Real Simple Reliable Software - Perfectly Suited for DevSecOps

© Ann Marie Neufelder, SoftRel, LLC, 2020

ann.neufelder@missionreadysoftware.com

http://www.missionreadysoftware.com

Mission Ready Software Founder Ann Marie Neufelder

- Chairperson of IEEE 1633 Recommended Practices for Software Reliability Working Group (2016 edition)
- 38 years of SW engineering and SW reliability experience
- Authored NASA's Software FMEA and FTA training webinar
- Has world's largest database of software failure events and root causes (almost 1 million)
- Has world's largest database of actual software reliability data (700 factors by hundreds of organizations)
- Authored Intel's Vendor Assessment
- Has taught Software Reliability to more than 5000 engineers
- Co-authored USAF Rome Laboratory "System and Software Reliability Assurance Notebook", Boeing Corp.
- Authored "Ensuring Software Reliability", Marcel-Dekker, 1993.
- Authored "Effective Application of Software Failure Modes Effects Analysis", published for CSIAC, 2014.
- U.S. Patent 5,374,731 for a predictive model

Agenda

- Overview
  - Reliable software– what is it and how does it differ from quality?
  - Software failures that have affected mission critical programs
  - Key metrics indicating reliable software
  - Reliability statistics for world class, mediocre and distressed software/firmware programs
  - Relationship between HW and SW reliability
- Root causes for nearly every major software failure and how they can be predicted
  Real simple ways to allocate system reliability objectives to software
  - Past history, R&D $, achievable failure rates
- Real simple ways to establish an early prediction for software
  - Ranked list of factors quantitatively proven to affect software reliability
  - What every failed and successful software project has in common myths - a few popular but overrated factors
- Real simple ways to track software reliability growth during testing
- How software reliability fits into the Agile/Scrum project execution and DevSecOps

Overview

- Reliable software– what is it and how does it differ from quality?
- Software failures that have affected mission critical programs
- Why tasks for reliable software aren't necessarily part of software quality
- How the underlying failure modes can be predicted before they cause a failure in operation
- Key metrics indicating reliable software
- Reliability statistics for world class, mediocre and distressed software/firmware programs
- Relationship between HW and SW reliability

This is why you are reading this presentation

Software/firmware is increasing in size at a hyper exponential rate.

Only two decades ago size was measured in 1000s of source lines of code. Millions of SLOC of relatively rare.

Now size is routinely measured in multi-millions.

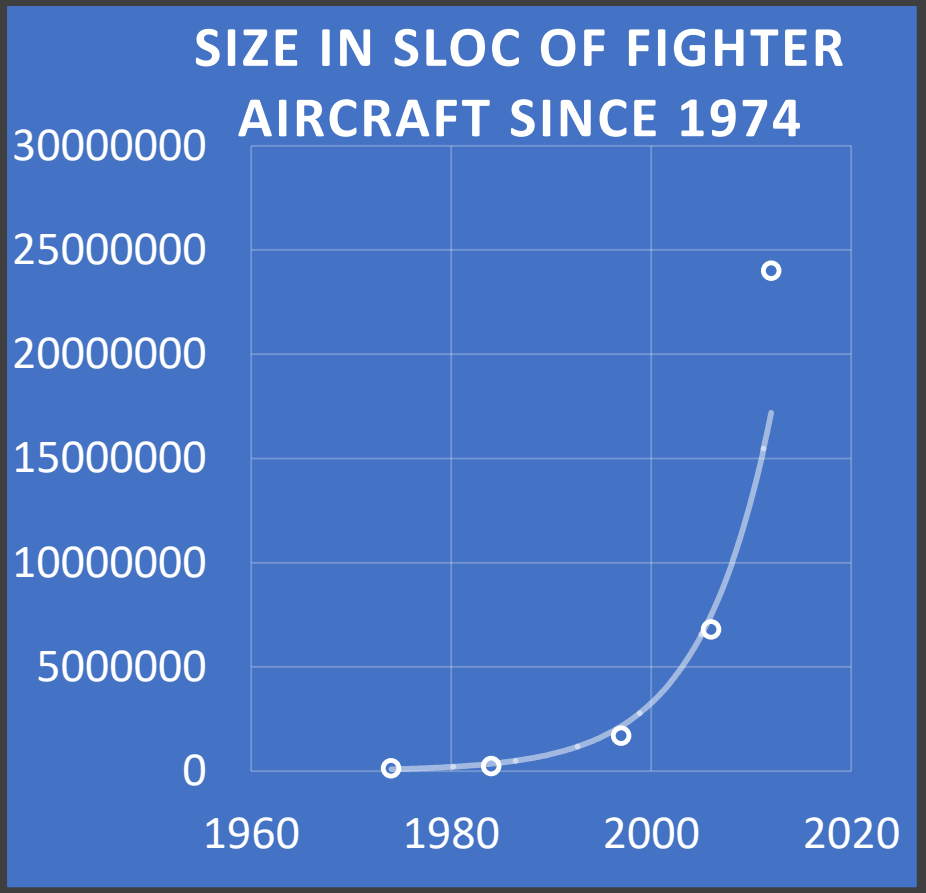
Compare your smart phone, HVAC, lighting, appliances to only 10 years ago. Less hardware, more software.

The increase in size of F16A to F35 software is just one example[1]

With increased size comes increased complexity and increased failures due to software as discussed in this class.

Things are not getting better as long as software size is increasing.

Unfortunately, the methods for developing software haven't improved at the same rate. You will learn more about this in this class.



Software failures that have affected mission critical programs

- Fratricide due to software failures is no longer theoretical.
  - IFF and target ID failures in Patriot March 2003 which shot down British Tornado Aircraft [1]
  - SCUD missile attack in 1991 which killed 27 soldiers[2]
  - AFATDS friendly fire Fort Drum in 2002[3]
- Lost missions due to software failures is no longer theoretical
  - F22 international date line defect [4] – Loss of entire mission and nearly lost 11 aircraft
  - As per the Joint System Safety Engineering Handbook Appendix F[5]
    - Missile Launch Timing Error Causes Hang-Fire
    - Reused Software Causes Flight Controls to Shut Down
    - Flight Controls Fail at Supersonic Transition
    - Incorrect Missile Firing from Invalid Setup Sequence
    - Operator's Choice of Weapon Release Overridden by Software Control
- Cancelled projects due to software failures and gross underestimates of software size
  - Future Combat System (FCS) originally planned to have 33.7M lines of code. Then it was 63.8 M and then 114 M. *It would have been significantly bigger than any other system on earth.[6]*
  - F35 software was grossly underestimated and was the cause of several GAO investigations [7]
  - As we will see in upcoming modules, gross underestimates of size are the leading factor in unreliable software whether the program is cancelled or not

Failures aren't limited to only shutdown or total failures

- This is a common myth which is not supported in any IEEE standard.
- Reliability engineers assume that the software shutting down is equivalent to wear-out for hardware.
- The definition of failure has never been from the root cause viewpoint – only the effects viewpoint.
- Example: A missile launcher is required to launch when a) launcher is operational b) there's no abort c) the launch window can be met

Failure	Root causes (defects)
Inadvertent missile launch	Prohibited state transitions allowed by the code
Launch is executed despite an abort	Faulty logic (failed to detect abort) or timing (detected abort too late)
Launch is executed when launcher is not operational	Missing or faulty logic for all faults that cause launcher to be operational.
Launch is executed when window cannot be met	Faulty launch calculator algorithm or faulty logic in detecting missed window
The system isn't in launch ready mode for specified <x> hours per day	Initialization code takes too long to execute on startup
Launch isn't executed when commanded	Faulty logic – all conditions are met but false negative logic
Target is missed	Faulty timing - missile is launched too early or too late
	Faulty launch calculator algorithm
	Faulty data – unhandled overflow or underflows with data
	Faulty timing – built up timing inaccuracies

FAQ: What's the relationship between SRE and SEI CMMi?

- The facts[1] show that all organizations with highly reliable software have defined processes
- However, the facts also show that organizations with defined processes aren't *guaranteed* to have highly reliable software
- Software processes such as ASPICE, SEI CMMi, etc. provide a *required foundation* for reliable software but are proven to be insufficient for highly reliable software. Reliable software also requires:
  - People who understand the industry and product
  - Execution – smaller cycles and team sizes
  - Avoidance of known risks
  - Techniques such as model based specifications
  - The maturity of the design itself
- We have 28 years of data[1] to show that
  - SEI CMMi level 2 and 3 organizations have fewer defects found in operation than SEI CMMi level 1
  - However, SEI CMMi level 4 and 5 organizations do not have fewer defects than SEI CMMi level 3.
  - Multiple CMMi level 4 and 5 organizations had failed software projects due to overconfidence.

SRE methods include this tasks

…which aren't typically conducted by software QA people

Allocate a portion of the system reliability to software

- Past history, R&D dollars, achievable failure rates

Predict these things before the code is even written

- Any reliability figure of merit
- Strengths and gaps that directly translate to more or fewer defects
- Release maturity- successful, mediocre or distressed
- ROI of changing a few specific development factors

Estimate these things during testing

- Progress towards a system reliability objective
- Test hours needed to reach specific level of maturity
- Maintenance staffing required to avoid defect pileup which causes the next release to be late

Identify these during development and test

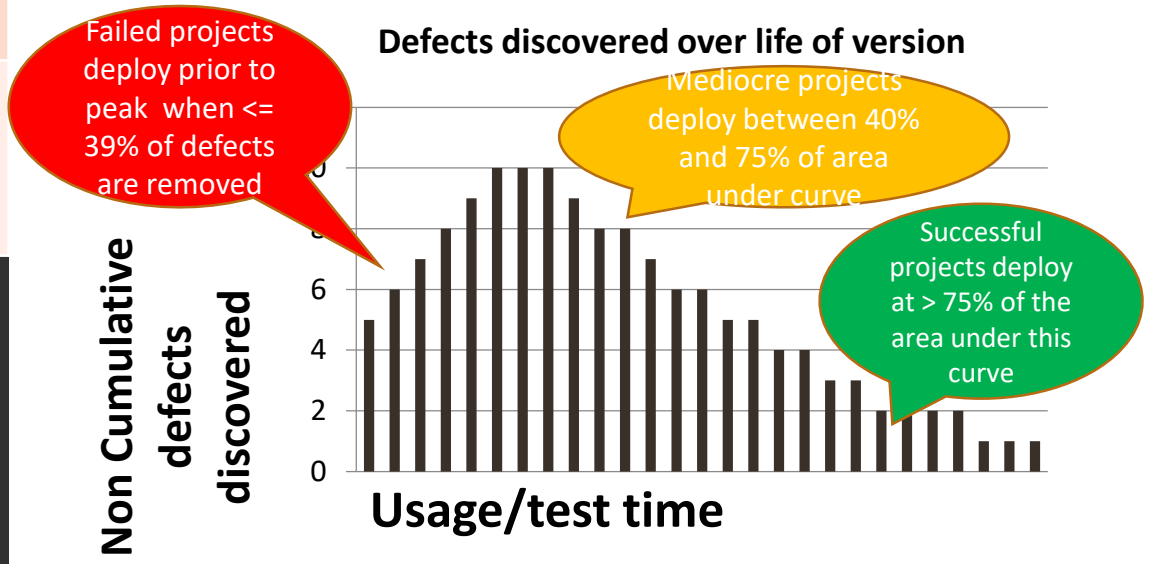
- Failure modes in plain sight in the specifications
- Testing for failures instead of success

All SRE methods apply to both SW and FW

Key metrics indicating highly mature and reliable software

Metric	World Class	Mediocre	Distressed
Fault rate trend	Steadily decreasing	Peaking or recently peaked	Increasing
Percentage of defects identified prior to deployment	>=75%	40-74%	<=39%

- **It's been proven since the 1970s that software fault rate increases, peaks and then decreases prior to maturity**
- Maturity level at deployment separates the world class from the distressed
  - Increasing fault rate– the customers will see it as a failed product in 100% of all cases
  - Fault rate barely decreasing- customers will be unhappy with it
  - Fault rate is steadily decreasing – customers won't notice the SW *which is exactly what you want*
- With agile or incremental development there are multiple peaks until the final burn down of defects
- **None of the distressed and most of the mediocre software projects were tracking their faults or fault rate maturity prior to deployment.**
- **We cover how to track fault rate during testing in the Integrating Software and Hardware Reliability Class.**

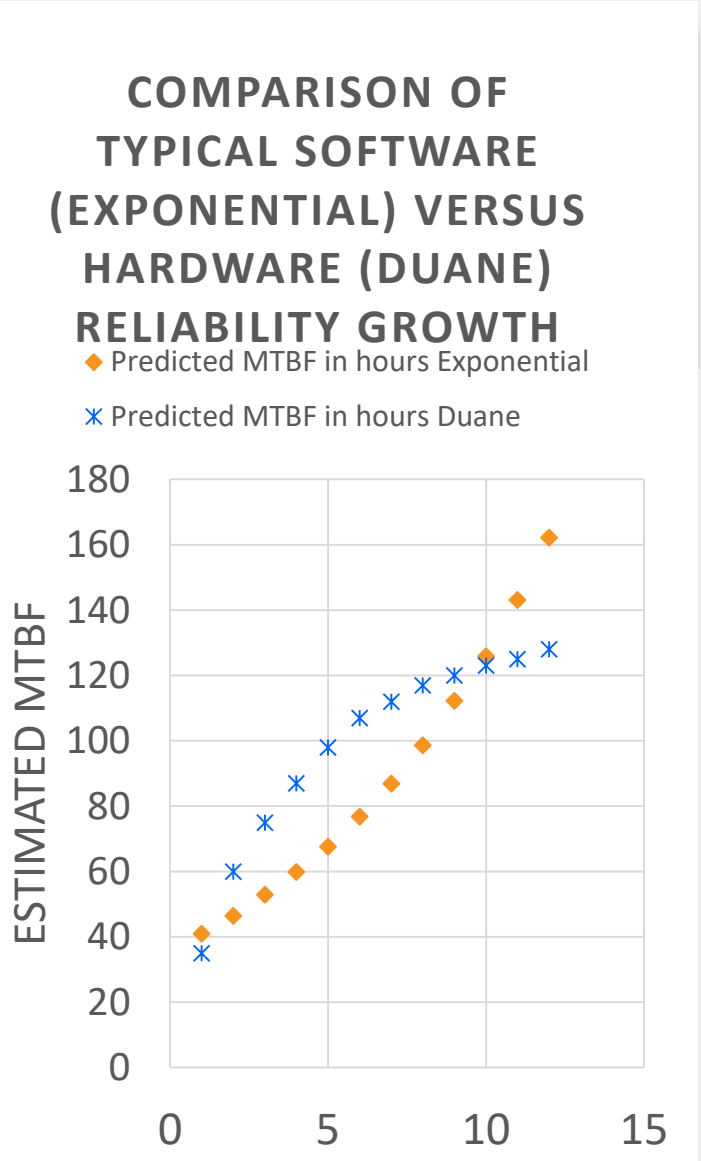


SW versus HW MTBF

- Hardware MTBF represents mean time between the same type of failure in the same replaceable LRU.
- Example: If a lightbulb has an MTBF of 7 years, you may be replacing it in 7 years.
- Software doesn't wear out so software MTBF presents the mean time between *any* failure in the *entire* software program
  - Many different types of defects in many lines of code contribute to the MTBF
  - Replacing the software has no effect unless the defect is removed and the software LRU rebuilt
  - If we knew where they were in the code, we could remove them so MTBF is measuring time until someone discovers one we don't know about
  - Typically it can take 2-8 years for every defect in the software to be discovered in operation
- However, it's been shown that there is a predictable trend for the software failures as shown on next slide

Software/Hardware Reliability

HW reliability prediction	Software Reliability Prediction
Mean Time To Repair	Mean Time To SoftWare Restore
Burn in phase of Bathtub curve	Reliability growth via operation and defect correction
Wear out phase of Bathtub curve	1. Obsoletion – Target hardware or environment has changed and software must change with it. Obsoletion can happen overnight. 2. Software design and code becomes too unstructured to make any changes to (can take 10+ years).
Reliability growth	Can be more limited for SW than for HW
FMEA	SFMEA on requirements, interfaces, detailed design/code, usability, maintenance actions
FTA	Software is included on the system FTA



Software reliability growth versus hardware

- Software reliability growth typically starts off slow. If/when defects are removed then it increases.
  - No wear out encountered, however, once new features are injected the growth resets.
  - If environment become obsolete software can become unusable quickly.
- Hardware reliability typically starts out better and then encounters wear-out.

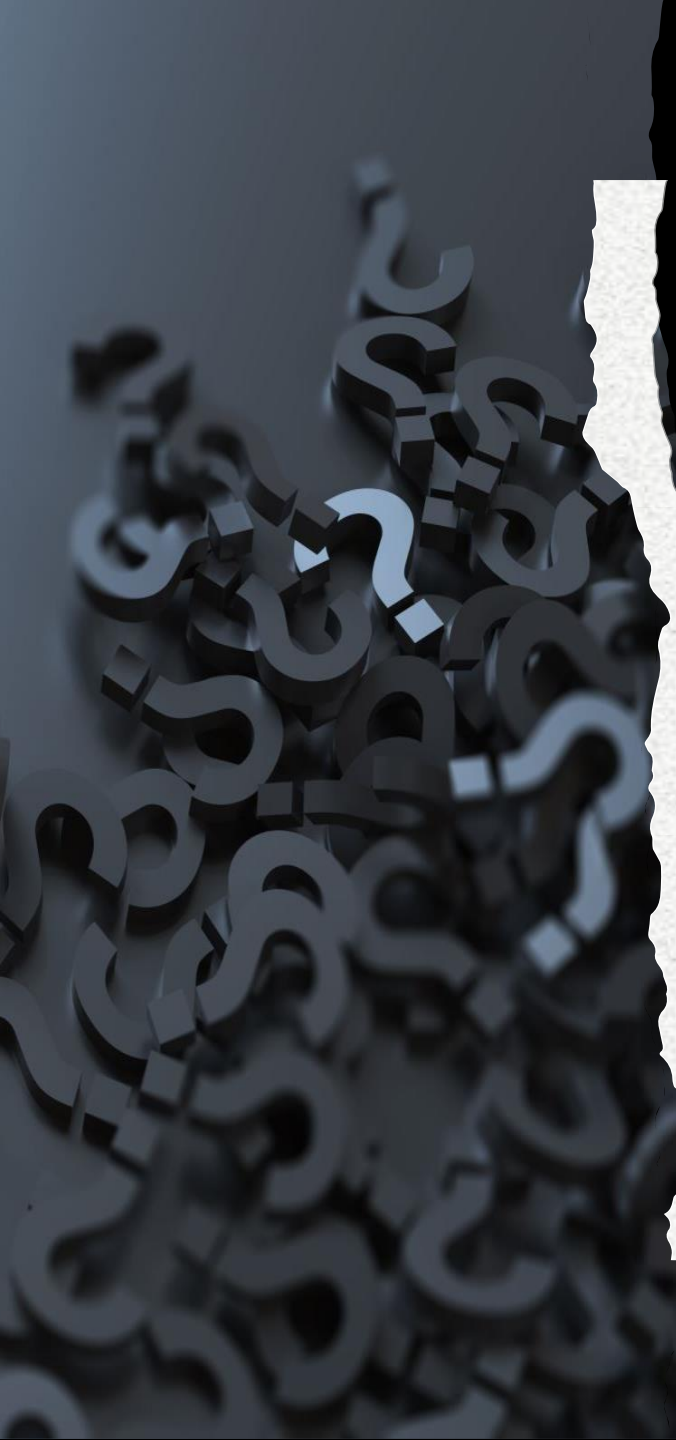
The root causes behind virtually every major software failure

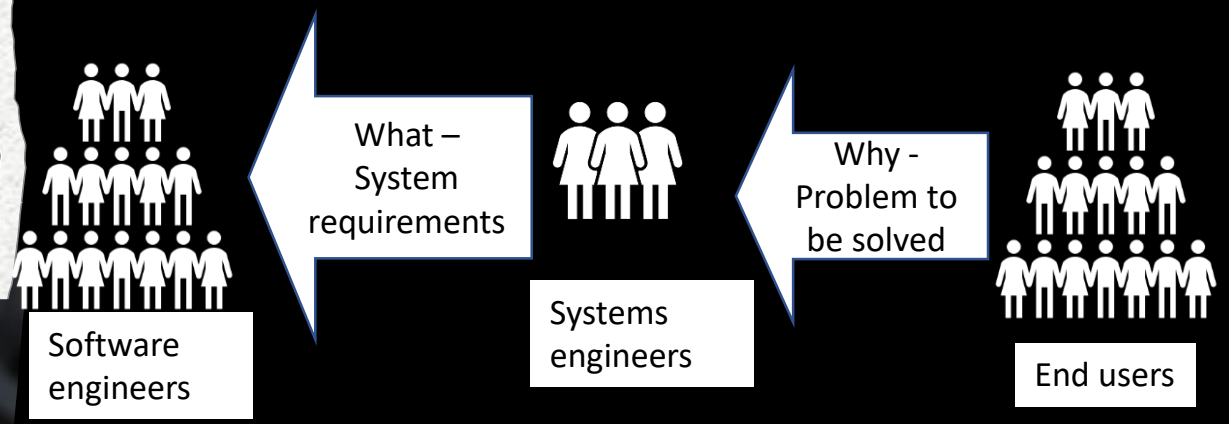
And how you can predict them a priori

Where software failures originate

At highest level software failures are caused by

- Software specifications are missing crucially important details *(research by multiple organizations has found this to be as much as 50%)*
- Software specifications are inherently faulty
- Software code isn't written to the specifications or misses some specification
- **The underlying root cause for many software failures is that the software engineers not fully understanding the "whys" or the problem to be solved.**





Software failure modes are very predictable –

Just a short list of the software root causes that keep repeating themselves

- **Faulty error handling** – Apollo 11 lunar landing, ARIANE5, Quantas flight 72, Solar Heliospheric Observatory spacecraft, Denver Airport, NASA Spirit Rover (too many files on drive not detected), F22 International Dateline
- **Faulty data definition** - Ariane5 explosion 16/64 bit mismatch, Mars Climate Orbiter Metric/English mismatch, Mars Global Surveyor, 1985 SDIO mismatch, TITANIV wrong constant defined, Flight Controls Shut Down, Incorrect Missile Firing from Invalid Setup Sequence
- **Fault logic**– AT&T Mid Atlantic outage in 1991
- **Timing** - SCUD missile attack Patriot missile system, 2003 Northeast blackout, Therac 25, Missile Launch Timing Error Causes Hang-Fire
- **Faulty state transitions -**Incorrect Missile Firing from Invalid Setup Sequence
- **Faulty algorithms -** Flight Controls Fail at Supersonic Transition, 2003 Inadvertent shooting of British Aircraft
- **Faulty functionality –** Operator's Choice of Weapon Release Overridden by Software Control
- **Peak load conditions** - Affordable Health Care site launch, 2020 Iowa Caucus Primary
- **Faulty usability**
  - **Too easy for humans to make mistakes** – AFATDS friendly fire, PANAMA city over-radiation
  - **Insufficient positive feedback of safety and mission critical commands** – 2007 GE over-radiation

Lesson to be learned – history keeps repeating itself because everyone thinks it won't happen to them

Organizational mistakes that lead to failure modes are also very predictable

Each of the below are easily predictable well in advance of a loss of mission due to software

- Faulty functionality - Faulty assumption that the software engineer understands the mission and environment. Failing to test the software in an end to end environment
- Faulty state management - Insufficient level of detail in the state design and specifications
- Faulty timing – failing to design the timing and scheduling
- Faulty logic – failing to use simple diagrams such as logic or flow diagrams in design.
- Faulty algorithms - Failing to consider the entire range of possibilities for the system.
- Faulty data definition – failing to define default values, units of measure, scale, size, type for every data element
- Faulty error handling - Failing to design for known failures in HW, computations, power, communications, File I/O, etc.
- Faulty endurance - Failing to test the software over a passage of time and under realistic conditions
- Faulty peak loading – Failing to consider maximum users, operations, etc.
- Faulty usability – Failing to think about the environment around the end user and the job that must be performed by the user

What you do is rocket science. Predicting the weak link that leads to certain failure modes is not.

When properly applied in a timely manner SRE has made a difference

Every one of the root causes discussed on previous pages is detectable during development - *but only if someone is looking for them*

Every one of the process root causes discussed on previous pages is detectable during development - *but only if someone is looking for them*

Since 1962 the same software root causes have resulted in thousands of world events in space, defense, medical devices, energy, etc.

Yet software engineering continues to overlook the same root causes

Real simple Methods for Establishing an Allocation for Software

All of these have been proven to be more accurate than subject matter expert guessing

Simple and accurate methods for allocation

Method	Description
Past history	Compute relative portion of SW versus HW failures from a past similar system
R&D $	Compute relative portion of R&D $ dedicated to software development
Achievable failure rates	Use prediction models to determine failure rate for HW, SW. The predicted values for each determine their allocation.

The first method is very accurate if the past history is recent and is calibrated for changes in technology. As a rule, software grows 10-12% per year. So, history data should be calibrated to assume that the software portion is growing 10-12% per year.

Real example: An engineering company produced a system in 2015. Of all of the deployed failures, 25% were due to software. In 2017 they were deploying a similar system. Since historical data was 2 years old, 25% is adjusted by 10-12 % per year. So, the prediction is between 30.25% and 31.36%. When the equipment was deployed in 2019 - the actual portion of failures due to software was 33%. Much more accurate than the 5% estimated by subject matter experts.

Simple and accurate methods for allocation

Method	Description
Past history	Compute relative portion of SW versus HW failures from a past similar system
R&D $	Compute relative portion of R&D $ dedicated to software development
Achievable failure rates	Use prediction models to determine failure rate for HW, SW. The predicted values for each determine their allocation.

If no historical data available then R&D dollars or achievable failure rates can be used.

Example #1: The R&D budget for software is 100 million. The R&D budget for hardware design is 200 million. Hence software gets 33% of the allocation and hardware gets 67% of the allocation.

Example #2: The software MTBF is predicted to be 500 hours. The hardware MTBF is predicted to be 1000 hours. The software is then allocated 33% of the objective and the hardware is allocated 67%.

Either of the above is more accurate than “subject matter expertise”

Predicting reliable figures of merit for software before testing

What you do is rocket science. Predicting reliable software isn't. Prediction models for software have been around since 1980s. The problem is that reliability engineers don't want to use them.

Just a few decades ago weather predictions had 50% accuracy to the day.

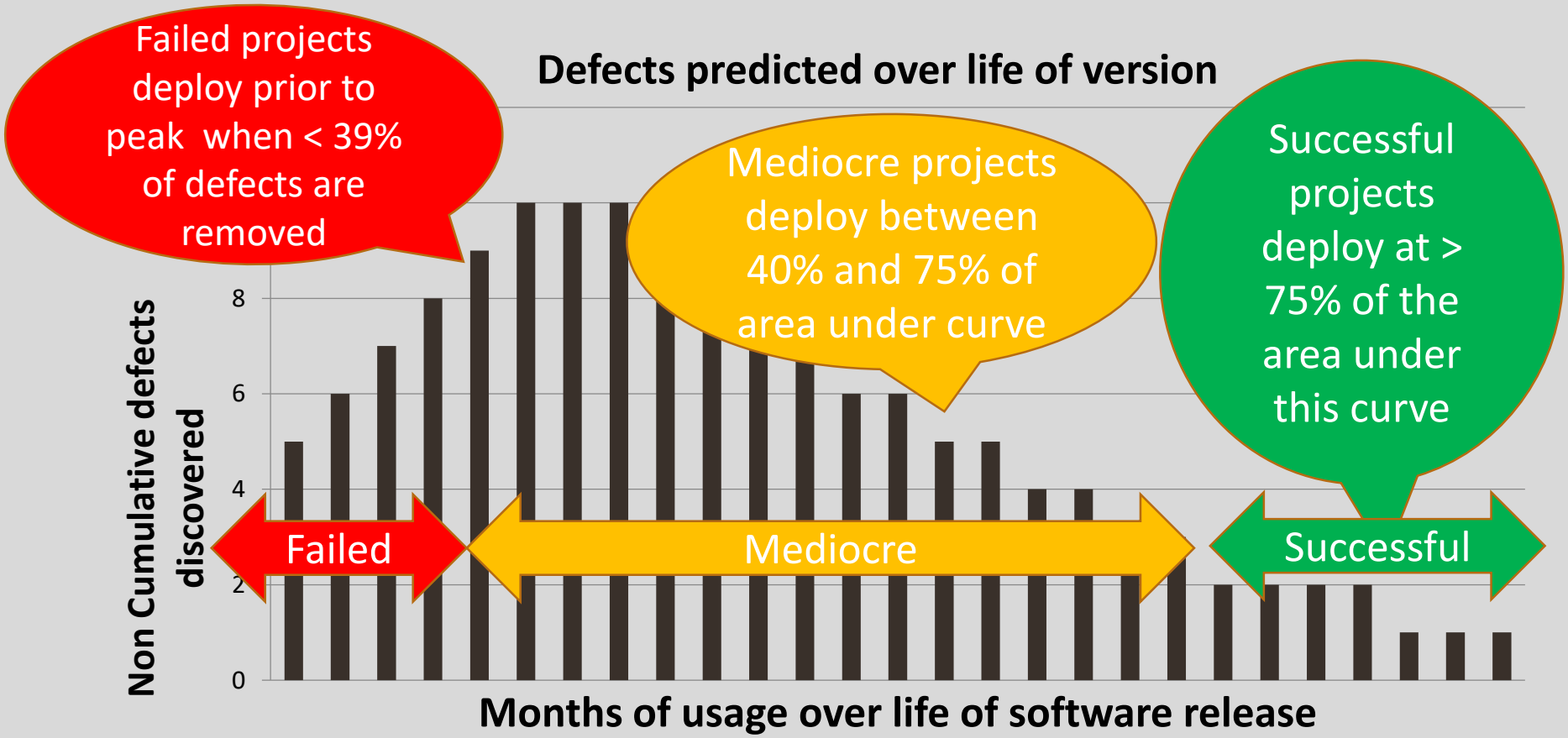
Now they are very accurate to the hour.

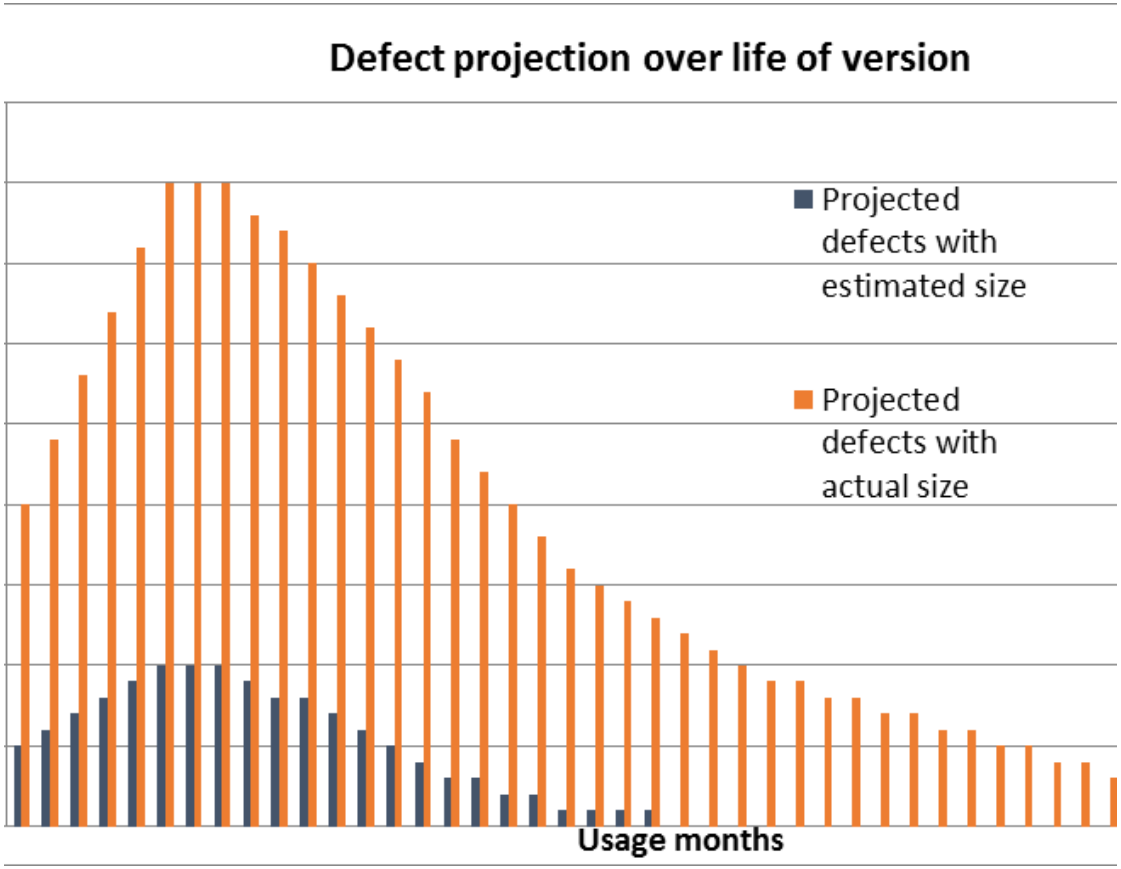
If you collect enough data, you can predict anything.

SRE predicts defect discovery profile/project outcome

Early in program you can manage defects, size and on time delivery and identify a failed project

During testing you can measure the actual faults to determine when to release



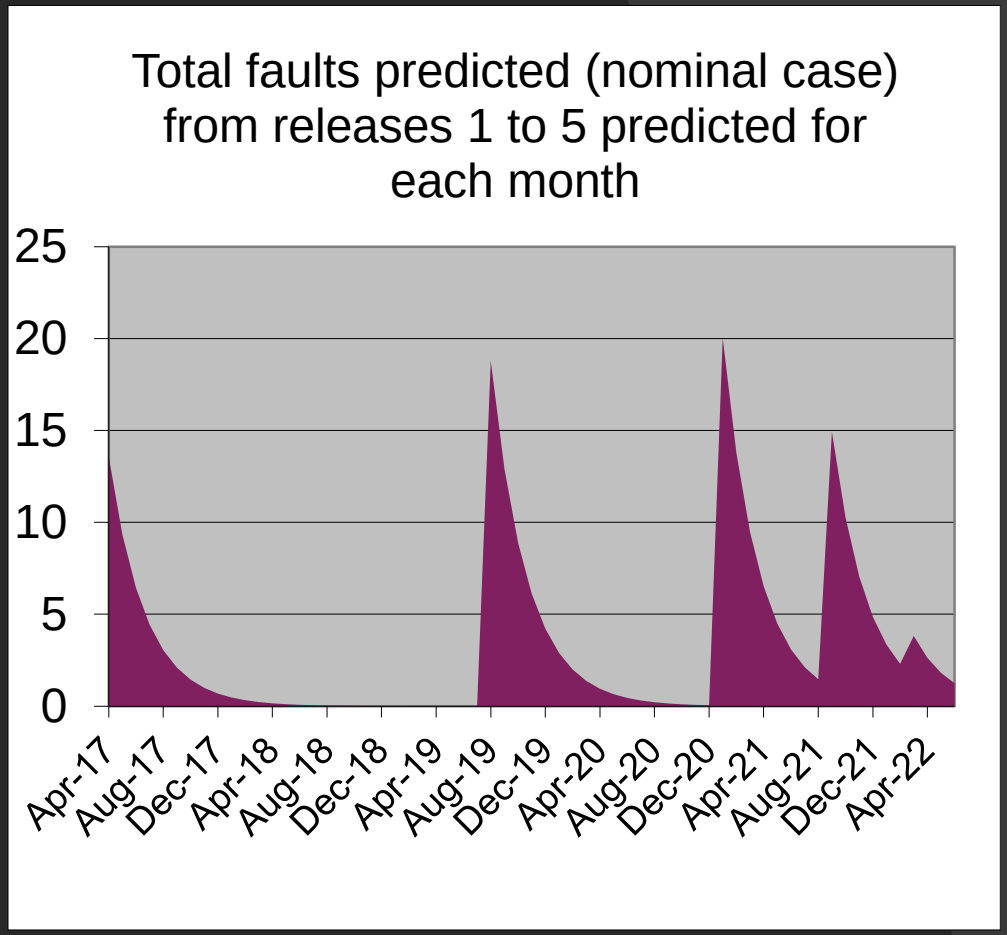


SW releases are often late and unreliable when SRE isn't used because of underestimates of scope and defect potential

- **No one sets out to release half baked software**
- It happens when SRE metrics aren't used early in project when there is time to do something about it
- Team is expecting a small number of defects when the larger number could have been predicted and managed before code was even written

Real example of how predictions detected future defect pileup on a large DoD program

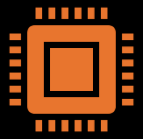
- In this real example, “kicking the can” predicted to cause defect pileup
- Releases are too far apart initially and too close together later on
- SRE predictions allowed for leveling of features before code was even written



Real examples of how predictions were significantly closer to SME guess for MTBF

- #1 - On a real DoD program - subject matter expert guess for MTBF was 500,000 hours.
  - Actual MTBF upon initial deployment was in single digits
  - Predicted MTBF was in low double digits.
  - Predictive model was 5 orders of magnitude more accurate than expert guess
- #2 – On real DoD program - subject matter expert guess for MTBF was so high as to be virtually infinity.
  - Actual software MTBF upon initial deployment was in double digits
  - Predicted software MTBF was low triple digits
  - Predictive model was 5 orders of magnitude more accurate than expert guess
- Hardware reliability prediction is rarely pinpoint accurate. Software reliability models don't have to be pinpoint accurate to be useful. In both of the above cases, the prediction models accurately predicted that system objective would not be met.

Software reliability prediction



Means to predict software defects, failure rate, etc. before the code is even finished

Nearly all software reliability prediction models are based on an assessment or survey

Based entirely on the factors that have quantitatively been correlated to reduced operational defects

Type of factor	Number /% of characteristics in this category	Examples of characteristics in this category
Product	50 – (10%)	Size, complexity, whether the design is object oriented, whether the requirements are consistent, code that is old and fragile, etc.
Product risks	12 – (2%)	Risks imposed by end users, government regulations, customers, product maturity, etc.
People	38 – (7%)	Turnover, geographical location, amount of noise in work area, number of years of experience in the applicable industry, number of software people, ratio of software developers to testers, etc.
Process	121 – (23%)	Procedures, compliance, exit criteria, standards, etc.
Technique	302 – (58%)	The specific methods, approaches and tools that are used to develop the software. Example: Using a SFMEA to help identify the exceptions that should be designed and coded.

Static analysis tools measure these

These are often overlooked

SEI CMMi and ASPICE assess this

These are often overlooked

Factors that have been mathematically proven to be related to software reliability

USAF Rome Laboratories developed first prediction model in 1987. It was based on these factors.

A few more have been developed since then.

Facts don't lie.

All predictive models agree that how the software is developed is a good predictor for it's ultimate reliability.

Category	Examples
Decomposition	• Code a little, test a little philosophy. • Release development/test time < 18 months long and preferably <12 months. • Each developer has a schedule that is granular to day or week.
Visualization with pictures and tables	A picture is worth 1000 words. Specifications with diagrams/pictures/tables are associated with fewer defects than text.
Requirements focus	Developing requirements that aren't missing crucially important details
Testing focus/rigor	Explicitly testing the requirements, design, stresses, lines of code, operational profile
Unit testing focus	Unit testing by every software engineer is mandatary and as per a defined template. Branch coverage tools and metrics.
Defect reduction techniques	Software fault trees, software FMEA, etc.
Design focus	Designing states, sequences, timing, logic, algorithms, error handling **before coding**
Regular monitoring of the software engineers	Monitoring software progress daily or weekly, identifying risks early, etc.
Planning ahead	Planning the scope, personnel, equipment, risks before they become problematic, planning the timing of the tasks

Techniques that effect software reliability that are often overlooked

Visualization ranked in top 5 and is also one of cheapest ways to reduce software failures

Visualization is augmenting words with pictures, diagrams, tables, etc.

State, Logic, Timing, Data flow, Function Flow

Because of requirements management tools such as DOORs, software organizations employ “text” based requirements.

Organizations that draw pictures or tables as informative references have fewer defects in testing and operation than those who don’t

Absence of diagrams can be used to predict specific failure modes such as faulty state management, faulty timing, faulty error handling, faulty data handling, etc.

Mission Ready Software Statistics for various SRE capabilities

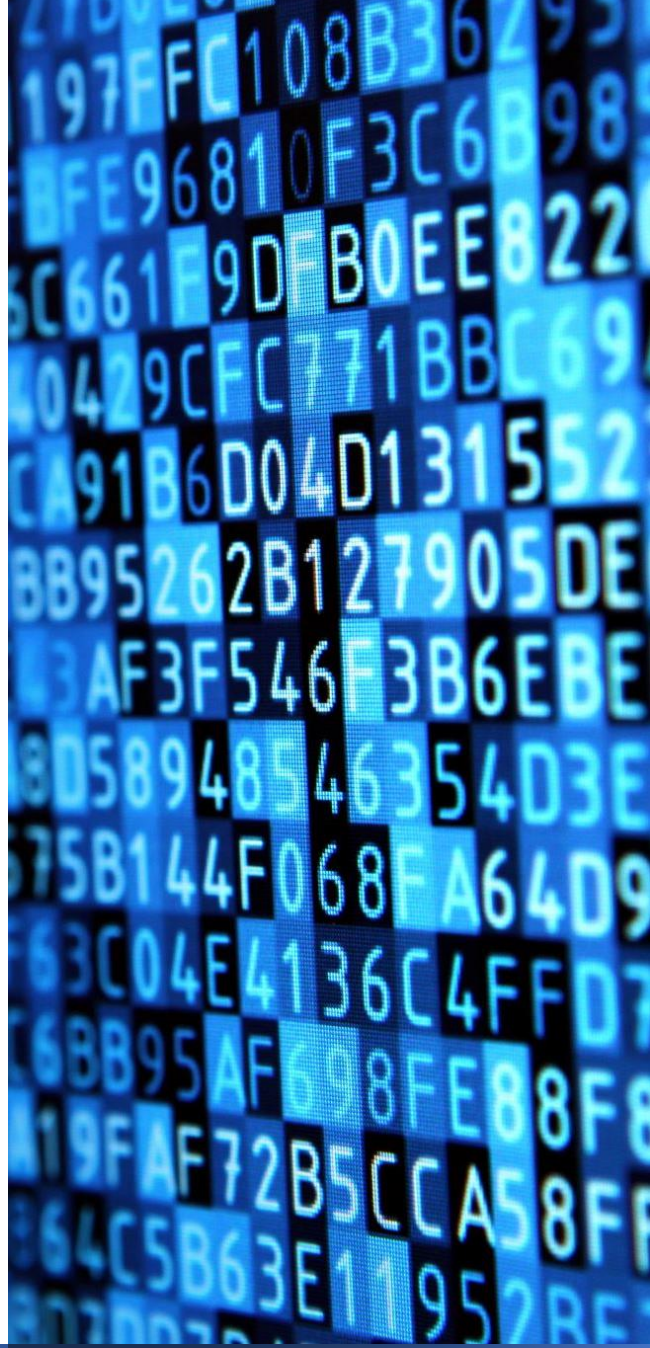
Cluster	Outcome	Defect metrics			Late deliveries (as per SW estimates)	
		Average defects per 1000 source lines of code	% defects removed prior to release	Fault rate	Prob (late)	How much project is late by as % of schedule
3%	World Class	.0269	>75%	Steadily decreasing	40	12
10%	Successful	.0644			20	25
25%	Above average	.111	40-75%	Recently peaked or recently decreasing	17	25
50%	Average	.239			34	37
75%	Below average	.647			85	125
90%	Impaired	1.119	<40%	Increasing or peaking	67	67
97%	Distressed	2.402			83	75

In the IEEE 1633 class, learn how to predict one of these outcomes

- All 100+ software/firmware projects in Mission Ready Software database fall into one of seven clusters
- Organizations with lowest deployed defect density were also late less often and by a smaller amount
- SRE for any given project can be predicted by answering a simple survey

These are things every world class/successful software program has in common in the Mission Ready Software database

- They have at least one software engineer who understand the product and the industry
- They don't try to tackle too many risky things in the same release (see next page for list of risky things)
- They have release cycles < 18 months
- They always deploy the software with a decreasing failure rate
- They test it from mission standpoint and from a design standpoint
- They have written software specifications, design, test procedures which they kept up to date as things changed
- They kept track of the software defects and control over the source code
- They aren't overly confident about their ability to develop the software
- They track progress against schedule from the beginning of the release



These are things every impaired/distress software program in the Mission Ready Software database have in common

- They don't track progress against schedule until they are already late
- They ship the software with an increasing failure rate
- They have < 10% of total effort in testing
- They grossly underestimate the size of the software
- They try to conquer too many learning curves in the same release instead of spreading them out
  - Software people who don't have industry experience
  - Sudden significant turnover in software engineers
  - Software technology they've never used before
  - Brand new product
  - Significantly changed interfaces
  - Software development tools they've never used
  - An obsolete development environment
- They have many excuses for cutting corners but are still late anyhow
- Track record of grossly underestimating the size of the software
- Track record of faulty assumptions that reused code doesn't need to be tested or failing to reuse code when it makes

Ball park method for predicting software MTBF based on effort size

Size range in Man Years	Worst MTBF at initial delivery	Average MTBF at delivery	Best MTBF at initial delivery	Worst MTBF after 1 year	Average MTBF after 1 year	Best MTBF after 1 year
1-9 MY	117	669	10368	469	2640	40927
10-49 MY	23	134	2074	92	529	8485
50-99 MY	8	45	691	31	176	2768
100-149 MY	5	27	415	18	106	1637
150-200 MY	3	19	296	13	75	1169
200 MY+	3	15	230	10	59	909

At acceptance/delivery

After 1 Year of defect removal and usage with no new features introduced

MTBEFF historically 2.5 times higher than MTBF
MTBSA historically 11.1 times higher than MTBF
High risk programs (See page 33) use the worst case column.
Low risk programs (See page 32) use the best case column.

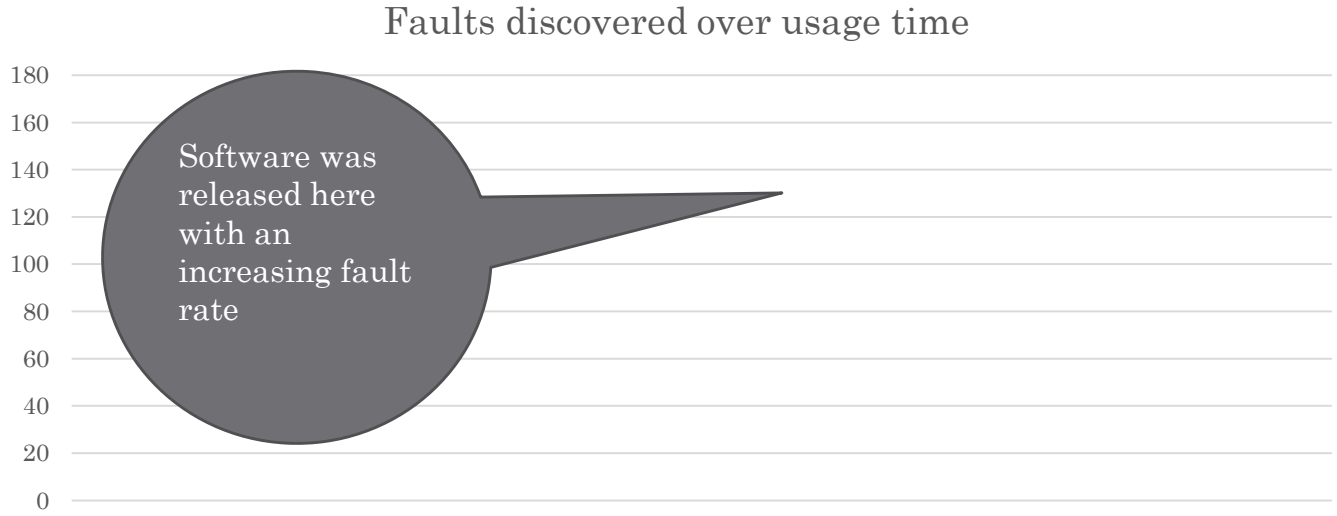
Overview of software reliability growth models

Once the software is testable these models can be used to forecast future reliability

Any simple tool, like Excel or JMP, can be used to track the reliability growth.

Lessons learned from a real DoD program in which SWRG models were not used

This is the fault rate from a distressed DoD software program



The contractor released the software to operational deployment before the fault rate peaked.

That's because no one was trending the fault rate.

More than 800 software failures were discovered by DoD after deployment.

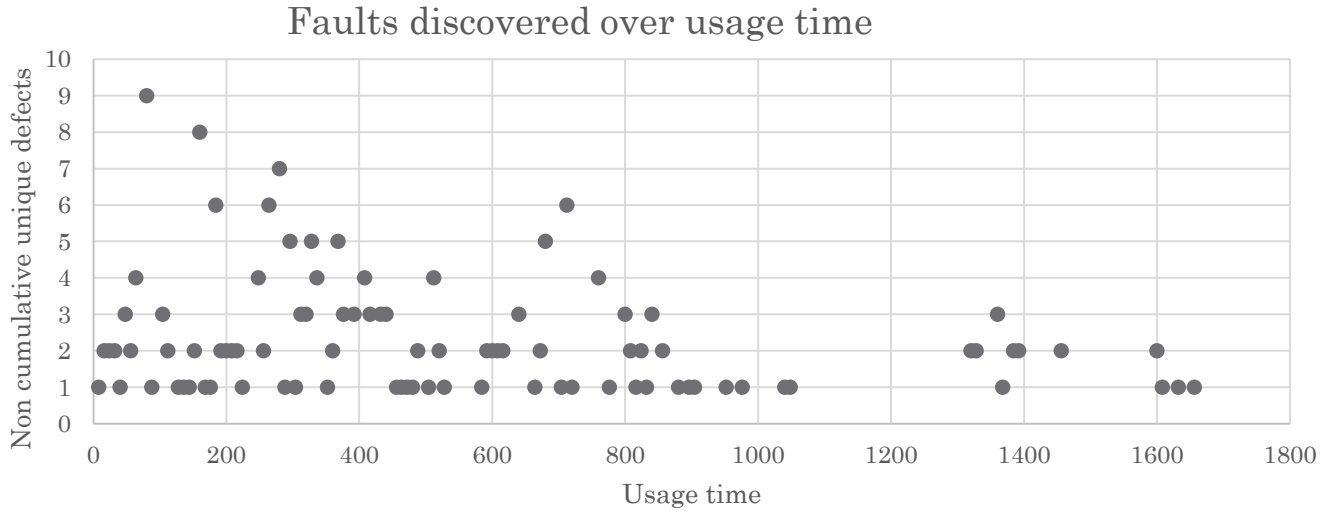
Upon deployment, the actual system reliability was **8 %** of the required reliability objective **because of the software failures**.

If SWRG models had been used prior to deployment, the service would not have accepted the software as is since the RAM goal had not been met.

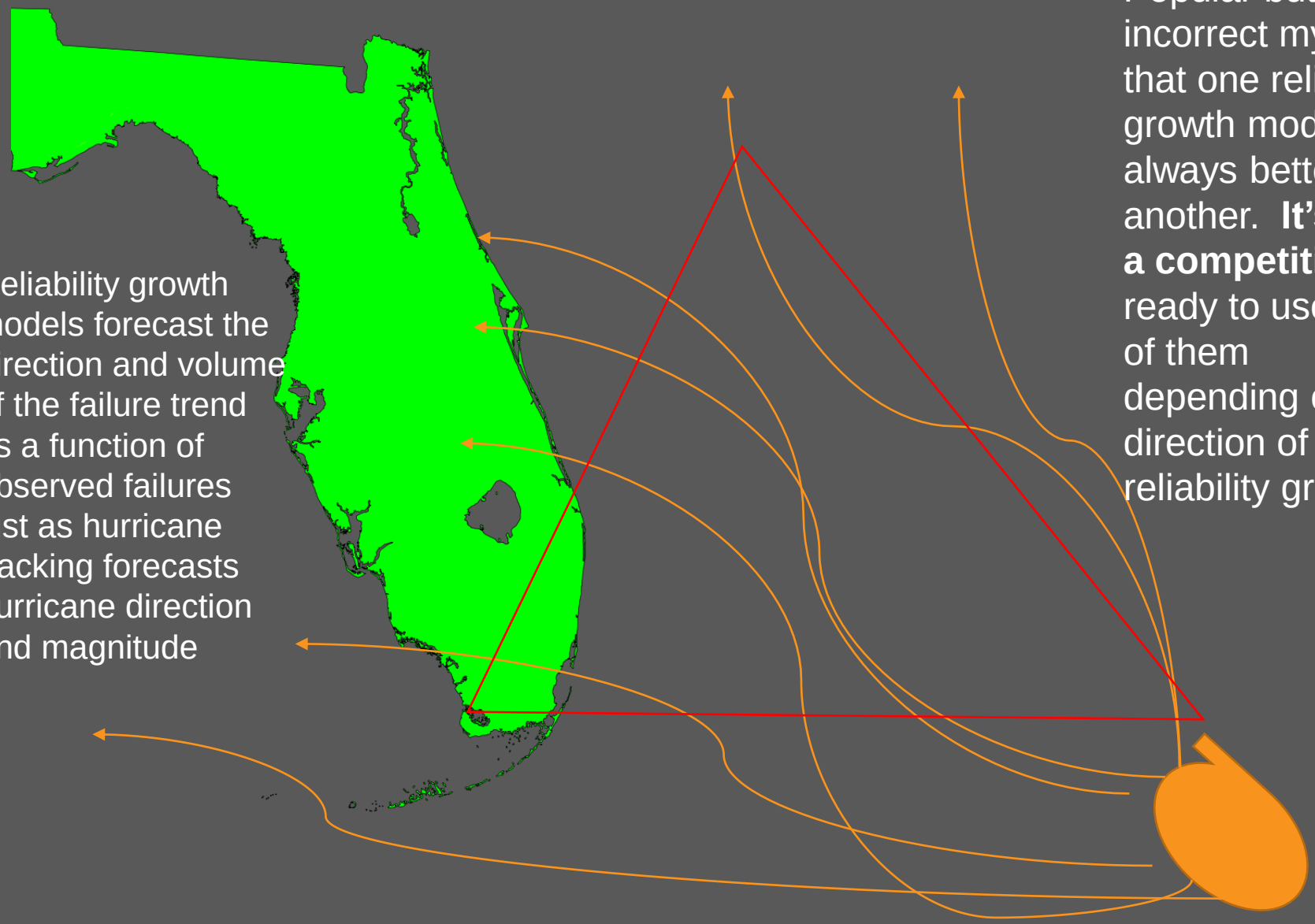
Lessons learned from a real DoD program in which SWRG models were used

This is the fault rate from a DoD software program

- The fault rate is clearly trending downwards
- By the end of the trend, approximately 80% of defects had been discovered
- The time between catastrophic failures was about 1600 hours as there was only 1 during the entire usage
- The time between any serious defect was 20 hours which doesn't meet the system objective
- There was still work to be done with regards to defect removal but the software is stable.
- The SWRG model provides confidence that the overall RAM objective ***can*** be met and the work required to meet it



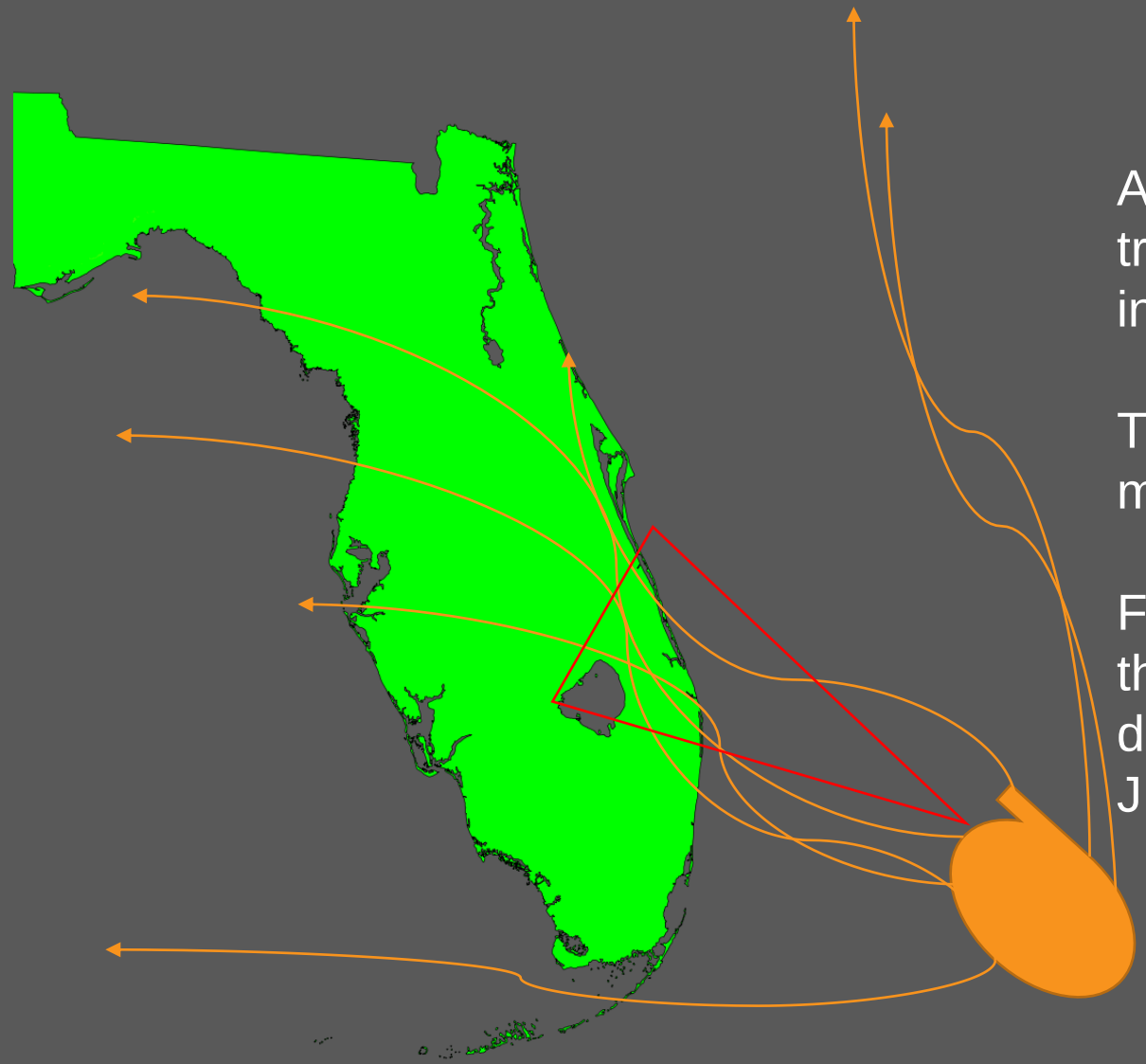
Think of SW reliability growth models as hurricane trackers



Reliability growth models forecast the direction and volume of the failure trend as a function of observed failures just as hurricane tracking forecasts hurricane direction and magnitude

Popular but incorrect myth that one reliability growth model is always better than another. **It's not a competition**, be ready to use any of them depending on the direction of the reliability growth.

The closer the hurricane gets to land the smaller the cone

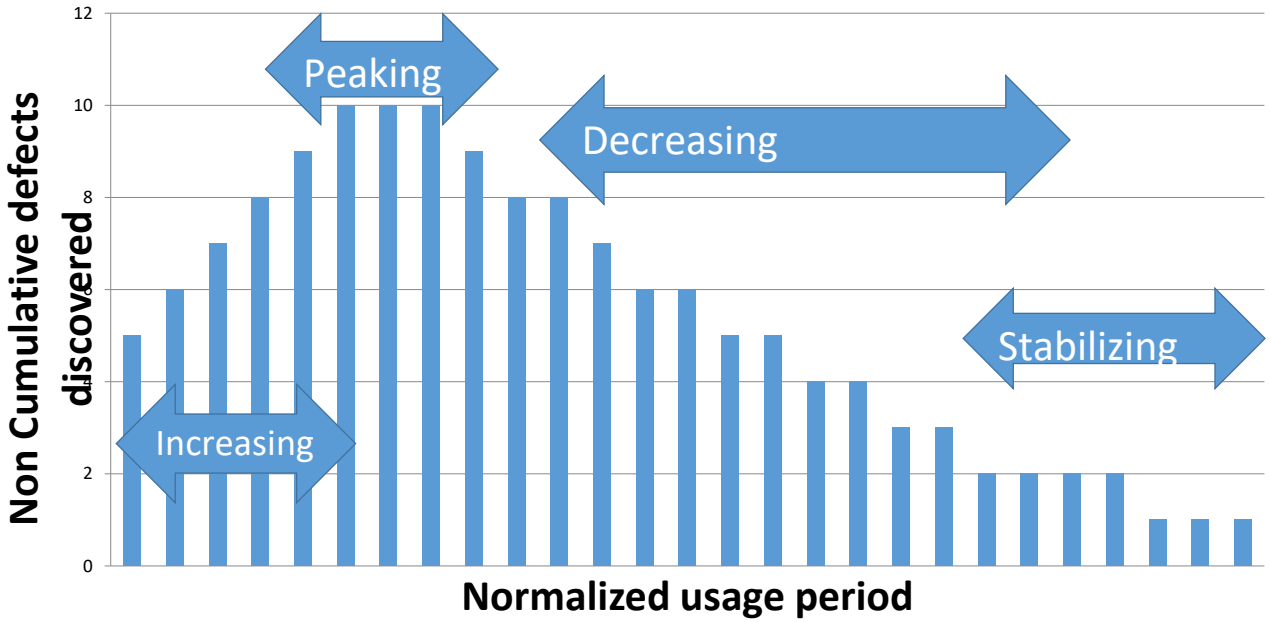


As with hurricane trackers, the expense is in the data collection.

The cost of each of models is insignificant.

For best ROI, have a tool that automates several different models such as JMP.

Software fault rates can increase, peak, decrease or some combination



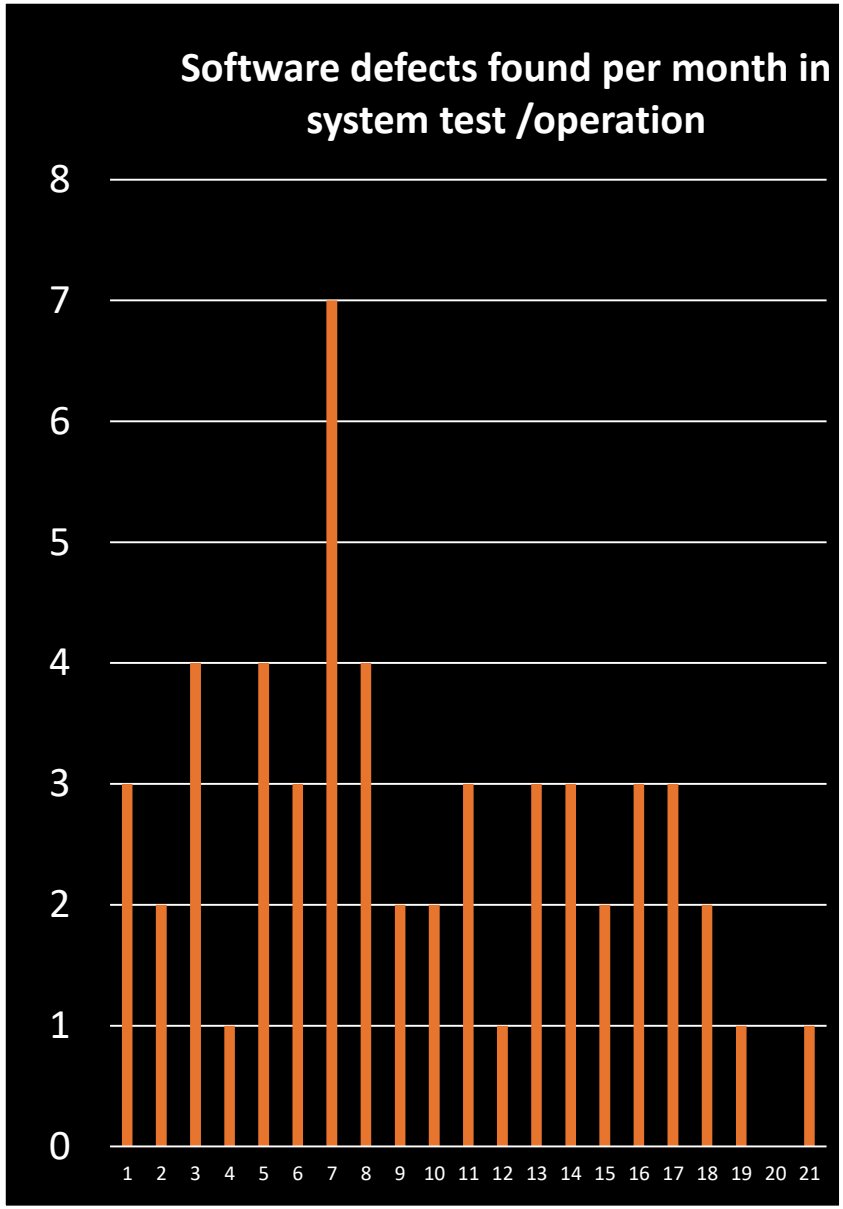
Just as there are different models for hurricanes, there are different models for software reliability depending on the direction of the fault rate and other factors.

The first step in SWRG modeling is to plot the faults over usage time and see what the fault rate is.

The fault rate direction is itself a key indicator of stability. If the testing is almost over and the fault rate is increasing – THAT'S NOT GOOD.

VERY SIMPLE SW RELIABILITY GROWTH MODEL

1. Plot the unique observed defects found during an operational test .
2. Identify the peak. Add all defects up to an including the month of that peak. Ex: The peak below is at month 7 and there were 24 defects found prior to and including that time.
   - If there is no clear peak then the release is probably not mature enough.
   - If there are multiple peaks, choose the biggest peak.
3. Multiply the result of the previous step by 2.5. Ex: 24*2.5 = 60
4. Count up the total defects found so far. Ex: 54 have been found so far.
5. Divide the total found so in step 4 far by the total estimated in step 3. If the result < 75% the software has not reached the minimum % associated with a successful project.
6. If there is no visible peak then statistically < 39% of defects have been found



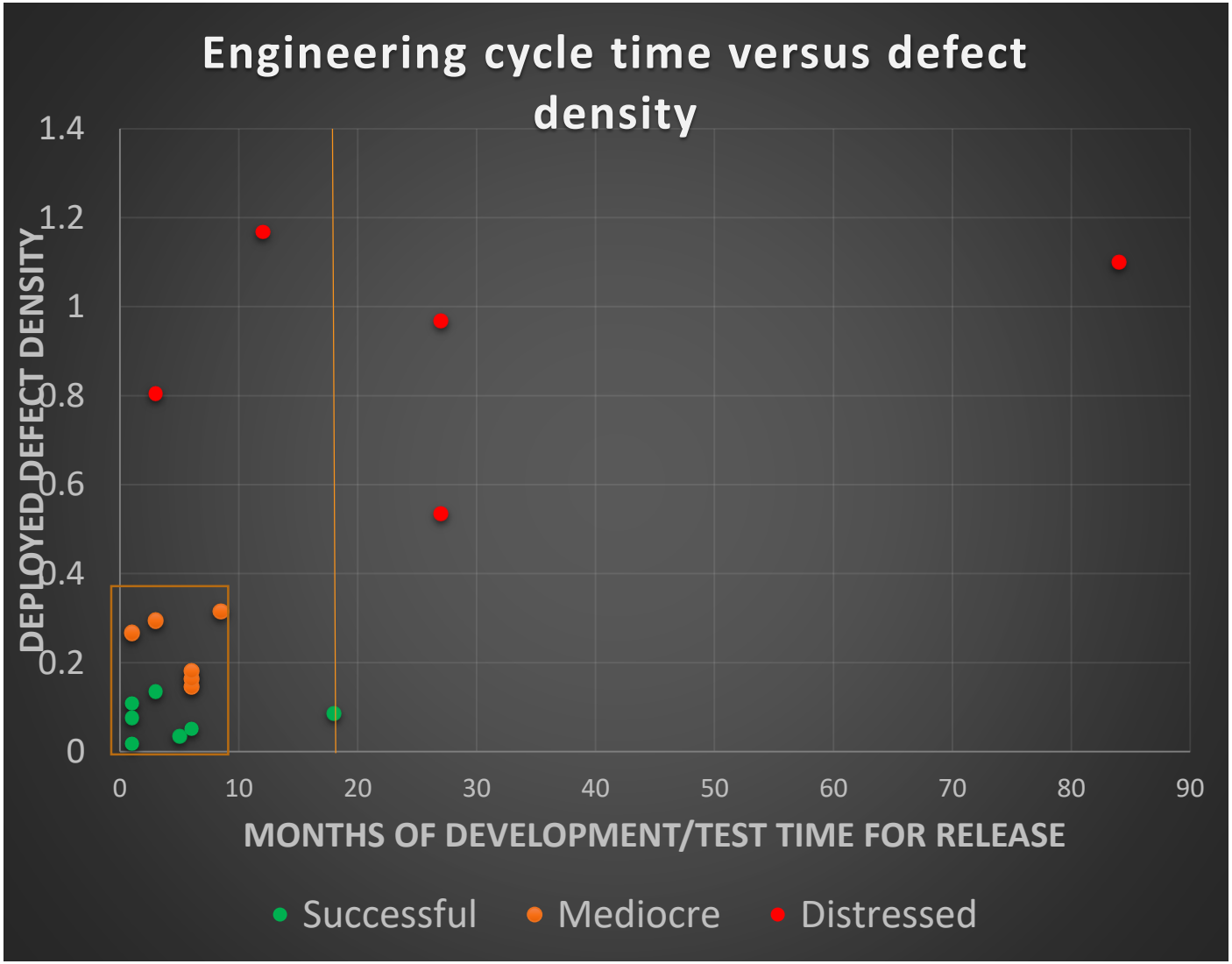
How SRE fits within Agile/Scrum Execution

SRE works within any life cycle model

Agile and incremental execution actually make SRE easier to employ

- Failure modes effects analysis and fault tree analysis are more effective when development is incremental/iterative
- Quantitative models work the same for incremental/iterative development – simply apply them to each engineering cycle
- The below agile principles have been correlated to fewer defects
  - **Break the silos**
  - Deliver value frequently (smaller cycles)
  - Simplest solution possible
  - **Regular face to face meetings between software engineers and leads**
- But our data also shows that the following bad practices (which people justify with Agile) don't correlate
  - Using Agile principle #2 as an excuse to be overly reactive to the loudest customer at the expense of satisfying most customers
  - Using Agile principle #3 as excuse for having a poor design
  - Using Agile principle #5 as an excuse to not review the product
  - Using Agile principle #7 as an excuse to not fix serious defects or to not test failure modes
  - https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/

One of the most sensitive factors is release cycle



In Mission Ready Software DB there have been

- No successful releases when engineering cycle exceeds 18 months
- All successful releases have <=18 month engineering cycle
- When the engineering cycle time is <= 8.5 months few SW projects fail

SWRG Model Goals aren't dramatically different for Agile/Incremental versus Waterfall development

Agile/Incremental sprints	Final Agile/Incremental Release or Waterfall Model Release
• Goal #1 – Verify that the fault rate is decreasing before adding any more code	• Goal #1 – Verify that the fault rate is decreasing before final deployment
• Goal #2 – Predict the total number of defects to ensure at least 75% discovery prior to adding any new code	• Goal #2 – Predict the total number of defects to ensure at least 75% discovery prior to final deployment
• Goal #3 – Predict the failure rate/MTBF to ensure that system failure rate/MTBF goal can be met	• Goal #3 – Predict the failure rate/MTBF to ensure that system failure rate/MTBF goal is met

Agile/Incremental versus Waterfall

Agile/Incremental

- Smaller cycles called sprints
- A sprint can be a “release” but typically isn’t released to operation
- Duration of a sprint can be weeks or months
- Several sprints lead to a final release
- Requirements, design and code evolve

Waterfall

- One big release
- No interim sprints
- Duration of development cycle is often fairly long (i.e. several months or years)
- Requirements are cast in stone before design or code begins

How to apply SWRG models within a sprint

Go through same process shown in this presentation for each sprint

Capture the defect metrics by originating sprint if possible. In other words

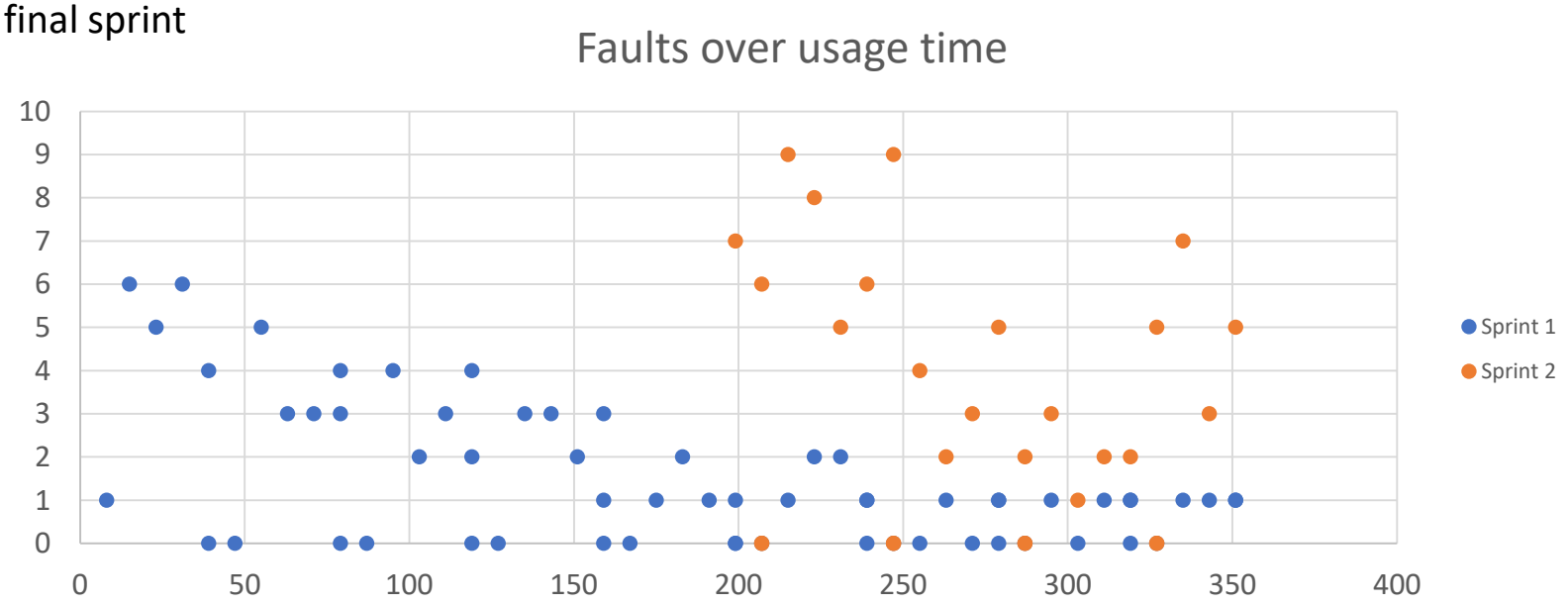
- If a defect found in testing was introduced in sprint 1 but found in sprint 2 it's part of the sprint 1 dataset
- The software engineers who fix the defects know which sprint it was introduced in. The SCR system will need to have a field to identify this information.

Generate a trend and growth model for each sprint and final sprint to ensure

- The fault rate isn't increasing prior to adding in more code
- The estimated number of remaining defects won't lead to defect pileup
- The fault rate is trending towards a final failure rate that meets the objectives

Example

- The sprints last 3 months of which one month is testing. The faults per usage day are plotted over total usage time. The defects found in sprint 2 testing are separated so that those introduced in sprint 1 during sprint 2 testing are trended with sprint 1.
- Sprint 1 faults were decreasing at time 199 when the next sprint was released to testing
- However, faults from sprint 1 spilled into the test effort for sprint 2
- The faults for sprint 2 has just peaked at the end of the testing for that sprint
- At this rate, sprint 3 is not likely to have a decreasing failure rate by the time sprint 3 testing is over. That means that the final sprint isn't on track for a decreasing failure rate as per it's scheduled end of test.
- Conclusions – there isn't enough test effort for each sprint to sustain stable software by the final sprint



References

- [8] “The Cold Hard Truth About Reliable Software, Edition 6h”, A. Neufelder, SoftRel, LLC, 2018
- [9]Four references are
  a) J. McCall, W. Randell, J. Dunham, L. Lauterbach, Software Reliability, Measurement, and Testing Software Reliability and Test Integration RL-TR-92-52, Rome Laboratory, Rome, NY, 1992
  b) "System and Software Reliability Assurance Notebook", P. Lakey, Boeing Corp., A. Neufelder, produced for Rome Laboratory, 1997.
  c) Section 8 of MIL-HDBK-338B, 1 October 1998
  d) Keene, Dr. Samuel, Cole, G.F. “Gerry”, “Reliability Growth of Fielded Software”, Reliability Review, Vol 14, March 1994.

Backup slides

Definitions

Software reliability as per IEEE/ ISO standards

Metric: 1) Probability of success of the software over some specified mission time. 2) Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time

Also used to describe an entire collection of software metrics or the overall maturity of the software.

A function of

- Inherent defects which is a function of
  - Development and test factors
  - Product maturity
  - Organization and experience in industry
  - Inherent risks
  - Process
- Operational profile
  - How the software is used (mission profile)
  - Duty cycle
  - Number of install sites/end users

Errors, defects, faults and failures

- Errors – This term is used incorrectly quite often. It is simply the human mistake made by the software engineer(s) when constructing the code.
  - Example – the software engineer forgets that dividing can cause an overflow when the denominator approaches zero.
- Defects – This is the manifestation of that mistake into the code.
  - Example – the software engineer writes the code a = b/c but doesn't have any checking for c approaching zero
- Fault – This is a defect that has been exercised during runtime. If there is fault handling the fault may not become a failure.
  - Example – During runtime c approaches zero and there is an overflow
- Failure – This is when a fault results in the system requirements not being met.
  - Example – The software is unable to perform it's function because of the overflow. Depending on the system design it might crash.

MISSION READY SOFTWARE

SOFTREL LLC

Software reliability and software FMEA training courses

Software reliability prediction tools

Software FMEA tools

Software reliability services

https://missionreadysoftware.com

sales@missionreadysoftware.com