Software Reliability

Ashima Tyagi
Assistant Professor
School of Computer Science & Engineering

Outline

- Introduction
- Hardware reliability vs. Software reliability
- Reliability metrics
- Failure and Faults – Prevention, Removal, Tolerance, Forecast;

Software Reliability

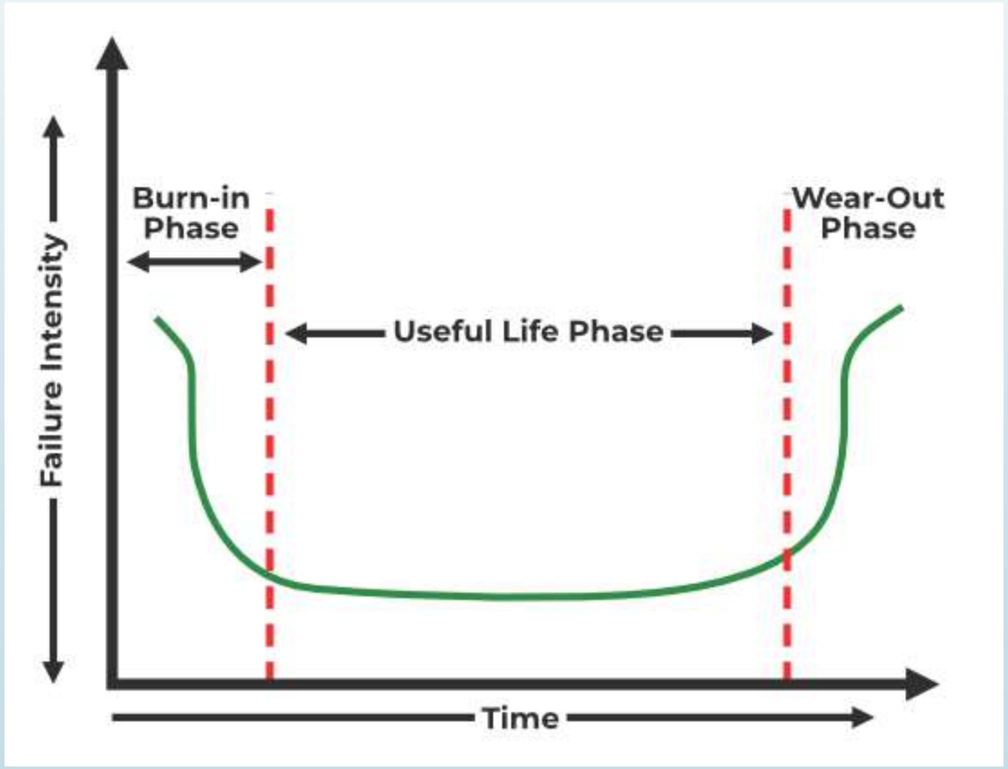
- Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment.
- It is a measure of how dependable and error-free the software is.
- High reliability means fewer bugs and more stable performance.

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

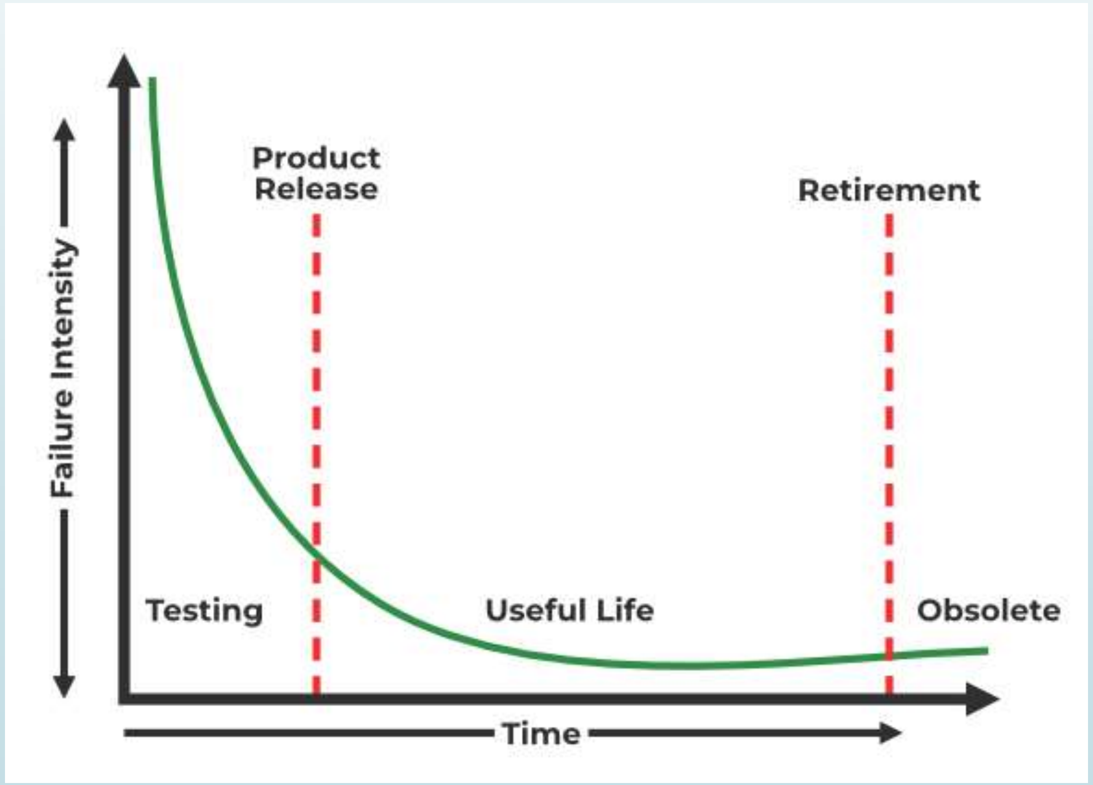
- ➨ Physical Wear and Tear → Components degrade over time.
- ➨ Environmental Factors → Temperature, humidity, dust, and power surges affect performance.
- ➨ Manufacturing Defects → Some components may have defects from production.
- ➨ Aging and Corrosion → Metallic parts may corrode, and circuits may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



S/W Reliability

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

- **1️⃣ 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

2⃣ Mean Time to Repair (MTTR)

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

Formula:

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

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

MTTR=20 / 4 = 5 hours

➨ 3️⃣ 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:

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

- Example: If MTTF is 100 hours,

  $\lambda = 1 / 100 = 0.01$ failures per hour

5️⃣ Availability (A)

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

Formula:

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

- 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,

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

7️⃣ 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).

8️⃣ **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.

Error and Bug

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.

✅ 1️⃣ 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

Thank You