Perspectives on Operationalizing Scientific Software

Ryan Adamson*, Addi Malviya Thakur$
*National Center for Computational Sciences, $Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA.

***Abstract*—Software best practices have evolved to speed up feature development, to minimize the introduction of software flaws, and to reduce security risk associated with software changes. Given a well-understood deployment platform target, common DevSecOps workflows lead to productivity gains. Unfortunately, there is a gap between the development of *scientific* research software and the deployment of that software in an operational scope. We explore the fundamental differences between the deployment of research and enterprise software and provide a framework for minimizing the rework necessary to deploy software when a research software prototype proves viable. We propose a balanced approach to the development of scientific software; the best outcomes are achieved when scientific and operations teams collaborate in a deliberate manner.**

***Index Terms*—Scientific Software, Operational Software, Toil, DevSecOps**

I. Introduction

In this paper, we define research software to be novel algorithms or the implementation of research ideas within a software context. Research software engineering [1] is performed for science's sake, typically to support the hypothesis of a related research question in order to provide value to a sponsor. These research scopes of work generally have a science question, perhaps an accompanying publication, and a period of performance in which funding can be spent. Funds typically expire at the end of the period of performance, and under-spending or lack of progress can necessitate a catch-up flurry of activity at the end that can be hard to accommodate.

In contrast, operational software deployed in production environments is rarely developed for the software's sake alone. Operations teams deploy this type of software to provide *capabilities* that an operations team is responsible for delivering, often times to the satisfaction of an Service Level Agreement (SLA) with the software customer. As such, operational software is a living entity and is subject to replacement with another tool at any point in time that does a "better job" at addressing the operational needs and responsibilities of the operations team.

Regarding differences between research and operational software, we make the following observations:

1) Research software is by definition unproven. Failure of research prototypes to prove a hypothesis are common and even expected. Valid operational software tools tend to emerge from a litany of research software prototypes.
2) Operational software is by definition production ready. Best practices for Site Reliability Engineers [2] call for reducing toil and other non-valuable work by building software best practices into the code base and deployment methodologies.
3) Because of the uncertainty of research prototypes, work performed early in a research project regarding requirements gathering may be wasted if the prototype fails, thus this work is a form of toil.
4) Operational software is a living system. Underlying libraries, hardware, provisioning tools, and operational requirements change over time, and software maintenance is an ongoing necessity.
5) All operational software began as a research idea. A research software prototype successfully validate that idea at some point and future software was either written or hardened to operational standards successfully.

Given these observations, we will explore the gap between these types of software through the lens of the DevSecOps methodology.

II. Characterizing Scientific Software Effort

With the paradigm shift to an integrated DevSecOps team structure [3][4], scientific software developers in mature organizations are now able to work much closer to the operations and security teams. Site Reliability Engineers (SREs) embody the concept of integrated security and operations capabilities. SREs are a relatively new concept and are employed to maintain the deployment platforms that software developers use for deployment of their software services. Many SRE best practices are freely available [2]; among the central tenets of the SRE mantra are the concepts of continuous automation of daily tasks and infrastructure as code. "Useful Work" such as software engineering, systems engineering, and even some amount of overhead tasks are healthy, but tasks that are manual, repetitive, tactical, or that have no enduring value can be thought of as wasteful "toil".[5] The mantra that emanates from software engineering best practices is actually quite similar. Solid software design principles, developing incremental feature enhancements, enforcing strong testing frameworks, better documentation strategies, continuous deployment methodologies, and complexity reduction [6] [7] serve to identify and remove potential wasted work and enhance developer productivity. Why then is research software that follows these design principles often so hard to deploy in a production environment as an operational capability?

A. The Deck is Stacked

It turns out that even when research software engineers follow best practices, scientific software can still have issues

at deployment time. Operational, security, budgetary, and strategic policies of the deployment platform and managing organization must be adhered to. Unfortunately, those policies are often not considered by research software engineers or are worst-case *unknowable* during the development of research software.

Let's consider the plight of a research software development team that is testing a novel capability and wants to eventually deploy the tool in an operational context:

1) *Even if* operational policies are known at the outset, the constraints may change in the time between design, successful demonstration, and deployment.
2) Funding profiles, compressed timelines, or other sponsor constraints may prevent the research team from spending much effort on the operationalization problem, kicking the can down the road in the hopes of obtaining more funding at a later date.
3) If there is not a clear targeted infrastructure for the eventual deployment of the application, the research team cannot even bring in an operations SRE to help advise them on the early design decision tradeoffs.
4) In certain secure areas, specific security policies may not even be shareable between SRE and software development teams when need-to-know cannot be established before prototype viability is demonstrated.

To the sponsor's surprise, of course, more work – and in the worst cases, a complete rewrite – needs to be performed to meet the eventual production goal once prototype validity has been demonstrated.

B. A Mathematical Definition of Effort

If we consider the path that a research idea takes from prototype software to demonstration to operationally deployed software, we can model it as a mathematical function. Let's assume that a new piece of scientific software x will be viable, and also assume that there is perfect information about the operational deployment requirements for the specific deployment targets, we can define the total work required to deploy an initial operational version of research software x on n target platforms with:

$$Work(x) = P(x) + \sum_{i=1}^{n} G_i(x) + R_i(x)$$

where P is the development effort required to design, build, and test the research prototype, G_i is the work required to gather and integrate operational requirements throughout the development of the project and R_i is the rework required to retrofit the research prototype to operational requirements to the n arbitrary deployment platforms. Note that the real values of P, G_i, and R_i are very dependent on the similarity of operational and security requirements placed on each deployment platform and to the initial design of the prototype itself. Platforms that provide the same container deployment tools, reside within the same organization, and have the same set of users and security policies will have significant requirement overlap.

Of course, these assumptions regarding the viability of prototypes and a perfect understanding of the deployment requirements do not hold in practice. Thus, in order to minimize $Work(x)$, a balance must be struck between the amount of requirement gathering effort that is spent, the total number of platforms targeted, and the expected cost of reworking a prototype for a platform. The fundamental tradeoff for a research software team is clear: Given the expected probability of prototype success and the expected complexity of deployment requirements, should the upfront cost of integrating SRE into the design of the prototype be spent in order to minimize the deployment rework, keeping in mind that the failure of prototype demonstration results in wasted work regarding requirements gathering and building those into the prototype.

III. Scaling Scientific Applications

Critical challenges associated with the operational deployment of scientific applications include ensuring robustness, responsiveness, stability, resiliency, and security. Scientific development teams, security engineers, and operations SREs must address these challenges using a DevSecOps integration strategy. This is typically done through a Product Readiness Review (PRR) [8] to assess operational maturity. While considerations such as monitoring, logging, internal and external documentation, architectural diagrams, and branding are important, the most significant considerations that could cause impact to other users, programs, and capabilities sharing the platform are discussed below.

A. Network Latency and Bandwidth Scaling

An operational scientific application should be responsive to end-user queries and interactions and should respond to such requests in a reasonable timeframe. Deployments should be able designed to scale with the expected number of users and in such a way as to not adversely impact the shared network infrastructure.

B. Scientific Application Stress Testing and Load Balancing

The scientific application's stability and reliability should be evaluated using a set of test benchmarks to validate application readiness. Some examples include memory leakage, garbage collection, handling of run-time errors, graceful termination, load and performance testing, and availability, among others[9]. This conformance allows both developers and SREs to catalog the upper and lower bounds of the applications' performance under weak, normal, and heavy workloads.

C. Database Performance and Schema Considerations

Choosing an appropriate database organization strategy is important as it is often very difficult to retrofit poor design decisions without a disruptive and comprehensive extract, transform, load operation. Database normalization is important for this purpose, and enhances data retrieval, reduces errors, provides redundancy, and supports CRUD consistency[10]. A normalized database has a more future-proof logical structure

and it compliments the application architecture as well[11]. Recently, NoSQL databases (e.g. MongoDB) have became popular choices for data-intensive scientific applications.

D. Application Modularization

Software application modularity significantly increases maintainability as well as rapid development[12]. A modular application is less complex, is highly re-usable, increases collaboration, and has a far lower probability to contain bugs in the modules that have already been operationally hardened. If possible, a scientific application should be modular from the beginning unless significant effort is required. In case, such consideration to convert a monolithic application to be more modular should be made during the operationalization of any scientific application.

E. Improving Application Usability

Scientific applications' usability is one of the essential criteria for broad adoption. Good usability provides a better user experience and increases user productivity. An improved human-computer interaction indicates greater intuitiveness and understandability of the application which thereby increases its adoption.

F. Cybersecurity Compliance

Cybersecurity is a key consideration when operationalizing scientific applications. Use of secure protocols and authentication methods improves network protection and provides defense to the applications from common attacks. Secure systems are more resilient to unauthorized data access, identity theft, ransomware attacks, perform proper logging, and appropriately enforce user separation and user/system boundaries.

G. Documentation and Training

Finally, to better attract and retain a user base, scientific applications must provide proper documentation, guidebooks, and relevant training material. Proactive documentation and support prevents user tickets and minimizes operational toil.

IV. Reducing Scientific Toil

In order to reduce the average amount of scientific toil a research team might experience, we present the following considerations. While these can be adopted solely by research software engineers, it is much more advantageous for a combined DevSecOps team to help navigate the journey from research idea inception, through prototyping, and ultimately to operational deployment.

A. Assess the deployment landscape

Perhaps the most important consideration is the *early* assessment of the potential target deployment landscape. If it is clear that the research software can be modularized and deployed using containerization technology, that provides a lot of flexibility. Many sites, however, do not support containerization or common cloud infrastructures. If certain assumptions are not valid at the deployment site, consider what features of the software will need to be reworked.

B. Empathize with other teams

Strong DevSecOps teams have well-balanced and well-defined roles and responsibilities, nurture a blame-free culture, and empathize with one another. Distribution of responsibilities may vary widely between different organizations, but the strongest teams tend to have both the *capability* as well as the *authority* to manage and improve their services. Unhappy developer, operations, or security teams take longer to reach shared consensus and often fall into one of two camps:

1) Teams with capability but not authority are not able to choose their own destiny, and suffer from the technical choices (often made at higher levels) that do not line up with their strengths.
2) Teams with authority but not capability typically lack training, staff, or budgetary resources to fully take advantage of the technical choices they do make.

Due to the tight integration of DevSecOps teams, team members migrate much closer to the center of the responsibility, availability, and capability Venn diagram as shown in Figure-1. Developers making decisions about their software are able to lean on both security and operations experts, giving back some authority to the security and SRE teams. Likewise, changes to security compliance constraints are much more likely to be discussed and engineered ahead of time, letting SRE and development teams have a voice in the conversation.

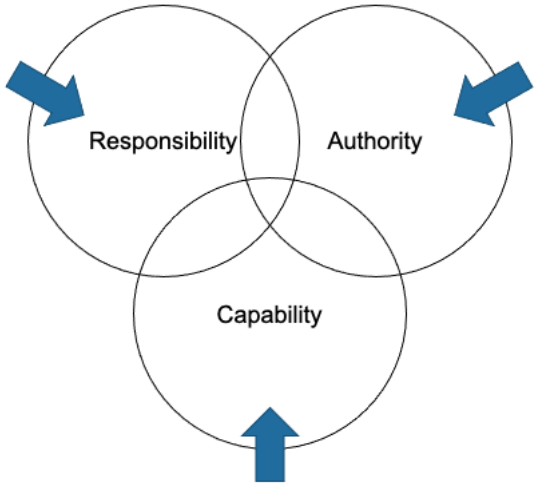

Fig. 1. Convergence of Responsibility, Authority, and Capability

C. Plan for some amount of rework

Some amount of rework will probably be necessary after any successful prototype demonstration, so planning *some* amount of cost and schedule contingency should be done up front. In the best case, no rework is necessary, and that contingency can be used to either further research or add support for the application to run on other non-targeted environments. The amount of planning and requirements gathering at the beginning of a project is inversely proportional to the cost of retrofitting the research prototype. Front-loaded effort, of course, must be balanced against the viability of the research itself, or there is risk of scientific toil.

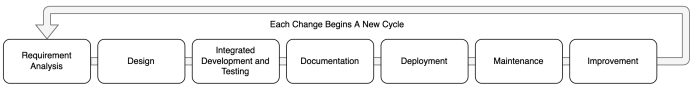

Fig. 2. Software Deployment and Maintenance Lifecycle

D. Employ best practices from other domains

One can argue that DevOps methodologies emerged from operations teams adopting developer tools as infrastructure-as-code became a design pattern. Similarly, DevSecOps teams should employ best practices from each of the development, security, and operational domains (and beyond!). One such best practice is coordinating feature development or design changes as shown in Figure-2. This is a standard model for following and tracking changes to a software product, operational service, or security tool so that all stakeholders can be notified and retain some authority for their responsibility areas that might be affected.

E. Manage the sponsor

Finally, the sponsor of a research effort is ultimately a deciding factor in the calculus of effort, and thus scientific toil. Sponsors with inflexible timelines and meager budget constraints should be managed in order to explain the retrofitting necessary should they want a viable-but-lean prototype deployed operationally. On the other hand, if a sponsor is clear about an eventual operational goal, front-loading effort before prototype viability is demonstration comes with risks that should also be discussed.

V. Conclusion

There is a gap in research software best practices when it comes to the operationalizing of prototype software. Essentially, any work towards that prior to demonstration of the prototype risks being wasted, but rework may be necessary after demonstration if the original design principles were based on faulty assumptions about eventual deployment. The software development research team should weigh the probabilities of prototype success with the cost of rework on the targeted deployment infrastructure in order to make informed decisions about reducing effort, thereby providing maximum value to the sponsor. A well-planned approach that includes anticipating such challenges and discovering associated patterns at the beginning would greatly advance the developmental effectiveness of scientific software applications. It is worthy to discuss potential concerns and their mitigation with the sponsors at the beginning and potentially include the operations team from the beginning to accelerate the transition from a research application to an operational application. An early participation as well as setting up a CI/CD pipeline resembling the target environment would enable the detection of potential gaps as well as improve the overall process.

VI. Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] C. R. Prause, R. Reiners, and S. Dencheva, "Empirical study of tool support in highly distributed research projects," *Proceedings - 5th International Conference on Global Software Engineering, ICGSE 2010*, pp. 23–32, 2010.

[2] Google, "Google - Site Reliability Engineering." [Online]. Available: https://sre.google/sre-book/table-of-contents/

[3] A. A. U. Rahman and L. Williams, "Software security in devops: Synthesizing practitioners' perceptions and practices," in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, 2016, pp. 70–76.

[4] V. Mohan and L. B. Othmane, "Secdevops: Is it a marketing buzzword? - mapping research on security in devops," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*, 2016, pp. 542–547.

[5] Google, "Google - Site Reliability Engineering." [Online]. Available: https://sre.google/sre-book/eliminating-toil/

[6] B. Moseley and P. Marks, "Tar Pit fossils," *Boreas*, vol. 15, no. 1, pp. 82–82, 2008.

[7] N. Wirth, "A Plea for Lean Software," 1995.

[8] Google, "Google - Site Reliability Engineering." [Online]. Available: https://sre.google/sre-book/evolving-sre-engagement-model/

[9] W. B. Nelson, *Accelerated testing: statistical models, test plans, and data analysis*. John Wiley & Sons, 2009, vol. 344.

[10] S. Friedrich and N. Ritter, *CRUD Benchmarks*. Cham: Springer International Publishing, 2019, pp. 529–533. [Online]. Available: https://doi.org/10.1007/978-3-319-77525-8_116

[11] C. Beeri, P. A. Bernstein, and N. Goodman, "A sophisticate's introduction to database normalization theory," in *Readings in Artificial Intelligence and Databases*. Elsevier, 1989, pp. 468–479.

[12] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 99–108, 2001.