Software Dependability

1a) Define Software Dependability. Describe the key dimensions of dependability with suitable examples. [7 Marks]

→

Software Dependability means the degree of trust that users can have in a software system. It shows how much confidence users have that the system will operate correctly and will not fail during normal use. A dependable system ensures reliability, availability, safety, and security so that users can rely on it for critical operations. For example, a banking application must be dependable because even small failures can cause financial loss.

The **key dimensions of dependability** are:

1. **Availability** – It is the probability that a system is operational and ready to deliver services when required.
   Example: An online payment gateway being available 99.9% of the time.
2. **Reliability** – It is the probability that a system operates failure-free for a specified time.
   Example: An ATM machine completing thousands of transactions without error.
3. **Safety** – It ensures the system does not cause harm to people or the environment.
   Example: An insulin pump in healthcare must not overdose a patient even if some fault occurs.
4. **Security** – It is the ability of the system to resist accidental or deliberate attacks.
   Example: A social media platform protecting user data from hackers.

Other related properties are:

- **Maintainability** (ease of fixing and updating software),
- **Repairability** (how quickly the system can be restored after failure),
- **Survivability** (ability to continue functioning during attacks or failures),
- **Error tolerance** (handling user mistakes gracefully).

Hence, dependability is an umbrella concept combining all these attributes to make a system trustworthy and acceptable for users.

1b) Differentiate between Discrete-Time and Continuous-Time Markov Chains with relevant examples in software systems. Also, explain the memoryless property of Markov models. Why might this property be a limitation? Provide a counter example to illustrate. [8 Marks]

→

A Markov Chain is a mathematical model used to represent systems that transition from one state to another with certain probabilities. They are widely used in software reliability modeling.

Discrete-Time Markov Chain (DTMC):

- State transitions occur at fixed, discrete time steps (e.g., t = 1, 2, 3...).
- Transition probabilities are defined for each step.
- Example in software: Modeling the reliability of a web server where at each request (discrete event), the system either succeeds or fails.

Continuous-Time Markov Chain (CTMC):

- State transitions can occur at any point in continuous time.
- Transition probabilities depend on time duration and are usually modeled using exponential distributions.
- Example in software: Modeling failure and repair of cloud servers where failures can occur at any random time, not just at fixed steps.

The memoryless property means that the future state of the system depends only on the current state, not on how the system reached that state.

- Formally:
  $P(X_{n+1} = j \mid X_n = i, X_{n-1}, \ldots, X_0) = P(X_{n+1} = j \mid X_n = i)$
- Implication: The past history is irrelevant; only the present matters.

In real software systems, the probability of failure often depends on past usage, accumulated stress, or aging effects. Markov chains ignore this dependency, which can make the model unrealistic.

Counter Example:

Suppose a software system runs continuously and tends to accumulate memory leaks. The longer it has been running, the higher the probability of failure. However, a Markov chain would always assume the same failure probability regardless of how long the software has been running. This underestimates the real failure risk.

2a) A software system is being tested, and the following parameters are given:
Total Expected failures (v0) = 150
Failures experienced so far = 75
Initial Failure intensity (λ0) = 25 failures/CPU hour
Target Failure intensity (λF) = 8failures/CPU hour
Find:

- **i) Current failure intensity λ(75)**
- **ii) Decrement in failure intensity per failure.**
- **iii) Failures and intensity after: 15 CPU hrs 80 CPU hrs**
- **iv) Additional failures and CPU hrs needed to reach λF = 5**

[8 Marks]

→

Formulas used:

- Failure intensity vs. failures removed: $\lambda(\mu) = \lambda_0 \cdot (1 - \mu/v_0)$
- Cumulative failures vs. time: $\mu(t) = v_0 \cdot (1 - e^{-(\lambda_0/v_0)t})$
- Intensity vs. time: $\lambda(t) = \lambda_0 \cdot e^{-(\lambda_0/v_0)t}$
  Given: $v_0 = 150$, $\lambda_0 = 25$ failures/CPU hr, failures so far $\mu = 75$.

i) **Current failure intensity λ(75)**
$\lambda(75) = 25 \cdot (1 - 75/150) = 25 \cdot (1 - 0.5) =$ **12.5 failures/CPU hr**.
ii) **Decrement in failure intensity per failure**
$\Delta\lambda$ per failure $= \lambda_0 / v_0 = 25/150 =$ **0.1667 failures/CPU hr per failure**.
iii) **Failures and intensity after time t**
Use rate $\lambda_0/v_0 = 25/150 = 1/6$ per hr.

- **t = 15 CPU hrs**
  $\mu(15) = 150 \cdot (1 - e^{-(1/6)\cdot 15}) = 150 \cdot (1 - e^{-2.5}) \approx$ **137.69 failures**
  $\lambda(15) = 25 \cdot e^{-2.5} \approx$ **2.052 failures/CPU hr**
- **t = 80 CPU hrs**
  $\mu(80) = 150 \cdot (1 - e^{-(1/6)\cdot 80}) \approx$ **150.00 failures** (essentially all found)
  $\lambda(80) = 25 \cdot e^{-(80/6)} \approx$ **0.0000405 failures/CPU hr** ($\approx 4.05 \times 10^{-5}$)

iv) **Additional failures and CPU hours needed to reach λF = 5**
Target μ^* from start: $\mu^* = v_0 \cdot (1 - \lambda F/\lambda_0) = 150 \cdot (1 - 5/25) = 150 \cdot 0.8 =$ **120 failures**.
Already have 75 ⇒ **additional failures = 120 − 75 = 45**.
Current intensity $\lambda_{now} = 12.5$. Extra time needed:
$\Delta t = (v_0/\lambda_0) \cdot \ln(\lambda_{now} / \lambda F) = (150/25) \cdot \ln(12.5/5) = 6 \cdot \ln(2.5) \approx$ **5.50 CPU hrs**.

2b) What is FMEA (Failure Mode and Effects Analysis)? Describe its process and importance in software reliability assessment. [7 Marks]

→

FMEA (Failure Mode and Effects Analysis) is a step-by-step technique used to identify all possible ways a system, process, or software might fail, analyze their causes and effects, and prioritize actions to prevent them. It was first developed by the U.S. military in the 1940s and is widely applied in industries like automotive, aerospace, healthcare, and software systems.

Process of FMEA

1. **Define Scope** – Decide whether the focus is on the entire system, a specific design, or a process.
2. **Assemble a Team** – Involve cross-functional experts (developers, testers, users).
3. **Identify Components/Steps** – Break the system or process into parts.
4. **List Potential Failure Modes** – Identify how each part could fail (e.g., incorrect output, crash, data loss).
5. **Analyze Effects and Causes** – Study the consequences of each failure on system operation and users.
6. **Rate Severity, Occurrence, Detection** – Assign scores (1–10) for each.
7. **Calculate Risk Priority Number (RPN)** = Severity × Occurrence × Detection.
8. **Prioritize and Recommend Actions** – Focus on the highest RPN failures.
9. **Implement Improvements** – Fix design flaws, add error checks, or strengthen testing.
10. **Reassess and Update** – Recalculate RPN after changes to ensure reduced risk.

Importance in Software Reliability Assessment

- **Proactive Risk Management:** Identifies potential software faults before they occur.
- **Improved Reliability:** Helps avoid critical failures that could harm users or cause downtime.
- **Prioritization:** Uses RPN to focus on the most critical risks.
- **Cost Reduction:** Preventing failures early is cheaper than fixing them post-release.
- **User Trust:** Enhances safety, security, and dependability of software systems.

Example: In a banking app, FMEA might reveal that "transaction failure due to network timeout" is high-risk. By addressing it early (adding retries or fallback mechanisms), reliability improves significantly.

Thus, FMEA is an essential preventive tool in software reliability, ensuring systems are dependable and risks are minimized.

3a) Describe the role of hazard analysis in safety engineering. Outline the steps to formulate safety requirements for an autonomous vehicle system. [7 Marks]

→

Role of Hazard Analysis in Safety Engineering:

Hazard analysis is a systematic process used to identify potential situations that can lead to accidents or harm in a system. In safety engineering, it plays a crucial role by:

- Detecting unsafe conditions and events early in design.
- Assessing the likelihood and severity of hazards.
- Guiding the creation of preventive and protective measures.
- Ensuring that safety requirements are explicitly included in the system specification.
- Reducing risks to an acceptable level, especially in safety-critical domains like aviation, healthcare, or autonomous vehicles.

Steps to Formulate Safety Requirements for an Autonomous Vehicle System:

1. **Hazard Identification**
   - List potential hazards like sensor failure, brake malfunction, GPS errors, or software bugs in navigation.
2. **Hazard Classification**
   - Categorize hazards by severity (e.g., collision = critical, route deviation = moderate).
   - Use risk matrices to combine severity and probability.
3. **Hazard Analysis**
   - Perform **Fault Tree Analysis (FTA)** or **Failure Mode and Effects Analysis (FMEA)** to understand causes and effects of hazards.
   - Consider both hardware faults and software errors.
4. **Define Safety Goals**
   - Translate hazard findings into high-level safety goals.
   - Example: *"The vehicle must avoid collision with pedestrians under all operating conditions."*
5. **Formulate Safety Requirements**
   - Make the goals measurable and testable.
   - Example: *"If object detection fails, the system must trigger emergency braking within 1 second."*
6. **Validation and Verification**
   - Ensure requirements are consistent, complete, and testable.
   - Simulate critical scenarios (e.g., obstacle in front at high speed).
7. **Iterative Refinement**
   - Update requirements as new hazards are discovered during testing and field trials.

Example in Autonomous Vehicles:

- Hazard: Camera sensor fails at night.
- Requirement: Vehicle must switch to radar/LiDAR inputs and reduce speed to safe limits.

In short, hazard analysis provides the foundation for deriving robust **safety requirements**, ensuring that autonomous vehicles operate without endangering people or the environment.

3b) Why is availability important in high-assurance software systems like healthcare or banking? Illustrate with an example. [8 marks]

→

Availability refers to the degree to which a system is operational and accessible when required for use. In high-assurance software systems such as healthcare, banking, aviation, or defense, availability is critical because even short downtime can lead to severe consequences.

Importance of Availability in High-Assurance Systems

1. **Uninterrupted Service Delivery** – Continuous availability ensures life-critical or finance-critical operations are not disrupted.
2. **Safety of Human Life** – In healthcare, downtime could prevent access to patient records or delay life-saving treatment.
3. **Trust and Reliability** – Users must have confidence that systems will be available whenever needed, otherwise trust is lost.
4. **Economic Loss Prevention** – In banking, even a few minutes of unavailability can cause huge financial losses.
5. **Regulatory Compliance** – Many industries have strict availability requirements (e.g., 99.99% uptime) to meet safety and legal standards.

Example

- **Healthcare System:** If a hospital's patient monitoring software becomes unavailable during surgery, doctors may not get real-time updates on patient vitals. This could directly risk the patient's life.
- **Banking System:** If an ATM network or online banking app goes down during peak hours, customers cannot withdraw or transfer money. This not only causes financial loss but also damages the bank's reputation.

Therefore, availability is not just a performance measure but a **critical dependability attribute** for high-assurance systems, ensuring that essential services are always accessible when needed.

4a) Define software fault tolerance and discuss the design patterns typically employed in building fault-tolerant systems. [7 Marks]

→

Software Fault Tolerance is the ability of a software system to continue operating correctly even when faults occur in hardware, software, or user operations. Instead of allowing a single fault to cause system failure, fault-tolerant systems use special design strategies to detect, isolate, and recover from faults. This ensures reliability, availability, and safety in critical applications like aviation, healthcare, and banking.

Design Patterns in Fault-Tolerant Systems

1. **Recovery Blocks**
   - Multiple alternative implementations are prepared for a function.
   - If the primary block fails (detected via an acceptance test), control switches to a backup block.
   - *Example: A flight control system trying a backup algorithm if the main altitude control fails.*
2. **N-Version Programming**
   - Independent teams develop multiple versions of the same software.
   - At runtime, outputs are compared, and the majority result is chosen (voting mechanism).
   - *Example: Space shuttle flight software using three independently developed versions to ensure correctness.*
3. **Checkpoint and Restart**
   - The system periodically saves its state (checkpoint).
   - If a failure occurs, the system rolls back to the last checkpoint instead of restarting from scratch.
   - *Example: Database systems recovering from crashes using saved transaction logs.*
4. **Exception Handling**
   - Detects abnormal conditions and transfers control to predefined error-handling routines.
   - *Example: Banking software catching failed transaction errors and rolling back without corrupting data.*
5. **Graceful Degradation**
   - The system continues to provide limited functionality instead of complete shutdown during failures.
   - *Example: A video streaming service lowering resolution when bandwidth is low instead of stopping playback.*

Conclusion:

Software fault tolerance is essential for dependable systems. By using design patterns like recovery blocks, N-version programming, checkpointing, and graceful degradation, systems can **survive faults and continue providing critical services** without major disruption.

4b) what is system survivability? Explain the concept of N-version programming and provide an appropriate diagram to illustrate it. [8 marks]

→

System survivability is the ability of a system to continue delivering essential services even when it is under attack, experiencing partial failures, or operating in a degraded environment. Unlike fault tolerance, which focuses on recovery from accidental faults, survivability emphasizes resilience against both **deliberate attacks** (like cyber intrusions) and **unexpected failures**.

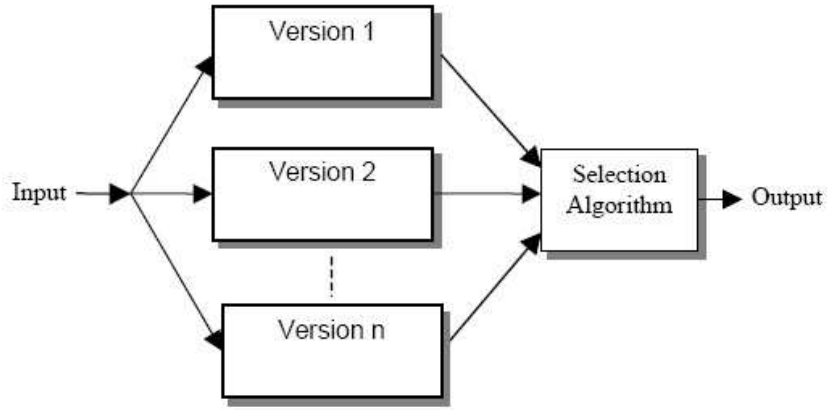
- Example: A cloud-based e-commerce website that keeps running with limited services (like basic browsing and offline payment option) even when its main payment gateway is attacked.

N-Version Programming (NVP)

N-Version Programming is a **fault tolerance technique** where multiple functionally equivalent versions of a software component are developed independently by different teams. All versions receive the same inputs at runtime, and their outputs are compared by a **voting mechanism (majority voting)**. The system then selects the output agreed upon by most versions.

- **Key Idea:** If one or more versions fail or produce incorrect results, the correct majority output ensures system reliability.
- **Use Case:** Safety-critical systems like **aircraft control software** or **nuclear plant monitoring**, where incorrect results could cause disasters.

Diagram of N-Version Programming



Conclusion

System survivability ensures continuous operation under stress or attack. N-version programming supports this goal by reducing the risk of failure due to software faults, as independent implementations with voting ensure correctness even when some versions fail.

5a) what do the terms software aging and software rejuvenation mean? How is software rejuvenation carried out in transaction-based systems? [7 Marks]

→

Software aging refers to the gradual performance degradation or increased failure rate of a software system due to accumulation of errors during long execution. Causes include memory leaks, unreleased file handles, data corruption, and resource fragmentation. Over time, these lead to reduced availability and reliability.

- *Example:* A web server running continuously for weeks may slow down or crash because of unreleased memory.

Software rejuvenation is the proactive technique of **cleaning the system state** to prevent failures caused by software aging. It restores the system to a "fresh" state before serious faults occur.

- *Example:* Restarting a web server periodically at off-peak hours to clear memory leaks and resource locks.

Rejuvenation in Transaction-Based Systems

In transaction-based systems (e.g., banking servers, online booking platforms):

1. **Monitoring:** The system is monitored for aging symptoms like increased response time, memory usage, or error rates.
2. **Triggering Rejuvenation:** When thresholds are exceeded (e.g., memory usage > 90%), rejuvenation is scheduled.
3. **Techniques Used:**
    - **Graceful Restart:** Ongoing transactions complete, new ones are queued, and then the system restarts.
    - **Process Swapping:** Critical processes are stopped and restarted without shutting down the entire system.
    - **State Cleanup:** Cache flush, memory garbage collection, log reset, or releasing unused resources.
4. **Minimizing Impact:** Rejuvenation is often done during low-load periods or in replicated systems where one server can take over while the other rejuvenates.

Software aging is inevitable in long-running systems, but software rejuvenation techniques such as **graceful restarts and state cleanup** in transaction-based systems help maintain performance, availability, and reliability.

5b) Describe the relationship between security and dependability. What are the security requirement of dependable system? [8 Marks]

→

Relationship between Security and Dependability

- **Dependability** is an umbrella concept that includes attributes like availability, reliability, safety, and maintainability. A system is dependable if users can trust it to deliver services as expected.
- **Security** is one of the core components of dependability. Without security, other dependability attributes cannot be guaranteed.
    - A **security breach** can reduce reliability (system manipulated), harm safety (life-critical systems attacked), or affect availability (denial-of-service attack).
    - Thus, **security is a prerequisite for dependability** – an insecure system can never be considered fully dependable.

Example: In online banking, even if the system is highly reliable (always processes transactions correctly), it cannot be trusted if hackers can steal user data.

Security Requirements of a Dependable System

A dependable system must fulfill the **CIA Triad** and related controls:

1. **Confidentiality** – Ensure that information is accessible only to authorized users.
    - *Example:* Encrypting patient medical records in healthcare systems.
2. **Integrity** – Prevent unauthorized modification or corruption of data and software.
    - *Example:* Digital signatures on banking transactions.
3. **Availability** – Ensure services are always accessible to authorized users, even during attacks.
    - *Example:* Protection against Distributed Denial of Service (DDoS).
4. **Accountability & Auditability** – System must log activities and allow tracing of malicious actions.
    - *Example:* Recording failed login attempts in secure servers.
5. **Attack Detection & Recovery** – Ability to detect intrusions, minimize impact, and restore normal operation.
    - *Example:* Intrusion Detection Systems (IDS) and automatic failover in cloud platforms.

6a) What are the key principles of designing a secure system, and what typical vulnerabilities and types of attacks can affect software systems. [7 Marks]

→

Key Principles of Designing a Secure System

1. **Least Privilege** – Give users and processes the minimum access rights they need.
   Example: A hospital receptionist should not have access to modify patient prescriptions.
2. **Fail-Safe Defaults** – Deny access by default, and only allow explicitly granted permissions.
   Example: A firewall blocking all ports except the ones specifically opened.
3. **Defense in Depth** – Use multiple layers of protection (authentication, firewalls, encryption).
   Example: Online banking requiring password + OTP + device verification.
4. **Separation of Duties** – Split critical tasks so no single user has full control.
   Example: One person initiates a transaction, another approves it.
5. **Open Design** – Security should not depend on secrecy of design but on robust algorithms.
   Example: Public cryptographic algorithms like AES, instead of hidden "custom" ones.
6. **Minimize Attack Surface** – Reduce the number of entry points to the system.
   Example: Disabling unused services and ports in a server.
7. **Regular Updates and Patch Management** – Fix vulnerabilities quickly to prevent exploitation.

Typical Vulnerabilities in Software Systems

- **Buffer Overflows** – Writing beyond memory boundaries.
- **Injection Flaws** – SQL injection, command injection.
- **Race Conditions** – Conflicts when multiple processes access resources simultaneously.
- **Insecure Authentication** – Weak passwords, poor session handling.
- **Unvalidated Input** – Accepting user input without sanitization.
- **Improper Error Handling** – Revealing system information to attackers.

Types of Attacks on Software Systems

1. **Denial of Service (DoS/DDoS):** Overloading the system to make it unavailable.
2. **Phishing & Social Engineering:** Tricking users into revealing credentials.
3. **Malware & Ransomware:** Inserting malicious software to steal or lock data.
4. **Man-in-the-Middle Attack:** Intercepting communication between two parties.
5. **Privilege Escalation:** Exploiting bugs to gain higher system rights.
6. **Data Breaches:** Unauthorized access leading to leakage of sensitive data.

6b) Discuss the significance of managing changes and controlling dependencies in ensuring successful software maintenance. [8 marks]

→

Significance of Managing Changes in Software Maintenance

Software maintenance is not only about fixing bugs but also about adapting the system to new requirements and environments. Effective **change management** ensures that modifications do not introduce new faults or break existing functionality.

- **Traceability:** Every change request should be linked to requirements, design, and test cases so its impact is clear.
- **Controlled Process:** Using formal procedures like Change Control Boards (CCB) avoids random or unnecessary changes.
- **Risk Reduction:** Analyzing the effect of changes prevents system instability.
- **Documentation:** Properly documenting changes ensures future maintainers understand what was modified and why.
- **Example:** In a hospital management system, if the billing module changes tax rules, change management ensures the update doesn't affect patient record handling.

Significance of Controlling Dependencies

Dependencies are relationships between modules, components, or external libraries. Poorly managed dependencies make maintenance costly and error-prone.

- **Reducing Ripple Effect:** If one module changes, tight coupling may force changes in many others. Dependency control limits this.
- **Modularity and Reusability:** Well-managed dependencies (using modular design or microservices) allow components to be updated independently.
- **Version Management:** External dependencies (like APIs, libraries) must be tracked to avoid compatibility issues.
- **Example:** An e-commerce website using a third-party payment gateway must manage dependency carefully so that updates in the gateway do not break the whole system.

7. Write short notes on: (Any two) 2×5

a) Granularity of 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:

i) Minimizing downtime
ii) Preserving user experience
iii) Efficient resource utilization

Types of Granularity:

i) System-Level Rejuvenation: Entire operating system and all running services are restarted.
ii) Application-Level Rejuvenation: Only the target application or service is restarted.
iii) Component-Level Rejuvenation: Rejuvenation is applied to a specific model, thread, or internal structure.

Best Practices:

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

b) MTTF

→

MTTF (Mean Time to Failure) is one of the reliability metrics tools, 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:

MTTF = t1+t2+…. +tn/n

Where, t1, t2, tn are the times between failures.

To measure MTTF, we collect and observe failure data for n failures.

Let the failures occur at time instants t1, t2, …., tn.

MTTF can be calculated as:

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

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.

c) Mathematical models of software reliability

→

Mathematical models of software reliability are quantitative models used to **measure, predict, and improve reliability** based on failure data collected during testing or operation. They help managers decide when software is ready for release and how reliable it will be in the field.

Key Software Reliability Models:

1. **Jelinski-Moranda (JM) Model**
   - One of the earliest models (1972).
   - Assumes a fixed number of faults exist initially, and each detected fault is perfectly removed.
   - Failure rate decreases linearly as faults are removed.
2. **Goel-Okumoto (GO) Model**
   - Based on a **Non-Homogeneous Poisson Process (NHPP)**.
   - Assumes finite number of faults and exponential decrease in failure intensity with time.
3. **Musa-Okumoto Logarithmic Model**
   - Another NHPP model.
   - Assumes infinite potential faults, but failure intensity decreases logarithmically with time.
   - Suitable for large, long-lived systems.
4. **Schick-Wolverton Model**
   - Considers that failure rate increases between fault fixes.
   - Useful when testing effort varies over time.
5. **Littlewood-Verrall Model**
   - Bayesian model that incorporates uncertainty in debugging.
   - Assumes fault removal may be imperfect.

d) Safety critical systems

→

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

i) Death or serious injury to people
ii) Loss or damage to equipment or property
iii) 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.

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 dose
Nuclear	Reactor cooling control system	Nuclear meltdown

Key Characteristics:

- High assurance level
- Fail-safe design
- Redundancy
- Verification & Validation
- Regulatory compliance

Typical safety process:

i) Hazard identification
ii) Risk assessment
iii) Requirements specifications
iv) Design with safety in mind
v) Implementation using best practices
vi) Verification & Validation
vii) Certification and audit