Overview of Software Dependability

Er. Rudra Nepal

Nepal College of Information Technology

April 17, 2025

Outline

What is System Dependability?

- For many computer-based systems, the most important system property is the dependability of the system.
- The dependability of a system reflects the user's degree of trust in that system. It reflects the extent of the user's confidence that it will operate as users expect and that it will not 'fail' in normal use.
- Dependability covers the related systems attributes of reliability, availability and security. These are all inter-dependent.

Importance of Dependability

- System failures may have widespread effects with large numbers of people affected by the failure.
- Systems that are not dependable and are unreliable, unsafe or insecure may be rejected by their users.
- The costs of system failure may be very high if the failure leads to economic losses or physical damage.
- Undependable systems may cause information loss with a high consequent recovery cost.

Causes of Failure

- **Hardware failure:** Hardware can fail due to design flaws, manufacturing defects, or simply because the components have worn out over time and reached the end of their useful life.
- **Software failure:** Software may fail because of mistakes in how it's specified, designed, or implemented. These errors can lead to unexpected behavior or complete breakdowns.
- **Operational failure:** These failures happen when people using or operating the system make mistakes. In complex systems, human error is often the biggest cause of failure.

Key Dimensions of Dependability

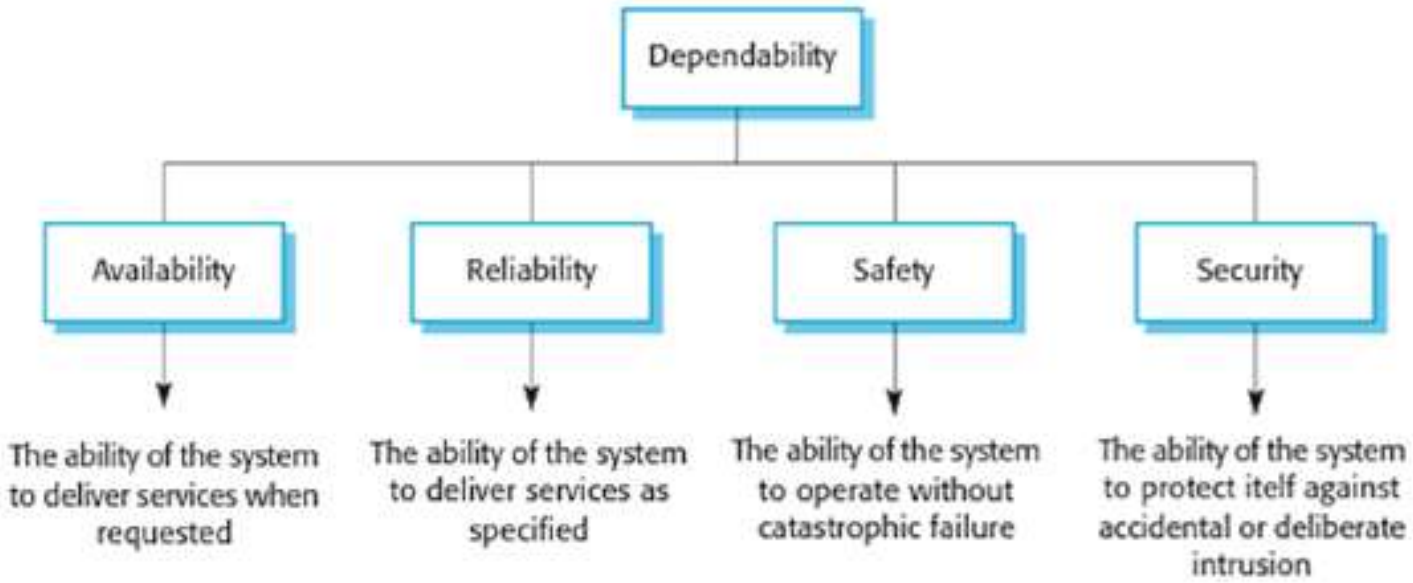


Figure: Key Dimensions of Dependability

Principal Dependability Properties

- **Availability:** Probability the system is operational and delivering services.
- **Reliability:** Probability of failure-free operation over a specified period.
- **Safety:** Likelihood the system avoids causing harm to people or the environment.
- **Security:** Ability to resist accidental or deliberate intrusions.

Other Dependability Properties

- **Repairability:** How easily the system can be repaired after failure.
- **Maintainability:** Ease of adapting or repairing the system for new requirements.
- **Survivability:** Ability to deliver services during/after attacks or failures.
- **Error tolerance:** Ability to avoid, detect, and tolerate user errors.

Repairability

- Ability to quickly fix the system after a failure.
- Involves identifying the problem, accessing the failed component, and applying a fix.
- Considered a short-term solution to restore service.
- Difficult to assess before the system is deployed.

Maintainability

- Ease of making long-term changes or repairs to the system.
- Includes fixing bugs and adding new features.
- A highly maintainable system reduces the chance of introducing new faults.
- Critical for systems that undergo frequent updates.

Survivability

- System's ability to keep working during attacks or failures.
- Especially important for distributed systems with security concerns.
- Related to resilience — continuing operation despite component failures.

Error Tolerance

- Describes how well the system handles user mistakes.
- Errors should be detected and corrected automatically.
- Helps prevent user errors from turning into system failures.
- Part of the broader concept of usability.

Dependability Attribute Dependencies

- Safe operation requires availability and reliability.
- Security breaches can undermine reliability and safety.
- Denial-of-service attacks affect availability.
- Virus infections can compromise reliability and safety.

Dependability Achievement

- Avoid accidental errors during development.
- Use effective verification and validation (V&V) processes.
- Implement protection mechanisms against attacks.
- Configure systems correctly for their environment.
- Include recovery mechanisms for restoring service after failure.

Dependability Costs and Economics

- Dependability costs tend to increase exponentially as increasing levels of dependability are required.
- There are two reasons for this
  - The use of more expensive development techniques and hardware that are required to achieve the higher levels of dependability.
  - The increased testing and system validation that is required to convince the system client and regulators that the required levels of dependability have been achieved.

Dependability costs

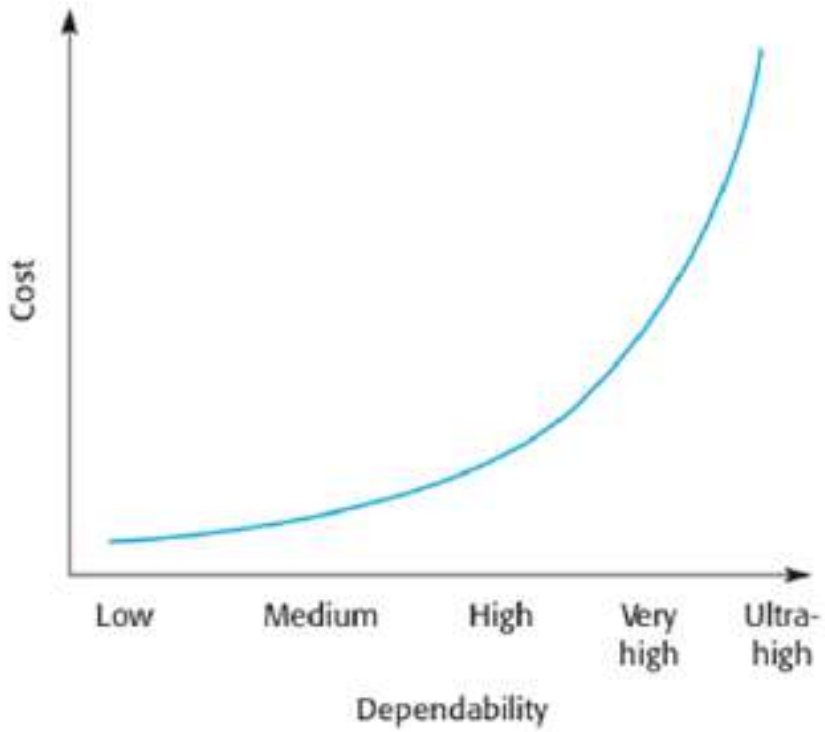


Figure: Cost/dependability curve

Dependability Economics

- Achieving high dependability can be very expensive.
- Sometimes, it's more cost-effective to accept some failures and cover the cost of fixing them.
- However, this approach depends on social and political factors.
- Poor dependability may hurt a company's reputation and future business.
- The required level of dependability also depends on the type of system:
  - For many business systems, moderate dependability may be sufficient.

Availability vs Reliability

- **Reliability:** Probability of failure-free operation over a period.
- **Availability:** Probability the system is operational at a specific time.
- Availability considers repair time; quick repairs can keep availability high even if reliability is low.
- Both can be expressed quantitatively (e.g., 99.9% uptime).

Perceptions of Reliability and Availability

- User perception may differ from formal definitions.
- Environment and consequences of failure affect perception.
- Availability as a percentage does not account for number of users or length of outages.

Faults, Errors, and Failures

- **Human error:** Mistake introducing a fault.
- **System fault:** Defect that can lead to an error.
- **System error:** Erroneous state that can cause unexpected behavior.
- **System failure:** System does not deliver expected service.
- Not all faults cause errors, not all errors cause failures.

Fault Tolerance and Recovery Techniques

- **Fault avoidance:** Minimize introduction of faults during development.
- **Fault detection and removal:** Use V&V to find and correct errors.
- **Fault tolerance:** Run-time techniques to prevent faults from causing errors/failures.
- **Recovery:** Restore normal service after failure (repairability, maintainability, survivability).

Safety and Safety-Critical Systems

- **Safety:** Ability to operate without causing harm.
- **Primary safety-critical:** Failure can directly threaten people (e.g., insulin pump).
- **Secondary safety-critical:** Failure leads to faults in other systems with safety consequences.
- Safety requirements often exclude undesirable situations.

Safety vs Reliability

- Reliability and availability are necessary but not sufficient for safety.
- Reliability: Conformance to specification.
- Safety: Avoidance of harm, even if specification is followed.
- Specification errors or rare faults can make a reliable system unsafe.

Security Concepts

- **Security:** Ability to protect against accidental/deliberate attacks.
- Essential for networked systems.
- Prerequisite for availability, reliability, and safety.
- Insecure systems undermine dependability.

Security Terminology

- **Asset:** Valuable item to be protected (e.g., data).
- **Exposure:** Possible loss/harm (e.g., data loss).
- **Vulnerability:** Weakness that can be exploited.
- **Attack:** Exploitation of a vulnerability.
- **Threat:** Potential cause of loss/harm.
- **Control:** Measure to reduce vulnerability (e.g., encryption).

Threat Classes and Damage from Insecurity

- **Confidentiality:** Unauthorized disclosure of information.
- **Integrity:** Unauthorized modification of software/data.
- **Availability:** Restriction of access for authorized users.
- **Damage:** Denial of service, data corruption, information disclosure.

Security Assurance

- **Vulnerability avoidance:** Design to prevent vulnerabilities.
- **Attack detection and elimination:** Detect and neutralize attacks.
- **Exposure limitation and recovery:** Minimize consequences of successful attacks.

Dependability vs Reliability vs Safety vs Security

Attribute	Focus	Key Points
Dependability	User trust (umbrella term)	Includes availability, reliability, safety, security, repairability, maintainability, etc.
Reliability	Consistent, failure-free service	Necessary but not sufficient for safety; can be formally measured, user perception matters
Safety	Avoidance of harm	May require more than reliability; specification errors can lead to unsafe but "reliable" systems
Security	Protection from threats	Prerequisite for reliability and safety; an insecure system cannot be considered

Key Points Summary

- Dependability reflects user trust in the system.
- It is achieved through avoidance, detection, and tolerance of faults.
- Security is foundational; without it, other dependability attributes are undermined.
- Balancing cost and required dependability is essential for system design.

ELSEVIER

Information and Software Technology 42 (2000) 889–894

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

A Markov chain model for predicting the reliability of multi-build software

J.A. Whittaker[a,*], K. Rekab[b], M.G. Thomason[c]

[a]*Software Engineering Program, Florida Tech, Melbourne, FL 32901 USA*
[b]*Department of Mathematical Sciences, Florida Tech, Melbourne, FL 32901 USA*
[c]*Department of Computer Science, University of Tennessee, Knoxville, TN 37996 USA*

Abstract

In previous work we developed a method to model software testing data, including both failure events and correct behavior, as a finite-state, discrete-parameter, recurrent Markov chain. We then showed how direct computation on the Markov chain could yield various reliability related test measures. Use of the Markov chain allows us to avoid common assumptions about failure rate distributions and allows both the operational profile and test coverage of behavior to be explicitly and automatically incorporated into reliability computation.

Current practice in Markov chain based testing and reliability analysis uses only the testing (and failure) activity on the most recent software build to estimate reliability. In this paper we extend the model to allow use of testing data on prior builds to cover the real-world scenario in which the release build is constructed only after a succession of repairs to buggy pre-release builds. Our goal is to enable reliability prediction for future builds using any or all testing data for prior builds.

The technique we present uses multiple linear regression and exponential smoothing to merge multi-build test data (modeled as separate Markov chains) into a single Markov chain which acts as a predictor of the next build of testing activity. At the end of the testing cycle, the predicted Markov chain represents field use. It is from this chain that reliability predictions are made.

Keywords: Software failure; Software reliability model; Software testing

1. Introduction

Software reliability estimates are generally made by building probability models of data collected during testing. Testing data is comprised of two different types of events: success events are instances in which the software correctly processes input and *failure events* are situations in which the software exhibits out-of-spec behavior, e.g. premature termination, erroneous computation or mishandling of stored data.

In many cases, software reliability models represent success events only implicitly, usually as they pertain to some definition of *time*. Examples are measuring the number of inputs correctly processed (without tracking exactly which inputs) or keeping track of the so-called wall clock time of failure-free runs. Failure events are treated by assuming that they obey a specific probability law and using that assumption to make estimations and predictions about future failure occurrences.

In some cases, such treatment has delivered valuable and accurate analysis [4]—particularly when the assumed distribution governing failure events has empirical support and substantial testing has covered much of the application's functionality. But without empirical data and the resources to perform substantial testing, other approaches are needed. We proposed in Ref. [14] to model success and failure events as a finite-state, discrete-parameter, recurrent Markov chain. The Markov model has wide applicability as shown in several subsequent case studies [1,3,5,7,9,10]. The model encapsulates success events in advance of testing as states and state transitions in a Markov chain. Thus, one can apply a series of tests and maintain counts of state transitions, providing convenient bookkeeping for test coverage and progress. Obviously, the more complex an application, the larger the state set, requiring a larger test set to properly cover. Missing test inputs and sections of behavior which are poorly covered can be detected by analysis of the model [14].

Failure events are associated with the last known success

* Corresponding author.
E-mail addresses: jw@se.fit.edu (J.A. Whittaker), rekab@cs.fit.edu (K. Rekab), thomason@cs.utk.edu (M.G. Thomason).

0950-5849/00/$ - see front matter
PII: S0950-5849(00)00122-1

state and integrated directly into the Markov model as a special *failure state*, say state s_f, such that out-of-spec software behaviors cause transitions to s_f. Each transition to s_f marks a distinct failure event and may differ substantially in its probability of occurrence. This is a desirable characteristic of the model since each failure event is likely to occur at a different rate in the field. Thus, all failures are not treated the same. Their individual probability of occurrence (which reflects their contribution to unreliability) can be computed from the chain. In Ref. [14] we derive equations for reliability, *MTTF*, and a stochastic test stopping criteria.

Thus, the Markov chain may alleviate the need for some assumptions common in reliability models that do not explicitly incorporate software behavior into their formulae. Moreover, recent advances in automating the generation of states and state transitions have made the modeling process practical for increasingly larger systems [2,10]. The contribution of this paper is to combine past and current testing data into a single Markov chain representing predicted testing results for the next build of the software. This will allow us to detect patterns in the way failure events occur and recur and use these patterns to predict future failure events.

Sections 2 and 3 define Markov chains for software testing and discuss their construction. Sections 4 and 5 present a method for statistical analysis of prior testing chains, each of which represents a specific build of the software during testing. Section 6 describes an example use of our new method wherein we predict known Markov chains and compare the accuracy of our predicted chains with the actual chains.

2. A Markov chain model for software testing

A Markov chain model for software testing consists of:

- the finite set S of operational states of a software system;
- the finite set I of externally generated inputs to the software; and
- the probability-weighted transition function $\delta : S \times I \times [0, 1] \rightarrow S$.

Operational states describe internal or external objects that influence the behavior of the software under test. In general, we are interested in objects that affect the way the software reacts to external stimuli. We say that software is in state s_j when a collection of objects have certain values and in a different state s_k when at least one of those objects has a different value. The current state also designates which external inputs are allowed and which are prevented (or, at least reacted to in a different manner) by the software. State s_k may have a different set of allowable inputs that mark it as distinct from s_j. Two special states s_{inv} and s_{term} are established to represent invocation and termination of the software and are the initial and final states of the Markov chain, respectively.

The transition mapping δ describes probabilistic changes of state as a function of internal state and application of external input. The model's probability distributions are established through collection of frequency counts during testing. Before testing begins, each transition is set to frequency count 0 to reflect the fact that the software has yet to receive any input. As inputs are applied by testers, the corresponding transitions are identified and their frequency counts incremented to maintain an accurate record of transition coverage during testing. This can be accomplished by instrumenting existing test harnesses[1] or by using the Markov chain itself to generate test cases [5,14].

When failures are encountered, new states are integrated into the chain to model these events. Suppose input j is supposed to move the application from state s_i to state s_k but instead causes a failure in the software to be identified. Create a transition from state s_i to fail state s_f with frequency count 1 (assuming that this is the first failure from state s_i) and an outgoing transition from s_f to state s_k, unless the failure caused the system to halt in which case the outgoing transition is made to the terminate state s_{term}. Additional failures are modeled by establishing new transitions to s_f and failure recurrence is modeled by incrementing existing counts on the appropriate transition.

Whenever estimation using the model is required, the frequency counts can be normalized to probabilities by dividing each frequency by the sum of the outgoing frequencies for each state $s \in S$. When probabilities are assigned to transitions, the model is a finite state, discrete parameter Markov chain and many standard analytical results can be computed without additional assumptions [6]. Moreover, the distribution associated with failure occurrence is embedded in the Markov chain and arises from the statistics associated with real testing data.

3. Modeling software repair

In previous work [14] we modeled software-testing data with a single Markov chain. A problem not addressed, however, was continuity of the model when the underlying software is modified in an effort to repair one or more faults. If the software is changed to eliminate one or more faults, we expect the new build to have fewer embedded defects than previous builds; however, a "repair" that changes the software may in fact result in any of the following:

1. The repair may be successful with no unwanted side-effects. This is the perfect repair situation and allows us to reset to 0 the frequency on the affected transition(s) to the failure state s_f. The new build of the software is an unambiguous improvement over the old.
2. The repair may be locally successful but, as a side-effect, cause a part of the software once deemed fault-free now to be faulty. The implication for the test Markov chain is that, while the transitions to s_f associated with the repair

[1] Web applications can be easily instrumented with page counters to automate this task.

can be reset, new transitions with nonzero frequency count must also be established.
3. The repair may be unsuccessful but without unwanted side-effects. This situation is status quo: there is no improvement overall in the software but no degradation either. The test Markov chain remains unchanged.
4. The repair may be unsuccessful and also introduce unwanted side-effects. In this case the relevant, existing transitions are unchanged and new transitions must be created to represent the side-effects. There is degradation of the software in this unfortunate situation.

There is no way to determine which of these four situations has occurred without further analysis, usually in the form of additional testing. By resetting the frequency counts each time a new build is compiled, our original model took the most conservative approach.

An alternative approach suggested by Walton et al. [12] is to continue incrementing frequency counts across software builds so that one may estimate various aspects of the development process for those builds. However, they too suggest resetting frequency counts for product-oriented estimation such as reliability.

We propose the following method of combining results obtained during additional testing with results from previous testing. The objective is to incorporate the entire testing-and-repair history into a tractable model based on practical, but rigorous, statistical analysis of any or all test data.

Let the initial testing chain T_1 be established and updated with frequency counts and failure state as described above until a repair is attempted. At this time no further change to T_1 is allowed and T_2 is established when testing resumes on the modified software. Consecutive testing chains T_3, T_4,… are similarly constructed.

Thus we establish a single and separate testing chain for each build of the software. The chain T_{k+1} represents testing of the next software build (or usage of the software after release) and has unknown transition probabilities. We are interested in predicting the parameters of this chain using any or all of the chains $T_1, \ldots, T_k$.

Since current state of the practice is to use only the most recent chain T_k to achieve this result [5,12], we believe that improvement in predictive accuracy can result. Since modifications to fix faults have changed T_k, there is every reason to believe (or to hope) that T_{k+1} will differ from T_k in general. Thus, we propose that an analysis of the trends in changes to the transition probabilities over the series $T_1, \ldots, T_k$ will potentially increase the accuracy of our prediction of T_{k+1} rather than using T_k alone.

4. Estimating the parameters of T_{k+1}

Let T^*_{k+1} denote the predictor of T_{k+1}. Obviously, predicting T_1 is not possible since no testing data exists. Likewise, predicting T_2 is problematic since there is only one prior data point and no patterns or trends will have surfaced in the data. However, T^*_3 can be established by analysis of T_1 and T_2. Once this has been accomplished T_4 can be predicted using the real data T_3 combined with its prediction, T^*_3 which contains the history data and trends established from prior testing. Thus, we carry historical data along in the form of our prediction chains throughout the testing process. Note also that since we make predictions for each intermediate build and then later establish actual data, we may discard inaccurate predictions and get feedback on the quality of the predictive process.

Our research suggests that a good prediction of T_{k+1} should involve two prediction techniques: *multiple linear regression* and *exponential smoothing*. Recall that our predictor of T_{k+1} is denoted by T^*_{k+1} and has the form

$$T^*_{k+1} = \alpha^* T_k + (1 - \alpha^*) T^*_k$$

where T^*_k is a "good predictor" of T_k based on the transition matrices of the previous chains $T_1, T_2, \ldots, T_{k-1}$.

Specifically, let $\hat{T}_k$ be a predictor of T_k determined via multiple linear regression and let $\tilde{T}_k$ be a predictor of T_k determined through the application of exponential smoothing. $\tilde{T}_k$ will depend on α^*, a suitable constant satisfying $0 < \alpha^* < 1$ and

$$\sum_{i=3}^{k} d(\tilde{T}_i(\alpha^*), T_i) = \min_{0<\alpha<1} \sum_{i=3}^{k} d(\tilde{T}_i(\alpha), T_i),$$

(see Section 6.1 for more details).

T^*_k is chosen so that

$$d(T^*_k, T_k) = \min\{d(\hat{T}_k, T_k), d(\tilde{T}_k, T_k)\}$$

where $d(T^*_k, T_k)$ denotes a distance from transition matrix T^*_k to matrix T_k. We will use a distance measure of the form

$$d(T^*_k, T_k) = \sum_m \sum_n \frac{|(T^*_k(m,n) - T_k(m,n))|}{T_k(m,n) + 1}$$

where m is the matrix-row index and n is the matrix-column index. This distance allows for a relative difference instead of an absolute difference. See Ref. [11] for more details.

Thus, we determine the parameters of T^*_{k+1} using the previous T_k established with real testing data and T^*_k which is a predictor of T_k encapsulating the history of chains $T_1, \ldots, T_{k-1}$. As we continue testing, we will eventually get the actual T_{k+1} and can compare it to our estimate T^*_{k+1} using the distance measure above. In this manner we can track the accuracy of our predictions through each successive build of the software. In fact, the distance measure may indicate that certain builds of testing chains may be atypical for one reason or another, allowing us to drop those data points from our analysis.

The ultimate goal is to determine T^*_{k+1} for which no T_{k+1} will be known, i.e. where T_k represents the final build before release. Thus, T^*_{k+1} becomes our prediction of field usage and estimates computed from it represent post-release reliability.

Table 1
Selection of α for exponential smoothing

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
$D(\alpha)$	270	228	201	175	174	167	163	152	161

5. Predictors of the transition matrix T^*_{k+1}

We use one of two methods to predict T^*_{k+1} based on the transition matrices $T_1, T_2, \ldots, T_{k-1}$. $\tilde{T}_k$ is derived using *exponential smoothing* and $\hat{T}_k$ is derived using *multiple linear regression*. The one, which provides the smaller distance measure, as defined above gets selected.

5.1. Derivation of $\tilde{T}_k$ (exponential smoothing)

$\tilde{T}_k$ is determined through the recursion

$$\tilde{T}_k = \alpha T_{k-1} + (1-\alpha)\tilde{T}_{k-1}$$

$$\tilde{T}_{k-1} = \alpha T_{k-2} + (1-\alpha)\tilde{T}_{k-2}$$

$$\vdots$$

$$\tilde{T}_3 = \alpha T_2 + (1-\alpha)\hat{T}_2$$

where $\hat{T}_2$ is an estimate of T_2 determined by linear regression.

5.2. Derivation of $\hat{T}_k$ (multiple linear regression)

The general form of $\hat{T}_k$ is

$$\hat{T}_k = a_0 + \sum_{i=1}^{k-1} a_i T_i + \sum_{i<j} \sum_{j=2}^{k} a_{ij} T_i T_j$$

where a_i for $i = 1, \ldots, k-1$ and a_{ij} for $i < j, j = 1, \ldots, k$ are determined through the least square method. In certain cases, the value of the distance is not increased significantly by including the interaction terms $T_i T_j$. In these cases, a simple multiple-linear regression model of the form

$$\hat{T}_k = a_0 + \sum_{i=1}^{k-1} a_i T_i$$

may be used.

Table 2
Regression model relating T_9 with $T_1, \ldots, T_8$

	Unstandardized coefficients	
	B	S.E.
(Constant)	-0.1×10^{-3}	0.141
T_1	−1.150	0.439
T_2	0.534	0.284
T_3	2.259	0.282
T_4	−1.308	0.287
T_5	0.378	0.074
T_6	−1.453	0.221
T_7	0.865	0.110
T_8	1.200	0.180

6. Example

We have data from a small development effort conducted under industrial contract. During this development, a 12-state Markov chain was constructed to generate test cases and ten iterated builds of the software occurred before its release. A single failure state s_f was created in the chains as a thirteenth state for the transitions due to software faults. We are interested in using the first nine chains to predict the tenth. Since the tenth chain is known, we can compare the accuracy of our predictions. Given $T_1, T_2, \ldots, T_9$ as the nine chains for the respective pre-release builds, we seek to determine T^*_{10} as a suitable predictor of T_{10}.

6.1. Exponential smoothing

As described in Section 4, the predictor of chain T_9 has the form

$$\tilde{T}_9(\alpha) = \alpha T_8 + (1-\alpha)\tilde{T}_8$$

$$\vdots$$

$$\tilde{T}_3(\alpha) = \alpha T_2 + (1-\alpha)\hat{T}_2$$

$$\hat{T}_2 = 0.129 + 1.561 T_1$$

where $\hat{T}_2$ is a predictor of T_2, given T_1, determined through simple linear regression.

$\tilde{T}_9(\alpha^*)$ is chosen so that

$$\sum_{i=3}^{9} d(\tilde{T}_i(\alpha^*), T_i) = \min_{0<\alpha<1} \sum_{i=3}^{9} d(\tilde{T}_i(\alpha), T_i)$$

where the distance d is defined as in Section 4. Table 1 represents the values of $D(\alpha) = \sum_{i=3}^{9} d(\tilde{T}_i(\alpha), T_i)$. The minimum occurs for $\alpha^* = 0.8$ giving

$$\tilde{T}_9 = 0.8 T_8 + 0.2 \tilde{T}_8.$$

Also, note that

$$d(\tilde{T}_9, T_9) = \sum_{m=1}^{13} \sum_{n=1}^{13} \frac{|(\tilde{T}_9(m,n) - T_9(m,n))|}{T_9(m,n) + 1} = 13.$$

6.2. Multiple linear regression

Multiple linear regression relates transition matrix T_9 to $T_1, \ldots, T_8$ in one linear functional form. Results in least squares coefficients are presented in Table 2.

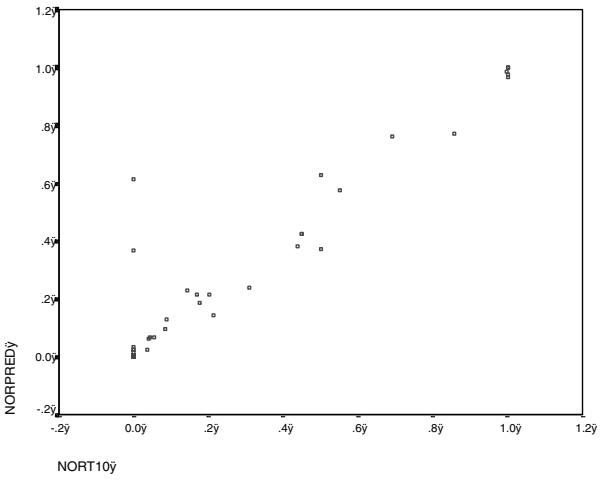


Fig. 1. Scatter plot: normalized predicted vs. normalized observed for transition matrix T_{10}.

The prediction equation is

$$\hat{T}_9 = -0.003 - 1.150T_1 + 0.534T_2 + 2.259T_3 + 1.308T_4 + 0.378T_5 - 1.453T_6 + 0.865T_7 + 1.200T_8,$$

and

$$d(\hat{T}_9, T_9) = \sum_{m=1}^{13}\sum_{n=1}^{13} \frac{|(\hat{T}_9(m,n) - T_9(m,n))|}{T_9(m,n)+1} = 16.32$$

which is larger than $d(\tilde{T}_9, T_9)$. The better predictor, therefore, of T_9 is $\tilde{T}_9$. Consequently, the predictor of T_{10} is

$$T_{10}^* = 0.8T_9 + 0.2\tilde{T}_9.$$

6.3. Adequacy of the predicted transition matrix

In our data, T_{10} contained no failures. This is common in some projects where release is forbidden unless a build tests clean. However, predicting no field failures has little value since it is unlikely that testing covered every possible input scenario combination.

This is a case in which our new method can predict failure likelihood in the absence of failures in the current build. Since it uses both current build data and prior build data, we have information about past failures and the failure occurrence trend from prior builds.

In order to compare our prediction T_{10}^* to the observed results T_{10}. We normalized each row of both transition matrices and performed exponential smoothing and multiple linear regression, accepting the best predictor for each successive build. Fig. 1 is a scatter plot for the normalized predicted matrix T_{10}^* (shown as the *y*-axis) vs. the normalized observed matrix T_{10} (shown as the *x*-axis). Most cases in the plot show that the predicted transition matrix is very close to the observed transition matrix (indicated visually by clustering along the diagonal). The two points that stray furthest from the diagonal correspond to failure states in chain T_{10}^*. Since T_{10} contained no failures, the failure values in T_{10}^* were compared to zero. Obviously, more testing would bring these two values closer to the diagonal.

7. Conclusion

We presented a technique to combine multi-build testing results each of which are modeled as separate Markov chains to predict testing results for the next software build. When such predictions are made using the last build before release, the predictions represent expected field use. Reliability analysis (presented in another paper [14]) on the predicted chains can be used to quantify software behavior in the field.

Currently, we are performing more substantial field trials under a contract from IBM Storage Systems Division in Tucson, Arizona. We hope to present this data in future papers.

Acknowledgements

The authors have had many helpful discussions with practitioners, most especially Mike Houghtaling of IBM and Kaushal Agrawal of Cisco. We would also like to thank the anonymous referees for many helpful suggestions.

References

[1] K. Agrawal, J.A. Whittaker, Experiences in applying statistical testing to a real-time embedded software system, Proc. Pac. Northwest Soft. Quality Conf., Portland OR, 1993, pp. 154–170.
[2] I. El-Far, Automated Construction of Markov Chains for Software Testing, MS Thesis, Comp. Sci. Program, Florida Inst. of Tech, Melbourne, FL 1999.
[3] O. Dahle, Statistical Usage Testing Applied to Mobile Telecommunications Systems, MS Thesis, Dept of Comp. Sci., Univ. of Trondheim, Norway, 1995.
[4] W. Farr, Software reliability modeling survey, in: M.R. Lyu (Ed.), Handbook of Software Reliability Engineering, IEEE Computer Society Press, New Jersey, 1996, pp. 71–115.
[5] M. Houghtaling, Automation frameworks for Markov chain statistical testing, Proc. Automated Software Test Evaluation Conf., EFPDMA Press, Washington DC, March 1996.
[6] J.G. Kemeny, L.J. Snell, Finite Markov Chains, Springer, New York, 1976.
[7] R. Oshana, Tailoring cleanroom for industrial use, IEEE Software 15 (6) (1998) 46–55.
[9] M. Rautakorpi, Application of Markov Chain Techniques in Certification of Software, MS Thesis, Dept of Mathematics and Systems Analysis, Helsinki Inst. Tech., Helsinki, Finland, 1995.

[10] H. Robinson, Finite model-based testing on a shoestring budget, Proc. Software Testing Analysis Review Conf., San Jose, CA, Nov. 1999.
[11] V.K. Rohatgi, An Introduction to Probability Theory and Mathematical Statistics, Wiley, New York, 1976.
[12] G.H. Walton, J.H. Poore, C.J. Trammell, Statistical testing of software based on a usage model, Software Practice and Experience 25 (1) (1993) 98–107.
[14] J.A. Whittaker, M.G. Thomason, markov chain model for statistical software testing, IEEE Transactions on Software Engineering 20 (10) (1994) 812–824.

Software Reliability

Er. Rudra Nepal

May 5, 2025

Introduction to Software Reliability

- Software reliability is a key aspect of software quality.
- It refers to the probability of a software system operating without failure under given conditions for a specified period of time.
- It focuses on the system's ability to perform its intended functions correctly and consistently.
- Unlike hardware, software doesn't degrade physically over time.
- Software can fail due to:
  - Design flaws
  - Coding errors
  - Unexpected inputs
  - Integration issues
- As software systems become more complex and critical (e.g., in healthcare, finance, aviation), ensuring high reliability becomes increasingly important.

Software Aging vs. Hardware Aging

- Software is not susceptible to environmental maladies like hardware.
- However, software does deteriorate over time.
- During its lifetime, software undergoes many changes.
- Each change can introduce new errors.
- These errors may increase the failure rate temporarily.
- Unlike hardware, software has no physical spare parts for replacement.

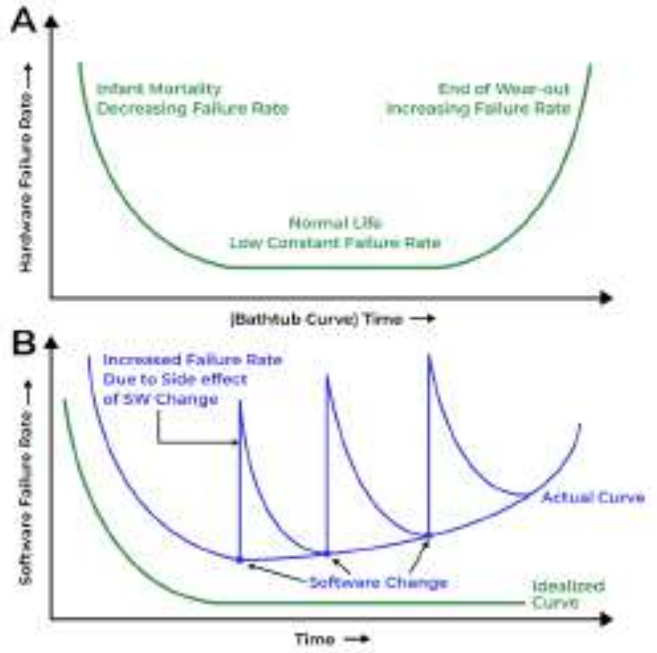


Figure: Failure rate spike after changes

Significance of Reliability in Software Systems

Software Reliability is one of the most critical attributes of a software system, especially in today's world where software controls everything from personal devices to life-critical systems. A reliable software system consistently performs its intended functions without failure over a specified period under stated conditions.

Why Reliability Matters:

- ✓ **User Trust:** Reliable systems are trusted and more widely adopted.
- ✓ **Safety Risk Reduction:** Critical systems (e.g., healthcare, aviation) require high reliability to prevent disasters.
- ✓ **Cost Efficiency:** Fewer failures mean less time and money spent on support and maintenance.
- ✓ **Reputation Business Impact:** Unreliable software can damage a company's image and financials.
- ✓ **Operational Continuity:** Essential for uninterrupted services in finance, logistics, etc.

Reliability Metrics

Reliability metrics are used to quantitatively express the reliability of a software product.

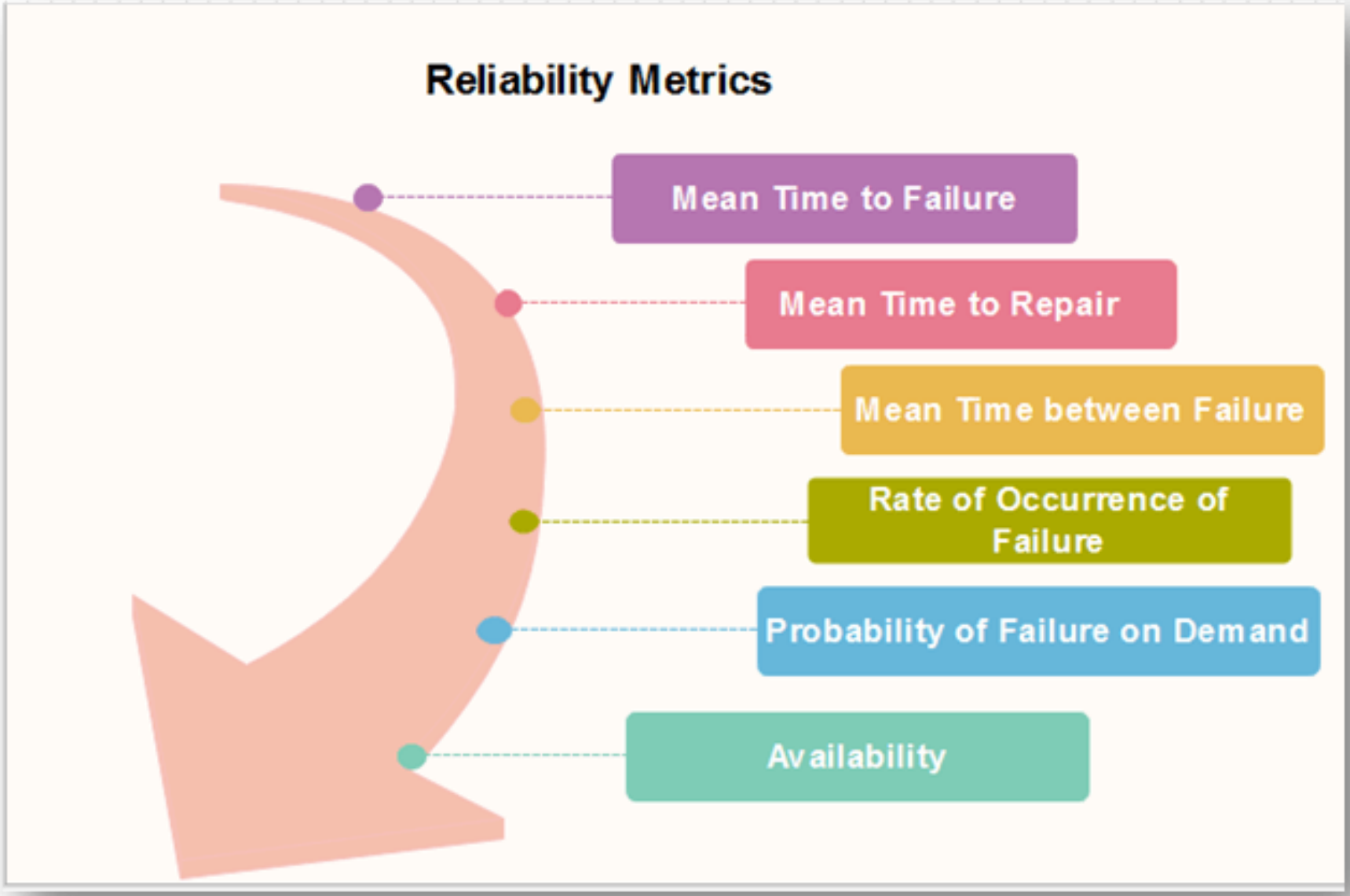
The choice of which metric to use depends on:

- The type of software system
- The operational environment
- The reliability requirements of the application domain

Some commonly used reliability metrics to quantify software reliability include:

- Mean Time To Failure (MTTF)
- Mean Time Between Failures (MTBF)
- Failure Rate
- Probability of Failure on Demand (POFOD)
- Availability

Reliability Metrics



1. Mean Time to Failure (MTTF)

- MTTF is the average time between two successive failures.
- Example: MTTF of 200 means one failure is expected every 200 time units.
- Time units can be hours, transactions, or operations.
- Suitable for systems like CAD or word processors.

MTTF Calculation:

$$\text{MTTF} = \frac{t_1 + t_2 + \cdots + t_n}{n}$$

where $t_1, t_2, ..., t_n$ are times between failures.

Calculation of MTTF

- To measure MTTF, we collect and observe failure data for n failures.
- Let the failures occur at time instants $t_1, t_2, \ldots, t_n$.

MTTF can be calculated as:

$$\text{MTTF} = \frac{1}{n} \sum_{i=1}^{n} t_i$$

where:

- t_i is the time to the i^{th} failure
- n is the total number of failures observed

Thus, the Mean Time to Failure represents the average operational time before a system or component fails.

MTTF Calculation for a Software System

Question:

A software company monitors a web application running on multiple servers. Over the course of a month, the application runs for a total of **12,000 hours** across all servers. During this period, the application experiences **6 failures** (crashes requiring a restart).

Calculate the Mean Time to Failure (MTTF) for the application. What does this value indicate about the software's reliability?

Solution:

1. **Identify the Given Data:**
   - Total operational time: $12,000$ hours
   - Number of failures: 6
2. **MTTF Formula:**

$$\text{MTTF} = \frac{\text{Total Operational Time}}{\text{Number of Failures}}$$

Calculation

- **Substituting the values Values:**

$$\text{MTTF} = \frac{12,000 \text{ hours}}{6} = 2,000 \text{ hours}$$

- **Interpretation:**
  - The MTTF is $2,000$ hours.
  - On average, the web application runs for $2,000$ hours before experiencing a failure.
  - A higher MTTF indicates better reliability.
  - This metric helps the company plan maintenance and improve robustness.

Numerical Example of MTTF

Suppose we observe the following times to failure (in hours) for a single device

Failure Number	Time to Failure (hours)
1	120
2	150
3	130
4	140

Step 1: Add all the failure times:

$$\text{Total Time} = 120 + 150 + 130 + 140 = 540\,\text{hours}$$

Step 2: Count the number of failures:

$$n = 4$$

Step 3: Apply the MTTF formula:

$$\text{MTTF} = \frac{540}{4} = 135\,\text{hours}$$

2. MTTR

MTTR – Mean Time to Repair

- Once failure occurs, some-time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
- Includes fault diagnosis, repair, and restart.

Numerical Example of MTTR

Suppose a machine experiences the following downtime (in hours) for 3 repair events:

Repair Event	Downtime (hours)
1	4
2	6
3	5

solution

$$\text{Total Downtime} = 4 + 6 + 5 = 15\,\text{hours}$$

$$\text{Number of Repairs} = 3$$

$$\text{MTTR} = \frac{15}{3} = 5\,\text{hours}$$

3. MTBF

MTBF – Mean Time Between Failures

- Combines MTTF and MTTR:

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

- Example: Thus, an MTBF of 300 denoted that once the failure appears, the next failure is expected to appear only after 300 hours. In this method, the time measurements are real-time not the execution time as in MTTF.

Numerical Example of MTBF

Let's say:

- **MTTF** = 100 hours (this is the average time the system works before it fails),
- **MTTR** = 10 hours (this is the average time it takes to repair the system).

Then, we can calculate **MTBF**:

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

$$\text{MTBF} = 100\,\text{hours} + 10\,\text{hours} = 110\,\text{hours}$$

4. ROCOF and 5. POFOD

ROCOF – Rate of Occurrence of Failure

- Number of failures per unit time.
- ROCOF = 0.02 means 2 failures in every 100 operational time units.
- Also known as failure intensity.

POFOD – Probability of Failure on Demand

- Likelihood that the system fails on a service request.
- POFOD = 0.1 means 1 failure in 10 service requests.
- Especially critical for safety or protection systems.

6. Availability (AVAIL)

- Probability that the system is operational at a given time.
- Takes into account both planned and unplanned downtime.
- Example: AVAIL $= 0.995$ means 995 hours of availability in every 1000 hours.
- If a system is down for 4 hours out of every 100, AVAIL $= 96\%$.

Formula:

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

Numerical Example of Availability

Let's say:

- **MTBF** = 110 hours (the system operates for 110 hours on average between failures),
- **MTTR** = 10 hours (the system takes 10 hours on average to repair after each failure).

We can calculate **Availability** using the formula:

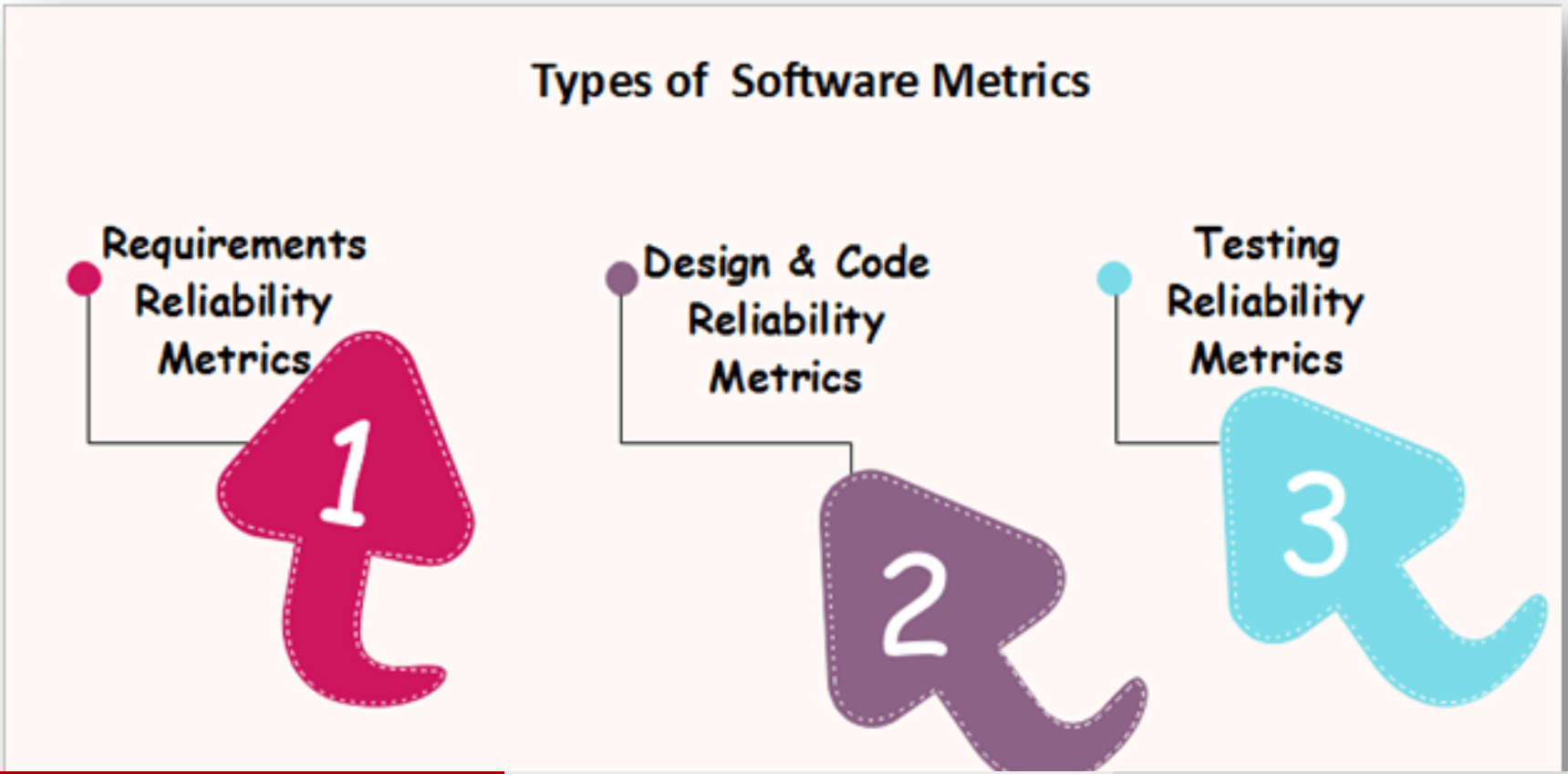
$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

$$\text{Availability} = \frac{110}{110 + 10} = \frac{110}{120} = 0.9167$$

So, the **Availability** is approximately 91.67%.

Software Metrics for Reliability

Software metrics are tools used to improve the reliability of a system by identifying areas for improvement in requirements, design, code, and testing. These metrics help in understanding the quality and performance of the software.



Different Types of Software Metrics

The different types of software metrics are:

- Reliability Metrics
- Requirements Reliability Metrics
- Design and Code Reliability Metrics
- Testing Reliability Metrics

1. Reliability Metrics

Reliability metrics assess the overall reliability of a system. They measure the system's behavior during normal operation and failure events.

- Frequency of failures
- System recovery ability
- Impact of failures on the system

These metrics are crucial for understanding how well the system performs under varying conditions without failure.

2. Requirements Reliability Metrics

Requirements define the expected features and functionality of the software.

- Requirements must be clear, thorough, and detailed to avoid misconceptions between developers and clients.
- Well-structured requirements help avoid loss of valuable data.
- The metrics evaluate the quality of the requirements document based on factors such as completeness, consistency, and unambiguity.

Reliable requirements are key to ensuring the software design and implementation align with expectations.

3. Design and Code Reliability Metrics

Design and code metrics assess the quality of the system's design and implementation.

- **Complexity**: High complexity increases the chance of errors.
- **Size**: Larger modules are harder to test and maintain.
- **Modularity**: Well-structured modular systems are more reliable.
- **Object-Oriented Metrics**: Include coupling and cohesion.

Balancing complexity, size, and modularity improves reliability.

4. Testing Reliability Metrics

Testing reliability metrics focus on ensuring the system meets its requirements and works as expected.

- **Functionality Coverage**: Ensure all requirements are met by testing for functionality.
- **Bug Detection and Fixing**: Identify and fix bugs through testing.

These metrics ensure the system performs reliably under real-world conditions.

Conclusion

By applying software reliability metrics across various stages (requirements, design, code, and testing), you can ensure the system is built with high reliability, low risk of failure, and effective handling of failures when they occur.

Software Reliability Models

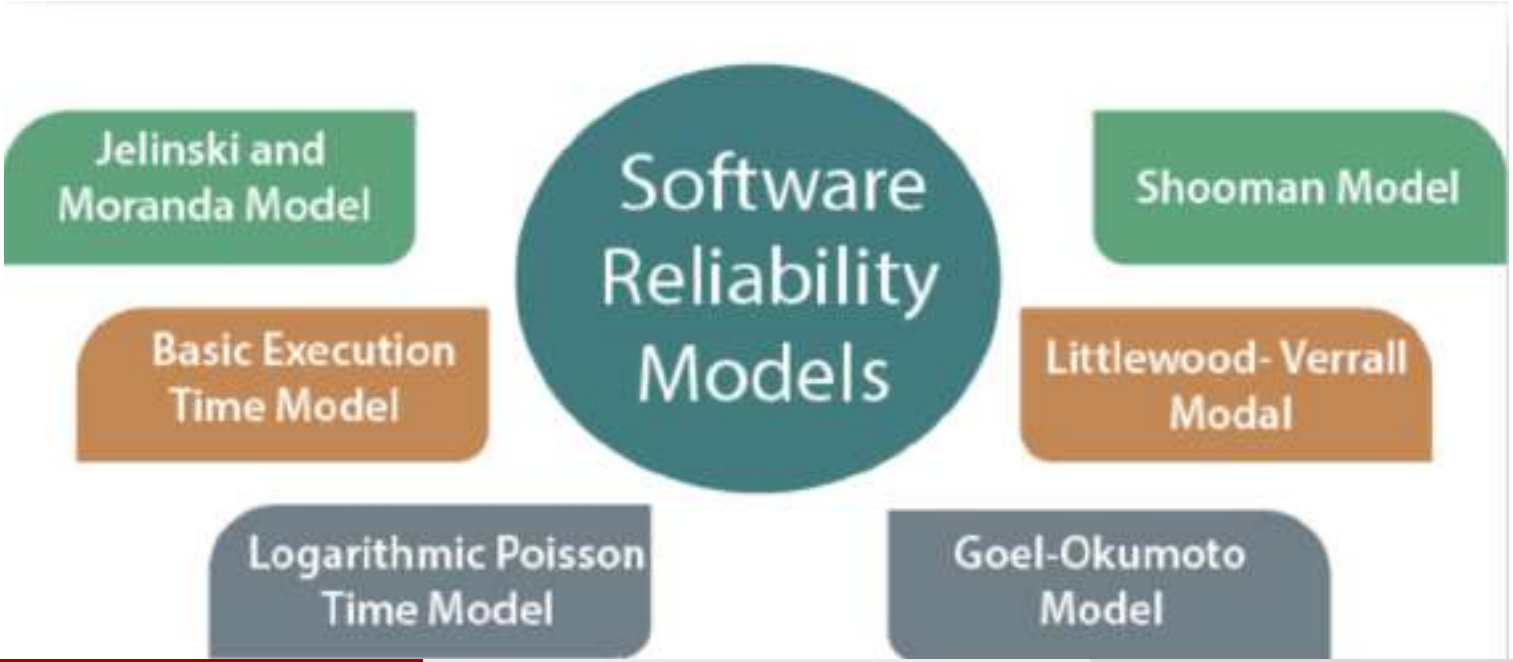
- A software reliability model describes how software failures occur over time.
- These models help us understand why software fails and try to measure reliability.
- Since the 1970s, over 200 models have been created, but no universal solution exists.
- No single model works for all cases; each has its limits.
- Most models include:
  - Assumptions
  - Influencing factors
  - A mathematical function (often exponential or logarithmic)

Prediction vs. Estimation Models in Software Reliability

Basics	Prediction Models	Estimation Models
Data Reference	Use historical data	Use data from current project
When Used	Early in the lifecycle (e.g., concept or design phase)	Later in the lifecycle (after data collection begins)
Time Frame	Predict future reliability	Estimate current or near-future reliability

Reliability Models

- A reliability growth model is a mathematical model that shows how software reliability improves over time.
- As errors are found and fixed, the model predicts increasing reliability.
- These models help project managers decide how much effort to invest in testing.
- The goal is to continue testing and debugging until the software reaches the required reliability level.



Jelinski-Moranda (JM) Model

- One of the earliest and most influential software reliability models.
- Based on a Markov process and assumes perfect debugging.
- Assumes a fixed initial number of faults, N, with constant fault detection rate ϕ.
- Each failure reduces the number of remaining faults by one.

Key Equations:

$$\mu(t_i) = N\left(1 - e^{-\phi t_i}\right) \quad \text{Mean Value Function}$$

$$\varepsilon(t_i) = N\phi e^{-\phi t_i} \quad \text{Failure Intensity Function}$$

$$\lambda(t_i) = \phi[N - (i-1)] \quad \text{Failure Rate at } i^{\text{th}} \text{ interval}$$

Reliability Measures of JM Model

Measure	**Formula**
Probability Density Function (PDF)	$f(t_i) = \phi[N-(i-1)]e^{-\phi[N-(i-1)]t_i}$
Software Reliability Function	$R(t_i) = e^{-\phi[N-(i-1)]t_i}$
Failure Rate Function	$\lambda(t_i) = \phi[N-(i-1)]$
Mean Time to Failure (MTTF)	$\text{MTTF}_i = \frac{1}{\lambda(t_i)}$
Mean Value Function	$\mu(t_i) = N(1-e^{-\phi t_i})$
Failure Intensity Function	$\varepsilon(t_i) = N\phi e^{-\phi t_i}$
Median Time to Failure	$m = \frac{\ln 2}{\phi[N-(i-1)]}$
Cumulative Distribution Function (CDF)	$F(t_i) = 1-e^{-\phi[N-(i-1)]t_i}$

Assumptions of the JM Model

- Initial number of software faults N is fixed and constant.
- Faults are independent and equally likely to cause failure.
- Time between failures follows exponential distribution.
- Each detected fault is removed perfectly (no new faults introduced).
- The failure rate decreases as faults are removed.
- It is a binomial-type black-box model.
- Known to produce optimistic reliability predictions.

Original JM Model Assumptions

- Fixed number of initial faults (N)
- All faults contribute equally to failure rate
- Faults removed perfectly upon detection
- Failure rate decreases linearly with each fix

Real-world scenarios often violate these assumptions

1. Generalized Jelinski-Moranda (GJM) Model

Modifications

- Different fault detection rates (ϕ_i)
- Accounts for varying fault severities

$$\lambda(t_i) = \sum_{j=1}^{N-(i-1)} \phi_j$$

- ϕ_j = fault detection rate for j-th fault
- Use Case: Faults with different criticality levels

2. Moranda Geometric Model (1975)

Modifications

- Failure rate decreases geometrically
- Diminishing returns in debugging

$$\lambda(t_i) = D \cdot k^{i-1}$$

- D = initial failure intensity
- k = reduction factor $(0 < k < 1)$
- Use Case: Later fixes have smaller impact

3. Schick-Wolverton (SW) Model (1978)

Modifications

- Time-dependent failure rate
- Rate increases between fixes

$$\lambda(t_i) = \phi \cdot (N - (i-1)) \cdot t_i$$

- t_i = time since last failure
- Use Case: Testing intensity matters

4. Littlewood-Verrall (LV) Model (1973)

Modifications

- Bayesian approach
- Accounts for debugging uncertainty

$$\lambda(t_i) = \frac{\alpha}{\beta + (i-1)}$$

- $\alpha, \beta =$ estimated parameters
- Use Case: Imperfect debugging

5. Delayed S-Shaped JM Model

Modifications

- Models learning effects
- S-curve failure rate

$$\mu(t) = N\left(1 - (1 + \phi t)e^{-\phi t}\right)$$

- Use Case: Testers improve over time

6. Modified JM with Imperfect Debugging

Modifications

- Allows imperfect fixes
- Probability p of successful fix

$$\lambda(t_i) = \phi\left(N - \sum_{j=1}^{i-1} p_j\right)$$

- p_j = success probability of j-th fix
- Use Case: Real-world debugging

Modified JM with Imperfect Debugging

- Fault removal probabilities: p, q, r with $p + q + r = 1$
- Failure Rate: $\lambda(t_i) = \phi[N - (i-1)(p-r)]$
- Reliability: $R(t_i) = e^{-\phi[N-(i-1)(p-r)]t_i}$
- MTTF: $\frac{1}{\phi[N-(i-1)(p-r)]}$

Comparison of JM Model Variations

Model	Key Modification	Failure Rate	Use Case
Original JM	Constant ϕ	Linear decrease	Simple systems
Generalized JM	Different ϕ_i	Variable decrease	Varying severity
Moranda Geometric	Geometric reduction	Exponential decrease	Diminishing returns
Schick-Wolverton	Time-dependent	Increases between failures	Testing intensity
Littlewood-Verrall	Bayesian approach	Probabilistic decrease	Imperfect debugging
S-Shaped JM	Learning effect	S-shaped curve	Tester improvement
Imperfect Debugging	Probabilistic fixes	Slower decrease	Real-world

Which Variation to Use?

- **Simple environments**: Original JM
- **Varying severity**: Generalized JM
- **Diminishing returns**: Moranda Geometric
- **Testing time matters**: Schick-Wolverton
- **Imperfect fixes**: Littlewood-Verrall
- **Tester improvement**: S-Shaped JM
- **Real-world**: Imperfect Debugging JM

Introduction

- Proposed by J.D. Musa in 1979
- Based on execution time (not calendar time)
- Most popular reliability growth model because:
  - Practical, simple, and easy to understand
  - Parameters relate to physical world
  - Accurate for reliability prediction

Key Characteristics

- Models failure behavior using execution time (τ)
- Nonhomogeneous Poisson process
- Equivalent to M-O logarithmic Poisson model (different mean value function)
- Mean value function based on exponential distribution

Key Variables

- λ - Failure intensity (failures/time unit)
- τ - Execution time
- μ - Mean failures experienced

Model Equations

Mean Failures Experienced

$$\mu(\tau) = v_0 \left(1 - e^{-\frac{\lambda_0 \tau}{v_0}}\right)$$

Failure Intensity

$$\lambda(\tau) = \lambda_0 \left(1 - \frac{\mu(\tau)}{v_0}\right) = \lambda_0 e^{-\frac{\lambda_0 \tau}{v_0}}$$

Where:

- λ_0 - Initial failure intensity
- v_0 - Total expected failures over infinite time

Reliability Projections

Additional Execution Time

Given current (λ_P) and target (λ_F) failure intensities:

$$\Delta\tau = \frac{v_0}{\lambda_0} \ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

Additional Expected Failures

$$\Delta\mu = v_0\left(\frac{\lambda_P - \lambda_F}{\lambda_0}\right)$$

Example Problem

Given:

- Total expected failures (v_0) = 200
- Current failures experienced = 100
- Initial failure intensity (λ_0) = 20 failures/CPU hr
- Target intensity (λ_F) = 5 failures/CPU hr

Find:

1. Current failure intensity
2. Decrement of failure intensity per failure
3. Failures and intensity after 20 and 100 CPU hrs
4. Additional failures and time needed to reach target

Example Solution (1-2)

1. Current Failure Intensity

$$\lambda_P = \lambda_0 \left(1 - \frac{\mu}{v_0}\right) = 20 \left(1 - \frac{100}{200}\right) = 10 \text{ failures/CPU hr}$$

2. Decrement per Failure

$$\frac{d\lambda}{d\mu} = -\frac{\lambda_0}{v_0} = -\frac{20}{200} = -0.1 \text{ failures/CPU hr per failure}$$

Example Solution (3)

3a. After 20 CPU Hours

$$\mu(20) = 200\left(1 - e^{-\frac{20\times 20}{200}}\right) = 200(1 - e^{-2}) \approx 173$$

$$\lambda(20) = 20e^{-2} \approx 2.71 \text{ failures/CPU hr}$$

3b. After 100 CPU Hours

$$\mu(100) = 200\left(1 - e^{-\frac{20\times 100}{200}}\right) = 200(1 - e^{-10}) \approx 200$$

$$\lambda(100) \approx 0 \text{ failures/CPU hr}$$

Example Solution (4)

4. Additional Requirements

$$\Delta\mu = 200\left(\frac{10-5}{20}\right) = 50 \text{ failures}$$

$$\Delta\tau = \frac{200}{20}\ln\left(\frac{10}{5}\right) = 10\ln(2) \approx 6.93 \text{ CPU hrs}$$

Summary

- Execution time-based model provides practical reliability estimates
- Exponential decay of failure intensity
- Useful for:
  - Reliability prediction
  - Test planning
  - Release decisions
- Requires good estimates of λ_0 and ν_0

Goel-Okumoto (GO) Model

- Proposed by Amrit Goel and Kazu Okumoto (1979)
- **Type**: Non-Homogeneous Poisson Process (NHPP) model
- **Key Feature**: Models reliability growth during testing
- **Assumptions**:
  - Failures occur randomly during testing
  - Failure count follows Poisson distribution
  - Failure rate decreases as faults are removed

Model Parameters

Mean Value Function

$$\mu(t) = a(1 - e^{-bt})$$

Failure Intensity Function

$$\lambda(t) = \frac{d\mu}{dt} = abe^{-bt}$$

Where:

- a: Expected total number of failures (as $t \to \infty$)
- b: Fault detection rate per fault
- t: Testing time (calendar or execution time)

Key Characteristics

- **Finite Failures**: $\lim_{t \to \infty} \mu(t) = a$
- **Exponential Decay**: Failure intensity decreases exponentially
- **Flexibility**: Can model various testing scenarios
- **Interpretability**: Parameters have clear physical meaning

Parameter Estimation

Two common methods:

Maximum Likelihood Estimation (MLE)

Solve simultaneously:

$$\frac{n}{a} = 1 - e^{-bt_n}$$

$$\frac{\sum_{i=1}^{n} t_i}{n} = \frac{1}{b} - \frac{t_n}{e^{bt_n} - 1}$$

Least Squares Estimation

Minimize:

$$\sum_{i=1}^{n} [\mu(t_i) - i]^2$$

Example Application

Given:

- Total expected faults (a) = 100
- Detection rate (b) = 0.02 per hour
- Testing time (t) = 50 hours

Calculations:

1. Expected failures:

$$\mu(50) = 100(1 - e^{-0.02 \times 50}) \approx 63$$

2. Current failure intensity:

$$\lambda(50) = 100 \times 0.02 \times e^{-0.02 \times 50} \approx 0.74 \text{ failures/hour}$$

3. Additional testing for $\lambda_F = 0.1$:

$$t_F = \frac{1}{0.02} \ln\left(\frac{1.48}{0.1}\right) \approx 136 \text{ hours}$$

Comparison with Other Models

Model	**Failures**	**Intensity Trend**
JM	Finite	Linear decrease
BET	Finite	Exponential decrease
GO	Finite	Exponential decrease
Musa-Okumoto	Infinite	Logarithmic decrease

Table: Comparison with Other Reliability Models

Advantages & Limitations

Advantages

- Simple yet effective for reliability prediction
- Parameters have clear interpretations
- Widely used in industry applications

Limitations

- Assumes perfect fault removal
- Doesn't account for fault severity
- Requires sufficient failure data for estimation

Practical Applications

- **Test Planning**: Estimate required testing time
- **Release Decisions**: Assess if reliability targets are met
- **Resource Allocation**: Optimize debugging efforts
- **Reliability Prediction**: Forecast field performance

Summary

- NHPP model with finite failures
- Exponential failure intensity decay
- Parameters: a (total faults) and b (detection rate)
- Useful for reliability growth modeling
- Foundation for many advanced models

Musa-Okumoto Logarithmic Model

- Developed by John Musa and Kazu Okumoto (1984)
- **Type**: Non-Homogeneous Poisson Process (NHPP)
- **Key Feature**: Models reliability growth with infinite failures
- **Applications**:
  - Large, complex software systems
  - Long-term reliability assessment
  - Systems with diminishing debug effectiveness

Model Formulation

Mean Value Function

$$\mu(t) = \frac{1}{\theta} \ln(\lambda_0 \theta t + 1)$$

Failure Intensity Function

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \theta t + 1}$$

Where:

- λ_0: Initial failure intensity
- θ: Reduction in failure intensity per failure
- t: Execution time

Key Characteristics

- **Infinite Failures**: $\lim_{t\to\infty} \mu(t) = \infty$
- **Logarithmic Growth**: Failure count grows logarithmically
- **Diminishing Returns**: Later fixes have less impact
- **Self-Healing**: Accounts for fault masking effects

Parameter Estimation

Maximum Likelihood Estimation

Solve:

$$\frac{n}{\lambda_0} = \sum_{i=1}^{n} \frac{t_i}{1 + \lambda_0 \theta t_i}$$

$$\frac{n}{\theta} = \sum_{i=1}^{n} \frac{\lambda_0 t_i}{1 + \lambda_0 \theta t_i} + \sum_{i=1}^{n} \ln(1 + \lambda_0 \theta t_i)$$

Practical Approach

1. Collect failure interval data $(t_1, t_2, ..., t_n)$
2. Use numerical methods to solve MLE equations
3. Validate with goodness-of-fit tests

Example Application

Given:

- $\lambda_0 = 10$ failures/CPU hr
- $\theta = 0.02$ reduction factor
- Current $t = 100$ CPU hours

Calculations:

1. Expected failures:

$$\mu(100) = \frac{1}{0.02} \ln(10 \times 0.02 \times 100 + 1) \approx 153$$

2. Current failure intensity:

$$\lambda(100) = \frac{10}{10 \times 0.02 \times 100 + 1} \approx 0.48 \text{ failures/CPU hr}$$

3. Additional time for $\lambda_F = 0.1$:

$$t_F = \frac{1}{10 \times 0.02} \left(\frac{10}{0.1} - 1 \right) = 49.5 \text{ CPU hrs}$$

Comparison with GO Model

Characteristic	**Musa-Okumoto**	**Goel-Okumoto**
Failure Assumption	Infinite	Finite
Mean Value Function	Logarithmic	Exponential
Failure Intensity	$O(1/t)$	$O(e^{-t})$
Parameter Count	2 (λ_0, θ)	2 (a, b)
Best For	Long-term	Short-term

Table: Model Comparison

Advantages

- **Realistic**: Captures diminishing debug effectiveness
- **Flexible**: Handles infinite failure scenarios
- **Practical**: Parameters have physical meaning
- **Analytic**: Closed-form solutions available
- **Evolvable**: Can model changing operational profiles

Limitations

- **Complex Estimation**: Requires numerical methods
- **Data Hungry**: Needs sufficient failure data
- **Over-Prediction**: May overestimate long-term failures
- **Debug Assumption**: Doesn't model imperfect fixes
- **Time Scale**: Requires careful time unit selection

Practical Applications

- **System Maintenance**: Predict long-term support needs
- **CI/CD Pipelines**: Assess reliability growth in DevOps
- **Safety-Critical Systems**: Conservative reliability estimates
- **Legacy Systems**: Model aging software behavior
- **Resource Planning**: Forecast testing/debugging effort

Extensions & Variants

- **Generalized MO**: Additional shape parameters
- **Time-Dependent** θ: Variable debug effectiveness
- **Coverage-Based**: Incorporates operational profile
- **Hybrid Models**: Combines with GO model features
- **Bayesian Versions**: Incorporates prior knowledge

Summary

- NHPP model with infinite failure assumption
- Logarithmic failure growth pattern
- Parameters: λ_0 (initial intensity), θ (reduction factor)
- Suitable for complex, long-lived systems
- Foundation for many advanced reliability models

When to Use

When you need conservative estimates for systems where:

- Faults become harder to find over time
- Complete fault removal is unrealistic
- Long-term reliability assessment is needed

What is FMEA?

Definition

A step-by-step approach to identify all possible failures in a:

- Design (Design FMEA)
- Process (Process FMEA)
- System (System FMEA)

- Developed in 1940s by U.S. military
- Widely used in automotive, aerospace, healthcare
- **Purpose**: Prevent failures before they occur

Types of FMEA

Type	**Application**
System FMEA	Entire systems and interactions
Design FMEA	Products before production
Process FMEA	Manufacturing/assembly processes
Service FMEA	Service delivery processes
Software FMEA	Software systems and code

The FMEA Process

1. **Define** scope and objectives
2. **Assemble** cross-functional team
3. **Identify** components/process steps
4. **List** potential failure modes
5. **Analyze** effects and causes
6. **Assign** severity, occurrence, detection ratings
7. **Calculate** Risk Priority Numbers (RPN)
8. **Recommend** corrective actions
9. **Implement** improvements
10. **Reassess** RPN after changes

Risk Priority Number (RPN)

$$RPN = \text{Severity} \times \text{Occurrence} \times \text{Detection}$$

Rating Scales (1-10)

- **Severity**: Impact of failure
- **Occurrence**: Frequency of cause
- **Detection**: Likelihood of detection

Example

- Severity: 8
- Occurrence: 3
- Detection: 5
- RPN $= 8 \times 3 \times 5 = 120$

Case Study: Automotive Brake System

- **Component**: Brake hydraulic line
- **Failure Mode**: Fluid leakage
- **Effects**:
  - Reduced braking power
  - Potential accident
- **Causes**:
  - Corrosion
  - Improper installation

RPN Calculation

- Severity: 9
- Occurrence: 3
- Detection: 2
- RPN = 54

Recommended Actions

- Use corrosion-resistant materials
- Implement installation torque checks
- Add pressure testing step

Benefits of FMEA

- **Proactive** risk identification
- **Systematic** approach to failures
- **Cross-functional** team involvement
- **Documented** risk assessment
- **Prioritized** improvement actions
- **Reduced** warranty costs
- **Improved** customer satisfaction

Limitations

- **Subjective** ratings
- **Time-consuming** process
- **Static** analysis (snapshot in time)
- **Overemphasis** on RPN values
- **Difficult** to predict all failures
- **Requires** expert knowledge

Best Practices

- **Start early** in design/process
- **Involve** diverse team members
- **Focus** on high RPN items first
- **Use** consistent rating scales
- **Combine** with other tools (FTA, DOE)
- **Update** regularly
- **Document** assumptions

Software Tools

- **ReliaSoft** XFMEA
- **Siemens** Teamcenter FMEA
- **IQ-FMEA**
- **EtQ** Reliance
- **Excel-based** templates

Emerging Trends

- AI-assisted FMEA
- Integration with PLM systems
- Automated RPN calculations

Summary

- FMEA is a **powerful preventive** tool
- **Systematic approach** to failure analysis
- **RPN helps prioritize** risks
- **Requires team commitment**
- **Living document** that needs updates
- **Complement** with other quality tools

Final Recommendation

"Incorporate FMEA early in your design and process development cycles to maximize its effectiveness."

Unit 4: Software Availability (4 Hours)

What is Software Availability?

Software Availability refers to the **degree to which a software system or service is operational and accessible when required for use.** It's a core measure of a system's **dependability** and **quality of service (QoS).**

It answers the question:

"**Is the system available when the user needs it?**"

It's typically expressed as a **percentage** over a given time period:

$$\textbf{Availability (A)} = \textbf{(Uptime / (Uptime + Downtime))} \times \textbf{100}$$

Importance of Availability in System Dependability

1. User Trust and Experience

- Systems that are unavailable frustrate users and erode trust.
- Even highly reliable systems can be **useless** if they are **not available** when required.

Example: A reliable online banking app that's down during payday is a failure in terms of user experience.

2. Business Continuity and Revenue

- Downtime directly impacts revenue, productivity, and customer satisfaction.
- Critical business operations rely on systems being accessible.

Example: Amazon can lose **millions per minute** of downtime during peak sales.

3. Foundation for Fault Tolerance

- Availability includes recovery time; hence it drives investment in:
  - **Redundancy**
  - **Failover systems**
  - **Self-healing infrastructure**

These technologies are vital for building **fault-tolerant** dependable systems.

4. Time-Critical Operations

- In real-time systems (e.g., air traffic control, healthcare), availability is essential to prevent disasters.

Example: An unavailable emergency dispatch system could cost lives.

5. Quantifiable SLA Guarantees

- Availability can be **measured** and included in **Service Level Agreements (SLAs)**.
- SLAs guarantee performance and **bind service providers** to certain uptime commitments (e.g., 99.99%).

6. Links with Maintainability

- A system might **fail often**, but if it's **restored quickly**, availability remains high.
- Availability highlights the **system's recoverability** and **repair efficiency**.

7. Dependency on Other Dependability Attributes

- High availability usually implies:
  - Good **reliability** (few failures)
  - Good **maintainability** (fast recovery)
  - Adequate **security** (protected from attacks)

Thus, it integrates and **reflects the effectiveness** of other dependability properties.

8. Availability Impacts Redundancy Design

- To meet availability targets (e.g., 99.999%), systems must use:
  - **Hot or cold standby systems**
  - **Clustering(group acting as a single system)**
  - **Load balancing**
  - **Distributed architectures**

This directly affects **system architecture decisions** and **costs**.

Availability Metrics:

Availability metrics provide **quantitative ways** to measure and analyze how often a software system or service is available for use. These metrics are essential for:

- Performance evaluation
- SLA (Service Level Agreement) compliance
- System reliability and design optimization
- Business continuity planning

These metrics help answer questions like:

- *How often does the system fail?*
- *How long does it take to recover?*

- *What is the impact of downtime on users and operations?*

1. Uptime

Definition:

Uptime is the total amount of time a system remains operational without any interruption.

Measured in:

- Hours, minutes, or seconds
- Typically over a day, week, month, or year

Example:

If a system runs for 23 hours without failure in a 24-hour window, uptime = 23 hours.

Downtime

Definition:

Downtime is the period when a system is **not available** due to failures, crashes, maintenance, or other issues.

Causes of Downtime:

- Software bugs
- Server crashes
- Network outages
- Cyberattacks
- Planned maintenance

Types:

- **Planned Downtime**: For upgrades, backups, patches
- **Unplanned Downtime**: Failures, crashes, security breaches

3. Availability Ratio

$$\textbf{Availability (A) = (Uptime / (Uptime + Downtime)) × 100}$$

Example:

If uptime = 720 hours, downtime = 2 hours:

A= 99.72%

Along with these three above metrics, the software availability also use most of the metrics like MTTF, MTTR, MTBF and Availability. Please refer the previously done numerical on those topics.

Mathematical Models of Availability

A **mathematical model of availability** is a formal way of quantifying how **often** and **for how long** a system is operational, using **probabilistic, statistical, and stochastic techniques**. These models help predict and optimize **uptime**, **recovery**, and **failure response** in software systems.

They answer:

- *What percentage of time is the system available?*
- *How often does it fail, and how quickly does it recover?*
- *What is the steady-state (long-term) behavior of system availability?*

1. Basic Availability Model

This is the **most commonly used model**, especially for systems with **constant failure and repair rates**.

Formula:
Availability (A) =MTBF/MTBF+MTTR

Where:

- **MTBF (Mean Time between Failures)**: Average operational time before a failure occurs.
- **MTTR (Mean Time to Repair)**: Average time taken to fix a failure.

2. Markov Model (Two-State Continuous-Time)

Markov models are used when systems switch **randomly** between **operational and failed states**, and the transition rates follow **exponential distributions**.

System States:

- **State 0 (Up)**: System is operational.
- **State 1 (Down)**: System is failed.

Transition Rates:

- λ: Failure rate (transitions from Up → Down)
- μ: Repair rate (transitions from Down → Up)

Availability Equation: A = μ/λ+ μ

This is **mathematically equivalent** to the basic MTBF/MTTR formula if:

- MTBF = 1/λ
- MTTR = 1/μ

3. Multi-State Markov Models

Used for **complex software systems** with:

- **Multiple degradation levels**

- **Sub-component failures**
- **Redundant systems**

Example States:

State	Description
S0	Fully operational
S1	Degraded (partial failure)
S2	Failed completely

Transitions between states have rates:

- λ_1, λ_2 for failures
- μ_1, μ_2 for repairs

To calculate availability:

1. Write all **balance equations**
2. Solve for **steady-state probabilities**
3. Sum the probabilities of all **"operational" states**

These models require solving **systems of linear equations** and are often computed using tools like **SHARPE**, **MATLAB**, or **PRISM**.

4. Instantaneous and Steady-State Availability

Type	Definition
Instantaneous A(t)	Availability at an exact moment t. Depends on time-dependent behavior.
Steady-State A(∞)	Long-run average availability. Assumes many failure-repair cycles have occurred.

Most systems use **steady-state availability** for SLAs and design decisions.

5. Redundant System Models (Parallel Systems)

In **redundant software systems**, availability is modeled using combinations of component availabilities.

Remember the formulas for Series System and Parallel System

6. Downtime Estimation from Availability

Given availability A, total time T:

$$\text{Downtime} = T \times (1 - A)$$

Example:

For A=99.9%, over a year (8760 hours):

$$\text{Downtime} = 8760 \times (1 - 0.999) = 8.76 \text{ hours/year}$$

Conclusion:

 Mathematical models of availability help **design, evaluate, and optimize** software systems.

 Choosing the right model depends on:

- Complexity of system
- Type of failure/recovery
- Need for time-specific or long-run estimates

 These models form the basis for achieving **highly available**, **dependable**, and **cost-effective** software solutions.

Unit 5: Software Safety (5 hours)

Software safety refers to the process of ensuring that software systems operate without causing unacceptable risk of harm to people, equipment, or the environment. It focuses on identifying, analyzing, and mitigating potential hazards that could result from software failures or unintended behavior, especially in critical systems such as those used in aviation, healthcare, automotive, military, and nuclear industries.

Explanation:

Software safety is not just about preventing bugs or crashes—it's about making sure that even if something goes wrong, the system doesn't lead to dangerous outcomes. For example:

- In a **medical device**, software safety ensures that a drug pump doesn't overdose a patient if a sensor fails.
- In a **car's braking system**, safety measures ensure the brakes still function properly even if part of the software malfunctions.

Key elements of software safety include:

- **Hazard analysis**: Identifying what could go wrong and what the consequences would be.
- **Risk assessment**: Determining how likely and severe those consequences are.
- **Design for safety**: Building the software to avoid or control hazards.
- **Verification and validation**: Making sure the software meets safety requirements.
- **Compliance**: Following safety standards like ISO 26262 (automotive), DO-178C (aviation), or IEC 62304 (medical).

Safety Critical Sytems

A **safety-critical system** is a system whose failure or malfunction may result in one or more of the following:

- **Death or serious injury** to people
- **Loss or severe damage** to equipment or property
- **Environmental harm**

These systems are often used in domains like **aerospace, defense, nuclear power, automotive, railways, and medical devices**. Because of the potential consequences, safety-critical systems require extremely high reliability, rigorous testing, formal verification methods, and compliance with strict safety standards.

Detailed Explanation of Safety-Critical Systems

1. Examples of Safety-Critical Systems:

Domain	Safety-Critical System Example	Risk If It Fails
Aviation	Flight control software	Plane crash
Automotive	Airbag deployment system	Injury or death in accident
Medical	Infusion pump software	Incorrect medication dosage
Nuclear	Reactor cooling control system	Nuclear meltdown

2. Key Characteristics:

- **High assurance level**: Must work as intended under all conditions.
- **Fail-safe design**: System must revert to a safe state if failure occurs.
- **Redundancy**: Multiple systems may perform the same task to ensure safety.

- **Verification & validation**: Must go through thorough analysis, simulation, and real-world testing.
- **Regulatory compliance**: Must follow industry-specific standards (e.g., DO-178C, ISO 26262, IEC 61508).

Typical Safety Process:

1. **Hazard identification**
2. **Risk assessment**
3. **Requirements specification**
4. **Design with safety in mind**
5. **Implementation using best practices**
6. **Verification and validation**
7. **Certification and audit**

Numerical Example

Problem Statement:
A medical ventilator has a software component that monitors oxygen levels. If oxygen levels drop below a threshold and no alarm is raised within **2 seconds**, the system is considered to have failed. Assume the following:

- The failure rate of the oxygen sensor is $\lambda = 2\times10^{-6}$ failures/hour.
- The failure rate of the software alarm module is $\lambda = 1\times10^{-6}$ failures/hour.
- The system is **series-connected**, meaning both must work for the system to work.
- Evaluate the **Mean Time between Failures (MTBF)** for the **combined safety-critical function**.
- Also, calculate the **probability of failure on demand (PFD)** for a **1-hour operation**, assuming no repairs during the hour.

Before solving this question lets understand what PFD is.

What is PFD?

***PFD** stands for **Probability of Failure on Demand**. It measures the **likelihood that a safety system will fail to perform its intended function when it's needed**—typically **on demand**, rather than during continuous operation.*

In simpler terms:

*If something goes wrong and you need the system to respond (like triggering an alarm or shutting down a machine), PFD tells you how likely it is that the system **won't** work correctly at that critical moment.*

Where is PFD Used?

*PFD is mainly used in **low-demand safety systems**, such as:*

- *Emergency shutdown systems*
- *Fire suppression systems*
- *Alarm systems in medical devices*
- *Backup systems in industrial processes*

How is PFD Calculated?

*For systems with **constant failure rates** and **no repair during the time interval**, a basic approximation is:*

$PFD \approx \lambda \times t$

Where:

- *λ is the **failure rate** (failures per hour)*
- *t is the **time interval** during which the system might be needed (hours)*

*This approximation is valid when $\lambda \times t \ll 1$, which is usually the case for safety-critical systems with **very low failure rates**.*

Example Calculation (from earlier):

*You have a system with a **combined failure rate**:*

$\lambda_{total} = 3 \times 10^{-6}$ *failures/hour*

*You want to know the PFD over a **1-hour period**:*

$PFD = \lambda_{total} \times t = 3 \times 10^{-6} \times 1 = 3 \times 10^{-6}$

*So, the **probability that the system fails when needed in 1 hour** is **0.000003**, or **0.0003%**.*

Now let's solve the previous question.

Solution:

Step 1: Combined failure rate (λtotal):
Since both components must work (series system), total failure rate:

$\lambda_{total} = \lambda_{sensor} + \lambda_{alarm} = 2 \times 10^{-6} + 1 \times 10^{-6} = 3 \times 10^{-6}$ failure / hour

Step 2: MTBF (Mean Time between Failures):

$MTBF = 1/\lambda total = 1/3 \times^{10-6} \approx 333{,}333$ hours

Step 3: Probability of Failure on Demand (PFD) for 1 hour:

Assuming constant failure rate and short interval:

$PFD \approx \lambda_{total} \times t = 3 \times 10_{-6} \times 1 = 3 \times 10^{-6}$

So, there's a **0.0003% chance** the system fails during a 1-hour period.

Safety Requirements and Hazard Analysis

Safety requirements are essential elements in the design, operation, and maintenance of systems, equipment, workplaces, and processes to ensure the protection of people, property, and the environment.

Definition of Safety Requirements

Safety requirements are **specifications or conditions** that a system, process, or product must meet to prevent accidents, minimize risk, and comply with laws or standards. These requirements ensure hazards are identified and mitigated through engineering, administrative, or procedural controls.

Purpose of Safety Requirements

- **Prevent harm to personnel** (injuries, illnesses, fatalities)
- **Protect the environment** (avoid contamination, pollution)
- **Safeguard equipment and assets**
- **Ensure compliance** with local, national, or international laws and regulations
- **Promote safe working conditions** and public confidence

Detailed Overview of **Software Safety Requirements**

1. Definition

Software Safety Requirements are specifications that ensure the software will function correctly and predictably in all situations that could impact safety. These requirements are intended to **prevent software-induced hazards**, **detect potential failures**, and **mitigate the impact** of software faults.

2. Importance of Software Safety Requirements

- **Prevent system-level hazards** caused by software errors
- Ensure **compliance** with safety standards (e.g., ISO 26262 for automotive, DO-178C for aviation)
- Facilitate **verification and validation (V&V)** of safety-critical software
- Reduce **operational risk** by ensuring reliable system behavior under fault conditions

3. Sources of Software Safety Requirements

Software safety requirements are typically derived from:

- **System-level hazard analysis** (e.g., FMEA, FTA, HAZOP)
- **Functional safety standards** (e.g., IEC 61508, ISO 26262, IEC 62304)
- **Safety integrity levels (SILs)**, **Automotive Safety Integrity Levels (ASIL)**, or other risk classifications
- **Interface requirements** with hardware and human operators

4. Types of Software Safety Requirements

A. Functional Safety Requirements

Describe specific behavior needed to maintain safety:

- "The software shall shut down the motor if the temperature exceeds 100°C."
- "The software shall trigger an alarm if pressure drops below 20 psi."

B. Non-Functional Safety Requirements

Address how the software performs under certain conditions:

- **Reliability**: "The software shall have an MTBF (Mean Time Between Failures) of at least 10,000 hours."

- **Response Time**: “The software shall respond to emergency input within 100 ms.”
- **Fault Tolerance**: “The software shall continue safe operation in the presence of a sensor failure.”

C. Design and Architectural Requirements

Ensure the software structure promotes safety:

- Use of **redundant computation paths**
- Separation of safety-critical code from non-critical code
- Use of **watchdogs** and **sanity checks**

D. **Process Requirements**

Specify how software should be developed and tested:

- Code must follow a **strict coding standard** (e.g., MISRA C)
- Safety-critical functions must be subject to **formal verification**
- All safety requirements must be **traced** from system-level hazards

5. Examples of Software Safety Requirements

Hazard	Software Safety Requirement
Patient overdose from infusion pump	"The software shall limit the maximum infusion rate to 25 ml/hr."
Autonomous car steering failure	"The software shall initiate a safe stop if sensor input is lost for more than 500 ms."
Industrial robot collision	"The software shall stop robot movement if a human is detected within 1 meter."

6. Life Cycle of Software Safety Requirements

1. **Hazard Analysis & Risk Assessment**
   → Identify what could go wrong.
2. **Requirements Derivation**
   → Define what the software must do (or not do) to prevent those hazards.
3. **Specification**
   → Write the requirements in clear, unambiguous, testable terms.
4. **Design & Implementation**
   → Architect the system to enforce safety behavior.
5. **Verification & Validation**
   → Prove through tests, reviews, and analysis that the software meets safety requirements.
6. **Maintenance & Change Management**
   → Ensure updates don't violate original safety intent (via regression testing, impact analysis).

7. Tools and Standards

Domain	Standard	Tool Example
Automotive	ISO 26262	Vector CAST, Polyspace
Aviation	DO-178C	SCADE, LDRA
Industrial	IEC 61508	Simulink, CodeSonar
Medical Devices	IEC 62304	IBM Rational DOORS, TortoiseSVN

8. Challenges in Implementing Software Safety Requirements

- **Ambiguity**: Poorly defined requirements lead to misinterpretation.
- **Complexity**: As systems become more complex, tracking safety logic becomes harder.
- **Change management**: Updates to software must be re-evaluated for safety.
- **Emergent behavior**: Interaction of multiple safe components might still lead to unsafe outcomes.

9. Best Practices

- Use **model-based design** and **simulation** early in development.
- Maintain strict **traceability** from system-level hazards to software requirements and test cases.
- Apply **formal methods** for high-assurance systems (e.g., theorem proving, static analysis).
- Conduct regular **safety audits and reviews**.

Hazard Analysis

Hazard analysis is a critical component of **software dependability** and **safety management**. It involves identifying potential sources of danger (hazards) that could arise from the malfunction or misuse of a software system, and determining how these hazards can be eliminated or controlled to ensure system safety and reliability.

What is Hazard Analysis?

Hazard analysis is the **systematic process of identifying, evaluating, and documenting hazards**—conditions or behaviors that could lead to accidents or undesired consequences. In the context of software systems, this means finding where and how **software behavior could lead to unsafe conditions**, especially in **safety-critical systems**.

Importance in Software Dependability and Safety Management

1. Improves Software Dependability

- Ensures **correct and predictable** software behavior under normal and abnormal conditions.
- Enables the design of **fail-safe**, **fault-tolerant**, and **robust systems**.

- Reduces **unexpected behavior** from software faults or unexpected interactions.

2. Core of Safety Management

- Forms the foundation for **safety requirements**, **mitigation strategies**, and **system design**.
- Provides traceability from **hazards** to **software requirements**, tests, and validation activities.
- Essential for **regulatory compliance** in industries like aerospace (DO-178C), automotive (ISO 26262), and medical (IEC 62304).

Key Concepts in Hazard Analysis

Term	Definition
Hazard	A potential source of harm or adverse effect
Risk	The combination of the likelihood and severity of a hazard
Cause	The initiating event or condition leading to a hazard
Mitigation	An action or control to reduce the risk associated with a hazard
Accident	An actual event where a hazard results in damage or injury

Types of Hazard Analysis Techniques

1. Preliminary Hazard Analysis (PHA)

- Conducted early in development.
- Identifies general system-level hazards.
- Helps in defining **high-level safety requirements**.

2. Failure Modes and Effects Analysis (FMEA)

- Analyzes possible **failure modes** of system components (including software).
- Determines the effect of each failure and its severity.
- Calculates **Risk Priority Number (RPN)** for prioritization.

3. Fault Tree Analysis (FTA)

- Top-down method that starts with a hazard and works backward to find causes.
- Uses logic gates to show how combinations of faults could lead to a hazard.

4. Hazard and Operability Study (HAZOP)

- Structured, qualitative method.
- Typically used in process industries.
- Explores deviations from normal operation using guidewords.

5. Software Hazard Analysis (SHA)

- Focuses specifically on software interactions.
- Identifies how software logic or failures can cause or contribute to hazards.

Process of Hazard Analysis

Step 1: System Understanding

- Define the system boundaries, interfaces, and intended functions.
- Identify all software components and their interactions with hardware and users.

Step 2: Hazard Identification

- List potential hazardous conditions related to software (e.g., incorrect output, timing errors, data corruption).
- Use checklists, past incidents, expert judgment, or modeling tools.

Step 3: Hazard Evaluation

- Assess each hazard in terms of:
  - **Severity** (how serious the outcome is)
  - **Likelihood** (how often it could happen)
- Tools: risk matrices, scoring systems

Step 4: Derivation of Safety Requirements

- Translate hazards into **software safety requirements**.
  - E.g., "The software shall shut off power if temperature exceeds threshold for more than 2 seconds."

Step 5: Mitigation and Control

- Apply strategies like:
  - **Redundancy** (e.g., backup systems)
  - **Software constraints** (e.g., input validation, range checking)
  - **Alarms and alerts**
  - **Safe states** (e.g., software shuts down gracefully)

Step 6: Documentation and Traceability

- Record all hazards, causes, consequences, and mitigation measures.
- Link each hazard to corresponding software requirements and tests.

Example: Software Hazard Analysis in a Medical Infusion Pump

Hazard	Cause	Consequence	Mitigation(relieve)
Overdose	Software miscalculates infusion rate	Harm to patient or death	Limit rate in software, double-check logic, alert if rate exceeds threshold
No delivery	Timer malfunction	Treatment interruption	Watchdog timer, alert on failure, backup delivery mode

Hazard	Cause	Consequence	Mitigation(relieve)
Wrong drug	Incorrect UI input	Administration of wrong medication	Confirmation dialog, barcode scanning, double-entry verification

Role in the Software Safety Lifecycle

Hazard analysis influences every phase of the **software safety lifecycle**:

Phase	Role of Hazard Analysis
Requirements	Derives safety-related requirements
Design	Guides architectural choices (redundancy, fault isolation)
Implementation	Ensures safe coding practices and defensive programming
Verification	Validates that safety mitigations work correctly
Maintenance	Assesses impact of changes on existing hazards

Conclusion

Hazard analysis is **vital** for ensuring the **dependability** and **safety** of software systems, especially where human lives or critical infrastructure are involved. By systematically identifying and managing hazards, it allows developers and engineers to:

- Proactively design out risks
- Build safer software systems
- Comply with international safety standards
- Increase user and regulatory trust

Safety Engineering Processes: Design, Implementation, Testing

Software is the set of instructions in the form of programs to govern the computer system and to process the hardware components. To produce a software product the set of activities is used. This set is called a software process.

What are Software Processes?

Software processes in software engineering refer to the methods and techniques used to develop and maintain software. Some examples of software processes include:

- **Waterfall**: a linear, sequential approach to software development, with distinct phases such as requirements gathering, design, implementation, testing, and maintenance.
- **Agile**: a flexible, iterative approach to software development, with an emphasis on rapid prototyping and continuous delivery. Based on Software Manifesto, a set of principles for software development that prioritize individuals and interactions, working software, customer collaboration and responding to change.
- **Scrum**: a popular Agile methodology that emphasizes teamwork, iterative development, and a flexible, adaptive approach to planning and management.
- **DevOps**: a set of practices that aims to improve collaboration and communication between development and operations teams, with an emphasis on automating the software delivery process.

SAFETY ENGINEERING PROCESSES

1. Design Phase

The design phase aims to create an architecture and components that minimize risk and support fault detection, containment, and recovery.

Key Concepts:

- **Hazard Analysis**
  - Identify potential hazards that the software might introduce or fail to mitigate.
  - Techniques: FMEA (Failure Modes and Effects Analysis), HAZOP, Fault Tree Analysis (FTA).
- **Safety Requirements Specification**
  - Define functional and non-functional safety requirements.

    - Include constraints (e.g., real-time deadlines, input validation) and safe states.
- **Design for Safety**
    - Use **safety patterns** and **defensive design**.
    - Apply **redundancy** (hardware/software), **diversity** (e.g., N-version programming), and **partitioning** (to isolate faults).
- **Formal Methods**
    - Mathematically prove that a design satisfies safety properties.
    - Examples: model checking, theorem proving (e.g., using TLA+, Z).
- **Safe State Design**
    - Ensure the system enters a safe state upon failure (fail-safe or fail-operational).
- **Separation of Concerns**
    - Use architectural styles like layered architecture to isolate safety-critical components from non-critical ones.

2. Implementation Phase

This phase turns design into code, with a focus on preserving safety goals and minimizing the introduction of errors.

Key Concepts:

- **Safe Programming Languages and Practices**
    - Use type-safe, memory-safe languages (e.g., Ada, SPARK, MISRA C).
    - Avoid unsafe constructs (e.g., pointer arithmetic in C).
- **Coding Standards**
    - Enforce guidelines that reduce fault likelihood (e.g., MISRA, CERT).
- **Static Analysis**
    - Analyze source code without executing it to detect bugs, vulnerabilities, or violations of coding standards.

- **Fault Containment & Error Handling**
  - Implement robust exception handling.
  - Ensure the software can gracefully degrade or switch to redundant systems.
- **Modularization and Encapsulation**
  - Isolate faults to prevent propagation.
- **Traceability**
  - Maintain traceability from requirements → design → code → tests.

3. Testing and Verification Phase

The goal is to validate that the software meets its safety requirements and behaves correctly under all (even faulted) conditions.

Key Concepts:

- **Verification vs. Validation**
  - *Verification*: Are we building the product right?
  - *Validation*: Are we building the right product?
- **Safety Testing Techniques**
  - **Unit Testing**: Isolated testing of components.
  - **Integration Testing**: Ensure safe interactions between modules.
  - **System Testing**: Ensure overall system behavior under normal and abnormal conditions.
  - **Fault Injection Testing**: Introduce faults to test fault tolerance.
  - **Stress and Load Testing**: Check behavior under high demand.
  - **Boundary Testing**: Examine edge cases where failures are likely.
- **Coverage Analysis**
  - **Code coverage** (e.g., statement, branch, path).
  - **Requirements coverage**: Are all safety requirements tested?

- **Model-Based Testing**
  - Derive test cases from formal models of system behavior.
- **Regression Testing**
  - Ensure new changes don't compromise safety.
- **Safety Audits and Reviews**
  - Conduct independent reviews of code, documentation, and test results.
- **Certification Testing**
  - Meet industry-specific safety standards (e.g., DO-178C for avionics, ISO 26262 for automotive, IEC 61508 for industrial control).

Safety Cases and Their Role in Certification

In the context of **software dependability**, especially in **safety-critical systems**, **safety cases** play a vital role in demonstrating that software is acceptably safe and supporting its **certification** by regulatory bodies. Here's a breakdown of their role and how they support certification:

What is a Safety Case?

A **safety case** is a structured argument, supported by evidence, intended to justify that a system is safe for a given application in a given environment.

It typically includes:

1. **Claims**: Statements about the system's safety (e.g., "System X will not cause harm under Y conditions").
2. **Arguments**: The reasoning that links the evidence to the claims (e.g., logical decomposition, fault-tree analysis).
3. **Evidence**: Artifacts supporting the claims (e.g., test results, formal verification, inspections, hazard analyses).

This structure is often visualized using tools like **Goal Structuring Notation (GSN)**

Role of Safety Cases in Certification

1. **Demonstrate Compliance**:
   Safety cases provide the evidence needed to show compliance with relevant **safety standards** (e.g., ISO 26262 for automotive, DO-178C for aviation, IEC 61508 for industrial systems).
2. **Support Regulatory Review**:
   Certification bodies (like the FAA, TÜV, or national rail authorities) review safety cases to assess whether the system meets safety requirements. A well-structured case makes this easier.
3. **Encourage Rigorous Development**:
   The process of building a safety case enforces discipline in **hazard identification**, **risk assessment**, and **mitigation planning** throughout the software development lifecycle.
4. **Manage Complexity**:
   In large systems, safety cases help structure complex safety arguments and trace them back to individual components or subsystems.
5. **Enable Change Management**:
   When systems are updated, the safety case can be modified to show that safety is still maintained. This is critical for maintaining certification during product evolution.

Connection to Software Dependability

Software dependability encompasses attributes like **reliability**, **safety**, **security**, and **maintainability**. Safety cases focus specifically on **safety**, but their structured approach and reliance on evidence-based arguments contribute to broader dependability by:

- Ensuring rigorous verification and validation.
- Promoting traceability from requirements to implementation.
- Identifying and mitigating failure modes.

Example: Automotive Software

In **automotive systems** governed by **ISO 26262**, the safety case might include:

- Functional safety requirements.
- Software architecture adhering to ASIL(Automotive Safety Integrity Level).
- Evidence from unit, integration, and system testing.
- Analysis of safety mechanisms (e.g., watchdogs, memory protection).
- Results of failure mode and effects analysis (FMEA).

The safety case is submitted during the **confirmation review** process and is essential for **certification** and market approval.

Summary

Role	Description
Justification	Provides structured argument that system is acceptably safe
Compliance	Supports certification to safety standards
Traceability	Connects safety requirements to implementation and testing

Role	Description
Lifecycle Support	Helps manage safety through changes and updates
Audit Facilitation	Organizes evidence for external review and approval

Unit 6: Software Fault Tolerance (5 hours)

Introduction to Software Fault Tolerance

What is Fault Tolerance?

Fault tolerance is the ability of a system (such as a computer, network or software) to continue operating properly even when one or more of its components fail. The goal is to ensure **high availability, reliability, and uninterrupted service** despite hardware failures, software bugs, or external disruptions.

Key Aspects of Fault Tolerance:

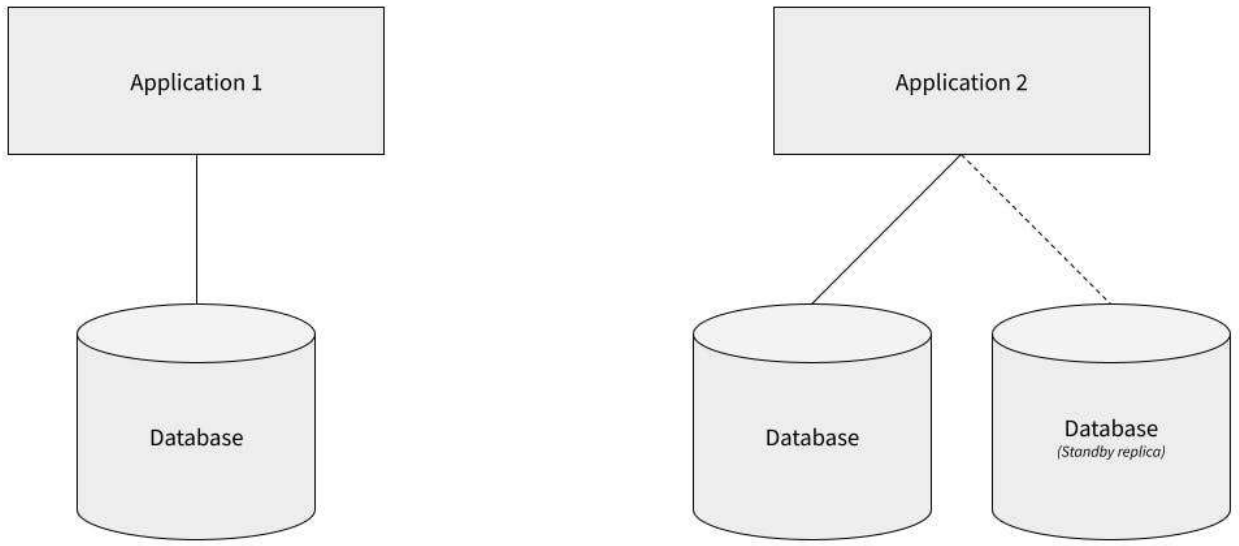
1. **Failure Detection** – The system must identify faults quickly (e.g., through health checks, timeouts, or redundancy checks).
2. **Recovery Mechanisms** – Automatically switch to backup systems or repair issues without human intervention.
3. **Redundancy** – Duplicate critical components (e.g., servers, databases, power supplies) so that if one fails, another takes over.
4. **Graceful Degradation** – If a failure occurs, the system reduces functionality rather than crashing entirely.

Examples of Fault Tolerance:

- **Hardware:** RAID (Redundant Array of Independent Disks) keeps data accessible even if one disk fails.
- **Cloud Computing:** AWS, Google Cloud, and Azure distribute workloads across multiple servers to prevent single points of failure.
- **Networking:** Routers and switches use backup paths if the primary connection fails.
- **Software:** Databases with replication (e.g., PostgreSQL, MongoDB) ensure data remains available if a node goes down.

Fault Tolerance vs. High Availability (HA):

- **Fault Tolerance** aims for **zero downtime** (seamless operation during failures).
- **High Availability** minimizes downtime but may have brief interruptions during failover.
-
- *Let's try to understand more about fault tolerance with example.*
- For example, here's a simple demonstration of comparative fault tolerance in the database layer. In the diagram below, **Application 1** is connected to a single database instance. **Application 2** is connected to two database instances — the primary database and a standby replica.



-
  - In this scenario, Application 2 is more fault tolerant. If its primary database goes offline, it can switch over to the standby replica and continue operating as usual.

- Application 1 is not fault tolerant. If its database goes offline, all application features that require access to the database will cease to function.

The above scenario is just a simple example though. In reality, fault tolerance must be considered in every layer of a system (not just the database), and there are degrees of fault tolerance. While Application 2 is more fault tolerant than Application 1, it's still less fault tolerant than many modern applications.

Fault tolerance can also be achieved in a variety of ways. These are some of the most common approaches to achieving fault tolerance:

Multiple hardware systems capable of doing the same work. For example, Application 2 in our diagram above could have its two databases located on two different physical servers, potentially in different locations. That way, if the primary database server experiences an error, a hardware failure, or a power outage, the other server might not be affected.

Multiple instances of software capable of doing the same work. For example, many modern applications make use of **containerization (Containerization** is a method of packaging software so that it can run reliably and consistently across different computing environments. It involves bundling an application along with all its dependencies (like libraries, configuration files, and binaries) into a single, lightweight unit called a **container**.

) platforms such as Kubernetes so that they can run multiple instances of software services. One reason for this is so that if one instance encounters an error or goes offline, traffic can be routed to other instances to maintain application functionality.

Backup sources of power, such as generators, are often used in on-premises systems to protect the application from being knocked offline if power to the servers is impacted by, for example, the weather. That type of outage is more common than you might expect

Fault tolerance goals

Building fault-tolerant systems is more complex and generally also more expensive. If we think back to our simple example from earlier, Application 2 is more fault tolerant, but it also has to pay for and maintain an additional database server. Thus, it's important to assess the level of fault tolerance your application requires and build your system accordingly.

Normal functioning vs. graceful degradation

When designing fault-tolerant systems, you may want the application to remain online and fully functional at all times. In this case, your goal is **normal functioning** — you want your application, and by extension the user's experience, to remain unchanged even if an element of your system fails or is knocked offline.

Another approach is aiming for what's called **graceful degradation,** where outages and errors are allowed to impact functionality and degrade the user experience, but not knock the application out entirely. For example, if a software instance encounters an

error during a period of heavy traffic, the application experience may slow for other users, and certain features might become unavailable.

Building for normal functioning obviously provides for a superior user experience, but it’s also generally more expensive. The goals for a specific application, then, might depend on what it’s used for. Mission-critical applications and systems will likely need to maintain normal functioning in all but the most dire of disasters, whereas it might make economic sense to allow less essential systems to degrade gracefully.

System Survivability and Measuring System Survivability

What is system Survivability?

System survivability is a subset of **dependability**, focusing on a system's ability to **continue functioning (possibly in a degraded mode) during and after failures, attacks, or adverse conditions**. Unlike basic fault tolerance (which handles known failures), survivability also deals with **unexpected disruptions**, such as cyberattacks, natural disasters, or unforeseen software errors.

Key Concepts:

1. Relationship with Dependability

Dependability encompasses multiple attributes:

- **Reliability** (continuous correct service)
- **Availability** (readiness for use)
- **Safety** (no catastrophic failures)
- **Security** (protection against attacks)
- **Maintainability** (ease of repair)
- **Integrity** (no unauthorized modifications)

Survivability extends these by ensuring the system can **detect, resist, recover, and adapt** to disruptions while maintaining critical functions.

2. Core Principles of Survivability

A survivable system must:

1. **Resist Attacks/Failures** – Prevent or mitigate damage (e.g., firewalls, redundancy).
2. **Recognize Threats** – Detect intrusions or failures (e.g., anomaly detection, logging).
3. **Recover Quickly** – Restore essential services (e.g., failover mechanisms, backups).
4. **Adapt & Evolve** – Modify behavior to maintain functionality (e.g., load balancing, throttling).

3. Techniques for Enhancing Survivability

a) Redundancy & Diversity

- **Hardware Redundancy** (backup servers, RAID storage)
- **Software Diversity** (multiple implementations of critical functions to avoid common failures)
- **Geographical Distribution** (cloud-based multi-region deployments)

b) Fault Containment & Isolation

- **Microservices Architecture** – Limits blast radius of failures.
- **Sandboxing** – Isolates compromised components.
- **Circuit Breakers** – Prevents cascading failures.(failures in one part leads to another part failures.)

c) Self-Healing & Autonomous Recovery

- **Rollback Mechanisms** – Reverts to a stable state after failure.
- **Dynamic Reconfiguration** – Adjusts resources based on demand/threats.

d) Intrusion Tolerance & Cyber-Resilience

- **Byzantine Fault Tolerance (BFT)** – Handles malicious nodes in distributed systems.
- **Honeypots & Deception** – Diverts attackers from real systems.

4. Real-World Examples

- **NASA Space Probes** – Must operate despite radiation, hardware faults, and communication delays.
- **Financial Systems (e.g., Stock Exchanges)** – Use redundant data centers to prevent trading halts.
- **Military Command Systems** – Designed to function even if partially compromised.
- **Cloud Services (AWS, Azure)** – Auto-scaling and multi-zone deployments ensure uptime.

5. Survivability vs. Fault Tolerance vs. High Availability

Aspect	Fault Tolerance	High Availability (HA)	Survivability
Goal	Zero downtime	Minimal downtime	Continued operation (possibly degraded)
Scope	Known failures	Planned failovers	Unknown threats (attacks, disasters)
Response	Immediate failover	Fast recovery	Adaptive recovery & evolution
Example	RAID storage	Load-balanced web servers	Cyber-resilient military systems

6. Challenges in Achieving Survivability

- **Complexity** – Managing redundancy and failover logic.
- **Performance Overhead** – Extra checks and replication slow down systems.
- **Cost** – Requires additional infrastructure.
- **Security Trade-offs** – More components mean more attack surfaces.

Measuring System Survivability

Survivability is a critical aspect of **system dependability**, but unlike traditional reliability or availability metrics, it focuses on **maintaining essential functionality during and after failures, attacks, or unexpected disruptions**. Measuring survivability requires a combination

of **quantitative metrics, qualitative assessments, and resilience evaluation frameworks**.

1. Key Metrics for Survivability

To assess survivability, we use a mix of **dependability metrics** (e.g., reliability, availability) and **resilience-specific measures** (e.g., recovery time, degradation tolerance). Below are the most important ones:

A. Core Dependability Metrics (Baseline)

These metrics help establish the system's baseline dependability:

1. **Mean Time Between Failures (MTBF)**
   - Measures how often failures occur.
   - Higher MTBF → More reliable system.
2. **Mean Time To Repair (MTTR)**
   - Time taken to recover from a failure.
   - Lower MTTR → Faster recovery.
3. **Availability (%)**
   - Availability=Uptime/Uptime +Downtime * 100
   - Survivability extends this by ensuring **critical functions remain available** even if some parts fail.
4. **Failure Rate (λ)**
   - Number of failures per unit time.

B. Survivability-Specific Metrics

These focus on **how well the system maintains operations under stress**:

1. **Degradation Ratio (DR)**
   - Measures how much performance drops during failure.
   - DR=Reduced Capacity/Full Capacity

- Example: A cloud service running at 60% capacity during an attack has DR=0.6

2. **Recovery Time Objective (RTO)**
   - Maximum acceptable time to restore critical functions.
3. **Recovery Point Objective (RPO)**
   - Maximum tolerable data loss (e.g., last backup time).
4. **Survivability Index (SI)**
   - Combines multiple factors (e.g., detection time, recovery success rate).
   - Example formula:
     SI=Successful Recoveries/Total Disruptions×1/MTTR

5. **Attack/Disruption Resistance Rate**
   - Percentage of attacks/failures the system resists without total failure.
6. **Adaptability Score**
   - Measures how well the system adjusts to new threats (e.g., via machine learning, dynamic reconfiguration).

2. **Evaluation Frameworks for Survivability**

A. Quantitative Models

1. **Markov Models**
   - Predicts survivability by modeling system states (normal, degraded, failed).
   - Used in safety-critical systems (e.g., aerospace, nuclear plants).
2.

3. **Fault Tree Analysis (FTA)**
   - Identifies potential failure paths and their impact on survivability.
4. **Reliability Block Diagrams (RBDs)**
   - Evaluates how redundancy and failover mechanisms improve survivability.

B. Qualitative Assessments

1. **Survivability Levels (SL)**
   - **SL0**: No survivability (total failure on disruption).
   - **SL1**: Minimal functionality retained.
   - **SL2**: Partial recovery with manual intervention.
   - **SL3**: Automatic recovery with minor degradation.
   - **SL4**: Full resilience (self-healing, no downtime).
2. **Cyber-Resilience Frameworks (e.g., MITRE, NIST SP 800-160)**
   - Assess survivability against cyber threats.
   - Includes metrics like:

- **Time to Detect (TTD): Time interval between the occurrence of a fault** and the **moment the system or monitoring mechanism detects it**. The shorter the TTD, the quicker a system can react or recover. Longer TTD means the attacker can attack for longer period.
- **Time to Respond (TTR):** The amount of **time taken to initiate a corrective or protective action** after a fault, failure, or attack has been detected. The shorter TTR means faster containment of issues and better survivability.
- **Time to Recover (TTR):** The **amount of time it takes to restore a system or service** back to its **normal operating condition** after a fault, failure, or disruption has occurred and a response has been initiated. The shorter TTR improves resilience and survivability and helps meet Service Level Agreements.

- **Design Patterns for Fault Tolerance**

1. Redundancy Pattern

Concept: Duplicate critical components so if one fails, another can take over.

Types:

- **Hardware Redundancy** (e.g., backup servers)
- **Software Redundancy** (e.g., N-version programming)
- **Data Redundancy** (e.g., RAID, replicated databases)

Example:
Cloud platforms using **multi-zone replication** for database availability.

2. Retry Pattern

Concept: If a transient failure occurs (like a network timeout), retry the operation after a short delay.

Features:

- Often used with exponential backoff
- Needs max retry count and timeout

Used In:
Microservices communication, HTTP API calls

3. Circuit Breaker Pattern

Concept: Prevents a system from repeatedly calling a failing service.

How it works:

- If failures exceed a threshold, the circuit "opens"
- Calls are blocked until the system stabilizes

Used In:
Distributed systems, to isolate failing services and avoid cascading failures

4. Failover Pattern

Concept: Automatically switch to a backup system when the primary fails.

Variants:

- **Active-Passive**: Backup is idle until needed
- **Active-Active**: All instances serve traffic, share load

Used In:
Databases, load balancers, clustered apps

5. Watchdog Timer Pattern

Concept: A separate component monitors the main system, and **reboots or restarts it** if it becomes unresponsive.

Used In:
Embedded systems, real-time systems, robotics

6. Health Check Pattern

Concept: Continuously monitor components to detect failure early.

Examples:

- Kubernetes **liveness** and **readiness** probes
- Load balancer health checks

7. Graceful Degradation Pattern

Concept: When full functionality can't be provided, offer a reduced (but still useful) level of service.

Example:

- A video app downgrading quality instead of stopping playback.
- E-commerce site disables recommendations during high load.

8. Bulkhead Pattern

Concept: Isolate parts of the system so a failure in one area doesn't bring down the entire system.

Named After: Ship compartments designed to limit flooding.

Used In:
Containerized apps, micro services, concurrent threads

9. Self-Healing Pattern

Concept: The system detects anomalies and autonomously attempts to fix them.

Examples:

- Auto-restart crashed containers

- Auto-scale services under load

10. Audit and Monitoring Pattern

Concept: Continuously log and analyze data to detect and respond to faults before they escalate.

Tools:
ELK Stack, Prometheus + Grafana, AWS Cloud Watch

- **Fault Minimization and Tolerance Techniques**

Fault minimization techniques are methods used in software engineering to reduce the number of errors or faults in a software system. Some common techniques include:

1. **Code Reviews**: Code review is a process where code is evaluated by peers to identify potential problems and suggest improvements.
2. **Unit Testing**: Unit testing involves writing automated tests for individual units of code to ensure that each component works as expected.
3. **Test-Driven Development**: Test-driven development (TDD) is a software development process where unit tests are written before writing the code, ensuring that the code meets the requirements.
4. **Continuous Integration and Continuous Deployment (CI/CD**): CI/CD is a software engineering practice where code changes are automatically built, tested, and deployed to production.
5. **Static Code Analysis**: Static code analysis is the process of automatically analyzing code to find potential problems without actually executing it.
6. **Design and Architecture Reviews**: Reviews of design and architecture help ensure that the overall design of the software is correct and scalable.

7. **Error Handling**: Proper error handling helps prevent the spread of faults and ensures that the software can handle unexpected situations gracefully.
8. **Documentation**: Clear and up-to-date documentation can help reduce the likelihood of errors by providing a clear understanding of the software's functionality.

A **fault** is a defect in the program that, when executed under particular conditions causes a different result of the program operation from its requirements. It is the condition that causes the software to fail to perform its required functionality. The following are the techniques used to reduce faults in software:

1. **Fault Prevention -** Fault Prevention/Avoidance strategies identify all potential areas where a fault can occur and close the gaps. These prevention strategies address system requirements and specifications, software design methods, re-usability, or formal methods. They are employed during the development phase of the software to avoid or prevent fault occurrence. They contribute to the system dependability through the rigorous specification of the system requirements, programming methods, and software re-usability. But it is difficult to quantify the impact of fault avoidance strategies on system dependability. So, despite fault prevention efforts, faults are created, so fault removal is needed.
2. **Fault Removal -** Fault removal strategies are dependability-enhancing techniques employed during verification and validation. They improve by detecting existing faults and eliminating the defected faults. They are employed after the development phase of the software to contribute to the validation of the software. Common fault removal techniques involve testing. It follows that minimizing component size and interrelationship maximizes accurate testing. The difficulties encountered in testing programs are often related to the prohibitive costs and exhaustive testing. Therefore, fault removal is imperfect, hence fault tolerance is needed.

3. **Fault Tolerance -** Fault tolerance includes dependability-enhancing techniques that are used during the validation of software to estimate the presence of faults. It is used to reduce system design faults and enhance the reliability of the software. Fault tolerance techniques are employed during the development phase of the software which enables the system to tolerate faults remaining in the system after its development and provide operation complying with the requirements specification in spite of faults. Therefore, when a fault occurs it prevents the system failure.

Fault prevention, fault removal, and fault tolerance represent the successive lines of defense against the contingency of faults of software systems and their impact on system. Despite the fact, that the benefits of each of these techniques are remarkable, the law of diminishing returns advocates that they should be used in unison where each one is applied wherever it is most effective.

Advantages and Disadvantages:

Advantages of fault minimization techniques in software engineering:

1. **Improved software quality**: By reducing the number of faults in a software system, the quality of the software is improved, resulting in a better user experience.
2. **Increased reliability**: With fewer faults, the software is more reliable and less likely to fail, leading to fewer downtime incidents and a better reputation for the company.
3. **Reduced maintenance costs**: By reducing the number of faults in a software system, the cost of maintenance is reduced as there are fewer issues to fix.
4. **Faster problem resolution**: By using techniques like unit testing and continuous integration, problems are caught early, making it easier and faster to resolve them.

Fault Tolerance Techniques

Fault-tolerance is the process of working of a system in a proper way in spite of the occurrence of the failures in the system. Even after performing the so many testing processes there is possibility of failure in system. Practically a system can't be made entirely error free. hence, systems are designed in such a way that in case of error availability and failure, system does the work properly and given correct result. Any system has two major components - Hardware and Software. Fault may occur in either of it. So there are separate techniques for fault-tolerance in both hardware and software. **Hardware Fault-tolerance Techniques:** Making a hardware fault-tolerance is simple as compared to software. Fault-tolerance techniques make the hardware work proper and give correct result even some fault occurs in the hardware part of the system. There are basically two techniques used for hardware fault-tolerance:

1. **BIST -** BIST stands for Build in Self-Test. System carries out the test of itself after a certain period of time again and again, that is BIST technique for hardware fault-tolerance. When system detects a fault, it switches out the faulty component and switches in the redundant of it. System basically reconfigure itself in case of fault occurrence.
2. **TMR -** TMR is Triple Modular Redundancy. Three redundant copies of critical components are generated and all these three copies are run concurrently. Voting of result of all redundant copies are done and majority result is selected. It can tolerate the occurrence of a single fault at a time.
3. **Check-pointing and Rollback Recovery -** This technique is different from above two techniques of software fault-tolerance. In this technique, system is tested each time when we perform some computation. This techniques is basically useful when there is processor failure or data corruption.

Fault Tolerance Architecture

N-version Programming: **N-Version Programming** is a **software fault tolerance technique** that involves developing **multiple functionally equivalent versions** of a program independently and running them in parallel to ensure correct execution, even if some versions contain faults. N-Version Programming involves implementing **N independently developed versions** of a software module from the same specification. The outputs of these versions are then compared using a **voting mechanism** to determine the correct result.

Goals of NVP

- Increase **reliability** and **fault tolerance**.
- Detect and tolerate **design and implementation faults** (as opposed to just transient faults).
- Avoid **common-mode failures** by introducing **design diversity**.

Key Components of NVP

Component	Description
N Versions	Multiple independently implemented versions (usually 3 or more) of the same function/module
Voter	A decision component that collects outputs from each version and selects the majority or correct one
Consensus Algorithm	Usually majority voting, but can use weighted or median voting
Specification	All versions are developed from the **same specification**, which must be unambiguous and complete

How It Works (Execution Flow)

1. **Input is received** by all N versions simultaneously.
2. Each version **processes the input independently**.
3. Outputs are sent to a **voter**.
4. Voter uses a **voting algorithm** to decide the final result.
    - Majority voting: If at least ⌈N/2⌉ versions agree, their result is accepted.
    - Median voting or consensus methods in case of numeric or approximate outputs.
5. If a discrepancy is detected, the system may:
    - **Log an error**,
    - **Isolate the faulty version**,
    - **Trigger alerts**, or
    - **Replace faulty module** (if dynamically updatable).

Example Scenario: NVP in Aerospace System

Imagine a flight control software that calculates the position of control surfaces on an aircraft:

- You develop 3 versions of the control logic:
    - **Version A**: Implemented in C by Team 1
    - **Version B**: Implemented in Ada by Team 2
    - **Version C**: Implemented in Rust by Team 3
- All versions receive the same sensor data.
- A **voter module** compares their outputs:
    - If A and C agree but B differs, then A and C's output is used.

This ensures that a bug in one version (e.g., due to a coding mistake or compiler issue) does not lead to failure.

Advantages

- High **resilience** to software faults.

- Can **detect and tolerate design bugs** (not just hardware faults).
- Especially useful in **safety-critical systems** (aerospace, nuclear, medical devices).

Challenges and Limitations

Challenge	Description
Cost	Very high cost – developing multiple versions triples effort.
Specification Consistency	Ambiguities in spec can cause all versions to be faulty in the same way (common-mode failure).
Voter Complexity	Designing a fair, deterministic, and reliable voter is complex.
Performance	Increased runtime overhead due to multiple versions running concurrently.
Development Independence	True independence is hard to achieve – teams might interpret specs similarly or share mental models/tools.

Recovery Blocks

Recovery blocks are a fault tolerance technique in computer programming that uses multiple alternative code blocks (called "alternates") to perform the same function. If the primary alternate fails an acceptance test, the block switches to an alternate and re-runs it until one passes. This mechanism helps ensure that the program can continue to function even if a fault occurs.

Core Concept:

- A recovery block consists of a primary alternate (the main code) and one or more secondary alternates.

- Each alternate is designed to perform the same task, but with potentially different algorithms or approaches.
- A recovery block also includes an "acceptance test" that verifies the results of each alternate.
-

2. **How it Works**:

- When a recovery block is executed, the primary alternate is first executed.
- After the primary alternate completes, the acceptance test is performed to verify that the results are correct and valid.
- If the acceptance test passes, the recovery block is considered successful, and the results are returned to the calling code.
- If the acceptance test fails, the recovery block switches to the next alternate and attempts to execute it.
- This process repeats until one of the alternates passes the acceptance test or all alternates have been tried.

Benefits of Recovery Blocks:

- **Fault Tolerance:**

  If one alternate fails, the program can switch to another alternate, potentially continuing to operate correctly.
- **Error Detection and Recovery:**

  The acceptance test provides a mechanism to detect errors and allows for recovery by switching to an alternate.
- **Increased Reliability:**

  By incorporating redundancy and error recovery, recovery blocks can increase the reliability of software systems.
- **Cost-Effective:**
  Recovery blocks can be a cost-effective way to implement fault tolerance compared to using multiple hardware channels.

4. **Example**:

- Imagine a function that calculates a value. The primary alternate might use a specific algorithm, while an alternate might use a different, more robust algorithm.
- The acceptance test might verify that the calculated value falls within a certain range or meets specific criteria.
- If the primary algorithm fails (e.g., due to a bug), the alternate algorithm can be used, and the acceptance test will ensure that the result is still valid.

Introduction to Software Aging and Software Rejuvenation

What is Software Aging?

Software aging refers to the **progressive degradation of software performance, reliability, or availability** during its continuous operation.

Symptoms of Software Aging

- Increased response time
- Memory leaks
- Resource overconsumption (e.g., CPU, disk space)
- Unexpected behavior or crashes
- System hangs or deadlocks

Causes of Software Aging

- **Memory Leaks:** Failure to release unused memory.
- **Data Corruption:** Gradual accumulation of inconsistent or incorrect data states.
- **Resource Exhaustion:** Depletion of limited system resources such as sockets, file handles, or threads.
- **Fragmentation:** Inefficient use of memory or disk due to fragmentation.
- **Software Bugs:** Logical errors that only manifest after long runtimes.
- **State Drift:** System state diverges from its ideal configuration due to continuous usage.

Note: Software aging is analogous to fatigue in materials — they don't fail instantly, but degrade with usage.

What is Software Rejuvenation?

Software rejuvenation is a **proactive fault management technique** used to mitigate the effects of software aging by refreshing the system state before failures occur.

Common Rejuvenation Actions

- Restarting the entire system
- Restarting individual applications or modules
- Reinitializing internal data structures
- Clearing memory or caches
- Releasing and reallocating resources

Goal

To **maximize system availability** and **reduce failure rates** due to aging.

Why Software Rejuvenation is Needed

- Long-running systems (e.g., servers, routers, ATMs) must operate continuously with minimal downtime.
- Traditional fault tolerance focuses on hardware faults, not logical degradation in software.
- Rejuvenation avoids unplanned outages and keeps systems healthy at a minimal cost.

Real-World Example: Web Server Scenario

A server running Apache or Nginx continuously over weeks starts to respond slowly. Logs show increasing memory usage. Instead of waiting for a crash, a scheduled reboot every Sunday at midnight (during low traffic) helps avoid failure.

Conceptual Model

Let:

- $\lambda(t)$: Failure rate due to aging (increases with time)
- T: Rejuvenation interval
- $A(T)$: System availability over time

Without rejuvenation: $\lambda(t) \uparrow$, and $A(T) \downarrow$
With rejuvenation: System is periodically refreshed, keeping $\lambda(t)$ low. Availability remains high, reducing total failure time.

Academic Insight

Yung H. Huang and Kalyanaraman Vaidyanathan introduced the concept in the late 1990s. Their research showed that timely rejuvenation significantly improves the **Mean Time To Failure (MTTF)** of systems.

Mathematical Illustration

If the failure probability over time follows a Weibull distribution:

$$F(t) = 1 - e^{-(\lambda t)^{\beta}}$$

Where:

- λ: Aging rate
- $\beta > 1$: Indicates aging (i.e., increasing failure rate over time)

Rejuvenation resets the system time $t \to 0$, effectively restarting the failure curve.

Analytical Models for Software Rejuvenation

Analytical models for software rejuvenation help in:

- Predicting software aging behavior
- Determining optimal rejuvenation schedules
- Balancing cost vs. availability trade-offs

These models use mathematical tools like **probability theory**, **stochastic processes**, and **optimization techniques**.

Types of Models

1. Deterministic Models

- Assume predictable aging behavior (e.g., rejuvenate every 24 hours).
- Simple but fail to capture real-world uncertainty.

2. Stochastic Models (Probabilistic)

These models incorporate randomness and better represent real-world behavior.

a. Markov Chain Models

- System transitions between states such as: *Healthy*, *Degraded*, *Failed*.
- Transition probabilities determine the chance of moving from one state to another.

Let:

$$\begin{bmatrix} P_{HH} & P_{HD} & P_{HF} \\ P_{DH} & P_{DD} & P_{DF} \\ 0 & 0 & 1 \end{bmatrix}$$

Where P_{ij} is the probability of transitioning from state i to j.

b. Semi-Markov Processes

- Allow time-dependent transitions (unlike standard Markov models).
- More accurate for long-running systems where time spent in a state affects transition likelihood.

c. Stochastic Petri Nets (SPNs)

- Graphical + probabilistic modeling
- Ideal for modeling concurrent systems with parallel processes

d. Renewal Processes

- Model rejuvenation as a renewal event.
- After each rejuvenation, the system is "as good as new".

Cost-Based Optimization Models

Objective: Minimize total expected cost over time.

Let:

- C_f: Cost of failure
- C_r: Cost of rejuvenation
- $P_f(t)$: Probability of failure at time t
- $R(t)$: Rejuvenation frequency

Then, total cost:

$$\text{Total Cost} = C_f \cdot P_f(t) + C_r \cdot R(t)$$

The optimal time T^* for rejuvenation minimizes this total cost.

Weibull Aging Model

Failure probability:

$$F(t) = 1 - e^{-(\lambda t)^{\beta}}$$

Where:

- λ: Aging rate parameter
- $\beta > 1$: Indicates increasing failure rate due to aging

Rejuvenation resets system to time $t = 0$, which restarts the aging process.

Applications

- Scheduling system restarts
- Predictive maintenance
- Cloud service auto-restart policies

Conclusion

Analytical models for software rejuvenation provide a mathematical basis to:

- Predict failures
- Reduce unplanned downtimes
- Maximize system availability and performance

These models are essential in building **dependable and resilient** software systems.

Software Rejuvenation in Transaction-Based Software Systems

Transaction-based software systems (TBSS) are systems where multiple concurrent users initiate time-sensitive, atomic operations known as **transactions**.
Examples include:

- Online banking platforms
- E-commerce applications
- Airline reservation systems
- Online payment gateways

Such systems are **mission-critical** and expected to run continuously with minimal interruption. Therefore, applying software rejuvenation requires special care to avoid:

- Transaction loss
- Data inconsistency
- User dissatisfaction

Challenges in TBSS Rejuvenation

1. **Continuous Operation**: TBSS often run 24/7 and have no ideal time window for downtime.
2. **Transaction Integrity**: Rejuvenation must ensure ongoing transactions are not lost or duplicated.
3. **Concurrency**: High volume of parallel operations complicates state tracking during rejuvenation.
4. **Data Consistency**: All committed transactions must reflect consistent states post-rejuvenation.

Techniques for Rejuvenating TBSS

1. Graceful Rejuvenation

- Wait until all in-flight transactions are completed.
- Block new transactions during rejuvenation preparation.
- Then safely restart the system.

2. Checkpoint and Recovery

- Save system state and transaction logs periodically.
- During rejuvenation, resume from the last consistent state.
- Ensures ACID (Atomicity, Consistency, Isolation, Durability) properties.

3. Transaction-Aware Load Balancing

- Distribute new transactions to other servers.
- Allow one node to rejuvenate while others maintain service.

4. Dual-System Architecture

- Maintain two systems: **Active** and **Standby**.
- Migrate transactions to the standby system before rejuvenating the active one.

Mathematical Consideration

Let:

- T_c: Average time to complete a transaction
- T_q: Queue waiting time
- T_r: Rejuvenation duration
- $\lambda(t)$: Failure rate due to aging

Objective: Schedule rejuvenation so that:

$$T_r < \min(T_c + T_q)$$

and $\lambda(t)$ remains low enough to avoid system crash during rejuvenation delay.

Best Practices

- Rejuvenate during low-traffic periods (if any).
- Monitor transaction volumes and memory usage to trigger rejuvenation adaptively.
- Use containers or microservices to rejuvenate smaller units independently.

Conclusion

In transaction-based software systems, rejuvenation strategies must prioritize **availability**, **transaction integrity**, and **data consistency**. Techniques such as *graceful shutdowns*, *checkpointing*, and *load balancing* make rejuvenation possible without service disruption.

Case Study: IBM X-Series Cluster Servers and Software Rejuvenation

IBM's X-Series cluster servers are high-availability, enterprise-grade servers widely used in:

- Data centers
- Cloud infrastructure
- E-commerce and financial systems

As with other long-running systems, these servers experience **software aging**, which can lead to:

- Memory exhaustion
- Performance degradation
- Sudden software crashes

Problem Statement

IBM engineers observed that:

- Aging effects such as memory leaks and resource fragmentation accumulated over time.
- Manual intervention and unexpected system failures led to decreased availability.
- Cluster-wide crashes could occur if aging was not proactively addressed.

Rejuvenation Strategy Employed

IBM implemented a software rejuvenation framework integrated into the cluster operating system.

1. Predictive Aging Detection

- Monitoring tools track memory usage, thread count, and CPU load trends.
- Statistical models predicted when aging would reach critical levels.

2. Rolling Rejuvenation (Staggered Restart)

- Rejuvenation is applied **node-by-node**, not all at once.
- One node is rejuvenated while others continue servicing requests.
- Load is redistributed during the rejuvenation of a node.

3. Automated Rejuvenation Manager

- A daemon process scheduled rejuvenation tasks automatically.
- Time-based and event-based triggers (e.g., memory threshold) were used.

1. Rejuvenation Architecture

- **Load Balancer:** Routes requests to healthy nodes.
- **Cluster Monitor:** Tracks aging indicators across all nodes.
- **Rejuvenation Scheduler:** Selects the next node for rejuvenation.
- **Failover Mechanism:** Ensures that rejuvenated nodes rejoin the cluster smoothly.

2. Benefits and Outcomes

- Improved Mean Time Between Failures (MTBF)
- Reduced total cost of operation and manual maintenance
- Enhanced system uptime and availability
- No interruption to end-users due to staggered rejuvenation

Conclusion

IBM's implementation of software rejuvenation in the X-Series cluster servers demonstrates the practical value of:

- Predictive failure analysis
- Rolling rejuvenation strategies
- Integration of monitoring and automation tools

This case study shows how rejuvenation can maintain **high availability and reliability** in mission-critical, distributed software systems.

Approaches and Methods of Software Rejuvenation

Software rejuvenation aims to proactively counter software aging through scheduled or event-driven refreshes of system components. The effectiveness of rejuvenation largely depends on the chosen **approach** (when to rejuvenate) and **method** (how to rejuvenate).

Approaches to Software Rejuvenation

1. Time-Based Rejuvenation

- Rejuvenation is scheduled at fixed time intervals (e.g., every 24 hours).
- Simple to implement.
- May rejuvenate too early or too late.

2. Event-Based Rejuvenation

- Triggered when specific system conditions are met.
- Conditions may include:
  - Memory usage exceeding threshold
  - Number of open files or threads above limit

  - Degraded response time
- More efficient than time-based rejuvenation.

3. Prediction-Based Rejuvenation

- Uses machine learning or statistical models to forecast failure risk.
- Rejuvenation is initiated before predicted failures.
- Requires historical data and model training.

Methods of Software Rejuvenation

1. Full System Reboot

- Entire system is restarted.
- Very effective but causes service interruption.
- Typically used during maintenance windows.

2. Application-Level Rejuvenation

- Only the application (not OS) is restarted.
- Reduces downtime and impact on other services.

3. Component-Level Rejuvenation

- Rejuvenate only specific modules (e.g., memory manager, thread pool).
- Minimizes impact and enables faster recovery.

4. Cache Clearing and Garbage Collection

- Manual or automated release of memory, disk cache, or temporary resources.
- May involve internal reinitialization of application data structures.

5. Virtual Machine (VM) Migration

- Move applications to another VM before rejuvenation.
- Used in cloud-based or containerized environments.

6. Container Re-deployment (Microservices)

- Rejuvenation via re-deployment of containers in orchestrated environments like Kubernetes.
- Allows near-zero downtime using rolling updates.

Conclusion

Choosing the right rejuvenation approach and method is crucial for maintaining:

- High availability
- Low recovery time
- Cost-efficiency

Modern systems often use a hybrid strategy combining prediction-based scheduling with component-level rejuvenation for best results.

granularity of software rejuvenation

The **granularity of software rejuvenation** refers to the *scope or level* at which rejuvenation actions are applied. It ranges from rejuvenating the entire system to rejuvenating only a specific component or service.

Choosing the right granularity is important for:

- Minimizing downtime
- Preserving user experience
- Efficient resource utilization

Types of Granularity

1. System-Level Rejuvenation

- Entire operating system and all running services are restarted.
- Suitable for severe or multi-component aging effects.
- Results in complete service interruption during reboot.

Example: Restarting a server hosting a database and web services.

2. Application-Level Rejuvenation

- Only the targeted application or service is restarted.
- Other applications running on the system remain unaffected.
- Reduces impact compared to system-level rejuvenation.

Example: Restarting a web server (e.g., Apache) without rebooting the OS.

3. Component-Level Rejuvenation

- Rejuvenation is applied to a specific module, thread, or internal structure.
- Minimally invasive and offers fine-grained control.
- Requires sophisticated monitoring and architecture support.

Example: Refreshing a database connection pool or regenerating session caches.

Comparison Table

Granularity Level	Scope	Impact	Complexity
System-Level	Entire system	High	Low
Application-Level	Single application/service	Medium	Medium
Component-Level	Subcomponent/module	Low	High

Factors Influencing Granularity Selection

- **Severity of aging symptoms**
- **Availability requirements**
- **System architecture (monolithic vs. microservices)**
- **Automation and monitoring capabilities**

Best Practices

- Use **fine-grained rejuvenation** (component-level) when possible to reduce impact.
- Combine granularity strategies based on observed aging symptoms.
- In distributed systems, prefer rolling rejuvenation with minimal granularity.

Conclusion

The effectiveness of software rejuvenation significantly depends on its granularity. A well-chosen level balances:

- System performance
- Availability
- Operational cost

Fine-grained rejuvenation strategies (like container restarts or thread pool resets) are preferred in modern, highly available systems.

~~Chapter~~

Software Security

↳ Software security is simply a collection of methods used to protect computer programs and the sensitive information handled by them against malicious attacks. It covers a wide range of functions to safeguard software and its correlated data privacy, accuracy and accessibility respectively.

↳ [Relⁿ betⁿ security & Dependability]

Relationship betⁿ security and ~~availability~~ Dependability

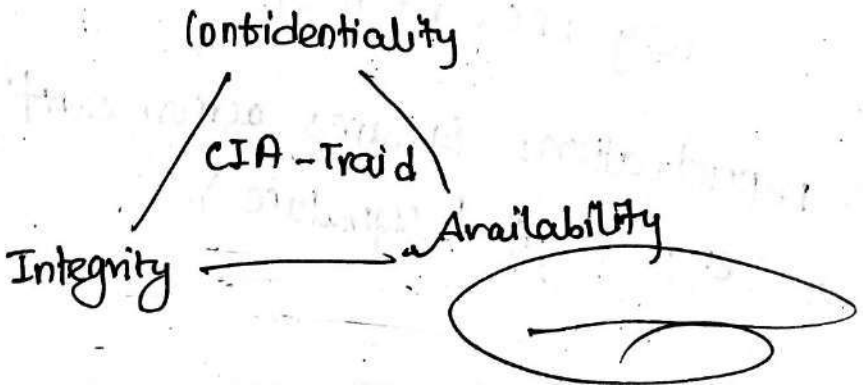
→ Dependability means the system performs correctly and reliably overtime (includes reliability, availability, safety, maintainability, integrity)

Security is a subset of dependability focusing on

confidentiality : prevent unauthorized access.

Integrity : prevent unauthorized modification.

Availability: The system is available 24/7 for intended users.



A system can not be full dependable without Security.

Example: If a banking system is reliable but not secure (hackers can steal money, it can't be considered reliable.

Security Requirements for Dependable Systems

Security Requirement ensures that dependable systems stay functional under attack. The security requirement for dependable systems are:

i) Authentication: Verify users identity (e.g. password, biometric)

ii) Authorization: Limit access to resources based on users rights

iii) confidentiality: Protect sensitive data from unauthorized access.

iv) Integrity: Ensure data is accurate and unaltered.

v) Availability: Prevent service destruction (e.g. DDOS Attack)

vi) Non repudiation: Ensures action can't be denied later (e.g. digital signature)

Secure systems Design Principles

Design principle help build secure systems from the start.

Principle	Description
least of privilige	Give minimum access necessary
fail state Defaults	Default should be deny access unless explicitly allowed
Economy of mechanism	keep design of simple, tiny to avoid hidden flaws
complete mediation	check every acess request
open design	Security should rely on secrecy of design.
seperation of Duties	Divide critical tasks among different peoples or components
Defense in Depth	multiple layers of security control
Psychological Acceptability	Security measure should not hinder usability.

Security Testing and Assurance Techniques

↳ Testing and assurance ensures that system is actually secure:

- Penetration Testing Ethical Hacking
  → simulated attack to find vulnerabilities.
- Static Analysis
  → Analyze the source code for vulnerabilities (without running the program)
- Dynamic Analysis
  → Analyze the program while it's running to find the issues
- formal verification
  → Use mathematical methods to prove correctness and security.
- Security Audit
  ↳ Manual reviews of security policies, system design and implementation.
- Fuzzy Testing
  ↳ to Input random data to detect crashes or unexpected behavior.

Common Vulnerabilities ~~and attack vectors~~

i) Buffer ~~overflow~~ overflow
→ writing more data ~~than~~ then a buffer can hold

ii) SQL injection
→ Inserting ~~sql~~ malicious SQL queries

iii) Cross-site ~~expecting~~ scripting (XSS)
→ Injecting malicious scripts into the web pages.

iv) Cross-site request Forgery (CSRF)
→ Forcing user to execute unwanted actions

v) Insecure Authentication
→ ~~use p~~ weak ~~with~~ password policies or login mechanism

vi) ~~mix con~~ mis configuration
→ pure security settings (e.g. default password)

common attacks

i) ~~Phising~~ Phishing
→ Trick users to give credentials

ii) Malware
→ Software that damages or steals data

iii) Social Engineering
→ manipulating people to gain acces.

Intersecting Communication

iv) Denial of service (DOS)

→ overloading system to make service unavailable.

Software Maintenance and Evolutions

1.1 What is Software Maintenance?

Software maintenance is the process of modifying a software system or component after delivery to:

- Correct faults (bugs)
- Improve performance or other attributes
- Adapt the product to a changed environment
- Enhance functionalities based on user needs

IEEE Standard 1219 defines maintenance as:
"The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

1.2 Importance of Software Maintenance

- 70–90% of total software lifecycle cost is spent on maintenance (depending on system complexity).
- Keeps the system relevant, secure, and efficient.
- Ensures that user requirements and technological environments are continuously met.
- Essential for long-term reliability, security, and compliance.

1.3 Types of Software Maintenance

Type	Purpose	Example
Corrective Maintenance	Fixing known defects or bugs after the software has been released.	Fixing a crash when users input invalid data.
Adaptive Maintenance	Modifying software to work in a new environment (hardware, OS, regulations).	Updating a system for a new OS version or browser.

Perfective Maintenance	Enhancing features, usability, or maintainability based on user feedback or performance analysis.	Improving response time of a dashboard query.
Preventive Maintenance	Making changes to reduce future problems or improve future maintainability.	Refactoring legacy code to reduce technical debt.

Real-world Distribution (Typical Estimates):

- Corrective: 20%
- Adaptive: 25%
- Perfective: 50%
- Preventive: 5%

(These numbers vary depending on system domain, e.g., critical systems require more preventive work.)

1.4 Software Maintenance Activities

Phase	Description
Analysis	Identify the need for maintenance, analyze problem reports or change requests.
Impact Assessment	Determine the effect of changes on system modules and other components.
Change Implementation	Modify code, update configurations, and adjust data structures.
Testing	Perform regression, integration, and unit testing to validate changes.
Release and Deployment	Package and deploy updates to production environments.
Documentation Update	Update system and user documentation as needed.

1.5 Key Challenges in Maintenance

- **Understanding Legacy Code:** Often poorly documented or written by developers no longer with the team.
- **Ripple Effects:** Small changes can cause unexpected bugs in other areas.
- **Dependency Management:** Changes in libraries or APIs can break compatibility.

- **Maintaining Quality:** Risk of introducing new bugs while fixing old ones.
- **Time and Cost Constraints:** Tight deadlines and limited budgets often conflict with proper maintenance processes.

1.6 Tools for Maintenance

Category	Tools
Version Control	Git, SVN, Mercurial
Bug Tracking	Jira, Bugzilla, GitHub Issues
Code Quality	SonarQube, ESLint, PMD
CI/CD	Jenkins, GitLab CI, GitHub Actions
Monitoring	New Relic, Datadog, Prometheus

1.7 Standards and Models

- **IEEE 1219** – Standard for Software Maintenance
- **ISO/IEC 14764** – International standard for software maintenance processes
- **Boehm's Spiral Model** – Highlights the iterative nature of evolution and maintenance
- **Lehman's Laws of Software Evolution** – Predict patterns in long-term software change, e.g.:
  - *"Continuing change"* – Systems must be continually adapted
  - *"Increasing complexity"* – Systems become more complex unless actively managed

2. Managing Changes and Updates in Dependable Systems

2.1 What Are Dependable Systems?

Dependable systems are software systems that require:

- High availability
- Safety
- Security
- Reliability
- Fault tolerance

Failures in these systems can lead to loss of life, financial damage, or legal issues.
Examples:

- Aircraft navigation and control systems (avionics)
- Medical device control software (e.g., pacemakers)
- Nuclear power plant control systems
- Banking systems
- Autonomous vehicle systems

2.2 Key Challenges in Managing Changes

Challenge	Description
High Risk of Failure	Even small errors can cause catastrophic outcomes
Complex Interdependencies	Changes in one module may affect many others
Strict Regulations	Compliance with industry standards (e.g., FDA, DO-178C, ISO 26262)
Hard-to-Test Features	Safety features might only trigger under rare conditions
Legacy Code and Poor Documentation	Understanding and modifying old code is risky

2.3 The Change Management Process

A structured change management process helps control, analyze, and implement modifications safely and systematically.

Step-by-Step Process:

1. **Change Request (CR) Submission**
    - Submitted by developers, testers, users, or system monitors.
    - Must include purpose, scope, affected modules, and justification.
2. **Impact Analysis**
    - Determine how the change affects:
        - System architecture
        - Safety and reliability
        - Performance
        - Downstream modules
    - Tools: Static code analyzers, dependency graphs
3. **Risk Assessment**
    - Identify:

  - Safety risks (e.g., incorrect dosage in medical software)
  - Security risks (e.g., new attack vectors)
  - Reliability risks (e.g., increased crash probability)
- Perform Failure Mode and Effects Analysis (FMEA) or Hazard and Operability Study (HAZOP).

4. **Change Control Board (CCB) Review**
   - Multidisciplinary team (QA, dev, domain experts, safety engineers).
   - Decisions: Approve, Reject, Defer, or Modify.
   - Documentation: All approvals/rejections must be traceable and archived.
5. **Implementation**
   - Developers apply changes using version-controlled branches.
   - All code must follow strict coding standards (e.g., MISRA for C in automotive systems).
6. **Validation and Verification (V&V)**
   - Regression testing, unit testing, integration testing, and acceptance testing.
   - For critical systems: Formal verification methods may be used.
   - Tools: Model checkers (e.g., SPIN), theorem provers (e.g., Coq, Z3)
7. **Deployment**
   - Often done in phased rollout, canary deployments, or A/B testing (in non-critical environments).
   - For real-time or embedded systems, updates may require firmware reflashing and certification checks.
8. **Post-Deployment Monitoring**
   - Use real-time monitoring tools to detect regressions.
   - Tools: Prometheus, Nagios, ELK Stack
   - Maintain error logs, crash reports, and usage analytics.

2.4 Tools for Change Management

Category	**Tools**
Version Control	Git, Subversion
Change Tracking	Jira, Bugzilla, Azure DevOps
Impact Analysis	Lattix, Understand, CodeScene
Test Automation	Selenium, JUnit, Robot Framework
Verification	SPIN, UPPAAL, CBMC

2.5 Regulatory Standards for Dependable Systems

Different domains require formal change control and traceability:

Industry	Standard
Aerospace	DO-178C (Software Considerations in Airborne Systems)
Medical Devices	FDA 21 CFR Part 820, ISO 13485
Automotive	ISO 26262 (Functional Safety for Road Vehicles)
Railways	EN 50128
Nuclear	IEC 60880

These standards demand:

- Rigorous documentation
- Traceable V&V results
- Risk management procedures
- Configuration management records

2.6 Best Practices

Best Practice	Explanation
End-to-End Traceability	Track changes from requirement → design → code → test cases
Code Reviews	Conduct peer reviews to detect issues early
Automated Testing Pipelines	Prevent regressions and improve consistency
Audit Trails	Maintain logs of every change for accountability and regulatory compliance
Redundant and Fail-Safe Design	Ensure that even if part of the system fails, the whole doesn't crash

Example: Medical Infusion Pump Software

Change request: Add alarm sound when flow rate drops.

Process:

- Change request submitted by hospital user.
- Impact analysis shows effect on audio module, power usage, and user interface.
- Risk assessment identifies possibility of alarm failure during low battery.
- Approved by medical safety board.
- Code change in `alarm_controller.cpp`.
- Regression + safety testing done in simulation and real hardware.
- Documentation updated; version 2.3.1 released after FDA review.

3. Handling Dependencies and Ensuring Consistency

3.1 What are Dependencies?

A dependency is any external module, library, API, framework, or component your software relies on.

Example: Your Python project depends on `NumPy`, `pandas`, and the OS it runs on. A change in any of them can affect your system.

3.2 Challenges in Dependency Management

Challenge	Example
Version Conflicts	Library A needs v1.2.0 of X, but B needs v2.0.0
Transitive Dependencies	Your dependency depends on another, which may introduce risks
Breaking Changes	Updating a library may break your code due to API changes
Security Vulnerabilities	Old or unpatched libraries may have known exploits
Dependency Drift	Different environments may install different versions over time

3.3 Dependency Management Techniques

a. Use Dependency Managers

Language	Tool
Python	`pip`, `poetry`, `conda`
Java	`Maven`, `Gradle`
JavaScript	`npm`, `yarn`
.NET	`NuGet`

These tools handle installation, upgrades, resolution, and locking of versions.

b. Semantic Versioning (SemVer)

Format: `MAJOR.MINOR.PATCH`

- **MAJOR:** Incompatible API changes
- **MINOR:** Backward-compatible features
- **PATCH:** Bug fixes

Example: Upgrading from `1.2.0` to `2.0.0` may break your code.

c. Pin and Lock Versions

- Pin versions: Explicitly specify versions (e.g., `pandas==1.3.5`)
- Lockfiles: Freeze the full dependency tree:
  - `package-lock.json` (npm)

- `Pipfile.lock` (Python)
- `yarn.lock`

d. Track Vulnerabilities and Updates
Use tools to scan for security flaws:

- OWASP Dependency-Check
- Snyk
- GitHub Dependabot
- npm audit

e. Automated CI Pipelines
Integrate dependency checks and tests into your CI/CD workflows:

- Run tests every time a dependency changes
- Alert if a new version breaks the build
- Tools: GitHub Actions, Jenkins, GitLab CI

f. Isolate via Interfaces

- Use interface abstraction or facades to reduce tight coupling
- **Example:** Wrap a 3rd-party API in your own class/interface

This way, if the 3rd-party API changes, only your wrapper needs updates — not your whole codebase.

g. Consistent Environments

- Use containers (e.g., Docker) to define a consistent runtime environment
- Use virtual environments for local development (`venv`, `conda`, etc.)

Best Practices

- Document all external dependencies and licenses
- Use dependency update bots (e.g., Renovate, Dependabot)
- Audit dependencies quarterly
- Retire or replace abandoned libraries

4. Reliability Maintenance Over Time

4.1 What is Reliability?

Reliability is the probability that software will function correctly under specified conditions for a specified time.

Goal: "The system runs without failure."

4.2 Threats to Long-Term Reliability

Threat	Example
Software Aging	Performance degradation due to memory leaks, unused resources
Code Rot / Entropy	Accumulated bad code practices
Increasing Complexity	More features → harder to maintain
Uncontrolled Changes	Regression bugs introduced during updates
Obsolete Dependencies	Unsupported or broken third-party tools/libraries

4.3 Strategies to Maintain Reliability

a. Preventive Maintenance

- Regularly refactor code to reduce technical debt
- Use modular design to isolate failures
- Fix latent bugs before they cause failures

b. Automated Monitoring and Logging
Set up:

- Uptime monitors (Pingdom, UptimeRobot)
- Error trackers (Sentry, Rollbar)
- Performance dashboards (Grafana + Prometheus)
- Log aggregators (ELK Stack, Fluentd)

Monitored metrics include:

- Response time
- Error rate
- System throughput
- CPU/memory/disk usage

c. Fault Tolerance and Resilience

- Implement redundancy (e.g., server replicas, backup DBs)
- Use retry logic with backoff strategies
- Use circuit breakers to prevent cascading failures (e.g., Hystrix pattern)
- Allow graceful degradation when part of the system fails

d. Regular Regression Testing

- Maintain automated test suites

- Include edge cases and error conditions
- Use mutation testing to validate test effectiveness
- Run tests in CI pipelines on all code changes

e. Software Configuration Management (SCM)
Ensure:

- Consistent environments
- Reproducible builds
- Version control over code, config, and data

Tools: Git, Ansible, Terraform, Helm (for Kubernetes)
f. Patch Management and Updates

- Apply security and performance patches regularly
- Monitor CVEs (Common Vulnerabilities and Exposures)
- Communicate breaking changes clearly in release notes

g. Documentation and Knowledge Sharing

- Update system design diagrams, user guides, and SOPs
- Conduct regular knowledge transfer sessions
- Document known failure modes and mitigation plans

4.4 Metrics to Track Reliability

Metric	Description
MTBF (Mean Time Between Failures)	Average time system runs without failure
MTTR (Mean Time To Repair)	Time taken to fix a failure
Uptime %	Availability over a time period
Error Rate	Number of failed requests / total requests

Example: Banking Transaction System

Problem: System crashes under high load at month-end billing.
Actions:

- Added automated load testing to CI pipeline
- Refactored transaction queue handling
- Introduced caching layer for account info
- Setup Grafana dashboards with alerts for slow queries

Result: 99.99% uptime achieved, with MTTR < 30 mins