Critical Analysis Techniques in Software Engineering

What is Critical Analysis

- Definition: A systematic evaluation process to assess the strengths, weaknesses, and validity of software artifacts
- Purpose in Software Engineering:
  - Identify defects early
  - Improve system reliability
  - Optimize development processes

Why Critical Analysis Matters

- Statistics:
  - 60% of software defects originate in the requirements phase (Source: General industry consensus).
  - Fixing a bug in production is 100x more expensive than in design.
- Real-world impact:
  - Therac-25 incident (1985-1987): Software flaws led to fatal radiation overdoses.

Critical Thinking vs. Critical Analysis

- Critical Thinking: General problem-solving and reasoning.
- Critical Analysis: Focused, structured evaluation of specific artifacts.
- Overlap:
  - Logical reasoning, skepticism.
- Difference:
  - Critical analysis uses domain-specific tools and methods.

Learning Objective

- Understand the role of critical analysis in software engineering.
- Learn key techniques (e.g., FMEA, Root Cause Analysis).
- Apply these techniques to real-world scenarios.
- Develop a critical mindset for software development

Techniques and Tools for Critical Analysis

List of Techniques:

1. **Failure Modes and Effects Analysis (FMEA)**: Identifies potential failures and their impact.
2. **Root Cause Analysis (RCA)**: Traces problems back to their origin.
3. **SWOT Analysis**: Evaluates strengths, weaknesses, opportunities, threats.
4. **Code Review**: Systematic examination of source code.
5. **Requirements Validation**: Ensures requirements are clear, complete, and testable.

Failure Modes and Effects Analysis (FMEA)

FMEA is proactive—it's a planning tool to stop trouble before it starts.

FMEA lists all possible failures (failure modes) across a system, rates them (S, O, D), and prioritizes fixes.

You're building an online store. You predict that the payment system might fail (e.g., "Credit card doesn't process"). You rate how bad that'd be and plan to add extra security.

Failure Modes and Effects Analysis (FMEA)

Steps:

1. List components/functions of the system.
2. Identify potential failure modes.
3. Assess severity, occurrence, detection (Risk Priority Number: RPN)

After the engineers completed the list of FMEA, they sort the list by RPN to see which failures have the highest priority.

Components/Functions of the System

- Which functions of the system should I include?
  - Failures have **high impact** (Severity, S).
  - Failures are **likely** (Occurrence, O).
  - Failures are **hard to catch** (Detection, D).

Components/Functions of the System

Steps to Identify Functions for FMEA

- 1. Understand the System’s Purpose and Scope
- 2. Assess Criticality and Impact
- 3. Evaluate Usage and Exposure
- 4. Check Complexity and Change Frequency
- 5. Assess Detection Challenges
- 6. Leverage Historical Failures
- 7. Consider External Interfaces

Assess Severity, Occurrence, Detection

Overview of S, O, D

- **Severity (S)**: How bad is the impact of the failure on the system, user, or business? (1 = minor, 10 = catastrophic)
- **Occurrence (O)**: How often does the failure happen? (1 = rare, 10 = frequent)
- **Detection (D)**: How likely are we to catch the failure before it reaches the user? (1 = always detected, 10 = undetectable)

- Collect data from a mix of **historical records**, **team expertise**, **testing**, **user feedback**, and **industry benchmarks**

Gathering Data for Severity (S)

- **User Impact Analysis**:
  - **Source**: User stories, requirements docs, or customer support tickets.
  - **Method**: Review how the failure affects users.
- **Stakeholder Input**:
  - **Source**: Product managers, developers, or business analysts.
  - **Method**: Conduct a workshop or survey. Rate impact on a 1-10 scale based on business goals (e.g., revenue, reputation)

Gathering Data for Severity (S)

- **Industry Standards**:
  - **Source**: FMEA guidelines (e.g., AIAG, IEC 60812) or software risk frameworks (e.g., ISO 27001 for security).
  - **Method**: Use predefined severity tables (e.g., 10 = legal/safety violation, 5 = moderate disruption)
- **Historical Data**:
  - **Source**: Incident reports, post-mortems, or bug trackers (e.g., Jira).
  - **Method**: Analyze past failures' impact.

Gathering Data for Occurrence (O)

- **Log Analysis**:
  - **Source**: System logs, error tracking (e.g., Sentry, New Relic).
  - **Method**: Count failure instances over time (e.g., failures/hour)
- **Testing Data**:
  - **Source**: Unit, integration, or load test results.
  - **Method**: Run tests to simulate conditions
- **Developer Estimates**:
  - **Source**: Software engineers or architects.
  - **Method**: Brainstorm likelihood based on code complexity or past bugs

Gathering Data for Occurrence (O)

- **Usage Patterns**:
  - **Source**: Analytics (e.g., Google Analytics, server metrics).
  - **Method**: Correlate failures with user activity (e.g., peak load spikes)
- **Benchmarks**:
  - **Source**: Industry data or vendor SLAs (e.g., AWS uptime stats).
  - **Method**: Use typical failure rates (e.g., network drops = 0.01/hour)

Gathering Data for Detection (D)

- **Test Coverage**:
  - **Source**: Test reports (e.g., unit test coverage from tools like JaCoCo, pytest).
  - **Method**: Measure % of code tested.
- **Monitoring Systems**:
  - **Source**: Logs, alerts (e.g., Prometheus, Datadog).
  - **Method**: Check if failures trigger alerts
- **QA Processes**:
  - **Source**: Development lifecycle (e.g., code reviews, static analysis).
  - **Method**: Assess controls' strength

Gathering Data for Detection (D)

- **User Reports**:
  - **Source**: Support tickets, bug reports.
  - **Method**: If users catch it first, D is high.
- **Design Reviews**:
  - **Source**: Architecture docs, peer reviews.
  - **Method**: Evaluate built-in checks

Example 1: Game App

- **Requirements FMEA**:
  - Function: Run the game on all phones.
  - Failure Mode: “Game lags on old phones.”
  - RPN
    - Severity: 8 (players quit)
    - Occurrence: 4 (many old phones)
    - Detection: 3 (easy to test)
    - RPN = 96.
  - Fix: Optimize graphics.

Example 2: Banking App

- **Design FMEA**:
  - Function: Transferring money to another account
  - Failure Mode: “Transfer fails if server’s down.”
  - RPN
    - Severity: 10 (money lost)
    - Occurrence: 2 (rare)
    - Detection: 5 (hard to catch)
    - RPN = 100.
  - Fix: Add offline queue.

Exercise

- **Task**: Perform an FMEA on an Online Voting System
- Components: User authentication, vote submission, result tallying.
- Steps:
  - Identify 3 failure modes.
  - Assign Severity (1-10), Occurrence (1-10), Detection (1-10).
  - Calculate RPN (Severity × Occurrence × Detection).

Exercise - FMEA Table

Component	Failure Mode	Severity	Occurrence	Detection	RPN
Authentication	Password leak	10	3	4	120
Vote Submission	Double voting	8	2	5	80
Result Tallying	Miscounted	10	1	3	30

Beyond Basic FMEA

- Recap:
  - FMEA identifies failure modes, assesses risks via RPN (Severity × Occurrence × Detection).
- Advanced Focus:
  - **Quantitative RPN**: Use probabilistic models (e.g., failure rates, MTTF).
  - **Mitigation Prioritization**: Optimize resource allocation using cost-benefit analysis.

Quantitive FMEA

- Traditional RPN:
  - Subjective (1-10 scales).

- Advanced:
  - Severity: Map to metrics (e.g., % downtime, revenue loss).
  - Occurrence: Use historical data or MTTF (Mean Time To Failure).
  - Detection: Probability of test coverage (e.g., 80% coverage = Detection 2).

- Formula: RPN = S × O × (1 - D%), where D% is detection probability.

Mitigation Prioritization

- Cost-Benefit Model:
  - Cost = Development effort (hours) + Tooling cost.
  - Benefit = Risk reduction (ΔRPN × Impact).

- Example:
  - Fix “DB crash” (RPN 200, $10K loss) with 50-hour effort ($5K).
  - ΔRPN = 150, Benefit = 150 × $10K = $1.5M, ROI = High.

Exercise

- FMEA with Quantitative RPN
- **Task**: Analyze a cloud storage API
  - Failure Mode: “Upload fails under load.”
  - Data:
    - MTTF = 1000 hours
    - Downtime = $5K/hour
    - Test Coverage = 90%.
  - Calculate RPN, suggest mitigation.

Answer

- **Failure Mode**: Upload fails under load.
- **Metrics**:
  - Severity: $5K/hour × 1-hour outage = $5K impact → 8 (high but not catastrophic).
  - Occurrence: MTTF = 1000 hours → Failure rate = 1/1000 = 0.001/hour → 2 (rare).
  - Detection: 90% coverage → D% = 0.9 → (1 - 0.9) = 0.1 → 1 (very detectable).
  - **RPN**: S x O x D = 8 × 2 × 1 = 16.
- **Mitigation**: Add autoscaling to handle load spikes.

Root Cause Analysis (RCA)

- **RCA** is reactive—it's a detective tool to fix trouble after it happens.

- The online store launches, and payments crash. You ask "Why?"—turns out the server couldn't handle 1,000 users because no one tested it under load.

Root Cause Analysis (RCA)

- Method:
  - 5 Whys
    - Problem: System crashes during peak load.
  - Why? → Server overload. Why? → Insufficient scaling. Why? → No load testing. (etc.)

Example 1: Game App (Space Shooter)

- **Coding RCA**:
  - Problem: Game crashes. Why? Enemies spawn too fast. Why? Loop has no limit.
  - Root cause: Missing cap on spawn rate.
  - Fix: Add a limit.

Example 2: Banking App

- **Testing RCA**:
  - Problem: Transfers duplicate. Why? Retry button resends. Why? No unique ID check.
  - Root cause: Missing transaction ID.
  - Fix: Add IDs.

Advanced RCA

- Basic RCA digs to root causes with 5 Whys

- In this section:
  - **Fault Tree Analysis (FTA)**: Logic trees with probabilities.
  - **Bayesian Inference**: Stats to refine guesses.

Fault Tree Analysis

- **FTA is top-down, deductive** approach to identify and analyze the causes of a specific system failure, called the “top event.”

- The top event (e.g., “website crashes”) is the root at the top, and the branches below are all the possible causes, sub-causes, and conditions that could lead to it.

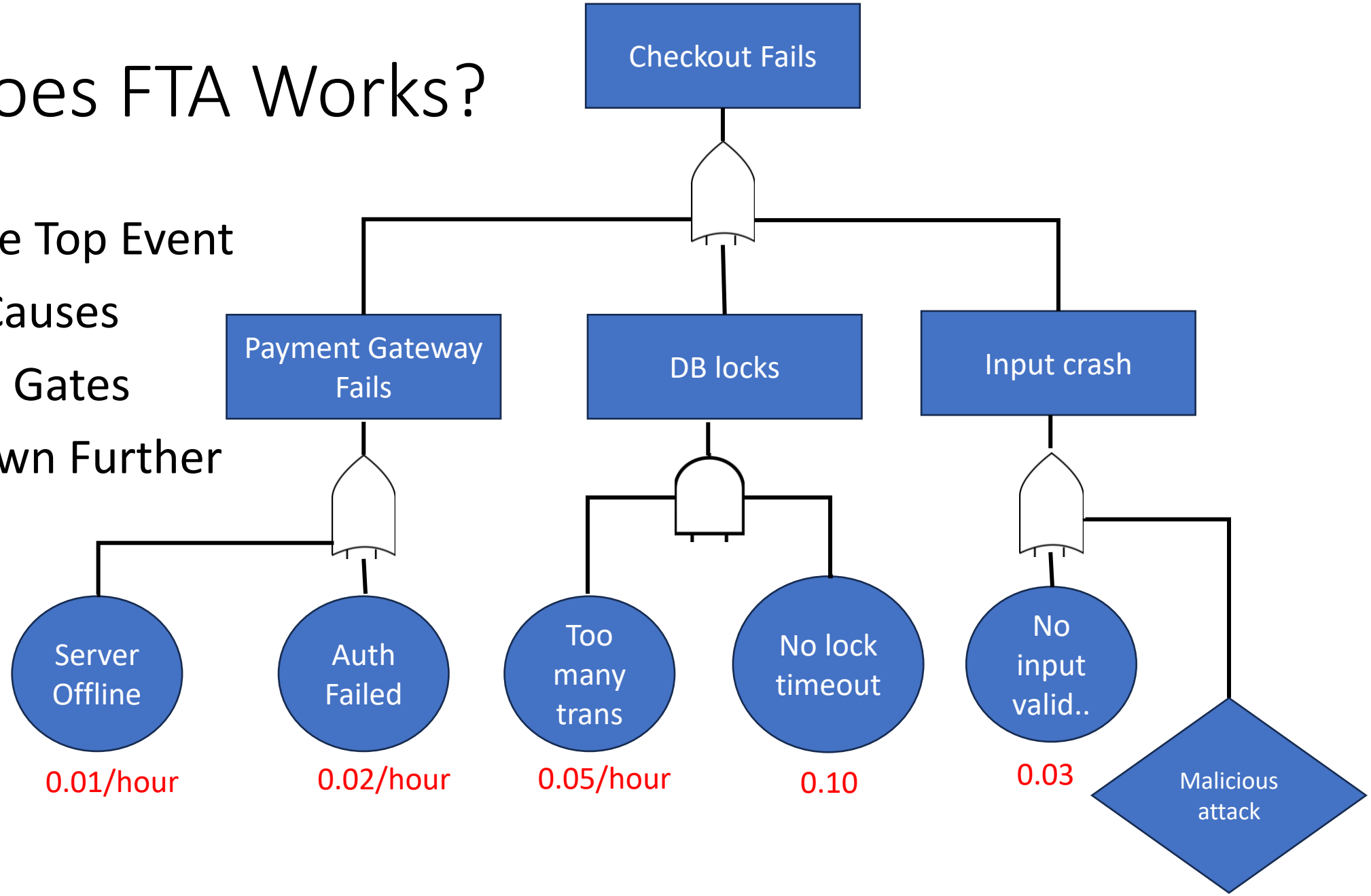
- It’s visual, logical, and systematic

Fault Tree Analysis

- **Purpose**
  - **Root Cause Identification**: Pinpoints what triggers a failure, down to basic events (e.g., “server runs out of memory”).
  - **Risk Assessment**: Quantifies the probability of the top event using failure rates or logic.
  - **Prevention**: Highlights critical paths to fix before disaster strikes.

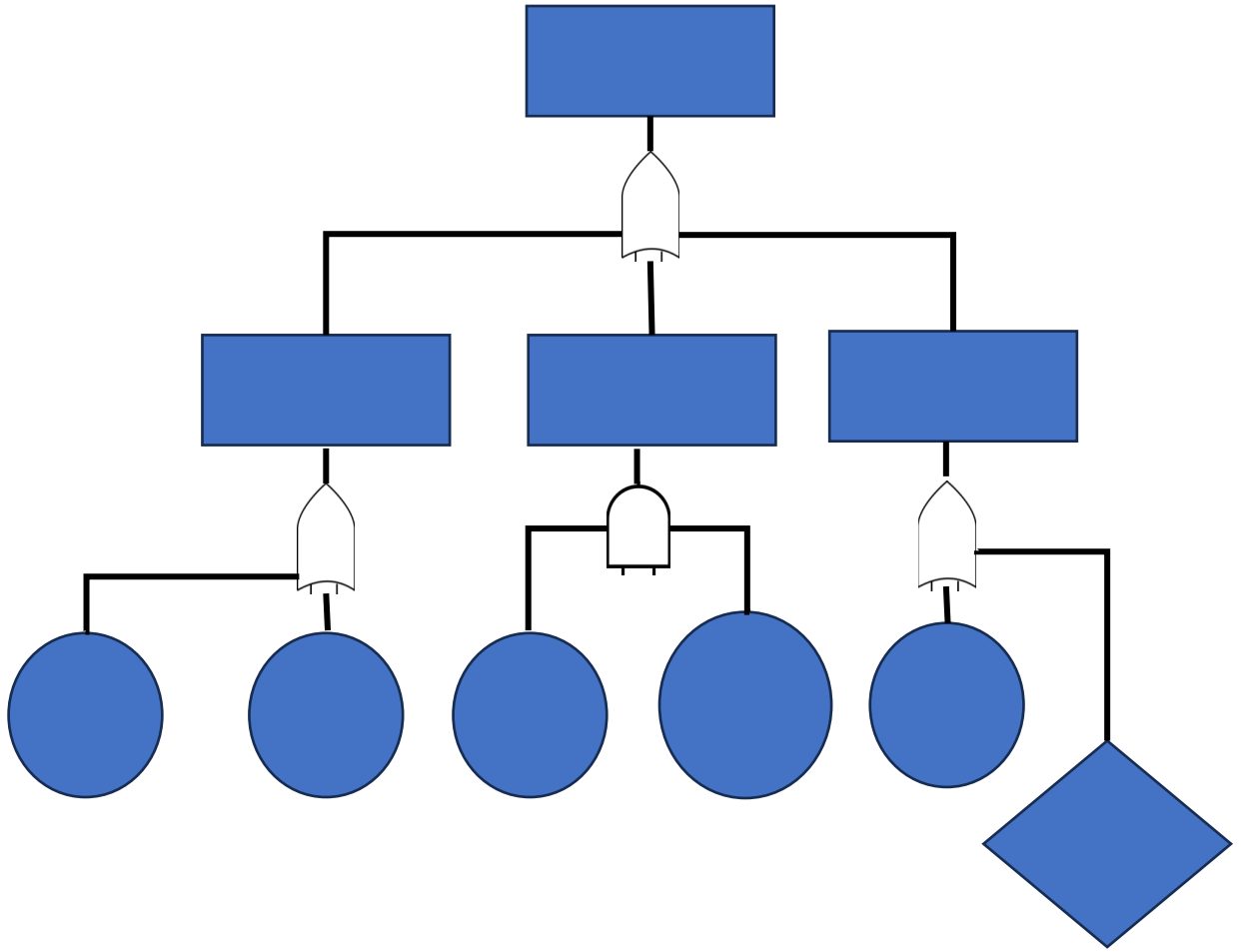
How Does FTA Works?

- Define the Top Event
- Identify Causes
- Use Logic Gates
- Break Down Further
- Quantify
- Analyze



Elements of the FTA diagram

- Events
  - Top event
  - Intermediate event
  - Basic event
  - Undeveloped event
- Logic Gates
  - AND
  - OR



Calculate Failure Probabilities

- p(Server offline) = 0.01/hour
- p(Auth failed) = 0.02/hour
- p(Too many transactions) = 0.05/hour
- p(No lock timeout) = 0.1
- p(No validation) = 0.03

- p(Payment Gateway failed) = p(Server offline) OR p(Auth failed)
  = 1 – (1 – 0.01 * 1 – 0.02)
  = 1 – (0.99 * 0.98)
  = 0.0298/hour

Calculate Failure Probabilities

- p(DB locks) = p(Too many transactions) AND p(No lock timeout)
  = 0.05 * 0.01
  = 0.005/hour

- p(Input crash) = p(No input validation) OR (undeveloped event)
  = 0.03

- p(Checkout fails) = 1 – (1 – 0.0298) * (1 – 0.005) * (1 – 0.03)
  = 1 – (0.9702 * 0.995 * 0.97)
  = 1 – 0.9364 = 0.0636/hour (We explore further in the later section)

Calculation Details

- p(Payment Gateway failed) = p(Server offline) OR p(Auth failed)
  = 1 – (1 – 0.01 * 1 – 0.02)
  = 1 – (0.99 * 0.98)
  = 0.0298

- Why not 0.01 + 0.02 = 0.03, instead 0.0298?

Calculation Details

- p(Payment Gateway failed) = p(Server offline) OR p(Auth failed)

- OR gate represent **at least one** of its causes occurs
  - ”server offline” or
  - “auth failed” or
  - Both

- The formula 0.01 + 0.02 adding directly overestimates because it double-counts the tiny overlap when *both* fail.

Calculation Details

- To calculate the probability that **at least one** event occurs. A handy way to calculate this is:
  - First, find the probability that **neither event occurs** (both don't fail).
  - Then, subtract that from 1 to get the probability that **at least one fails**.

- Step 1:
  - Server offline NEVER occur = 1 – 0.01
  - Auth failed NEVER occur = 1 – 0.02
  - Neither event occurs = 1 – 0.01 * 1 – 0.02
- Step 2:
  - 1 – (1 – 0.01 * 1 – 0.02)

Exercise: Login Failure

- **Scenario**: Users can’t log into a banking app.

- Task:
  - Build a fault tree:
    - Top Event: “Login fails.”
    - Causes:
      - “Login fails” if either “DB down” or “Auth error”
      - “DB down” if either “Power outage” [0.01/hour] or “DB crash” [0.02/hour]
      - “Auth error” if both “Bad credentials” [0.04/hour] and “No retry” [0.5]
  - Calculate the probability.

Dynamic and Static Nature

- p(Server offline) = 0.01/hour
- p(Auth failed) = 0.02/hour
- p(Too many transactions) = 0.05/hour
- p(No lock timeout) = 0.1
- p(No validation) = 0.03

- **Per Hour**: “Gateway server offline” (0.01/hour), “API authentication error” (0.02/hour), “Too many transactions” (0.05/hour).
- **No Time Unit**: “No lock timeout set” (0.1), “No input validation” (0.03).

Dynamic Nature of Probability

- Probabilities specified as **per hour** (e.g., 0.01/hour) are **failure rates**—they describe how often an event happens over time. These are typically used for:
  - **Dynamic Events**: Things that occur randomly or intermittently, like hardware failures, network drops, or load spikes.
  - **Time-Based Systems**: When we're analyzing a system's behavior over a period (e.g., an hour of checkout activity).

Dynamic Nature of Probability

- In FTA, failure rates are tied to **Mean Time To Failure (MTTF)**

- Example: “Gateway server offline” = 0.01/hour → MTTF = 1 / 0.01 = 100 hours. It fails once every 100 hours on average.
- In software, “per hour” often comes from:
  - **Logs**: Historical data (e.g., “Server crashed 5 times in 500 hours” → 0.01/hour).
  - **Metrics**: Monitoring (e.g., “Traffic spikes 50 times in 1,000 hours” → 0.05/hour).
  - **Vendor Specs**: Hardware or service uptime stats (e.g., 99.9% uptime → 0.001/hour downtime).

Static Nature of Probability

- Probabilities without a time unit are **static probabilities**—they're not rates but single, dimensionless chances. These apply to:
  - **Constant Conditions**: Design flaws, configuration errors, or missing features that don't "happen" over time—they just *exist*.
  - **State-Based Events**: The likelihood something is true at any given moment, not how often it occurs.

Static Nature of Probability

- “No Lock Timeout” - 10% chance the system lacks a timeout mechanism (e.g., a design oversight present in 10% of deployments)
- “No validation” - 3% chance the code lacks validation—maybe from a rushed feature or untested module.
- In software, these come from:
  - **Code Reviews**: “3% of commits skip validation”
  - **Design Audits**: “10% of configs miss timeouts”
  - **Assumptions**: Rough estimates when data is scarce.

Mixed Nature of Probability

- Does it impact the previous calculation?

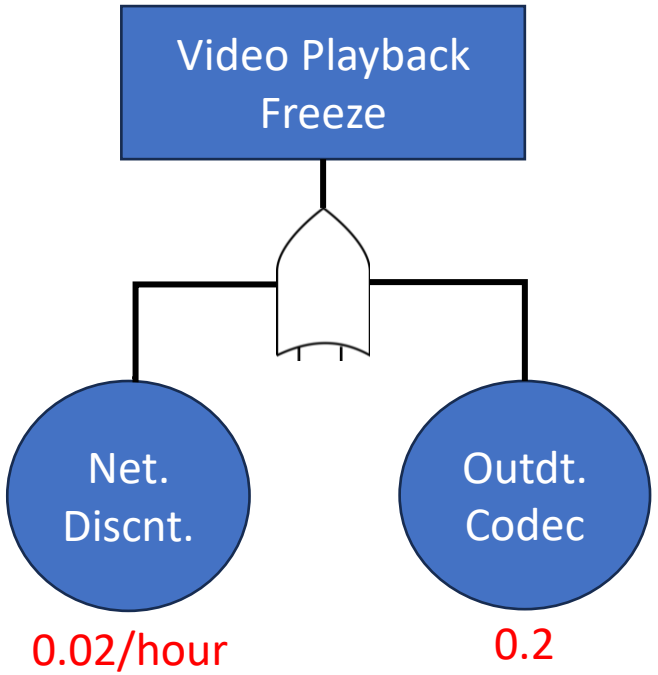
p(Checkout fails) = 1 – (1 – 0.0298) * (1 – 0.005) * (1 – 0.03)

- The standard OR gate formula—**P = 1 - (1 - P1) × (1 - P2)**—assumes all probabilities are comparable

- 0.0298 and 0.005 are time-based, while 0.03 is non time-based

Example: Video Streaming App Freezes

- **Network disconnect**: A time-based failure rate of 0.02/hour (2% chance per hour the network drops, based on ISP stats).
- **Outdated codec**: A non-time-based probability of 0.1 (10% chance the app uses an outdated codec, from version audits).



Approach 1: Convert Static to Time-Based

- If 10% of instances have an outdated codec, suppose it triggers a freeze 10% of the time under load.

- **Assumption**: Let's set it as 0.1/hour—10% chance per hour the codec fails to decode, freezing playback.

- p(Video playback freeze) = p(Network disconnected) OR p(Outdated codec)
  = 1 – [(1 – 0.02) * (1 – 0.1)]
  = 1 – (0.98 * 0.9)
  = 1 – 0.882
  = 0.118/hour (MTTF 1/0.118 = 8.47 hours)

Approach 2: Treat Static as Conditional

- Approach 1 forced “Outdated codec” (0.1) into a rate (0.1/hour), assuming it independently causes freezes 10% of hours.

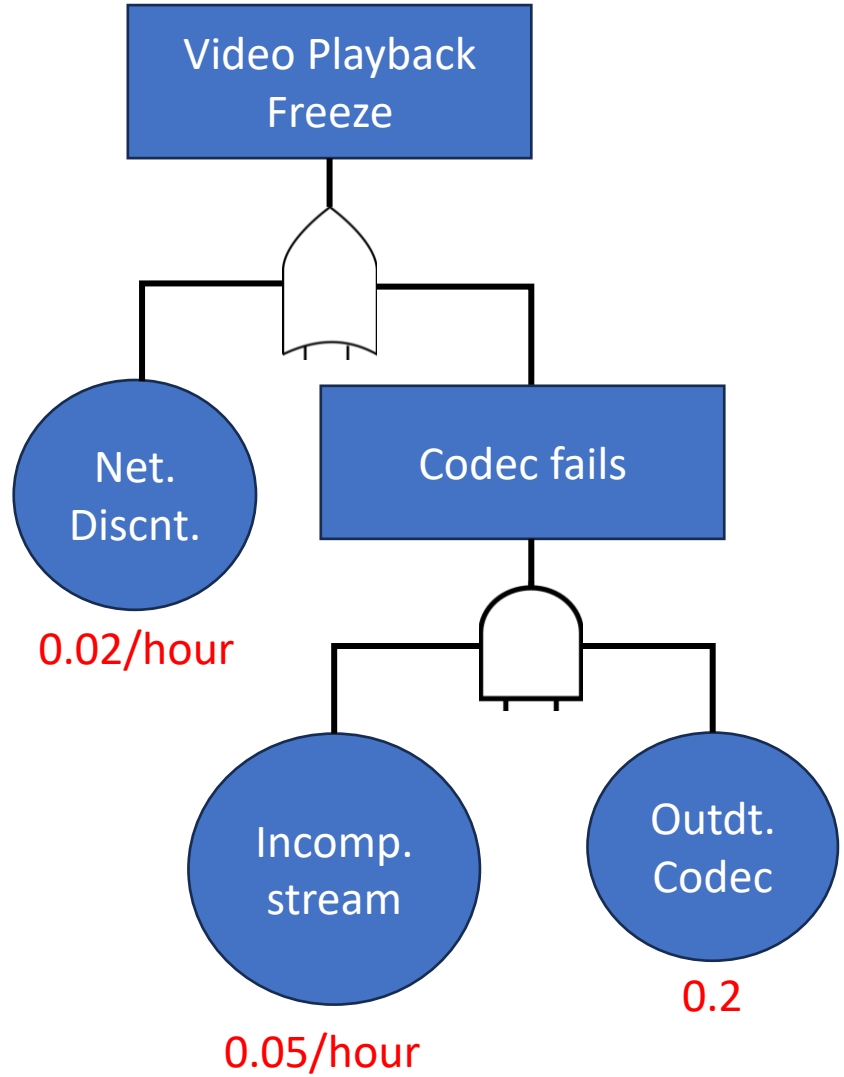
- But 0.1 originally meant “10% chance the codec is outdated”—a design flaw, not a failure frequency.

- In software, static conditions like this often need a trigger to cause failure.
  - **Static Probability**: 0.1 is the chance the system has an outdated codec.
  - **Trigger Needed**: It only freezes playback when paired with an event (e.g., playing an incompatible stream).

Approach 2: Treat Static as Conditional

- "Outdated codec" as a condition suggests it's not an independent hourly failure.
- Approach 2:
  - **Network Disconnect**: 0.02/hour
  - **Codec Failure**:
    - **Incompatible Stream**: 0.05/hour (hypothetical rate).
    - **Outdated Codec**: 0.1 (10% chance the app has an old codec).
  - p(Video playback freeze) = 0.025/hour

MTTF = 1 / 0.025 = 40 hours



Refine “Checkout Fialure” Example

- Since “malicious attack” is an undeveloped event, we ignore it for simplicity.
- Treat static probability for ”No validation” event as conditional.
- What will you get?

Exercise: Real-Time Chat App Disconnect

- **Scenario:** You're analyzing a real-time chat app where the top event is **"User disconnects unexpectedly"**—a critical failure that disrupts conversations. The system has multiple failure points, and the fault tree includes a top-level OR gate with both time-based and static probabilities
- The causes identify:
  - **Network Drop**: 0.03/hour
  - **Server Crash**: "Server overload": 0.04/hour and "No retry logic": 0.2
  - **Client Bug**: "Bug in client code": 0.02/hour or "No client update": 0.1
- **Task**
  - Build the fault tree
  - Calculate the probability per hour of "User disconnects unexpectedly"