Software Reliability in a DevOps Continuous Integration Environment

Mary "Lisa" Williams Bates, Raytheon Missiles & Defense
Enrique I. Oviedo, Ph. D., Raytheon Missiles & Defense

Key Words: Software Reliability, DevOps

2021 Annual Reliability and Maintainability Symposium (RAMS) | 978-1-7281-8017-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/RAMS48097.2021.9605768

1 SUMMARY & CONCLUSIONS

Government customers are starting to ask for software reliability in request for proposals as they seek to gain visibility into the effectiveness of the development processes and the maturity of the product. Since manually tracking software reliability is a labor intensive task, this paper explores the feasibility of obtaining reliability metrics autonomously by integrating into an existing DevOps/Continuous Deployment (CD) environment pipeline. Automating the collection and reporting of reliability data within a DevOps pipeline would provide the following benefits:

- Instantaneous visibility into the effectiveness of software verification
- Gauge of software maturity at any point in development cycle
- Facilitates dynamic recalibration of resource scheduling
- Minimize impact to developers while providing feedback about their progress
- Provide dashboards to allow real-time comparisons between predicted vs estimated reliability

2 INTRODUCTION

Reliability is a prediction or estimated measure of the failure free performance of software over a specific interval under specific conditions. As a prediction, reliability is a tool to anticipate the resources needed to discover and resolve defects throughout the software development lifecycle. As a risk reduction measure, coupled with an initial prediction, reliability is an instantaneous gauge of the efficiency of the software development process in achieving failure free performance over time. In a rapid development environment, where continuous feedback is key to monitoring and recalibrating the development process, reliability is an indispensable metric to demonstrate the efficiency of its ability to discover and resolve defects early in the development lifecycle, and provide a level of confidence that the software is approaching its desired maturity.

3 PLAN OF ACTION

In order to form a path forward on implementation of software reliability in a DevOps Continuous Integration Environment understanding of the current state is key. The plan of action is comprised of the following steps:

- Review documentation on Software reliability.
- Document definitions and types of data needed for software reliability models.
- Work to find alignment between critical DevOps metrics, company software reliability guidelines and the IEEE 1633.[1]
- Research current programs using DevOps tools across the company, gather their names and study lessons learned.
- Conduct a survey to gather data about current state of software reliability across the company.
- Compile results from the survey into a master list.
- Create a list of manual steps needed to have a robust automated software reliability program.
- Create a matrix providing the relationships between base data gathered in a DevOps/ Continuous Integration (CI) environment, metrics, stakeholders, and tools.

4 DEFINITIONS

There are no industry standards (i.e., MIL-STD or IEEE) on software reliability in a DevOps /CI environment. This paper suggests that the individual program leadership understand the current definitions pertaining to software reliability in general. The program then needs to adapt them for the instantaneous environment and educate their customers.

- *Defect:* A problem that, if not corrected, could cause an application to either fail or to produce incorrect results. [2] NOTE—For the purposes of this standard, defects are the result of errors that are manifest in the system requirements, software requirements, interfaces, architecture, detailed design, or code. A defect may result in one or more failures. It is also possible that a defect may never result in a fault if the operational profile is such that the code containing the defect is never executed.
- *Fault:* A defect in the code that can be the cause of one or more failures. (B) A manifestation of an error in the software. [2]. NOTE—There may not necessarily be a one-to-one relationship between faults and failures if the system has been designed to be fault tolerant or if a fault is not severe enough to result in a failure.
- *Error:* A human action that produced an incorrect result, such as software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, incorrect translation, or omission of a requirement in the design specification. [2]. NOTE—For the purposes of this standard, an error can also include incorrect software interfaces, software architecture, design,

978-1-7281-8017-5/21/$31.00 ©2021 IEEE

or code.

- *Feature Cycle Time:* The elapsed time of a feature's lifecycle from start of development (i.e. feature requirements, design, implementation, integration and test) through delivery. [3]
- *Time to First Deployment:* The time from contract award to deployment of the first batch of useful functionality.
- *Defects in Near Ops/Ops per Deploy:* Number of product defects found in near-ops or ops for all deployments to those environments. [3]
- *Automate Test Requirements Coverage:* A measure of the degree to which automated tests (system, component, or unit) verify system level requirements. The measure is based on the number of requirements tested as a percentage of total requirements of the system.[3]
- *Deployment Frequency:* The average count of deployment to Ops or near-Ops over a period of time. The deployment must incorporate a change.[3]
- *Mean Time to Recover:* Duration of time it takes to recover to a good state after introduction of a Failure to Ops or near-Ops. A Failure results from a deployment that causes degraded service or requires [3]
- *Lead Time:* Time elapsed from the time a feature is planned, through delivery. [3]
- *Automate Structural Coverage %:* A measure of the degree to which automated unit tests verify application functionality. The measure is based on the number of structures tested as a percentage of total structures.
- *Static analysis:* complexity checking, dead code, duplicate code checking which is mentioned in IEEE 1633[1] when talking about allocation, and software reliability and software defect per million hours
- *Testing Coverage*: Baseline of code / how many lines of code did I exercise during testing
- *Feature*: A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option. Features must have a definition of Done and acceptance criteria.[3]
- *LRU*: A software Line Replaceable Unit (LRU) is the lowest level of architecture for which the software can be compiled and object code generated. Software configuration items are commonly referred to a computer software configuration item (CSCI). However, a CSCI may be composed of more than one LRU. Hence the term LRU will be used in this document. While hardware components are replaced with identical hardware components, software components are updated whenever software defects are corrected. The lowest replacement unit for software will be either a dynamically linked library (DLL) or an executable. Software predictions are conducted at the program or executable level. Predictive models are not performed on the modules or units of code because the lowest replaceable unit is the application, executable, or DLL. Usually each LRU has a one-to-one relationship with a CSCI). However, in some cases CSCIs are composed of more than one software LRU. Hence, a listing of CSCIs does not always provide a listing of LRUs. [1].
- *Complexity*: Programming complexity (or software complexity) is a term that includes many properties of a piece of software, all of which affect internal interactions. According to several commentators, there is a distinction between the terms complex and complicated. Complicated implies being difficult to understand but with time and effort, ultimately knowable. Complex, on the other hand, describes the interactions between a number of entities. As the number of entities increases, the number of interactions between them would increase exponentially, and it would get to a point where it would be impossible to know and understand all of them. Similarly, higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes. In more extreme cases, it can make modifying the software virtually impossible. [4]
- *Stress testing from Critical Items List*: Generate a critical items list (CIL)—The output of the software failure modes and effects analysis (SFMEA) is the list of critical items and their associated mitigations. This list should be used as input to several other software reliability and engineering tasks such as developing the test suite to test the failure modes as part of the stress case coverage testing.[1]
- *CSCI*: Before beginning the analysis, determine which Computer Software Configuration Items (CSCI) will be included. Generally, the analysis should cover all CSCIs that can affect system critical mission performance or personnel safety. Other CSCIs may be included if the nature of the program justifies it. For each CSCI, determine the amount of reused versus new code. [3]

5 RELATIONSHIP MATRIX

The DevOps/CI environment and software reliability relationship was the primary focus for this study. The relationship matrix developed details the base data needed for most software reliability models, a list of metrics aligned to measure key DevOps outcomes along with software reliability, the stakeholders involved, and the tools that are currently used. The matrix identified 4 areas of automation related to tools which would give us access to 11 base data measures tied to 7 metrics. The body of metrics and data would provide a way to fulfill any software reliability/maturity requirements along with the foundations needed for modeling reliability or reliability growth.

The colors in the Relationship Matrix correspond to ability to automate the data. Automation will be enabled by scripts that grab the data already produced by the tools, store the data in an environment, pull the stored data to a dashboard environment and visualize the data on the dashboard. One script for each tool type will be needed. Our recommendations for priorities are as follows:

Scripts for the Lines of Code Tool to pull Kilo Source Lines of Code (KSLOCS)

Scripts for the Project tracking Tools

Scripts for the requirements tracking tool

Legend	Base Data - measure - one value no equation													Stakeholders					
Automated																			
Easy to Automate																			
More difficult to Automate																			
Most Difficult to Automate																			
Metrics of Interest																			
Metrics of not in the Scope of th project																			
Metrics	Actual KSLOCS	Actual Faults/Defects/Errors Autonomous	SW Version Number or timestamp	# of Software requirements	Test Completeness/Effectiveness	# Software LRUs/ CSCI/ Features	# Unit Tests Failures	# Verification Test Failures	Test Execution Start/End Timestamp	Software LRUs/ CSCI/ Features TimeStamp	Fault/Failure Discovery Timestamp	Fault Recovery Timestamp	Actual Complexity	Program Reliability Lead	Reliability Discipline Owner	Program Manager	Customer	Software development metrics owner	Process Group
Defects in Ops (near-Ops) per Deploy (DevOps)	x	x	x											x	x	x	x	x	x
Automated Test Requirements Coverage	x			x	x									x	x	x	x		x
Structural Code Coverage					x	x								x	x	x	x	x	
Pass/Fail Rates				x	x	x	x	x						x	x	x	x		x
Stress test coverage- testing critical faults					x			x	x					x	x	x	x		x
Integration Test results and coverage					x		x	x	x					x	x	x	x		x
Mean Time to Recover											x	x		x	x				
Feature Cycle Time						x				x				x	x				
Tools (sources of Base Data)																			
LCT/Sizer																			
Project Tracking Tool (**Jira, TFS**, RTC, iTracker, ...)																			
Requirements Tracking (**Doors, DNG**...)																			
Continous Integration Environment **Jenkins**																			
Static Analysis (Coverity, SonarQube, Clang-tidy, Find Bugs, CheckStyle, PMD, ...)																			
Unit Test (CPP Unit, Google Test, Boost Test, Junit, Py Nose2, ...)																			
Software Verification Test (Cucumber, Eggplant, Squish, ...)																			
Structural Coverage (Bullseye, gcov, jcov, Clover, Covertura, ...)																			
Code Review Tool (ReviewBoard, Gerrit, Collaborator, ...)																			
Autonomous Deployment (Artifactory, Nexus, ...)																			

Figure 1 Relationship Matrix

6 MANUAL PROCESSES

There are few manual steps that must happen to estimate software reliability in a DevOPs / Continuous Integration environment. During the proposal stage, in order to bid software estimate the KSLOCs for completion of the project, one must understand the Capability Maturity Model Integration (CMMI) level of your functional business area, and the actual software reliability achieved in similar programs. During software development trace your requirements to the automated tests performed. Understand the amount of KSLOCs each test will cover. Software safety must define which parts of the code are safety critical and their criticality level. A software failure analysis board must be created and software failures in field tests assigned. The software reliability engineer must define the reliability model for the program whether that be

The Planning Model based on Projection Methodology (PM2). The U.S. Army Materiel Systems Analysis Activity (AMSAA) AMSAA PM2 model, the exponential, the Weibull or some other model. The software reliability engineer in concert with the software development engineer must review each failure to see if it is a true failure of the code or a test failure. Defects would be counted when the engineer checks the code into the DevOps source code control tool.

7 SOFTWARE RELIABILITY REQUIREMENTS

Since DevOps can be deployed enterprise wide, it is important to understand how many programs across a company that have software reliability requirements, how each program is meeting those requirements, and if the programs are using any DevOps tools. Our recommendation is a survey of all reliability engineers and software engineer leads to understand the requirements and tools. A sample of pertinent survey questions is as follows:

1) Program Name
2) Point of Contact
3) Functional Organization
4) Program State – early, mid or late (after qualifying testing)
5) Does the program have software assessment guides and are they followed?
6) Does the program have a software reliability plan?
7) If there any mention of software reliability in the systems engineering management plan or software development plan?
8) Does the program have software reliability requirements?
    a. If yes, what software reliability tools are being used?
    b. Is Software reliability metrics collected?
    c. Is Software defect count collected?
    d. Where is the software reliability data stored and in what format?
9) Is there a Software configuration control board and what tracking tool does it use?
10) What is currently being automated in a continuous integration environments utilized by the program?
11) Is there any part of the data gathering / data analysis process that is causing a significant amount of labor and time?
12) Types of software metrics collected:
    a. Source Lines of Code (SLOC)

b. Software Complexity
c. Static / Dynamic analysis results
d. Unit test results and coverage
e. Integration Test results and coverage
f. Stress Test Results and coverage
g. Defect Density
h. CMMI Level
i. Technology Readiness Level

8 CONCLUSIONS

The DevOps/CI environment allows programs to track, measure and increase reliability across software programs. .. The base data exists in these environments, and its collection can be automated to reduce the man hours needed to estimate software reliability. By combining the existing base data and utilizing the software reliability model approach, the ability to quantify software reliability and return on investment can be fully realized.

9 RECOMMENDATIONS

This white paper only concentrated on automating software reliability within the highly automated framework of DevOps. Within that constraint, depending on the level of DevOps employed by each program, base data collection is automatic. The priorities recommendations for scripts to automate tools in the Relationship Matrix correspond to our understanding that all programs using some form of DevOps are using a Line Count Sizer Tool therefore writing a script to gather that data should be a high priority. If most of your programs moving to DevoOps a project tracking tools to track Agile software development. Then, the next logical step is automating the pull of data from the agile software development requirements tracking tools. Lastly, automating all the data from continuous integration environment. The scripts to automate any of these processes given the DevOps environment and a storage hardware area are all of medium difficulty. Once written for one program, they could easily be reused or adapted for other users. The programs would need a dedicated storage server for the data to be collected as most programs employing DevOps continually overwrite the data instead of storing it for metrics capture.

Most programs and companies defined defect containment on non DevOps/CI projects. Their definition of not declaring a defect unless it crosses an increment boundary works well for that environment but not for the DevOps/CI environment. Our recommendation on when to start counting defects should be when the software enters DevOps (it gets checked into source code control). This type of environment takes more of an earned value approach to defects. The program plans number of defects per KSLOC, the actual defects are measured. The software reliability model measures how well the actual program execution performed against the predicted values, how effective are the software fixes, and how well the program understood the software project. If the program performs to plan, then the program understood the software project and there is a very low risk of failure.

Educating the customer is critical for contextual understanding of a defect / error/ fault in the DevOps/CI environment. A common understanding between program and customer that software reliability is more difficult to estimate than hardware reliability is crucial.

All definitions of errors/ faults / defect/ units/ features/ CSCI/ LRU need to be clarified at the program level. Normally, the customer recognizes the IEEE standards and any deviation from that standard needs customer education and agreement.

REFERENCES

1. IEEE Recommended Practice on Software Reliability, IEEE 1633-2016
2. Systems and software engineering - Vocabulary (ISO/IEC/IEEE 24765-2010)
3. Raytheon DevOps Measurement Workshop
4. MM Lehmam LA Belady; Program Evolution - Processes of Software Change, 1985

BIOGRAPHIES

Mary “Lisa” Williams Bates
Raytheon Technologies
315 Bob Heath Drive
Huntsville, AL 35804

Mary.e.bates@rtx.com

Mary “Lisa” Williams Bates is a Principal Systems Engineer with Raytheon Missiles & Defense. She holds a Masters of Operations Research, Masters of Operations Research, Reliability Eng. & Engineering Mgmt, from the University of Alabama in Huntsville. Currently enrolled in Mississippi State University as a doctoral candidate in Computational Engineering. Currently working on the following programs: A&E Pillar Splunk Lead, SEATN Facilitator, CBM+ IRAD Investigator.

Dr. Enrique I. Oviedo
Raytheon Technologies
3350 E. Hemisphere Loop
Tucson, AZ 85706

Enrique.I.Oviedo@rtx.com

Enrique I. Oviedo is a Sr. Principal Software Engineer with Raytheon Missiles and Defense. He holds a Ph.D. in Computer Science, and a masters degree in Mechanical Engineering. Currently, he is the Process Lead. . He has led Software Engineering Process Groups and has presented several papers on lean and agile software development in RTX symposia.