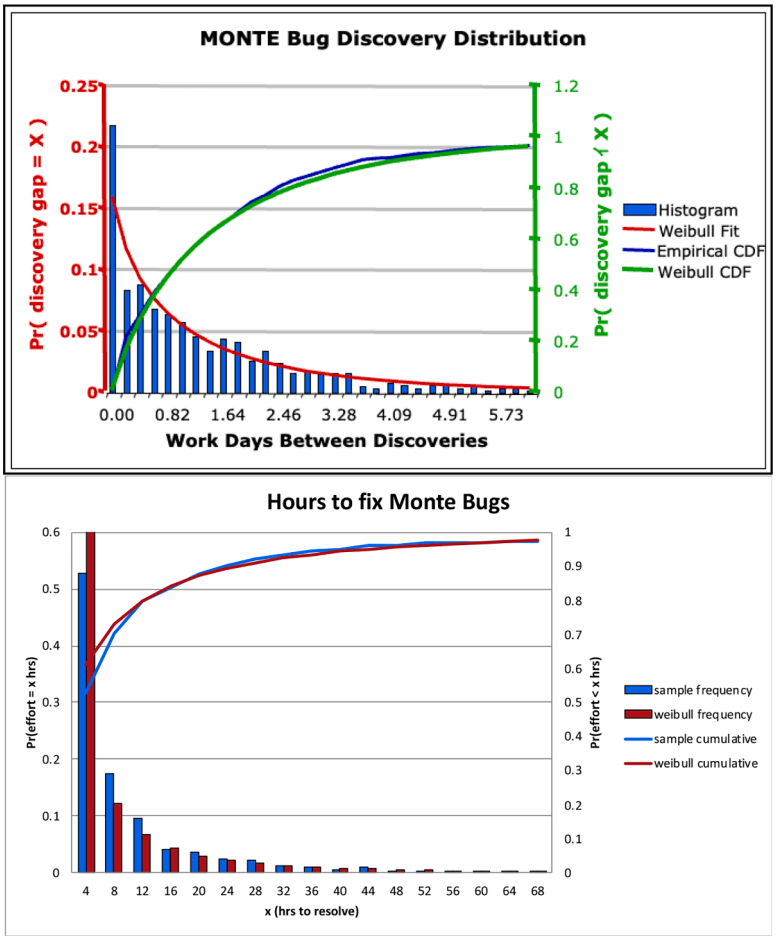


Fig. 5. MONTE bug discovery time and repair effort.

variability in how long enhancement requests go unimplemented and variability in the effort to implement them (Fig. 6).

Due to the time criticality of responding to bugs and implementing enhancements in operations, we must be able to reliably predict bug discovery, bug repair time, enhancement requests, and enhancement implementation effort. The distributions discussed here are the primary tool we use to for these predictions. However, the reliability of these predictions depends greatly on managing the variability in productivity which is our primary motivation for utilizing DevOps.

4. Data and methodology

Our maintenance process relies on accurate and reliable data. For integrity and practicality, we use a variety of automated and semi-automated data collection tools. Beginning in 2002, the software group has used the open-source bug tracking tool *bugzilla* to capture all anomalies in the operation of navigation software. One of the important features of *bugzilla* is that all bug reports as well as updates easily traceable to previously reported bugs (even from bug from earlier releases not resolved) to avoid multiple reporting of the same bug. Moreover, all users of the software (i.e., non-developers) are trained in the use of *bugzilla* and use it to report any suspected problems in the software. This provides us with reliable and consistent bug discovery data. All code, including tests, are managed in a shared repository (e.g., *perforce*) enabling us to gather data on code changes and where they occur. At each release we collect a variety of size measures and use the SLIC code count tool to obtain logical source lines (Lsloc) and documentation (source lines of comments). In addition, developers report their effort to resolve particular issues (both bug repair and enhancement requests).

4.1. Data collected

For the purposes of this investigation a data set was collected for each release of MONTE:

Version	Version label
Date	Release date
C2 Ops Bugs	Cumulative number of criticality 2 (major) bugs reported by users post release.
C3 Ops Bugs	Cumulative number of criticality 3 (average) bugs reported by users post release.
C4 Ops Bugs	Cumulative number of criticality 4 (minor) bugs reported by users post release.
Developer Bugs	Cumulative number of bugs reported by developers post release.
Lsloc	Logical source lines of code of the release reported by SLIC.
Comments	Lines of comment code of the release reported by SLIC.
Test Lsloc	Logical source lines of code in test suite of the release reported by SLIC.
Test Comments	Lines of comment code reported in test suite of the release by SLIC
Bugs repaired	Total bugs fixed for the release.
Repair effort	Total person-hours expended fixing bugs for the release.

All bug counts are the total bugs reported in bugzilla up to the release date. We do this as bugs introduced in a given release may not always be resolved in the next release. Also, a bug may have existed in a previous release but only discovered in the current release. For maintenance purposes we act only when a bug is reported, so for our process it matters little when a bug was introduced.

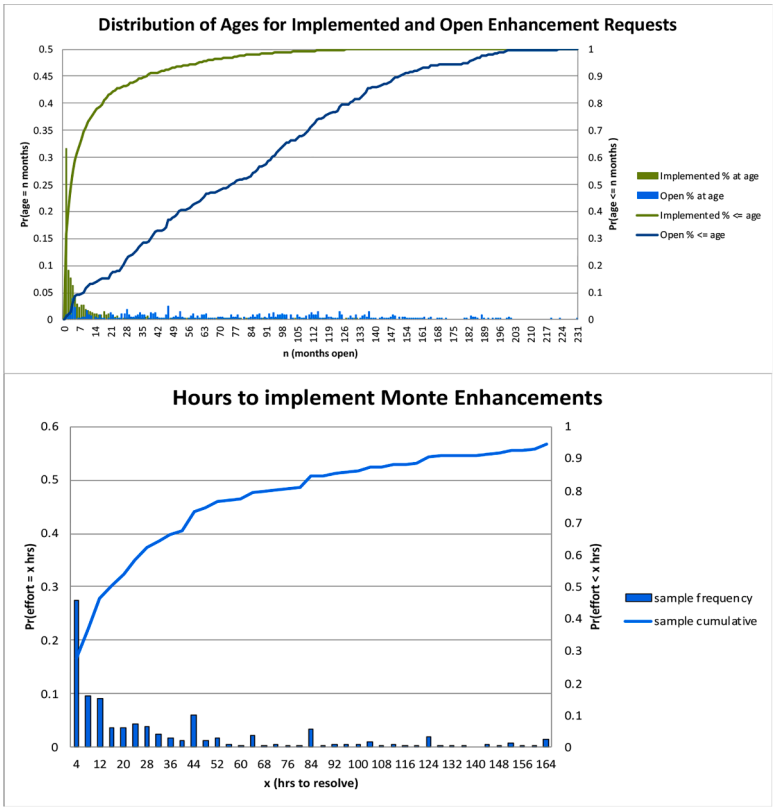


Fig. 6. MONTE bug enhancement requests and implementation repair effort.

Since the initial release of MONTE on Aug 19, 2001 there have been 173 releases up to Apr 27, 2018. Twenty of the releases are not on the primary development branch such as a revision of an earlier version (i.e. not the previous version) or a "programmatic release" done to satisfy some organizational requirement (such as a contractual milestone). These releases are not meaningful to our investigation of our DevOps practices and were removed from the data set as they are not representative of our typical release processes and were not subject to the six policies under investigation. In particular, these releases do not move forward in the primary development branch, and as such confound our productivity and quality measures calculations where the time window is measured as days between continual releases in the primary development branch. After removal of these unrepresentative releases, we have 36 releases from the development (or pre-DevOps) phase and 117 releases from maintenance.

We note that the data collected and the way it was collected has remained the same pre- and post-implementation of DevOps practices.

4.2. Data analysis and analysis methods

We are investigating our DevOps practices for the impact of its policies on managing quality and productivity in the maintenance of MONTE. This can be observed by comparing quality and productivity characteristics and associations difference *before* and *after* implementation of DevOps. We will analyze (1) variability of indicators of productivity and quality, (2) characteristic indicators of DevOps of implementation (e.g., frequent releases), (3) relationships between such characteristics and productivity and quality.

Maintenance is a continual release process i.e., a repeated sequence of activities over a block of time to deliver a new version of the system. For the items 1–3 above, we analyze the process data as an irregular time-series over release dates. Because the intervals between releases vary, we will have to take care to avoid statistics and analysis that assume uniform time intervals. This in general is not an issue as we are not interested in characteristics of individual measures over time. We primarily want to investigate trends (e.g., increasing, decreasing) over releases and correlation of measures over releases. Hence, it is the sequence of releases that is important, not the duration between them. Time-series relationships are often non-linear or lagged, so correlations are rarely constant over time. Inspired by signal processing analysis, we will often consider the cross-correlation function in order to discern relationships between time-series.

We will consider the following derived variables and measures:

Days – number of workdays between releases
Size – total of Lsloc and Test Lsloc
Size change – difference in Size between releases. An indicator of total amount of code added or removed.
Bugs – the difference between releases of the total of C2 Ops Bugs, C3 Ops Bugs, C4 Ops Bugs, Developer Bugs. This is an indicator of the number of new bugs introduced in a release. It is not exactly the number of bugs reported in a release. There may be unresolved bugs from the previous release that are not reported or counted again.
Reported bug density - reported operational and developer bugs per logical lines of code (Bugs/Lsloc). This is not the true bug density. However, we have found that it is substantively related to and is a useful indicator of reliability (Taber and Port,2014; Fenton and Pfleeger,1997).
Reported bug rate density – The rate that bugs are reported relative to the rate that Size changes. This is estimated as the ratio of the slopes from fitting lines to a time window of cumulative reported bugs and for a time window of cumulative reported bugs. While this is a non-standard quality metric, it is more relevant than bug density as an indicator of our quality management. Our maintenance objectives focus on our ability to respond to bugs reported between releases, rather than the overall reliability of the system. This metric allows us to predict the number of bugs expected in a given release and ensure we have adequate resources to repair them within a

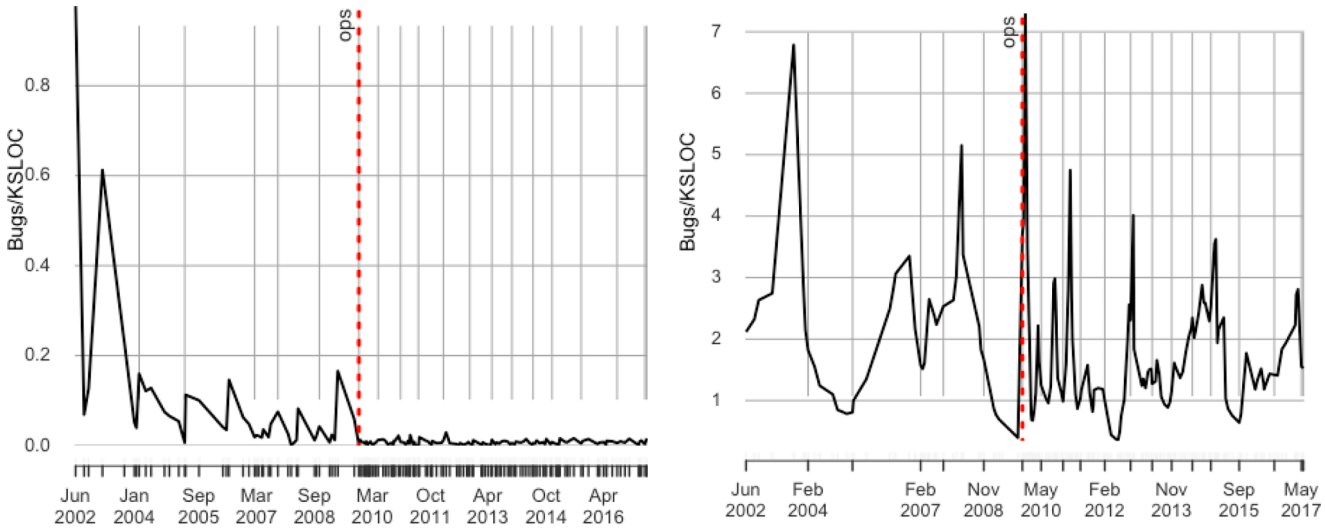


Fig. 7. **a**: MONTE reported bug density; **b**: MONTE bug rate.

constrained time period. It is not prone to the size issues that are problematic for bug density (FentonandPfleeger,1997). For this investigation, it is useful in identifying when developers are making productivity – quality tradeoffs.

Productivity – Number of enhancements implemented over the number of workdays to implement them. While productivity is better represented by tasks completed relative to effort expended, this crude measure is good enough to discern productivity trends and variability.

Comment density – comments per logical line of code computed as (Comments/Lsloc)

Test density – logical line of test code per logical line of code computed as (Test Lsloc / Lsloc)

Bug effort – person-hours expended per bug, approximately the mean time spent per bug computed as (Repair effort / Bugs repaired)

The data and measures presented above are practical to obtain and reasonably easy to analyze. We are aware that some are crude and perhaps atypical of measures used for DevOps process monitoring and evaluation (Trigo et al., 2022) Nonetheless, we find that they are satisfactory for our investigation of MONTE in addressing the questions (Q1-Q4) posed in Section 2.

5. MONTE DevOps practices

MDN adopted DevOps practices to align our release process with the distinctly different expectations for the maintenance of MONTE. Fig. 7 shows Reported bug density. The trend characteristically rapidly decays (Taber and Port, 2014) during development (before the "ops" event line) as the process drives the system towards an acceptable quality level for operations. When a bug density of less than 3 bugs/KSLOC in the Nov 17, 2009 release was achieved, the system was deemed reliable enough for the navigation users and officially released for operation. MONTE is now required for all new missions and users of the legacy system are encouraged to transition to it as soon as possible as it is no longer supported. This "ops" epoch designates the start of the maintenance phase for MONTE. While technically MONTE was already in operation during later stages of development, maintenance is distinctly different in terms of management where the users, rather than the sponsor (NASA), drive expectations and perhaps more importantly, funding.

At this point, MONTE shifted from navigation operations to providing new capabilities for mission design, development continues through enhancement requests from users. The mission design process is much more fluid than for navigation. For every mission in flight, a dozen or so are on the drawing board, most of which never make it past the design stage. These speculative missions often generate novel designs that go beyond MONTE's current capabilities. In addition to responding to bugs, development is driven through enhancement requests from users.

5.1. *Policies to implement DevOps practices*

Our DevOps practices were mainly developed in response to the needs of the trajectory design analysts. Trajectory design is the process of planning the path of a spacecraft from one point to another. It is a critical and complex part of spacecraft mission navigation and design, as it determines the spacecraft's speed, altitude, and orientation and whether the spacecraft can make it to its intended destination. Trajectory design is an area of active research with new design algorithms and requirements emerging much more quickly than the more mature navigation and maneuver design processes. However, we wanted to meet this more fluid demand without sacrificing the quality of the existing system. To achieve this, we automated and mandated many of the quality management and assurance activities that were not previously required in our release process. Six policies were established to implement the DevOps practices previously not required in the development phase:

P1. At least one release per month (provided there is user need)
P2. Nightly automated builds
P3. All functions must be documented. Documentation is automatically checked at build time. If not 100% function documentation the build fails.
P4. All functions must have unit tests that are automatically run at build time. Tests must cover at least 90% of the functional capability (100% is the goal but is often impractical due to the large number of functional variations).
P5. All usage examples of functions in the documentation must have corresponding tests for those examples.
P6. No "to be done" comments. Do or remove.

Appendix A has annotations depicting where the policies are implemented as DevOps practices within our maintenance process. All the policies are expected to be met before the beginning of a release cycle. To achieve this, as per DevOps practice, automated checking of these requirements is built into the nightly build process. If any of these requirements were not met, the nightly build is regarded as being broken. Since it is part of the daily workflow for a developer to fix any of

his/her submissions to configuration management that broke the nightly build, it is generally a small step to move the work on the development branch of the code base to the delivery branch and comply with the requirements for a release of the software. The expectation was that by automating the quality requirements as part of the nightly build we could quickly assemble a release for users. This expectation was indeed realized. However, as a side effect we have found that we have tended to improve overall quality of the releases as the quality is "baked" in from the beginning.

It's clear that P1 – P6 incur some additional effort overhead. This effort is "amortized" into small amounts over many frequent releases. The process overhead is expected, as per the quality is free paradigm, to be "paid back" in the near future through (1) fewer bugs reported in subsequent release, and (2) less effort fixing bugs and implementing enhancements down the line. We will investigate if this expectation is realized in our practice.

Successful implementation of the DevOps practices necessitated significant changes to our organizational culture. For example, in order to achieve "at least one per month release" (P1), we had to include users much earlier in the release process. Previously it was customary to include users only in the Software Acceptance Review. Now users are included much earlier in Test Readiness Reviews. Another cultural shift was in providing complete testing and documentation for each build with no TBD's that is automatically checked (P3, P4, P6). Prior to operations complete documentation and testing were not mandatory (and not automatically checked) and it was common for the developers to put this off and focus on implementing capabilities in order to make the delivery schedule. There was a firmly entrenched belief that capability was more important than quality which is counter to the expectation from the users for a reliable operation of their critical system. We know from experience with other systems that this belief carries over to maintenance. For such systems, the culture was that the reliability risk was secondary to the risk of not having a needed capability. This cannot be so for MONTE where the risk is very high for both reliability and capability and so there must be a cultural shift and the policies aim to institute this.

A vital component of this investigation is to verify compliance with the policies and determine if changes to the culture occurred and if these changes were beneficial. However, while these cultural shifts are important to DevOps generally, they are not the focus of our investigation unless we find a problem with the practices or policies that affect quality or productivity.

5.2. *Policies relation to contemporary DevOps practices*

When we switched from development to operations in late 2009, the DevOps concept was still in its early stages of development (circa 2008 by Patrick Debois). As a result, the policies we implemented do not fully reflect what is considered important to DevOps today. The purpose of the investigation is to justify organizationally institutionalizing the policies and contextualizing them within contemporary views of DevOps is helpful in this endeavor.

Since 2009 the DevOps concept has rapidly evolved and there are established principles and practices closely related to the six policies:

P1. At least one release per month (provided there is user need)

This policy is closely related to DevOps as it aligns with the principle of continuous delivery and deployment. By emphasizing the importance of frequent releases to meet user needs, organizations can gather feedback from users at a faster pace, enabling them to make rapid changes and improvements. A paper by Calvache et al. (2022) discuss a case study of a software development company that adopted the DevOps model and made the change to have regular software release was able to improve the quality of the software and the user experience. They found that implementing DevOps practices like Continuous Integration, Continuous Delivery, and Security Monitoring enhanced software quality, user experience, and provided benefits of clarity, flexibility, and stability to software development organizations. In a similar vein, Lwakatare et al. (2019) present a multiple-case study of five companies that successfully implemented DevOps in web application and service development, emphasizing the importance of cross-functional collaboration, automation, and toolchain support for achieving quick releases and minimum deployment errors. Supporting these findings, Mishra and Otaiwi (2020) provide evidence that DevOps can be an effective way to improve software quality by automating tasks, promoting collaboration, and delivering software changes frequently. However, it takes time, effort, and commitment to implement DevOps successfully. Furthermore, Grande et al. (2024) also further emphasize the critical role of communication and collaboration between development and operations teams in the context of DevOps, specifically highlighting the importance of CI/CD for enabling rapid and automated software delivery, improved quality, reduced risk, and shorter release cycles. Additionally, Gwangwadza and Hanslo (2023) also highlight the crucial role of a culture of continuous delivery and deployment in successful DevOps environments. They emphasize how frequent software releases enable prompt customer feedback, allow for timely adjustments, and minimize the risk of major bugs.

This iterative approach fosters a culture of continuous improvement, enhancing the quality of the software and optimizing the user experience. The policy's focus on a steady release cycle also promotes the DevOps principle of continuous delivery, advocating for regular and incremental software updates. Through this approach, teams can iterate quickly, incorporate user feedback into subsequent releases, and achieve a more efficient and responsive software delivery process. This policy is expected to help manage the variability in quality and productivity.

P2. Nightly automated builds

Automated builds, such as nightly builds, are a fundamental practice in DevOps. By automating the build process, teams can achieve consistent and reproducible builds, reducing human error and saving time. Nightly builds also promote early detection of integration issues and facilitate continuous integration, enabling teams to catch and resolve problems quickly. Yarlagadda (2019) explores how DevOps enhances software development quality through the CAMS framework, emphasizing automated builds, testing, and documentation checks, collaboration between teams, and the importance of continuous improvement, while recommending the adoption of additional DevOps practices and tools like behavior-driven development, test-driven development, Jenkins, JUnit, Selenium, and GIT to enhance automation and configuration management. In a similar vein, Narang & Mittal (2022) also compared traditional software development methodologies with DevOps using the Integrated Tool Chain (ITC) for automation, emphasizing DevOps as an emerging trend to improve efficiency, collaboration, security, and communication between development and operation teams. Building upon these insights, Blüher et al. (2023) implemented DevOps practices in an industrial production environment, resulting in a 90% reduction in deployment time and improved software quality through collaboration, automation, and efficient development and deployment strategies. Furthermore, Guerrero et al. (2020) study highlights the lack of consensus in defining DevOps, the need for standardized practices and guidelines, and the importance of automation in its adoption in software development organizations. Moreover, Mohammad's (2018) study highlights the crucial role of DevOps automation, supported by the CAMS model, in improving software quality through collaborative efforts between development and operations teams, ultimately leading to enhanced organizational performance and software development efficiency. Additionally, Offerman et al. (2022) highlighted the positive impact of DevOps practices, specifically automated builds and comprehensive test coverage, on software delivery performance and customer satisfaction.

This frees up developers to focus on more creative and strategic work, and it helps to ensure that the software is always in a releasable

state. This policy is expected to help manage the variability in productivity.

P3. All functions must be documented. Documentation is automatically checked at build time. If not 100% function documentation, the build fails

Documentation is a crucial aspect of DevOps as it promotes a culture of knowledge sharing, collaboration, and transparency. Requiring comprehensive documentation and enforcing its validation during the build process ensures that the documentation remains accurate and up to date. This policy aligns with the DevOps principle of "Infrastructure as Code" where documentation becomes an integral part of the software development process. By ensuring that all functions are documented, organizations can make it easier for developers to understand and maintain the code. Rafi et al. (2020) conducted case studies to validate the effectiveness of the RMDevOps model in assessing and improving best DevOps practices, including documentation as code, and identified challenges and recommendations for implementing documentation as code within software organizations. In a similar vein, Gwangwadza & Hanslo (2022) also identified key success factors for a DevOps environment, including documentation as Code, communication, sharing, organizational culture, automation, and continuous practices (CI/CD, monitoring, and experimentation). Moreover, Anandya et al. (2021) highlighted the challenges hindering DevOps adoption and emphasized the importance of addressing cultural, capability, and technological factors, including the implementation of documentation as code to improve clarity and accessibility in DevOps practices. Additionally, Díaz et al. (2018) presents an ongoing exploratory case study investigating DevOps adoption and practices, highlighting the importance collaboration between different teams, and addressing organizational culture challenges. This policy is expected to manage the variability quality of the software and enable developers to fix bugs and implement enhancements more efficiently.

P4. All functions must have tests that are automatically run at build time. Tests must cover at least 90% of the functional capability

Testing is a core tenet of DevOps. By mandating automated tests for all functions and aiming for high coverage, teams can ensure the reliability and quality of their software. Automated testing at build time helps catch potential issues early, facilitating continuous testing and enabling teams to deliver software with fewer defects and higher stability. In this aspect, Pando & Dávila (2022) conducted an SMS on DevOps automated tests, revealing growing industry interest, especially for web applications and SOA, with focus on unit and integration tests using Java and Jenkins, highlighting the need for automated testing skills training for small companies' global market competitiveness.

Moreover, Srivastav et al. (2023) discusses the implementation of DevOps practices, such as Git for version control, Jenkins for continuous integration and delivery, Chef for infrastructure automation, Puppet for infrastructure management, and Selenium for test automation and highlights their benefits and drawbacks in enhancing software automation, emphasizing the goal of improving software development and product quality through automation, monitoring, and collaboration. Furthermore, Hira et al. (2023) investigates that by integrating automated testing into the CI/CD pipeline, DevOps teams can ensure that software is thoroughly tested before deployment. Kubernetes can provide an environment to run automated tests, helping teams verify the functionality and performance of applications. Similarly, Faaiz et al. (2023) conducted a systematic literature review on DevOps challenges and mitigation strategies and found that the most common challenges are lack of understanding, cultural resistance, and technical challenges, which can be overcome by a number of mitigation strategies. They assessed automated testing as a mitigation strategy for the challenge of inadequate expertise in using automation tools. Additionally, Grande et al. (2024) discuss that incorporating automated testing, continuous testing, and feedback mechanisms, can enhance the quality of the software product. Moreover, Yarlagadda (2019) also mentioned the use of automated builds, automated testing, and automated documentation checks to automate processes and mitigate risks. Lastly, Petrović (2022) proposes an automated approach for code quality checks using a semantic-driven generator as part of their DevOps practices.

By ensuring that all functions have tests, developers can make it less likely that bugs will make it into operations. This can help to manage the variability in the quality of the software and the user experience.

P5. All usage examples of functions in the documentation must have corresponding tests for those examples

This policy emphasizes the importance of aligning documentation and tests, ensuring that usage examples are validated. It promotes a holistic approach where documentation not only provides instructions but also serves as a source of verified and executable examples. This can help to ensure that the documentation for users is accurate and up-to-date, and it can help to make it easier for developers to understand and maintain the code and promotes a culture of knowledge sharing, collaboration, and transparency. Petrović (2022) introduces a semantic-driven approach to automate code quality checks in the context of DevOps, specifically targeting Infrastructure as Code (IaC). The paper showcases its effectiveness through realistic case studies and provides an automated solution to enhance code quality within DevOps workflows. By employing the proposed semantic approach and automated code generation, documentation-driven testing becomes more efficient and accurate, leading to overall quality improvement in the DevOps environment. In their work, Gwangwadza and Hanslo (2022) identify continuous experimentation as a crucial success factor in DevOps, with documentation-driven testing playing a significant role in enhancing software quality and overall effectiveness. To address challenges hindering the adoption of DevOps culture, Khan et al. (2022) propose the DevOps Culture Challenges Model (DC2M). This model aims to improve team collaboration, understanding, and addresses issues such as the lack of collaboration, communication, and trust that can impact the implementation of documentation-driven testing and testing accuracy. Furthermore, Rafi et al. (2020) validate the RMDevOps model, which enhances DevOps practices, including documentation-driven testing, by addressing challenges such as the lack of suitable tools and processes, and providing recommendations to overcome them.

P6. No "to be done" comments. Do or remove

This policy reflects the DevOps principle of fostering a culture of accountability and action. By eliminating "to be done" comments, teams are encouraged to take ownership of their code and remove incomplete or unnecessary parts ensuring that tasks are either completed or removed from the backlog. This promotes a culture of continuous improvement and ensures that the software remains lean and efficient. This can help to manage the overall productivity of the team and the quality of the software. With regards to this, Rafi et al. (2020) validate the effectiveness of the RMDevOps model in three case studies, improving best DevOps practices, including continuous delivery, automated testing, and accountability within software organizations. In line with this, Azad & Hyrynsalmi (2023) emphasize the effectiveness of DevOps practices in improving productivity and quality in software maintenance, highlighting the importance of monitoring, continuous improvement, and addressing organizational challenges for successful implementation. Additionally, Basavegowda Ramu (2023) presents a comprehensive approach to optimizing DevOps pipelines by integrating performance testing, emphasizing the importance of ongoing validation, real-time monitoring, and iterative optimization. Furthermore, Grunewald et al. (2023) also propose the Hawk framework, which provides approaches and proof-of-concept implementations for ensuring transparency and accountability in cloud-native systems within the context of agile and DevOps practices, thus aiding data controllers in meeting regulatory obligations and aligning with current engineering practices.

Moreover, Noorani et al. (2022) used a SWOT-AHP approach to prioritize crucial factors for successful DevOps implementation in software development organizations, including leadership support, culture, processes, tools, and training, offering a readiness model based on the SWOT framework to enhance DevOps approaches.

We now refer to the six polices as "DevOps policies" as they align strongly with various DevOps principles such as continuous delivery, automation, infrastructure as code, testing, collaboration, and accountability. Although not entirely within contemporary views of DevOps, the six policies imply practices embraced by DevOps (which we now refer to as DevOps practices). For example, we do not use contemporary tools such as Jenkins and Docker. We have built our own tools to help automate the release process and are considering containerizing releases. Nonetheless, by having implemented DevOps policies and practices in the transition to maintenance, we expect to better manage quality, reliability, user experience, and overall productivity. This investigation aims to provide empirical evidence of this and that our practice complies and embraces (cultural shift) the six policies implemented in 2009.

However, it's important to note that we have only adopted a subset of DevOps and that it is not just about maintenance. Contextualizing the six practices within DevOps gives us a more general perspective enabling us to make use of knowledge and experiences outside the MDN group and is helpful in identifying adjustments needed and opportunities for improvement (and addressing Q4).

5.3. *Challenges to implementing DevOps practices*

There were some challenges in both implementing and managing the policies and practices, yet the developers adapted quickly and with surprising integrity. One reason for this is that much of the compliance with the new policies is automated (e.g., checking function documentation at build time). Adding non-automated compliance reporting to their tasks would be untenable, costly, and error prone. The developers could surely bypass or game this (for example by adding garbage documentation to the functions so it passes the documentation check). However, we found that they did not do this and took compliance with the policies seriously.

We could not simply ask our developers to comply with the policies and perform the practices. The culture had to be adjusted to have developers have more direct communication with users, sharing development information with them, collaborating on prioritizing bug repair and implementing enhancements, and sharing responsibility for quality. These culture changes are not easy to make as we have a fairly rigorous development process mandated by the sponsor (NASA) that does not include such extensive interaction with users (but they do not preclude such interaction). Fortunately, we have a culture with strong interest in institutionalizing process improvement. If we can make a case for improving our process, it will be institutionalized as required process. This investigation was undertaken in part to institutionalize the DevOps practices (in our context a delivery process improvement). In this context, the six polices are mandated and the job is not complete without complying to them. New staff are trained with these policies in place. Compliance is aided through the automated checking tools at check-in and build times. Thus, the DevOps practices are baked-in to the development process culture and new team members are brought up to speed on this process as a whole without specific training in our DevOps practices.

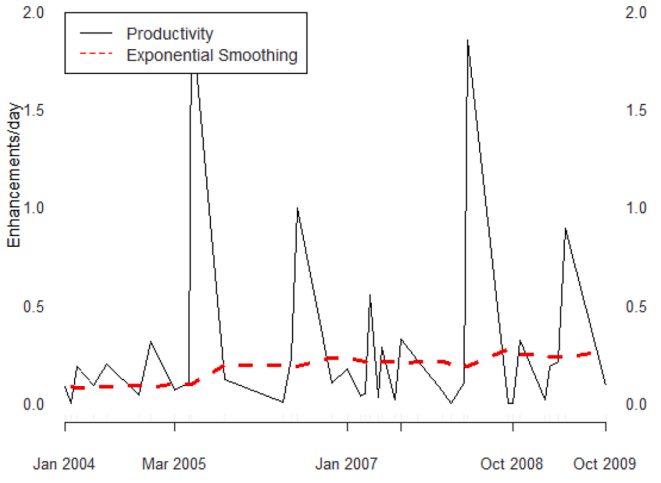


Fig. 8. Development productivity per release.

6. DevOps quality and productivity

We previously presented some conclusions about DevOps practices that begin to address Q1-Q4 discussed in the introduction (Section 3). To re-cap this:

- The higher density of event lines in maintenance compared to development in Fig. 3 indicate we have more frequent releases since instituting DevOps [compliance (Q3) to policy (P1)].
- The consistently low level and low variability of Reported bug density in maintenance from Fig. 7 indicates DevOps does not negatively impact quality [addresses (Q1)].
- The productivity in maintenance (Fig. 3) is generally higher and more stable than in development (Fig. 7) DevOps does not negatively affect productivity [addresses (Q1)].

The above results are comforting but not convincing. We are more interested in seeing if DevOps has a net positive effect rather than simply having no evidence of negative effects. We explore this now.

6.1. *DevOps practices improve and control quality*

Fig. 7 shows that since implementing DevOps (after the ops line), we have consistently maintained a low Reported bug density and lowered its variability (Fig. 7a). Given that there is significant ongoing new development implementing enhancements, the very low variability in Reported bug density is also notable. The precipitous increase before then large drop in bug density after the first release to operations is explained by the institution of 90% test coverage policy P4. This policy was put into place for this initial release resulting in considerably more developer bugs being reported prior to the release (note the increase in developer bugs before ops release in Fig. 4 and the enormous increase in bug rate after the release in Fig. 7b. The low bug density after the operational release is not an artifact of an increase in code size. In fact, the code base decreased significantly (see the drop in Lslock in Fig. 3 after the ops release) for the ops release as developers removed buggy non-critical features to meet the less then 0.03 Bugs/KSLOC ops release requirement.

It's vital that the bug density remain low with low variability to ensure the reliable operation of the system. Our ability to manage this relies on being able to predict the number of bugs and the effort needed to repair them. This predictability of these depends on the predictability of the bug rate. We need not only that its variability is low, but also that it's constant. In terms of time-series, we would like to see that the bug rate's statistical properties do not change over time i.e., that it is *stationary*.

Visually in Fig. 7b the bug rate during development it appears to trend down. This is a good sign for the process assuming it continues to decrease over time and is not a spurious fluctuation. However, for management concerns it is unwelcome as it makes predicting the bug rate unreliable. The Augmented Dickey-Fuller Test (ADF) (Said and Dickey,1984) is a hypothesis test that is used to determine whether a time series is stationary. The alternative hypothesis is the time-series is

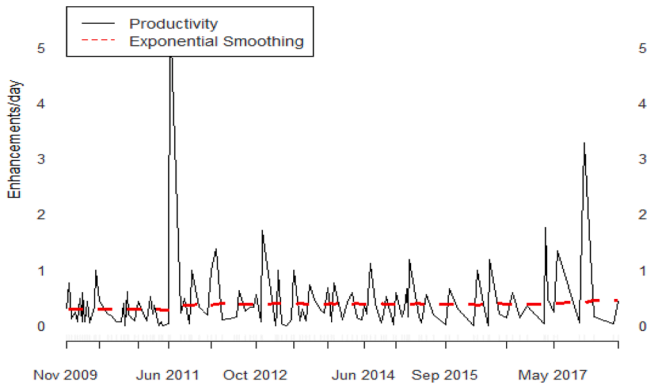


Fig. 9. Maintenance productivity per release.

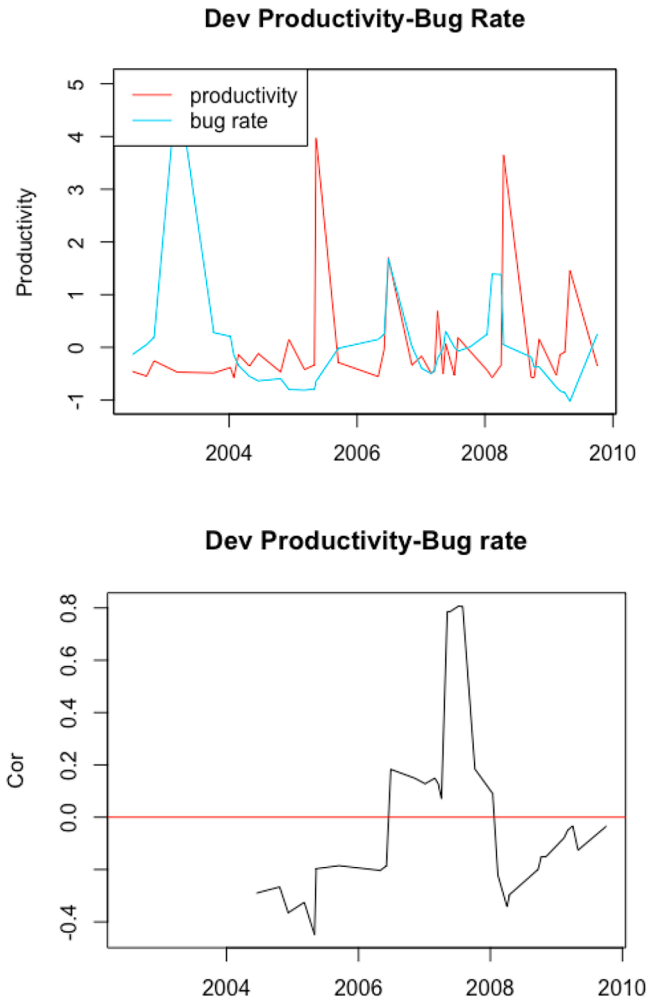


Fig. 10. Development productivity and bug rate cross-correlation.

stationary. In performing this test on the bug rate during development we observe a p-value over 0.4 indicating that we have insufficient evidence that the time-series is stationary confirming our suspicion that predicting the bug rate in development would be unreliable.

Looking at the bug rate after we introduced DevOps practices in Fig. 7b we see no notable trends or patterns in the time-series. The ADF test for this part of the time series produced a p-value of less than 0.01 confirming that the time-series is stationary and we can reliably estimate the bug rate.

We conclude from Fig. 7 that our DevOps practices reduced bug density and helps maintain it consistently at a low level.

6.2. *DevOps practices do not degrade productivity*

Our other concern is maintaining a consistent level of productivity. Figs. 8 and 9 show productivity indicators and trends in development and maintenance. The dashed line shows the trend derived from exponential smoothing (Holt, 1957) and is intended to aid in visualizing trends in noisy data. Exponential smoothing focuses on capturing short-term changes and minimizing the impact of random variations or outliers. This makes it beneficial in scenarios where the data has irregular fluctuations or is influenced by external factors that can introduce significant volatility as we observe in our productivity data. The upward trend in Fig. 8 was expected and the result of efforts to implement required features prior to release to operations. The ADF test produced a p-value of 0.12 indicating that the time series may be non-stationary and the observed is not explainable by chance variation. Therefore, we cannot claim that productivity is consistent in development. Fig. 9 shows the productivity in maintenance is notably less variable with no trend up or down and generally varies around 0.6 enhancements/day which is higher than in development which varies around 0.1 to 0.4 enhancements/day. The ADF test has a p-value less than 0.01 confirming that there is no trend in the productivity over time during maintenance. We conclude that DevOps reduced productivity variability and stabilized it to essentially constant over time and therefore more predictable and manageable.

6.3. *DevOps mitigates productivity and quality tradeoffs*

During development phase of MONTE, we observed productivity-quality tradeoffs (Fig. 1). The left side of Fig. 10 compares the productivity and bug rate for development releases on a normalized scale (z-scores of the values) to aid in comparison of the variabilities. There appears to be a pattern where spikes in productivity are followed by spikes in defect rate. This may indicate lagged positive correlation. That we do not see a large spike in productivity in the beginning but there is a large spike in bug rate is due to productivity accounting only for enhancements, not general development and the bug rate accounts for both developer and operational reported bugs. Releases during development were commissioned by missions wanting to use MONTE resulting in an increasing number of enhancement requests and hence spikes in productivity. The ongoing development produces bugs that are not accounted for in productivity complicates identifying the effect productivity has on bug rate. To examine the possible effect from simpler perspective, the right side of Fig. 10 also shows the running correlations (or moving window) over 10 releases between Productivity and Reported bug density in development.

Although the running correlations appear positively biased, it's difficult to assess if the correlation is significant or simply due to chance variation. To examine this more objectively, we consider the cross-correlation function (Shumway and Stoffer,2017) chart in Fig. 12. The cross-correlation chart is useful for identifying a number of possible associations between time-series such as lagged relationships, interactions, and non-constant correlation. In our case, we see in the development productivity – bug rate cross-correlation chart that at lag 10 the cross-correlation is past the upper critical value indicating a significant positive correlation. The critical values here indicate the 95% confidence interval for the sample correlation if the population correlation is zero. Therefore, if there is any sample correlation outside this interval, we can conclude with at least 95% confidence that the population correlation is non-zero. We conclude that in development, on average, an increase in productivity will be followed by an increase in bug rate. That is, during development, quality was often traded for productivity (often enough to be positively correlated on average). Since the bug rate is calculated over several releases after a given release, a positive correlation at lag 10 does not mean the bug rate increases 10

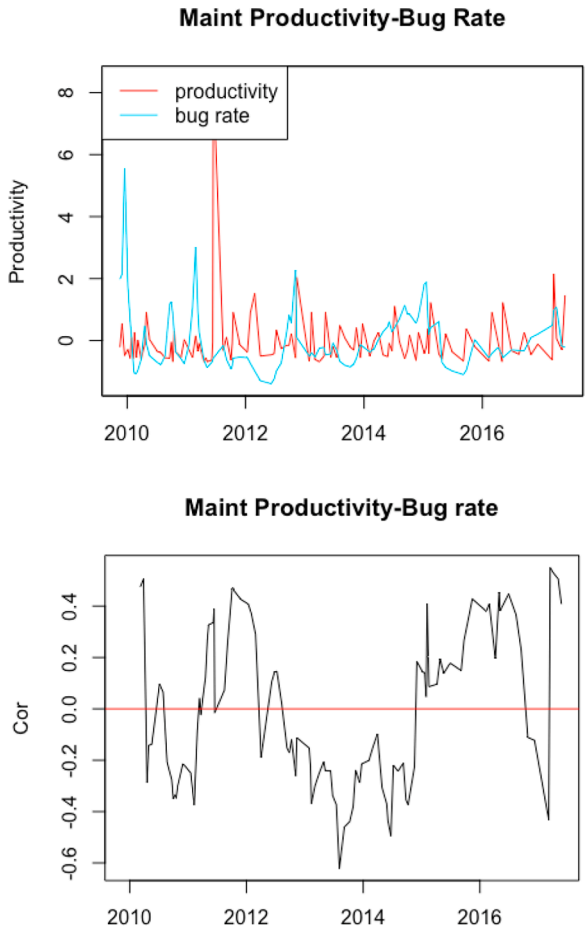


Fig. 11. Maintenance productivity and bug rate cross-correlation.

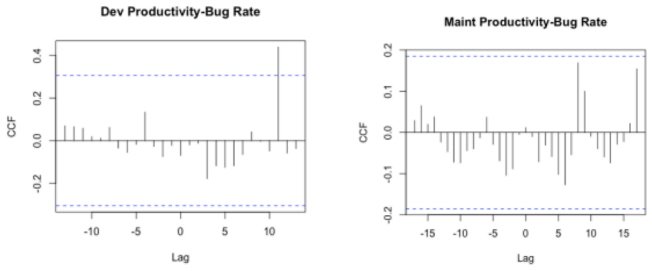


Fig. 12. Productivity and bug rate cross-correlations.

releases after an increase in productivity. It is sometime after but less than 10.

In Fig. 11 we see no discernable pattern in the productivity and bug rate spikes over releases in maintenance. While there is notably more variability, the correlations are generally smaller and not biased positively or negatively. Looking at the cross-correlation in Fig. 12 we see no significant correlations and they are generally lower than the cross-correlations during development. There is insufficient evidence to conclude that Productivity and bug rate are correlated during maintenance. This does not mean that quality – productivity tradeoffs are not being made. Just not often enough to be significant. Hence DevOps appears to be effective at suppressing productivity and quality tradeoffs, enabling us to sustain high quality and high productivity throughout maintenance releases. We note that this is also likely due to the upfront investment in quality during development. If maintenance did not start with high quality (in particular very low reported bug density) this effect would probably not be present.

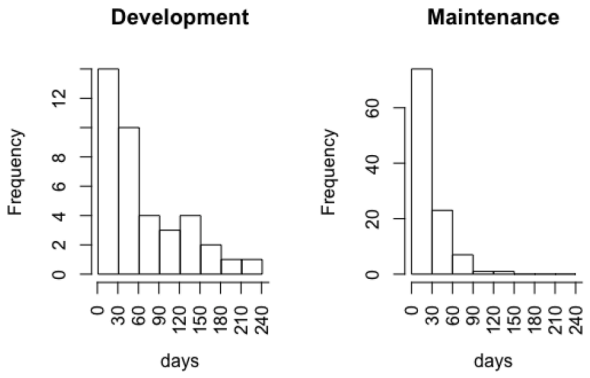


Fig. 13. Days between releases.

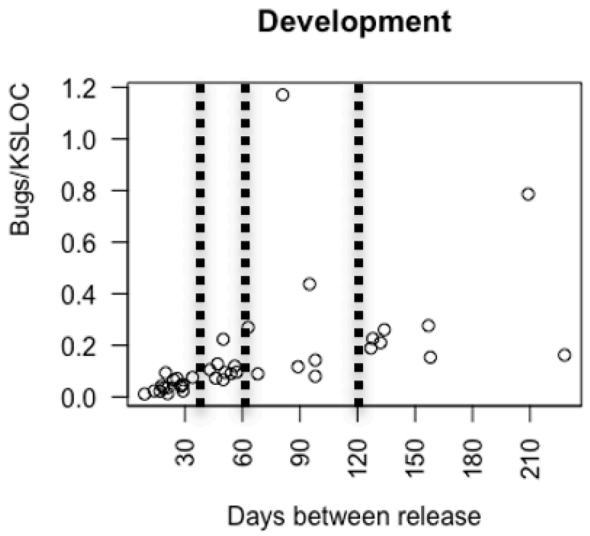


Fig. 14. Reported bug density and release frequency.

It's clear that the DevOps practices are having a beneficial effect on the maintenance process. We would like some insight into how the particular policies generate this effect and how effective they are in doing so. We also wish to understand how we can monitor compliance to policy and that practices are performed as expected. This is important not only for management purposes. We would like to have confidence that the effect is indeed a consequence of the policies, and if so, are there adjustments we might make to improve the effect or provide more flexibility in compliance.

6.4. *More frequent releases encourage higher quality*

We noted in Fig. 3 that there are considerably more releases in maintenance than in development – the x-axis ticks in maintenance are denser than in development. However, policy P1 states that releases should be monthly (typically).

Fig. 13 presents histograms of the days between releases in development and in maintenance.

The mean number of days between releases maintenance is 29 days, but the distribution is notably right-skewed indicating that releases are more frequently below this average than above it. The longest interval is 124 days. We see about 70% of the releases in maintenance happen in less than 30 days from the previous release. While technically this is not precisely compliant with (P1), the policy is meant to enable more frequent releases and thus we view this as evidence that we have in general been compliant to (P1).

As per our question Q4, having monthly releases was somewhat

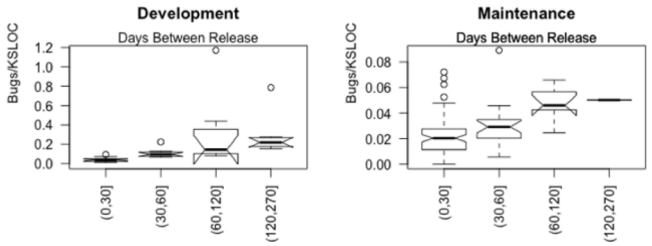


Fig. 15. Reported bug density and release frequency.

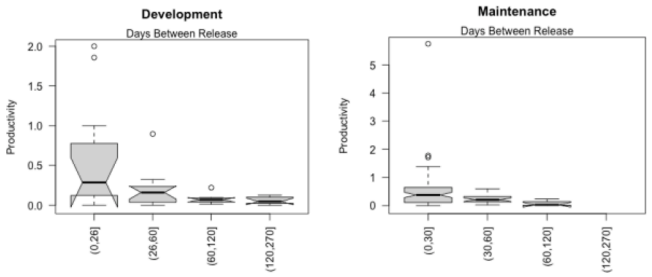


Fig. 16. Productivity and release frequency.

arbitrarily chosen and so we have latitude in the release frequency and possibly can improve the process by formulating the policy differently. Before undertaking this, let's examine why we have this policy in the first place and the effect it has on quality and productivity currently. We know that the frequency of releases will have an impact on quality and productivity since they are both related to code size. These relationships are complicated and difficult to analyze in relation to release frequency.

An alternative is to examine this empirically. Fig. 14 shows how Reported bug density varied over days between release in development.

There is a moderately strong increasing trend with increasing variability. We observe clusters around 30 days, 60 days, and 120 days (marked by the dashed lines in Fig. 14). Using these to create release frequency groups, Fig. 15 compares the Reported bug density between these groups.

We observe in the Development boxplot that releases less than 31 days from the previous release typically (as per the median) have significantly lower Reported bug density. In a notched boxplot, the notches indicate the 95% confidence interval for the median. This means that if the notches of two boxplots do not overlap, we can have high confidence that the population medians are not equal and we can say they are significantly different. While we cannot conclude significance in comparison to the 60–120-day group, the trend is clear. Less frequent releases lead to higher Reported bug density. While we could fiddle with the groups, perhaps perform a cluster analysis to find a release frequency that minimizes Reported bug density, our policy to encourage monthly releases is based in part by practicality and constraints on our process that limits how short we can make the interval between releases.

Looking at the Maintenance boxplot in Fig. 15 we see that more frequent releases still tend to have lower Reported bug density and that the less than 30 days group is significantly the lowest. We conclude that more frequent releases increase quality.

Although unlikely, in Fig. 16 we check if frequency of release effects productivity.

In Development, there appears no significant differences in productivity. There does appear to be a slight decreasing productivity trend as release time increases. But there this is not strong evidence that release frequency effects productivity. We see a similar situation in maintenance.

We conclude that DevOps policy P1 is generally being complied with and the approximately monthly release goal is effective in maintaining high quality (in terms of bugs reported between releases) without compromising productivity (in terms of enhancements implemented). There is little evidence that changing the release frequency will improve quality or productivity.

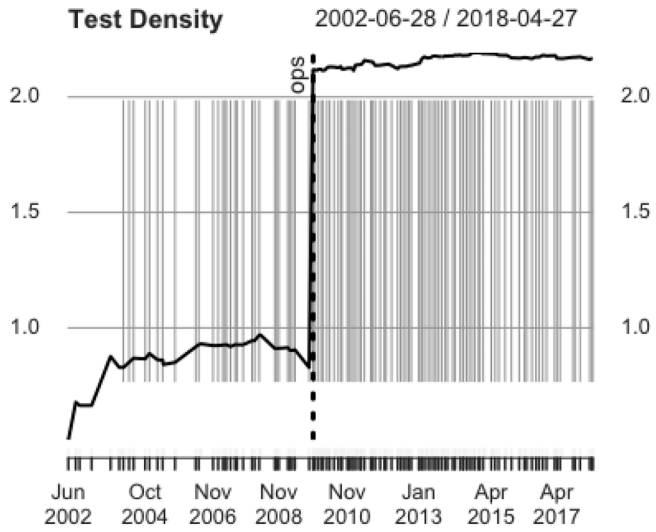


Fig. 17. MONTE Test density per release.

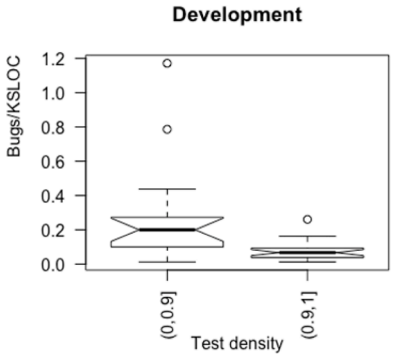


Fig. 18. Reported bug density for Test densities less than and greater than 90%.

6.5. *DevOps eliminates productivity and testing tradeoff*

DevOps requires at least 90% test coverage per release (P4). Fig. 17 presents the Test density per release.

We see clearly a dramatic increase in testing after DevOps was implemented. This is surely due to the 90% test coverage from policy P1 being implemented at this time. Compliance to P1 is clear. What we are more interested in here is if this high-test density be maintained over time. Our goal carried over from development is 1 Lsloc test per Lsloc code (an arbitrary choice). Curiously, since implementing DevOps practices, we have more than doubled this. Compliance with P1 is actually not in doubt as we have automated checking of test coverage at the nightly builds. However, it's nice to have verification that our tools are working. What we would like to see is if this increase in testing does not impact productivity too much.

Fig. 18 shows how Reported bug density compares between development releases that have Test density less than 0.9 test Lsloc per Lsloc with those that have more than 0.9.

Not surprisingly we see that the higher Test density group has significantly lower Reported bug density. That is, more testing increases quality.

Why the 0.9 cutoff? We simply kept increasing the percentage until we saw a significant difference (non-overlapping notches). So, we set our Test coverage policy (P4) to 90% assuming that this will always give

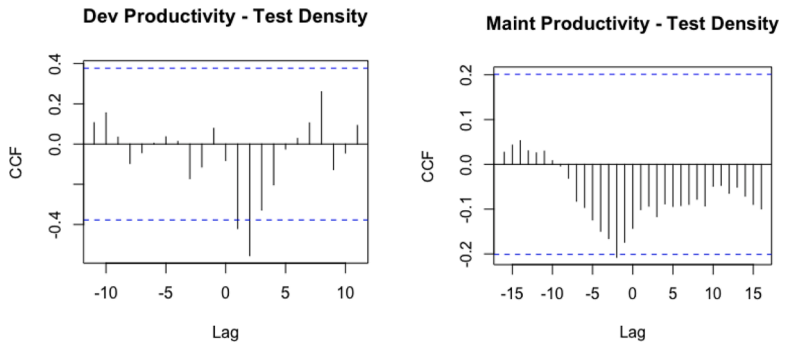


Fig. 19. Comparing development and maintenance productivity and test density cross-correlations.

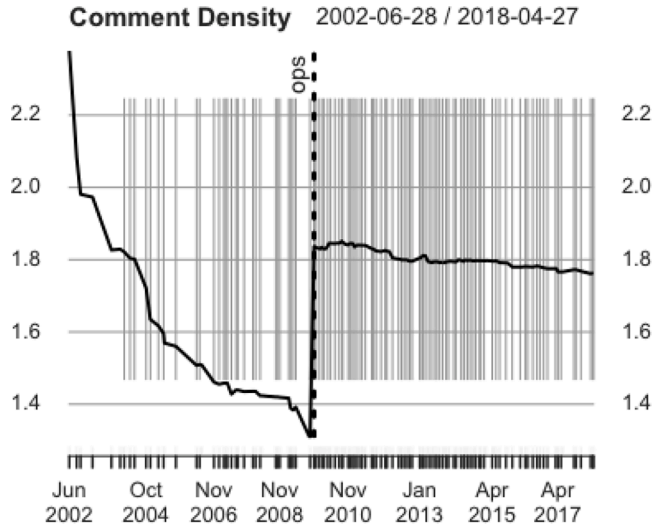


Fig. 20. MONTE Comment density per release.

a Test density over 0.9. This is automatically checked and enforced in maintenance so there is no need to investigate compliance. For evidence that this policy is effective we need only look again at the stable and very low Reported bug density after the ops event line in Fig. 7. What's notable and unexpected is that, as seen in Fig. 17, the policy seems to give a Test density greater than 2.1. Probably the higher the better here.

But must an increase in testing always decrease productivity? Fig. 19 compares development and maintenance Productivity and Test density cross-correlations.

In development, we see a strongly negative forward correlation (i.e., after lag 0). We see evidence that developers tend to increase productivity by sacrificing test. Given the strong correlation of testing to quality, this probably unwise when reliability is critical. However, in maintenance we see that enforcement of test coverage does not significantly affect productivity. This is not surprising given that test developed is counted in productivity and that test coverage is automatically checked and enforced. Although the correlations are generally negative, there are no significant forward correlations, and we note that the largest correlation is less than half the largest found in development. That we see a significant negative correlation at lag −2 is unlikely a concern as we are looking for what happens when productivity is increased, not when test density is increased. What's important here is that we do not see any evidence of productivity and testing tradeoff as we did in development and the effect of compliance to the DevOps policy (P4).

One technical note about interpreting the cross-correlation function: the Test density is significantly autocorrelated, which distorts the cross-correlation function making it appear more correlated with anything

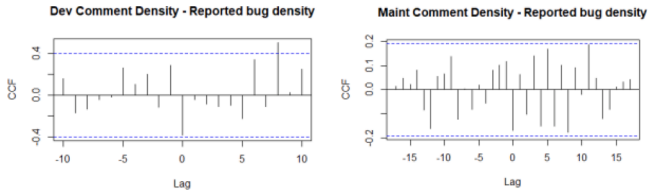


Fig. 21. Comparing development and maintenance comment density and reported bug density cross-correlations.

than it actually is. To address this issue, the series was pre-whitened (ShumwayandStoffer, 2017; Von Storch, 1995) before computing the cross-correlation.

6.6. *Enforcing documentation increases quality without sacrificing productivity*

Fig. 20 shows the evolution of Comment density per release. The decreasing Comment density leading up to the end of development may be due to developers increasing their productivity by shortcutting documentation. It is unnecessary to check compliance with DevOps policy (P3) as it is enforced in the nightly builds. This is evident in the sharp rise in Comment density at the ops event line in Fig. 20 and maintaining a high level thereafter. Here again, it's nice to see that our tools are working and that they are not being circumvented. That the Comment density has gradually been decreasing may simply be due to the increasing mature code base where fewer comments are needed as functionality is enhanced rather than added (and thus additional documentation is not needed). If this is the case, we should see this trend towards an asymptote. That is, it should achieve a nearly constant comment density sometime in the future.

We now consider the motivation for policy (P3) and its effect on quality. First, a large portion of the documentation for MONTE is generated automatically from the source code documentation. In order to make a release quickly, we need to know that the documentation is complete when the release process begins. Second, we believe that fully documented code will have higher quality as it forces developers to review and think through their designs and code more thoroughly. Bugs, especially conceptual and design bugs, that may have slipped past while developing may now be caught and fixed before they get reported as bugs after release. Thus, we expect to see a negative correlation between Comment density and Reported bug density.

Fig. 21 presents the cross-correlation between Comment density and Reported bug density. While we mainly interested in the general correlation between these variables independent of time, it is difficult to separate this from the process factors that may introduce forward or backward effects (i.e., lagged correlations). In particular, Comment density is strongly autocorrelated, even more so than Test density, and additionally, there is almost surely interaction with Test density. To

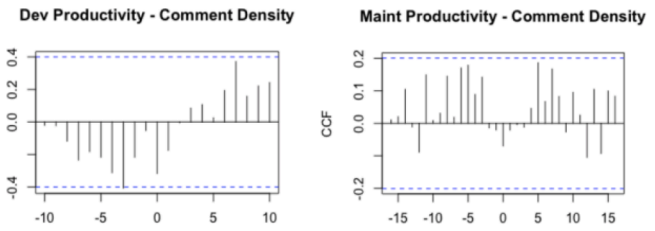


Fig. 22. Comparing development and maintenance productivity and comment density cross-correlations.

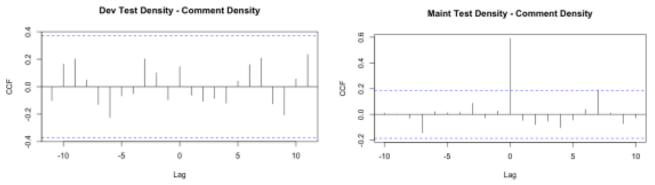


Fig. 23. Comparing development and maintenance test density and comment density cross-correlations.

untangle this, we again prewhiten the time series before considering the cross-correlation.

We see that in development there is a notable negative correlation at lag 0. Not unexpectedly, increases in documentation increase quality. We have no explanation for the large positive correlation at lag 8. In maintenance we also see negative correlation at lag 0 but much weaker (less than half that of development) and not as significant. This weaker significance may be due to the prewhitening. A concern is that the forced increase in documentation from policy (P1) negatively impacts productivity (Fig. 22).

In Fig. 18, the development cross-correlation between productivity and Comment density, we obeserve notable backward negative correlations. As we surmised, in development, developers may have been trading off productivity and documentation. Looking at the situation in maintinance we see little evedince of this. It seems that we can enforce substantal documentaion without impacting productivity.

6.7. Compliance with function documentation and tests

(P5) is a somewhat controversial policy and our developers have been skeptical of its value. However, we have developed automated tools to ensure that examples given in the documentation have associated tests. These are also part of the nightly build process. Thus, compliance is not in question. What is unclear is if developers are creating new tests for the examples as expected from (P5) or covering them with existing tests. Fig. 23 compares Test density with Comment density before and after institution of policy (P5).

What we observe is that pre-policy (P5), test code and comment code are independent. After instituting policy (P5), we see a strong positive correlation at lag 0. When more comments are written, more tests are written. The only process element that has changed from development that could associate comments and tests is (P5). Therefore, we have some indirect evidence that (P5) is being performed as expected. It is interesting that this kind of subjective process activity may be detectible from an analysis of Comment and Test density correlation.

6.8. Enforcing documentation decreases effort to fix bugs

Previously, we investigated the impact of enforcing documentation (P3) on quality and found that we can increase documentation to increase quality without sacrificing productivity. The primary motivation for (P3), however, was to manage downstream (i.e., in subsequent releases) effort required to repair bugs and implement enhancements. The

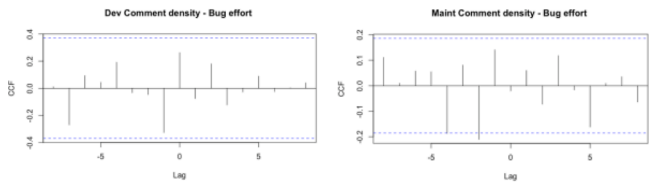


Fig. 24. Comparing development and maintenance comment density and Bug effort cross-correlations.

conventional thinking on this is that well-documented code is better organized and enables easier identification of the problem, its location, and how to fix it. The effect we expect to see is a negative correlation between Comment density and Bug effort. Fig. 24 compares the cross-correlations for these measures between development and maintenance.

In development we observe a non-significant backward (at lag −1) negative correlation. This is expected since the effort for the bugs repaired is expended for the release *one* after the release the comments were made. This is unlike the previous case looking at Comment density and Reported bug density where bugs are avoided for the given release and thus, we see the effect immediately (i.e., no lag).

During maintenance, we also see a similar significant backward negative correlation lagged at −2. The additional lag may be due to the (P5) policy which enforces much less variability and a higher Comment density. Also, in maintenance, given the more frequent release policy, a bug related to the commented code is not always reported or repaired in the next release. It may be observed or pushed to two or three subsequent releases. None the less, the effort reducing effect of commenting is present.

7. General considerations and caveats

We have discussed a number of correlations in process variables. It is important to put these in context and take care to not over-interpret the results. Our intent is to investigate our DevOps practices as a CMMI Level 3 process improvement within our environment and no claims are made as to their generalizability and applicability elsewhere. However, our results may indicate more general principles that would be of interest to other practitioners, so we now discuss some of considerations which may preclude generalizing our results that may not hold for software systems engineering in general.

7.1. Observed impacts are due to DevOps

We note that *all* of the policies P1-P6 were implemented at the same time for the first official operational release of MONTE (marking the transition to maintenance) and have not changed since. This includes the automated checking tools and changes to our delivery process such as including users earlier in system reviews (e.g., the Test Readiness Review). The data we collected and the way it was collected did not change from development. The measures and metrics also remained the same (e.g., Bug Rate). The only significant change between development and maintenance was the introduction of the DevOps practices as indicated by policies P1 – P6. In implementing these policies, there were a number of immediate impacts on the delivery process we expected to see that were observed:

- From policy P1 we expect approximately monthly releases. As per Fig. 13, the average release was 29 days with 70% of the releases less than 30 days.
- From policy P3 we expect to maintain a higher comment density with less variability. We observe this in Fig. 20, but to understand the extent we note the average comment density in development was 1.58 and this jumped to 1.81 since that start of maintenance. In

Table 1
Executive summary of conclusions to investigation questions Q1-Q4 for DevOps policies P1-P6.

	Q1: Effects productivity and quality?	Q2: Effective?	Q3: Compliance and performance?	Q4: Improvement?
P1	Yes	Yes	Yes	Yes
P2	Indirectly	Indirectly	Yes	No
P3	Yes	Yes	Yes	Yes
P4	Yes	Unsure	Partial	Yes
P5	Yes	Unsure	Partial	Yes
P6	Not checked	Not checked	Not checked	Not checked

development the standard deviation was 0.23 which dropped to 0.023 in maintenance indicating much lower variability.

- From policy P4 and P5, we expect more test code, test comments, and to maintain a higher test density with less variability. We clearly observe more test code and text comments in Fig. 3. In Fig. 17 we observe notably higher test density with lower variability. The average test density in development was 0.87 and jumped to 2.15 in maintenance. The standard deviation in development was 0.09 which dropped to 0.02 in maintenance.

There is a clear causal connection in principle between the DevOps practices and the impacts we observed. Therefore, we have high confidence that the impacts observed were due to implementing the DevOps practices.

7.2. Control of confounding variables

We have monitored the rate of bugs and production both before and after instituting the DevOps practices continually to ensure we had not degraded quality as measured by the rate bug discovery, bug density, and comment density, and test density.

Over 80% the development team has been with the project for over a decade. New staff members have joined the team, but they are mentored by the veteran developers.

No other process changes have been made other than DevOps.

The development environment and tools used (e.g., use of Python) have evolved, but very slowly over the many released. We have not observed any notable changes in the quality or productivity other than pre- and post- DevOps.

Thus, in this investigation there do not appear to be any confounding variables that might threaten our results.

7.3. Variable and process interactions

We have used cross-correlation functions to detect interactions and association confounding factors such as autocorrelation and addressed them as needed.

Overall, we are keenly aware that correlation in our process variables does not imply we can conclude that the effects we observe are in fact directly *caused* by particular DevOps policies. However, we are careful to note that we are mainly interested evidence of the effectiveness of DevOps for MONTE based on the data we have collected and not making general claims for other systems or processes. Given that we have evidence that the policies are being adhered to and we have controlled the possible confounding variables that may offer alternative explanations for our results, at least for MONTE, in this investigation, correlation does support causation.

7.4. DevOps practices for other systems

There is nothing particularly unique about MONTE or our development process. The main consideration in implementing the DevOps practices discussed in this investigation is there must be a great need to manage the variability in quality and productivity. In our case, users critically depend on the continual reliable operation of MONTE. A major bug or capability failure not addressed in a timely manner could spell doom for a mission. Similarly, needed new capabilities must be implemented within a non-negotiable time frame or a mission depending on having these capabilities may fail. Thus, for MONTE, variability in quality and productivity has to be carefully managed. Our DevOps practices target limit developers trading quality for productivity as this posed a major risk to our ability to manage this variability.

8. Summary and conclusions

The investigation discussed here is used by the Navigation Software Group at the Jet Propulsion Laboratory to provide insight for managing DevOps practices for maintaining our critical software systems. Therefore, we continually monitor our process to ensure we meet our productivity and quality releases goals and identify areas for improvement. We cannot afford the risk of trading off productivity and quality. The DevOps policies and measures were instituted in part to help mitigate this risk.

But how effective are DevOps policies and to what degree are the complied with? This work is our initial step in addressing this. We summarize our findings in relation to the questions Q1-Q4 posed in the Introduction with Table 1.

Overall, we find defensible and compelling evidence that DevOps practices are effective in contributing to the management of a high-quality and highly productive maintenance process for our critical systems here at JPL.

The investigation is ongoing as we gather more data and address the gaps of this investigation. We caution that our results are only valid within our particular investigation. However, the methods used to investigation productivity and quality are applicable in general and we hope our efforts inspire and guide other practitioners to utilize them for their own benefit.

CRediT authorship contribution statement

Dan Port: Conceptualization, Formal analysis, Investigation, Methodology, Software, Visualization, Writing – original draft. **Bill Taber:** Data curation, Funding acquisition, Investigation, Project administration, Software, Supervision, Validation, Visualization. **Parisa Emkani:** Investigation, Resources, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

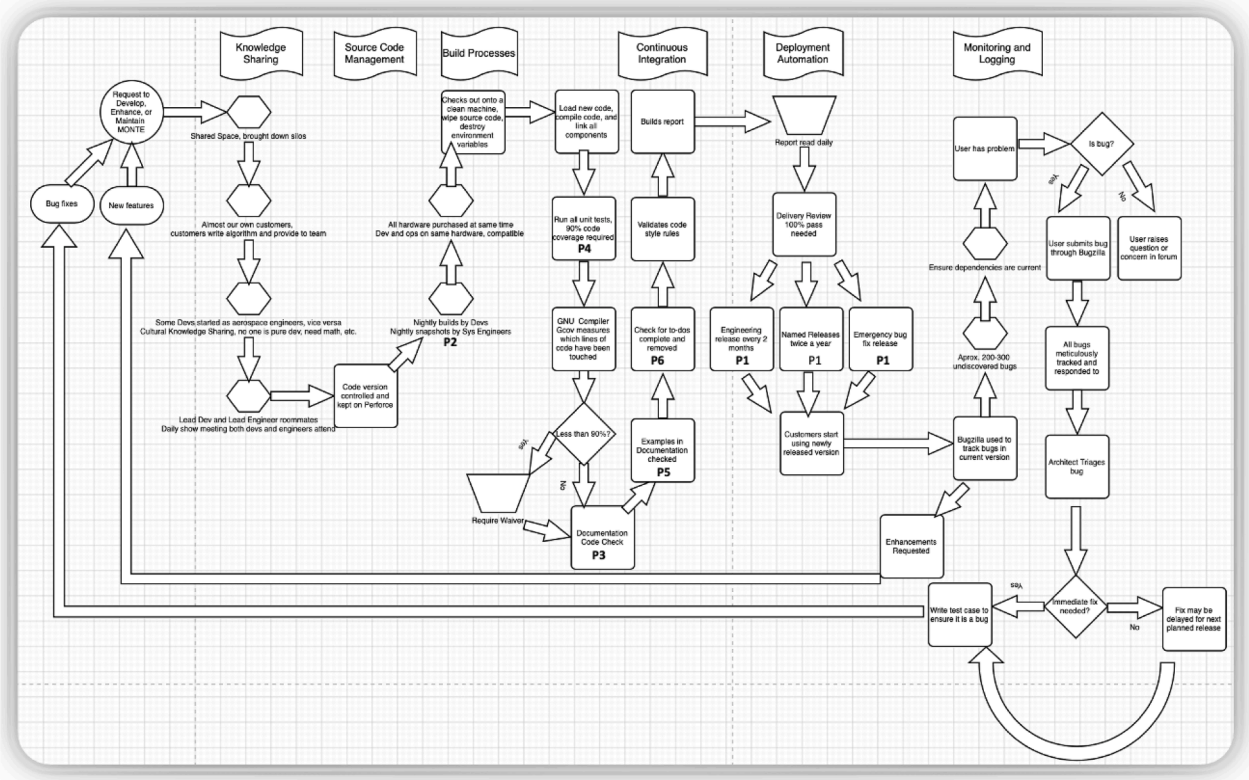
Data availability

Data will be made available on request.

Acknowledgments

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Appendix A. MONTE DevOps process



References

Afzal, M., Hameed, U., Zubair, S., Iqbal, M., Arif, S., Haseeb, U., 2023. Adoption of continuous delivery in DevOps: future challenges. J. Jilin Univ. 42, 20. https://doi.org/10.17605/OSF.IO/6NYPX.

Akbar, M.A., Rafi, S., Alsanad, A.A., Qadri, S.F., Alsanad, A., Alothaim, A., 2022. Toward successful DevOps: a decision-making framework. IEEE Access 10, 51343–51362. https://doi.org/10.1109/access.2022.3174094.

Ali, N., Daneth, H., Hong, J., 2020. A hybrid DevOps process supporting software reuse: a pilot project. J. Softw. 32 (7) https://doi.org/10.1002/smr.2248.

Amaro, R., Pereira, R., da Silva, M.M., 2023. Capabilities and practices in DevOps: a multivocal literature review. IEEE Trans. Softw. Eng. 49 (2), 883–901. https://doi.org/10.1109/tse.2022.3166626.

Anandya, R., Raharjo, T., Suhanto, A., 2021. Challenges of DevOps implementation: a case study from technology companies in Indonesia. In: 2021 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS. https://doi.org/10.1109/icimcis53775.2021.9699240.

Arvanitou, E.M., Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., Deligiannis, I., 2022. Applying and researching DevOps: a tertiary study. IEEE Access 10, 61585–61600. https://doi.org/10.1109/access.2022.3171803.

Azad, N., Hyrynsalmi, S., 2022. DevOps challenges in organizations: through professional lens. Lecture Notes in Business Information Processing, pp. 260–277. https://doi.org/10.1007/978-3-031-20706-8_18. CHALLENGES.

Azad, N., Hyrynsalmi, S., 2023. DevOps critical success factors — a systematic literature review. Inf. Softw. Technol. 157, 107150 https://doi.org/10.1016/j.infsof.2023.107150.

Basavegowda Ramu, V., 2023. Optimizing DevOps pipelines with performance testing: a comprehensive approach. Int. J. Comput. Trends Technol. 71, 35–41. https://doi.org/10.14445/22312803/IJCTT-V71I6P106.

Blüher, T., Maelzer, D., Harrendorf, J., Stark, R., 2023. DevOps for manufacturing systems: speeding up software development. Proc. Des. Soc. 3, 1475–1484. https://doi.org/10.1017/pds.2023.148.

Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., McLeod, G., Merritt, M., 1978. Characteristics of Software Quality. North Holland.

Calvache, C., Guerrero, J., Suescun Monsalve, E., 2022. DevOps model in practice: applying a novel reference model to support and encourage the adoption of DevOps in a software development company as case study. Period. Eng. Natur. Sci. 10, 221–235. https://doi.org/10.21533/pen. v10i3.3086.g1167.

Crosby, P.B., 1980. Quality is Free: The Art of Making Quality Certain. Signet.

Díaz, J., Almaraz, R., Pérez, J., Garbajosa, J., 2018. DevOps in practice. In: Proceedings of the 19th International Conference on Agile Software Development: Companion. https://doi.org/10.1145/3234152.3234199.

Evans, S., Taber, W., Drain, T., et al., 2018. MONTE: the next generation of mission design and navigation software. CEAS Space J. 10, 79–86. https://doi.org/10.1007/s12567-017-0171-7.

Faaiz, S.M., Khan, S.U.R., Hussain, S., Wang, W.L., Ibrahim, N., 2023. A study on management challenges and practices in DevOps. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering. https://doi.org/10.1145/3593434.3594240.

Faustino, J., Adriano, D., Amaro, R., Pereira, R., da Silva, M.M., 2022. DevOps benefits: a systematic literature review. Software 52 (9), 1905–1926. https://doi.org/10.1002/spe.3096.

Fenton, N.E., Pfleeger, S.L., 1997. Software Metrics: A Rigorous and Practical Approach, 2nd ed. International Thomson Computer Press. https://doi.org/10.1007/978-1-4303-6263-9.

Grunewald, E., Kiesel, J., Akbayin, S., Pallas, F., 2023. Hawk: DevOps-driven Transparency and Accountability in Cloud Native Systems.

Gokarna, M., 2021. DevOps Phases Across Software Development Lifecycle. https://doi.org/10.36227/techrxiv.13207796.v2.

Grande, R., Vizcaíno, A., García, F.O., 2024. Is it worth adopting DevOps practices in Global Software Engineering? Possible challenges and benefits. Comput. Stand. Interfaces 87, 103767. https://doi.org/10.1016/j.csi.2023.103767.

Guerrero, J.M., Zúñiga, K., Certuche, C.D., Pardo, C., 2020. A Systematic Mapping Study about DevOps.

Gwangwadza, A., & Hanslo, R. (2022). Factors that contribute to the success of a software organisation's DevOps environment: a systematic review. *ArXiv, abs/2211.04101*.

Gwangwadza, A., Hanslo, R., 2023. Towards the success of DevOps environments in software organizations: a conceptual model approach. In: Ndayizigamiye, P., Twinomurinzi, H., Kalema, B., Bwalya, K., Bembe, M. (Eds.), Digital-for-Development: Enabling Transformation, Inclusion and Sustainability Through ICTs. IDIA 2022, Communications in Computer and Information Science, vol 1774. Springer, Cham. https://doi.org/10.1007/978-3-031-28472-4_5.

Hernández, R., Moros, B., Nicolás, J., 2023. Requirements management in DevOps environments: a multivocal mapping study. Requir. Eng. https://doi.org/10.1007/s00766-023-00396-w.

Holt, C.C., 1957. Forecasting seasonals and trends by exponentially weighted moving averages. ONR Memorandum. Carnegie Institute of Technology, Pittsburgh. Vol. 52Available from the Engineering Library, University of Texas, Austin.

Humphrey, W., 1995. A Discipline for Software Engineering. Addison Wesley.

Jayakody, J.A.V.M.K., Wijayanayake, W., 2021. Challenges for adopting DevOps in information technology projects. In: 2021 International Research Conference on Smart Computing and Systems Engineering (SCSE). https://doi.org/10.1109/scse53661.2021.9568348.
Jeffries, R., 2009. Quality-Speed Tradeoff – You're Kidding Yourself. https://ronjeffries.com/xprog/blog/quality-speed-tradeoff-youre-kidding-yourself/.
João, N., Faustino, O., Pereira, R., d, M.M., Silva, A., 2023. The influence of DevOps practices in ITSM processes. Int. J. Serv. Oper. Manag. 44 (3), 390. https://doi.org/10.1504/ijsom.2023.129464.
Jones, C., 1991. Software Measurement and Control. Addison-Wesley Professional.
Khan, M.S., Khan, A.W., Khan, F., Khan, M.A., Whangbo, T.K., 2022. Critical challenges to adopt DevOps culture in software organizations: a systematic review. IEEE Access 10, 14339–14349. https://doi.org/10.1109/access.2022.3145970.
Khan, M.O., Jumani, A.K., Farhan, W.A., 2020. Fast delivery, continuously build, testing and deployment with DevOps pipeline techniques on Cloud. Indian J. Sci. Technol. 13 (5), 552–575. https://doi.org/10.17485/ijst/2020/v13i05/148983.
KnowledgeHut, 2022. History of DevOps. KnowledgeHut10Feb. https://www.knowledgehut.com/blog/devops/history-of-devops.
Krey, M., Kabbout, A., Osmani, L., Saliji, A., 2022. DevOps adoption: challenges & barriers [Conference paper]. In: Proceedings of the 55th Hawaii International Conference on System Sciences, pp. 7297–7309. https://doi.org/10.24251/HICSS.2022.877.
Lwakatare, L.E., Kuvaja, P., Oivo, M., 2016. An exploratory study of DevOps extending the dimensions of DevOps with practices. ICSEA 104, 2016.
Lazuardi, M., Raharjo, T., Hardian, B., Simanungkalit, T., 2021. Perceived benefits of DevOps implementation in organization: a systematic literature review. In: 2021 10th International Conference on Software and Information Engineering (ICSIE). https://doi.org/10.1145/3512716.3512718.
Lwakatare, L.E., Kilamo, T., Karvonen, T., Sauvola, T., Heikkilä, V., Itkonen, J., Kuvaja, P., Mikkonen, T., Oivo, M., Lassenius, C., 2019. DevOps in practice: a multiple case study of five companies. Inf. Softw. Technol. 114 https://doi.org/10.1016/j.infsof.2019.06.010.
Luz, W.P., Pinto, G., Bonifácio, R., 2019. Adopting DevOps in the real world: a theory, a model, and a case study. J. Syst. Softw. 157, 110384 https://doi.org/10.1016/j.jss.2019.07.083.
Mamdouh, A., Mohammad, Z., Sultan, A.S., 2019. DevOps development process awareness and adoption - the case of Saudi Arabia. I-Manager's J. Softw. Eng. 14 (1), 21. https://doi.org/10.26634/jse.14.1.16519.
Maroukian, K., Gulliver, R., 2020. Leading DevOps practice and principal adoption. In: 9th International Conference on Information Technology Convergence and Services (ITCSE 2020). https://doi.org/10.5121/csit.2020.100504.
Marques, P., & Correia, F.F. (2023). Foundational DevOps Patterns. ArXiv, abs/2302.01053.
Mishra, A., Otaiwi, Z., 2020. DevOps and software quality: a systematic mapping. Comput. Sci. Rev. 38, 100308 https://doi.org/10.1016/j.cosrev.2020.100308.
Mohammad, S.M., 2018. Improve software quality through practicing DevOps automation. In: International Journal of Creative Research Thoughts (IJCRT). ISSN, 2320-2882.
Morales, J.A., Yasar, H., Volkman, A., 2018. Implementing DevOps practices in highly regulated environments. In: Proceedings of the 19th International Conference on Agile Software Development: Companion. https://doi.org/10.1145/3234152.3234188.
Muñoz, M., Rodríguez, M.N., 2021. A guidance to implement or reinforce a DevOps approach in organizations: a case study. J. Softw. https://doi.org/10.1002/smr.2342.
Narang, P., Mittal, P., 2022. Performance assessment of traditional software development methodologies and DevOps automation culture. Eng. Technol. Appl. Sci. Res. 12 (6), 9726–9731. https://doi.org/10.48084/etasr.5315.
Noorani, N., Zamani, A., Alenezi, M., Shameem, M., Singh, P., 2022. Factor prioritization for effectively implementing DevOps in software development organizations: a SWOT-AHP approach. Axioms 11 (10), 498. https://doi.org/10.3390/axioms11100498.
Octopus Deploy, 2022. What is DevOps? https://octopus.com/devops/.
Offerman, T., Blinde, R., Stettina, C.J., Visser, J., 2022. A study of adoption and effects of DevOps practices. In: 2022 IEEE 28th International Conference on Engineering, Technology and Innovation (ICE/ITMC) & 31st International Association for Management of Technology (IAMOT) Joint Conference. https://doi.org/10.1109/ice/itmc-iamot55089.2022.10033313.
Petrović, N., 2022. Semantic approach to code quality improvement in DevOps. In: SAUM 2022, pp. 35–37.
Pando, B., Dávila, A., 2022. Software testing in the DevOps context: a systematic mapping study. Programm. Comput. Softw. 48 (8), 658–684. https://doi.org/10.1134/s0361768822080175.
Pardo, C., Orozco, C., Guerrero, J., 2023. DevOps Ontology - an ontology to support the understanding of DevOps in the academy and the software industry. Period. Eng. Nat. Sci. 11 (2), 207. https://doi.org/10.21533/pen.v11i2.3474.
Pedra, M.L., Silva, M.F., & Azevedo, L.G. (2021). DevOps Adoption: Eight Emergent Perspectives. ArXiv, abs/2109.09601.
Port, D., Taber, B., Huang, L., 2023. Investigating a NASA cyclomatic complexity policy on maintenance risk of a critical system. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Melbourne, Australia, pp. 211–221. https://doi.org/10.1109/ICSE-SEIP58684.2023.00025, 2023.
Rafi, S., Yu, W., Akbar, M.A., Mahmood, S., Alsanad, A., Gumaei, A., 2020. Readiness model for DevOps implementation in software organizations. J. Softw. 33 (4) https://doi.org/10.1002/smr.2323.
Rütz, M., 2019. DevOps: a systematic literature review. Inf. Softw. Technol.
Said, S.E., Dickey, D.A., 1984. Testing for unit roots in autoregressive-moving average models of unknown order. Biometrika 71, 599–607.
Senapathi, M., Buchan, J., Osman, H., 2018. DevOps capabilities, practices, and challenges. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. https://doi.org/10.1145/3210459.3210465.
Shumway, R.H., Stoffer, D.S., 2017. Time Series Analysis and Its Applications: With R Examples, 4th ed. Springer.
Sravan, S.S., Sai Ganesh, C., Kiran, K.V.D., Aakash Chandra, T., Aparna, K., Vignesh, T, 2023. Significant challenges to espouse DevOps culture in software organisations By AWS: a methodical review. In: 2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS). https://doi.org/10.1109/icaccs57279.2023.10113021.
Srivastav, S., Allam, K., Mustyala, A., 2023. Software automation enhancement through the implementation of DevOps. Int. J. Res. Publ. Rev. 4 (6), 2050–2054. https://doi.org/10.55248/gengpi.4.623.45947.
Taber, B., Port, D, 2014. Empirical and face validity of software maintenance bug models used at the jet propulsion laboratory. In: 8th International Symposium on Empirical Software Engineering and Measurement (ESEM 2014), September 18-19, 2014, Torino, Italy.
Tenzin, S., Tshering, Y., Choden, S., 2022. Adoption and Implementation of DevOps by the Software Companies of Bhutan. IARJSET 9 (9). https://doi.org/10.17148/iarjset.2022.9919.
Trigo, A., Varajão, J., Sousa, L., 2022. DevOps adoption: insights from a large European Telco. Cogent Eng. 9 (1) https://doi.org/10.1080/23311916.2022.2083474.
Von Storch, H., 1995. Misuses of statistical analysis in climate research. In: von Storch, H, Navara, A. (Eds.), Analysis of Climate Variability: Applications of Statistical Techniques. Springer-Verlag, Berlin, Germany, pp. 11–26.
Wahaballa, A., Wahballa, O., Abdellatief, M., Xiong, H., Qin, Z., 2015. Toward unified DevOps model. In: 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS). https://doi.org/10.1109/icsess.2015.7339039.
Yarlagadda, R.T., 2019. How DevOps enhances the software dévelopment quality. Int. J. Creat. Res. Thoughts VII (3), 358–364.
Zarour, M., Alhammad, N., Alenezi, M., Alsarayrah, K., 2020. DevOps process model adoption in Saudi Arabia: an empirical study. Jordanian J. Comput. Inf. Technol. 0, 1. https://doi.org/10.5455/jjcit.71-1580581874.
Zulkarnain, J., Mulya, R.F., Pratiwi, T., Pangestuti, W., Ilmawati, F.I., 2022. DevOps main area and core capabilities adopting DevOps in the last decade: a systematic literature review. Int. J. Res. Appl. Technol. 2 (2), 184–197. https://doi.org/10.34010/injuratech.v2i2.8364.

Dan Port is an associate professor in the Department of Information Technology Management at the University of Hawaii at Manoa. His primary research area is empirical software engineering; he specializes in software system assurance, strategic methods for development, and software and systems engineering education. Port received a PhD in applied mathematics from MIT. He's a member of IEEE, Beta Gamma Sigma, and the International Software Engineering Research Network.

Bill Taber is the technical group supervisor of the Mission Design and Navigation Software Group of the Mission Design and Navigation section at the Jet Propulsion Laboratory. His research interests include software reliability and mathematical modeling of the software development process. Taber received a PhD in mathematics from the University of Illinois Urbana-Champaign. He's a member of ACM and Sigma Xi.

Parisa Emkani is a graduate student in Information and Computer Science at the University of Hawaii at Manoa. Her current research interests are software systems maintenance and DevOps. She holds a Bachelor's degree in Information Technology Engineering and a Master's in Business Administration. She briefly pursued a Ph.D. in Business, but her passion for software engineering led her back to computer science.