Software Reliability

Ashima Tyagi
Assistant Professor
School of Computer Science & Engineering

Outline

- Introduction
- Hardware reliability vs. Software reliability
- Reliability metrics
- Failure and Faults
- Dependability Concept
- Reliability and Availability Modeling
- Reliability Evaluation Testing methods and models
- Static and Dynamic Analysis

Software Reliability

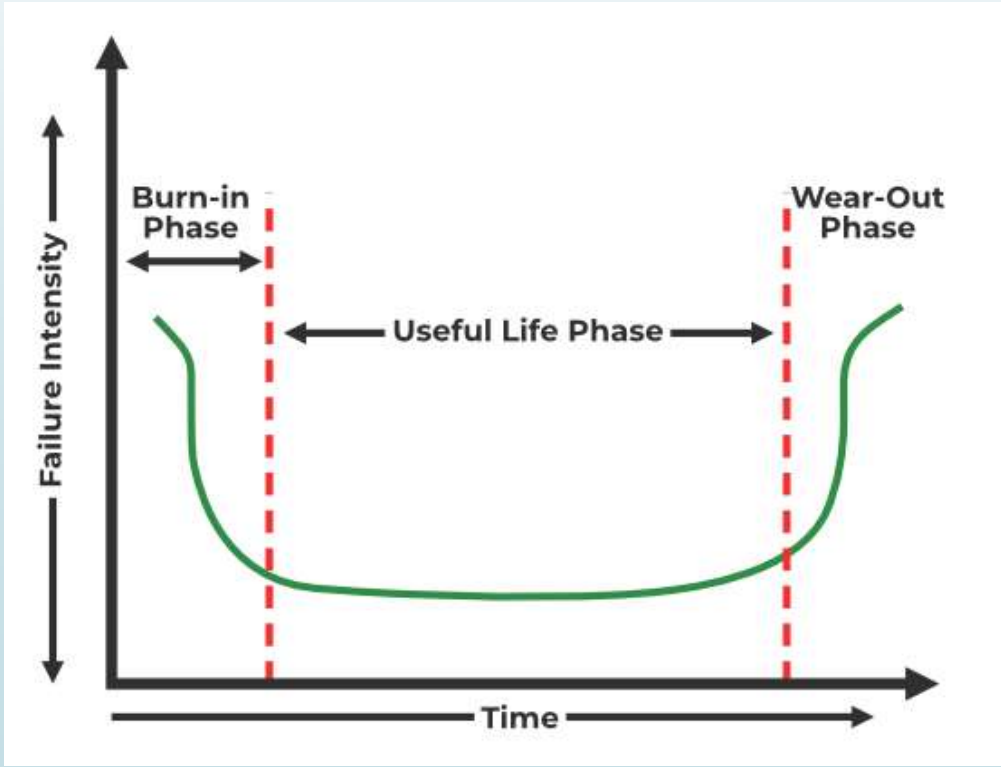
- Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment.
- It is a measure of how dependable and error-free the software is.
- High reliability means fewer bugs and more stable performance.

Examples:

- Consider a weather forecasting app. If the app consistently provides accurate weather predictions without crashes or incorrect information, then its reliability is high. If it often crashes or gives wrong data, its reliability is low.
- Reliability under stress: A reliable e-commerce website can handle the increased traffic (a high load) without crashing or slowing down excessively. It will load pages properly, process payments accurately, and allow users to complete transactions. The website can manage a large number of concurrent users (e.g., thousands or millions) without failure. If the website does crash under heavy load, it would be seen as having low reliability under stress.

Hardware Reliability

- Hardware reliability is the probability that the ability of the hardware to perform its function for some period of time.
- It may change during certain periods such as initial burn-in or the end of useful life.



Burn-in Phase (Early Failures)

- High failure intensity at the beginning occurs due to manufacturing defects, poor-quality components, or assembly errors.
- This phase is also called the infant mortality phase, where weak components fail early.
- Manufacturers often perform stress testing or burn-in testing to eliminate defective parts before reaching customers.

Useful Life Phase (Random Failures)

- Once weak components have failed, the remaining hardware operates reliably.
- The failure rate is relatively constant and low.
- Most failures during this phase are due to random external factors, such as power surges, accidental damage, or unpredictable wear.

Wear-Out Phase (Aging Failures)

- As time progresses, components age and degrade, leading to an increase in failure intensity.
- Common causes include material fatigue, corrosion, and wear.
- Preventative maintenance, replacements, or upgrades are required to extend the product's life.

Causes of Hardware Failures

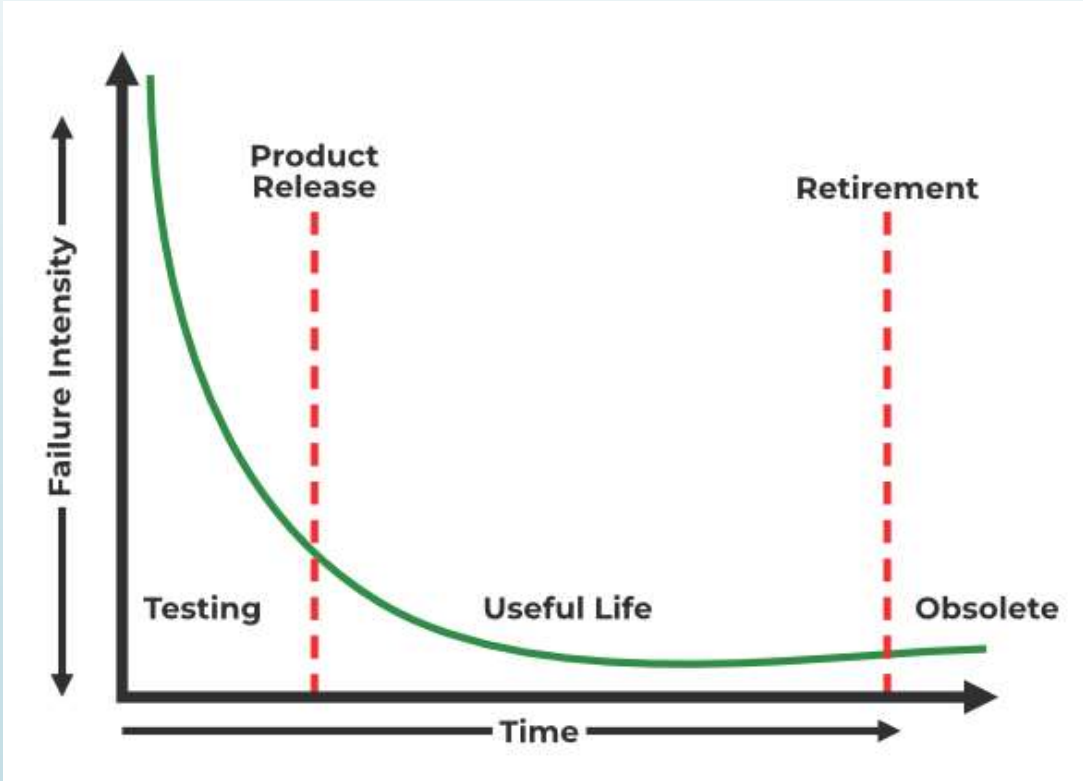
- Physical Wear and Tear → Components degrade over time.
- Environmental Factors → Temperature, humidity, dust, and power surges affect performance.
- Manufacturing Defects → Some components may have defects from production.
- Aging and Corrosion → Metallic parts may corrode, and circuits may weaken.

Improving Hardware Reliability

✓ Redundant Components → Use backup hardware like RAID for storage.
✓ Cooling Systems → Prevent overheating with fans and heat sinks.
✓ Quality Manufacturing → Use high-quality materials and strict testing.
✓ Regular Maintenance → Clean and inspect components regularly.

Software Reliability

- Software reliability is the probability that the software will operate failure-free for a specific period of time in a specific environment.
- It is measured per some unit of time.



Testing Phase (High Failure Intensity Initially)

- Failure intensity is high initially because:
  - Many undetected bugs and errors exist.
  - Code may not be optimized for real-world use.
  - Software engineers actively identify and fix defects.
- Over time, failures decrease as bugs are found and fixed.

Product Release (Start of Useful Life Phase)

- After rigorous testing, the software is released to users.
- Most critical bugs are resolved, but minor or unforeseen issues may still emerge.
- Reliability improves as updates and patches are released.

Useful Life Phase (Stable Period)

- The software operates reliably with minimal failures.
- Bugs and issues are resolved through patches, updates, and maintenance.
- The failure rate remains low and stable, similar to the useful life phase of hardware.

Retirement Phase (End of Software Life)

- The software becomes outdated due to:
  - New technologies and security concerns.
  - Better alternatives (newer software versions).
  - Lack of support from developers.
- The failure rate may slightly increase as updates and security patches are discontinued.
- Eventually, the software becomes obsolete.

Causes of Software Failures

- Bugs and Errors → Programming mistakes or logic errors.
- Inadequate Testing → Software not thoroughly tested for edge cases.
- Complexity → More features increase the chances of defects.
- External Dependencies → Third-party libraries may introduce issues.

Improving Software Reliability

✅ Code Reviews and Testing → Perform unit testing, integration testing, and debugging.
✅ Error Handling → Use proper exception handling to prevent crashes.
✅ Version Control → Maintain stable software versions with updates.
✅ User Feedback and Updates → Continuously fix issues based on reports.

H/W Vs S/W Reliability

FEATURE	HARDWARE RELIABILITY	SOFTWARE RELIABILITY
Nature of Failure	Physical damage or degradation	Logical errors or bugs
Failure Rate Trend	Increases over time (wear and tear)	Remains constant unless fixed
Maintenance	Requires physical repair/replacement	Can be updated or patched
Testing Methods	Stress tests, durability tests	Unit testing, regression testing
MTBF vs. MTTF	Mean Time Between Failures	Mean Time To Failure
Redundancy Use	Extra components for backup	Error handling, backups, and patches
Upgradability	Requires hardware replacement	Software updates and patches

Reliability Metrics

1. Mean Time to Failure (MTTF)

Definition: Average time a non-repairable system operates before failing.

Formula:

$$\text{MTTF} = \frac{\text{Total operational time}}{\text{Number of failures}}$$

Example: If a system runs for 500 hours and fails 5 times,

MTTF=500 / 5 =100 hours

2. Mean Time to Repair (MTTR)

- **Definition:** Average time taken to repair a system after failure.

Formula:

$$\text{MTTR} = \frac{\text{Total repair time}}{\text{Number of repairs}}$$

Example: If it takes 20 hours to fix 4 issues,

MTTR=20 / 4 = 5 hours

3 Mean Time Between Failures (MTBF)

- **Definition:** Average time between consecutive failures in a repairable system.

Formula:

$$\text{MTBF} = \text{MTTF} + \text{Mean Time To Repair (MTTR)}$$

- Example: If MTTF is 100 hours and MTTR is 10 hours,

  MTBF=100+10=110 hours

4. Failure Rate (λ)

- **Definition:** The number of failures per unit of time.

Formula:

$$\lambda = \frac{1}{\text{MTTF}}$$

- Example: If MTTF is 100 hours,

  λ=1 / 100 = 0.01 failures per hour

5. Availability (A)

- **Definition:** Probability that a system is operational at any given time.

Formula:

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

- Example: If MTBF = 110 hours and MTTR = 10 hours,

$$A = 110 / (110+10) = 110 / 120 = 0.9167 \ (\approx 91.67\%)$$

6. Probability of Failure-Free Operation (P)

- **Definition:** Probability that the system operates without failure over time t.

Formula:

$$P(t) = e^{-\lambda t}$$

- Example: If λ = 0.01 and time = 50 hours,

$$P(50) = e^{-0.01 \times 50} \approx 0.6065$$

7. Downtime

- **Definition:** Downtime is the total time a system is non-operational or unavailable due to failures, maintenance, or other interruptions.

Formula:

$$\text{Downtime} = \text{Total Time} \times (1 - \text{Availability})$$

Where:

- **Availability** is typically expressed as a decimal (e.g., 0.95 for 95%).
- **Total Time** is the time frame considered (e.g., total hours in a year).

8. Uptime

- **Definition:** Uptime refers to the total time a system is operational and available without failure.

Formula:

$$\text{Uptime} = \text{Total Time} - \text{Downtime}$$

Where:

- **Total Time** is the entire period considered (e.g., hours in a day, month, or year).
- **Downtime** is the time when the system was non-operational

➨ Where Are Downtime and Uptime Used?

1. Availability Calculation: Measures system reliability and operational performance (used in servers, networks, and cloud services).

$$\text{Availability} = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}}$$

2. Mean Time Between Failure: Predicts when a system might fail next and helps in maintenance scheduling.

$$\text{MTBF} = \text{Uptime} + \text{Downtime}$$

3. **Mean Time To Repair:** Assesses how quickly a system can recover after a failure—essential for service level agreements (SLAs).

$$\text{MTTR} = \frac{\text{Downtime}}{\text{Number of Failures}}$$

Practice questions on reliability metrics

- Check a Separate doc file.

Error and Bug

Error

Definition: An error is a human mistake made by the developer while writing the code, designing the system, or configuring the environment.

- *When It Happens:* During the development phase.
- *Example:* A developer accidentally writes x = y + z instead of x = y - z.
- Misconfiguring a database connection string.

Bug

Definition: A bug is a flaw or fault in the software that causes it to behave unexpectedly or incorrectly.

- *When It Happens:* Detected during testing or when the software runs.
- *Example:* The program crashes when a user clicks a button.
- A feature doesn't work as intended because of a coding error.

Feature	Error	Bug
What it is	Human mistake	Result of that mistake
Detected	During development	During testing or execution
Cause	Coding, logic, or design flaw	Code not working as expected
Example	Typing wrong logic in code	App crashes due to that logic

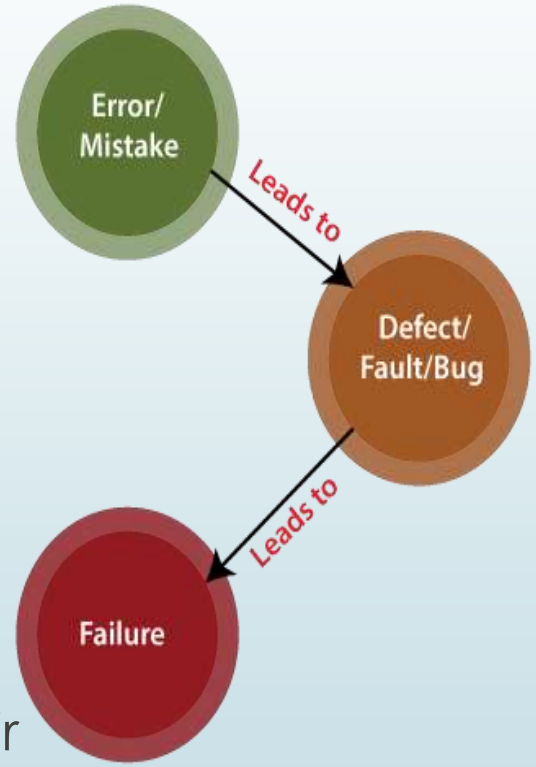
Fault and Failure

⚙ A **fault** (or defect/bug) is an underlying flaw or error in a system's code, design, or architecture. A fault may or may not lead to a failure.

- **Example:**
  If a developer writes incorrect logic for a login validation, that's a fault. However, users might not notice it until they input specific incorrect data.

✕ **Failure:** It occurs when a system doesn't perform as expected due to one or more faults being triggered during execution.

- **Example:**
  When the faulty login validation prevents users from accessing their accounts even after entering the correct credentials, that's a failure.



✓ 1 Fault Leading to Failure

When a fault gets triggered during execution, causing the system to behave unexpectedly or fail to meet requirements.

Examples:

Login System Bug

Fault: A developer writes the condition if (username == storedUsername) without ignoring case sensitivity.

Failure: The system rejects a valid username if the user enters "JohnDoe" instead of "johndoe" when it was meant to be case-insensitive for convenience.

Division by Zero in Calculator App

Fault: Missing a check for division by zero in the code.

Failure: When a user tries to divide a number by zero, the program crashes unexpectedly.

Automotive Cruise Control System

Fault: A sensor provides incorrect speed data due to a miscalibration.

Failure: The cruise control accelerates unexpectedly, causing a safety issue.

❌ 2️⃣ Fault Not Leading to Failure

The fault exists in the system but doesn't cause a failure because it's never triggered or has no impact on execution.

Examples:

🗓️ Dead Code in Calendar Application

Fault: A feature for event notifications has been written but never integrated into the final app flow.

No Failure: The notification feature code is never executed, so users don't experience any issue.

💻 Memory Leak in an Unused Function

Fault: A memory leak exists in a function that allocates memory but doesn't free it properly.

No Failure: Since the function is never called during execution, the memory leak doesn't affect the system.

📶 Incorrect Error Message in Network System

Fault: A wrong error message is set to display when the internet is disconnected.

No Failure: If the network connection remains stable, this fault never triggers and the users don't encounter it.

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.

Techniques:

- Well-defined requirements and design specifications
- Use of coding standards and best practices
- Regular team code reviews and pair programming
- Developer training and upskilling

Example:
Following strict coding guidelines and conducting frequent peer reviews to ensure logical consistency can help prevent faults from occurring in a banking application.

- 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.

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.

Techniques:

- Static code analysis (detects syntax and structural issues)
- Code inspections and peer reviews
- Comprehensive testing (unit, integration, system testing)
- Debugging during development

- **Example:**
  A software team running unit tests to catch and fix off-by-one errors before releasing a file-processing tool.
- 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.

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.

Techniques:

- Redundancy (backup systems or duplicate components)
- Exception handling and graceful degradation
- Recovery blocks and failover mechanisms
- Checkpointing for system recovery

Example:
In an e-commerce platform, if a payment gateway fails, the system automatically switches to a backup payment processor to complete transactions without affecting the user experience.

- 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 Forecast

- Predict the occurrence and impact of potential faults to prepare for or prevent failures.

Techniques:

- Reliability modeling and simulations
- Statistical analysis of failure data
- Risk assessment and management
- Predictive analytics

Example:
Using historical server downtime data to predict potential system failures and schedule preventive maintenance for a cloud storage service.

Aspect	Goal	Techniques	Example
Fault Prevention	Avoid introducing faults	Coding standards, peer reviews	Prevent logic errors in financial software
Fault Removal	Detect and eliminate faults	Testing, debugging, code inspections	Fix memory leaks found during testing
Fault Tolerance	Ensure system runs despite faults	Redundancy, failover systems, exception handling	Backup server activation during main server failure
Fault Forecast	Predict and minimize the impact of faults	Risk assessment, reliability models	Predict server failures using historical downtime data

Techniques for detecting software faults

There are several methods used to identify and resolve faults in software engineering, including:

- **Code Reviews:** A code review is a process in which other developers or team members review the code written by a developer to identify potential errors or areas for improvement. This can be done manually or with automated tools.
- **Testing:** Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not. There are several types of testing, such as unit testing, integration testing, and acceptance testing, which can help identify faults in the software.

- **Debugging:** Debugging is the process of identifying and resolving faults in the software by analyzing the program's source code, data, and execution. Debugging tools, such as debuggers, can help developers identify the source of a fault and trace it through the code.
- **Monitoring:** Monitoring is the ongoing process of tracking and analyzing the performance and behavior of a system. Monitoring tools, such as log analyzers, can help identify and diagnose faults in production systems.
- **Root Cause Analysis**: Root cause analysis is a method used to identify the underlying cause of a fault, rather than just addressing its symptoms. This can help prevent the same fault from occurring in the future.

Techniques for mitigating software faults

- **Code Reviews:** Code review is a process where code is evaluated by peers to identify potential problems and suggest improvements.
- **Unit Testing:** Unit testing involves writing automated tests for individual units of code to ensure that each component works as expected.
- **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.
- **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.
- **Static Code Analysis:** Static code analysis is the process of automatically analyzing code to find potential problems without actually executing it.
- **Design and Architecture Reviews:** Reviews of design and architecture help ensure that the overall design of the software is correct and scalable.
- **Error Handling:** Proper error handling helps prevent the spread of faults and ensures that the software can handle unexpected situations gracefully.
- **Documentation:** Clear and up-to-date documentation can help reduce the likelihood of errors by providing a clear understanding of the software's functionality

Dependability Concept

- Dependability in software engineering refers to the ability of a system to deliver reliable and correct services while handling faults and failures effectively.
- It ensures that the system remains available, secure, and maintains integrity even under failures or external threats.

Key Attributes of Dependability

1. **Reliability** – The system operates correctly over a given period.
2. **Availability** – The system is operational and accessible when needed.
3. **Safety** – The system does not cause unacceptable risk.
4. **Integrity** – The system prevents unauthorized data modification.
5. **Maintainability** – The system can be updated, repaired, or modified efficiently.

Failure Behavior

Failure behavior describes how a system responds when faults occur. Failures can be classified into different types based on their occurrence and impact:

📌 Types of Failures:

1. **Transient Failures** – Temporary failures that disappear after some time (e.g., network glitches).
2. **Intermittent Failures** – Failures that occur irregularly and unpredictably (e.g., hardware overheating).
3. **Permanent Failures** – Persistent failures that require intervention (e.g., corrupted database).

📌 Failure Models:

- **Fail-Safe** – The system enters a safe state when a failure occurs (e.g., circuit breakers in power grids).
- **Fail-Soft** – The system continues operation at a reduced level (e.g., battery-saving mode in devices).
- **Fail-Stop** – The system halts completely upon failure detection (e.g., emergency shutdown in industrial systems).

Characteristics of Dependability

1. *Reliability*
2. *Availability*
3. *Safety*
4. *Security*
5. *Maintanability*
6. *Integrity*
7. *Fault Prevention, Removal, Tolerance, Forecast*
8. *Recoverability*
9. *Consistency*

🛠 Example Scenarios

Case 1: Banking System Failure Prevention

A banking app prevents transaction failures by verifying user authentication before processing payments.

If the system detects an error, it rolls back the transaction instead of completing a faulty transaction.

Case 2: Flight Control System Failure Tolerance

A flight control system uses redundant processors. If one processor fails, the backup system takes over, ensuring safe operations.

Case 3: E-commerce Website Failure Forecasting

An online store predicts high traffic during sales and scales servers in advance to prevent downtime.

◆ Maintenance Policy in Dependability

Maintenance policies define how a system is maintained to ensure dependability. There are different approaches to maintenance:

📌 1. Corrective Maintenance

Fixing faults after they occur.

Example: Patching security vulnerabilities in an application.

📌 2. Preventive Maintenance

Regular updates and checks to prevent future failures.

Example: Scheduled software updates and database optimizations.

📌 3. Predictive Maintenance

Using monitoring tools to predict and address potential failures.

Example: AI-based failure detection in cloud infrastructure.

📌 4. Adaptive Maintenance

Modifying a system to accommodate changes in the environment.

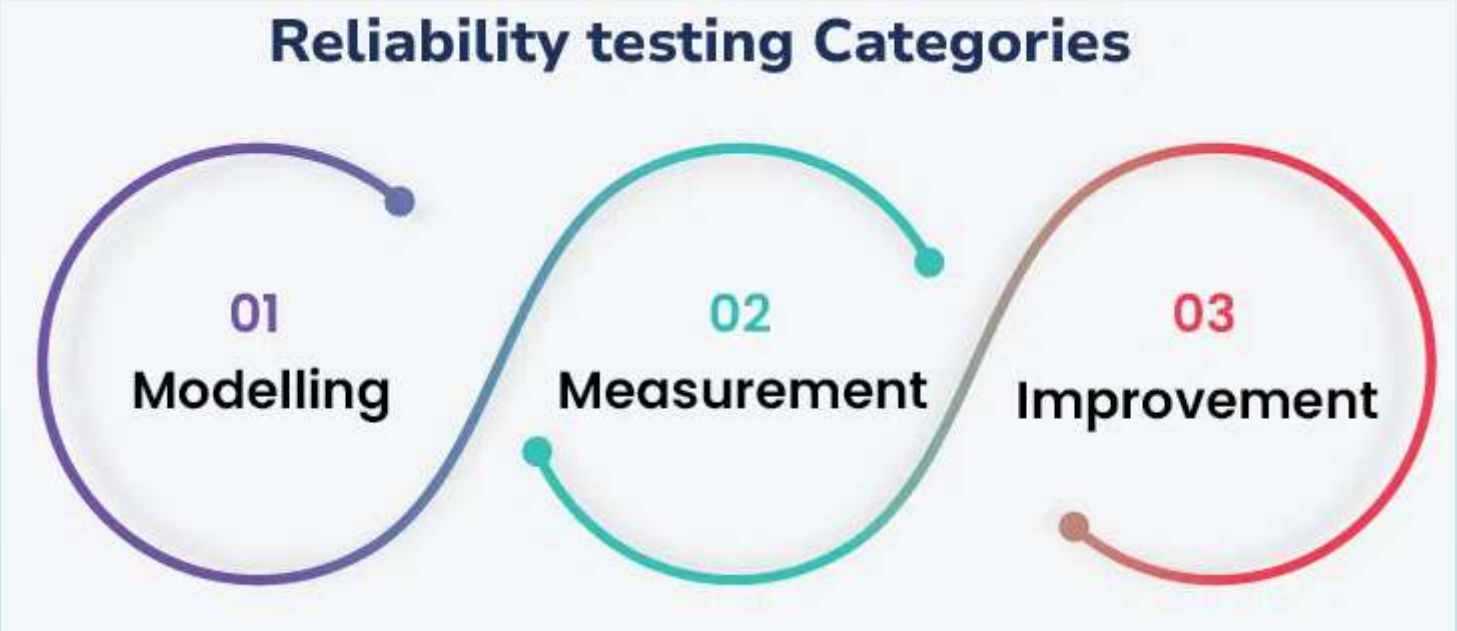
Example: Upgrading software to support new operating systems.

Reliability and Availability Modeling

- Reliability and Availability are two critical attributes of a system, particularly in software engineering, where ensuring a system's ability to function consistently and be accessible is essential for user satisfaction and business continuity.
- Modeling these attributes helps in designing and analyzing systems to ensure they meet the required performance standards.

Reliability Evaluation Testing methods

- Reliability testing is a type of software testing that evaluates the ability of a system to perform its intended function consistently and without failure over an extended period.



1. Modelling

Modelling in reliability testing involves creating mathematical or statistical representations of how a product or system might fail over time. It's like making an educated guess about the product's lifespan based on its design and components. This helps predict when and how failures might occur without actually waiting for the product to fail in real life.

Example: Engineers might create a model to estimate how long a new smartphone battery will last before it degrades significantly.

2. Measurement

Measurement focuses on collecting real-world data about a product's performance and failures. This involves testing products under various conditions and recording when and how they fail. It's about gathering concrete evidence of reliability rather than just predictions.

Example: A car manufacturer might test drive hundreds of cars for thousands of miles, recording any issues that arise during these tests.

3. Improvement

Improvement uses the insights gained from modelling and measurement to enhance the reliability of a product or system. This involves identifying weak points, redesigning components, or changing manufacturing processes to make the product more reliable.

Example: After finding that a particular part in a washing machine fails frequently, engineers might redesign that part or choose a more durable material to improve its lifespan.

Different Ways/Methods to Perform Reliability Testing

- **Stress testing:** Stress testing involves subjecting the system to high levels of load or usage to identify performance bottlenecks or issues that can cause the system to fail. Example: Testing a ride-booking app (Uber/Ola) during peak hours.
- **Endurance testing:** Endurance testing involves running the system continuously for an extended period runs the system continuously for long periods to check for memory leaks, slowdowns, or performance degradation. Example: A video streaming app (Netflix) runs non-stop for 48 hours to check if memory usage increases over time.
- **Recovery testing:** Recovery testing tests the system's ability to recover from crashes, failures, or network disconnections. Example: A user is making a money transfer, and the system crashes mid-transaction. The testing checks if the system can restore its last correct state without data corruption.

- **Environmental Testing:** Conducting tests on the product or system in various environmental settings, such as temperature shifts, humidity levels, vibration exposure or shock exposure, helps in evaluating its dependability in real-world circumstances. Example: A self-driving car must operate in extreme temperatures (-30°C in Canada, +50°C in deserts).
- **Performance Testing:** Evaluates if the system meets performance criteria under both normal and peak load conditions. Example: The website must handle millions of users during a Big Billion Day Sale without slowing down.
- **Regression Testing:** After every software update, regression testing ensures that previous features still work correctly. Helps detect new bugs introduced by recent changes. Example: Prevents new updates from breaking existing functionalities.
- **Fault Tree Analysis:** Identifies possible failures and examines how different system components contribute to failures. Uses a tree-like diagram to trace failures back to their root causes. Example: If a navigation failure occurs, fault tree analysis helps find the root cause, such as: GPS signal lost, Sensor failure, Software bug in autopilot mode.

Limits of Reliability Testing

Limitation	Explanation	Example
Time Constraints	Some reliability tests require long durations	Testing a space probe's software for a 10-year mission
Cost Constraints	Large-scale tests can be expensive	Cloud companies spend millions on server reliability tests
Unknown Failure Modes	Not all failure scenarios can be predicted	Unexpected cyberattacks affecting system performance
Environmental Variability	External factors change reliability results	Network reliability varies by location and traffic

Starvation in Reliability Evaluation

Starvation occurs when certain processes or requests are indefinitely delayed because system resources prioritize other operations.

📌 Examples of Starvation in Software Systems

1. A low-priority process may never get CPU time because high-priority processes keep running.
2. A low-priority query may never execute because the database always prioritizes faster queries.
3. In cloud computing, some virtual machines may be starved of resources if others consume too much bandwidth.

◆ How to Prevent Starvation?

✓ *Priority Aging:* Increases the priority of waiting tasks over time.
✓ *Fair Scheduling:* Ensures all requests get equal access to resources.
✓ *Resource Allocation Strategies*: Limits high-priority tasks from consuming all resources.

Coverage in Reliability Testing

Coverage refers to how much of the system is tested for reliability. Higher coverage means more test cases are executed.

📌 Types of Coverage in Reliability Testing:

Type of Coverage	Description	Example
Code Coverage	Ensures all lines of code are tested	Unit testing every function in a Python program
Path Coverage	Tests all possible execution paths	Checking all login scenarios in a banking app
Fault Coverage	Tests different failure cases	Simulating server crashes in cloud computing
Condition Coverage	Checks all if-else conditions	Testing both valid and invalid inputs in forms

✅ Higher coverage ensures fewer undetected failures.

Filtering in Reliability Evaluation

Filtering helps in removing irrelevant failures or noise to focus on critical failures.

📌 How Filtering Works?

1. Noise Reduction: Removes false failure reports caused by temporary glitches.
2. Severity-Based Filtering: Focuses only on high-impact failures that disrupt system performance.
3. Duplicate Failure Filtering: Removes duplicate error reports to avoid unnecessary debugging.

📌 Example: Filtering in Log Analysis

🚀 Web servers (Google, Facebook): If millions of error logs are generated, filtering removes duplicate errors and focuses on critical issues like security breaches.

Reliability Models

- Software reliability models help predict and measure the dependability of software by **analyzing failure behavior, fault rates, and system performance over time.**
- These models are crucial in ensuring software quality, improving maintainability, and reducing system failures.

Some of the Software Reliability Models that are used to achieve reliability in Software Development are:-

- Jelinski and Moranda Model
- Basic Execution Time Model
- Logarithmic Poisson Time Model
- The Bug Seeding Model
- Shooman Model
- Littlewood-Verrall Model
- Goel-Okumoto Model
- Musa-Okumoto Model

Classes of Software Reliability Models

Software reliability models are broadly classified into three main categories:

I. Deterministic Models

- These models assume a fixed number of faults in the software.
- They do not consider randomness in software failures.
- Used primarily for quality assurance and defect estimation.
- Example: Halstead's Software Metric Model (measures code complexity and defect likelihood).

II. Probabilistic Models

- These models assume that software failures occur randomly and follow a probability distribution.
- Used for reliability predictions based on historical failure data.
- Example: Jelinski-Moranda Model (models failures over time using a constant failure rate).

III. Fault-Based Models

- These models focus on fault detection and removal to predict software reliability growth.
- Assumes that debugging reduces failure rates over time.
- Example: Musa-Okumoto Model (models failure intensity and software improvement over time).

Statistical Reliability Models

Statistical models use **failure data** to estimate software reliability. These models fall into two major categories:

I. Time Between Failure (TBF) Models

- Measure time intervals between consecutive failures.
- Assume failure times follow a probability distribution (e.g., exponential or Weibull).
- Example: Musa Basic Model (predicts reliability based on failure times).

II. Failure Count Models

- Focus on the number of failures in a given time period.
- Used when failures occur at discrete time points.
- Example: Goel-Okumoto Model (predicts software failure rates based on observed defect counts).

Reliability Growth Models

The growth model represents the reliability or failure rate of a system as a function of time or the number of test cases.

Models included in this group are as follows:

- Coutinho Model
- Wall and Ferguson Model

Coutinho Model

In the log-log paper, Coutinho charted the cumulative number of defects identified and the number of corrective measures taken vs. the cumulative testing weeks. Let N(t) represent the total number of failures and t represent the entire testing duration. The model's failure rate, lambda (t), may be represented as,

$$\lambda_{(t)} = \frac{N(t)}{t} = \beta_0 t^{-\beta 1}$$

N′(t) (Failure Rate Function)
β0 (Fault Detection Rate Constant)
b(t) (Fault Detection Function)
b′(t)b'(t)b′(t) (Rate of Fault Detection)

Where,

β0 and β1 are model parameters. This model's parameters may be estimated using the least-squares approach.

Wall and Ferguson Model

- The total number of failures at time t, m(t), may be written as,

$$m_{(t)} = a_0 [b(t)]^3$$

a0 (Initial Number of Faults)
b(t) (Fault Detection Growth Function)

where alpha0: is unidentified parameter. The number of test cases or total testing time can be used to calculate the function b(t).

- Similarly, at time t, the failure rate function is given by,

$$\lambda_{(t)} = m'(t) = a_0 \beta b'(t)[b(t)]^{\beta-1}$$

a0 (Initial Failure Rate)
β (Decay Parameter)

Wall and Ferguson evaluated their model using a variety of software failure data and discovered that the failure data correlated well with the model.

Static Analysis

- Static analysis involves no dynamic execution of the software under test and can detect possible defects in an early stage, before running the program.
- Static analysis is done after coding and before executing unit tests.
- Static analysis can be done by a machine to automatically "walk through" the source code and detect noncomplying rules. The classic example is a compiler which finds lexical, syntactic and even some semantic mistakes.
- Static analysis can also be performed by a person who would review the code to ensure proper coding standards and conventions are used to construct the program. This is often called Code Review and is done by a peer developer, someone other than the developer who wrote the code.
- Static analysis is also used to force developers to not use risky or buggy parts of the programming language by setting rules that must not be used.

When developers performs code analysis, they usually look for

Lines of code
Comment frequency
Proper nesting
Number of function calls
Cyclomatic complexity
Can also check for unit tests

Quality attributes that can be the focus of static analysis:

Reliability
Maintainability
Testability
Re-usability
Portability
Efficiency

There are several types of static analysis methods-

Control Analysis :-

This software focuses on examining the controls used in calling structure, control flow analysis and state transition analysis. The calling structure is related to the model by identifying the calling and call structure. The calling structure can be a process, subroutine, function, or method. Control flow analysis checks the sequence of control transfers. Furthermore, it inefficient constructions in the model. A graph of the model is created in which the conditional branches and model junctions are represented by nodes.

Data Analysis :-

Ensures proper operation is applied to data objects such as data structures and linked lists. In addition, this method also ensures that the defined data is used properly. Data analysis involves two methods, namely, data dependency and data-flow analysis. Data dependency is necessary to assess the accuracy of synchronization across multiple processors. Data flow analysis checks the definition and context of variables.

Fault/Failure Analysis :-

It analyzes faults (incorrectly component) and failure (incorrect behavior of model component) in the model. This method uses the input-output transformation description to identify the conditions that are cause for the failure. To determine the failures in certain conditions the model design specification is checked.

Interface Analysis :-

This software verifies interactive and distribution simulations to check the code. There are two basic techniques for interface analysis and user interface analysis examines sub model interfaces and determines the accuracy of interface structure. User interface analysis examines the user interface model and for the precautionary steps taken to prevent errors during the user's interaction with the model. This method also focuses on how accurately the interface is integrated into the overall model and simulation.

Dynamic Analysis

- In contrast to Static Analysis, where code is not executed, dynamic analysis is based on the system execution, often using tools.
- Dynamic program analysis is the analysis of computer software that is performed with executing programs built from that software on a real or virtual processor (analysis performed without executing programs is known as static code analysis). Dynamic program analysis tools may require loading of special libraries or even recompilation of program code.
- The most common dynamic analysis practice is executing Unit Tests against the code to find any errors in code.

Dynamic code analysis advantages:

- It identifies vulnerabilities in a runtime environment.
- It allows for analysis of applications in which you do not have access to the actual code.
- It identifies vulnerabilities that might have been false negatives in the static code analysis.
- It permits you to validate static code analysis findings.
- It can be conducted against any application.

Dynamic code analysis limitations:

- Automated tools provide a false sense of security that everything is being addressed.
- Cannot guarantee the full test coverage of the source code
- Automated tools produce false positives and false negatives.
- Automated tools are only as good as the rules they are using to scan with.
- It is more difficult to trace the vulnerability back to the exact location in the code, taking longer to fix the problem.

Objectives of Dynamic Testing

- Find errors and bugs: Through comprehensive testing, find and expose flaws, faults, or defects in the software code and its functionality so that they can be fixed as soon as possible.
- Verify the behavior of the system: Verify that the software operates as expected and complies with company requirements, industry or regulatory standards, user expectations, and any applicable business regulations.
- Assessing Performance: To make sure the software satisfies performance requirements, evaluate its performance by monitoring reaction times, throughput, and use of resources under various scenarios.
- Assure Trustworthiness: Examine the software's dependability by determining how well it performs regularly under typical operating conditions, free of unexpected faults or crashes.
- Accuracy of Test Data: Verify the precision and consistency of the data handled by the software to guarantee reliable and uniform information handling.
- Assess Scalability: Examine whether the application can grow to handle more users, workloads, or data volumes without seeing an obvious decline in performance.

Dynamic Analysis Methods

- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Performance testing
- Security testing

Thank You