# Software and Security Engineering

## Tripos Revision Guide

**CST Part IA — Paper 2**

Complete coverage of the Engineering Mindset, Requirements, Process, Quality Assurance, Evolution, and the AI Era — with full worked answers to every Tripos and Example Sheet question

**Lecture Overview**

| # | Lecture | Key Content |
|---|---|---|
| 1 | Engineering Mindset & Cost of Failure | Software crisis, Brooks's Law, discrete state, case studies (Therac-25, MCO, LAS, Horizon, 737 MAX), technical debt |
| 2 | Requirements & Specifications | Problem vs. solution space, elicitation, FR/NFR, UML, PM/EM roles, MoSCoW, PRD |
| 3 | Engineering Process | Waterfall, Spiral, Agile Manifesto, XP, Scrum, story points, Kanban, scaling |
| 4 | Quality Assurance & Reliable Delivery | Git, testing pyramid, mocking, coverage, mutation testing, TDD, CI/CD/CDP, deployment strategies, cloud, containers |
| 5 | Software Evolution | Lehman's Laws, legacy code, SemVer, Hyrum's Law, refactoring, SRE, chaos engineering, supply chain |
| 6 | SE in the AI Era | AI as coder, risks & mitigations, AI workflow, building for AI, the new engineer |

Includes: full lecture notes • case-study deep dives • security appendix • model answers to the 2025 Tripos paper and both Example Sheets • quick-reference glossary • exam strategy

July 16, 2026

# 0 Contents

# 1 Engineering Mindset & The Cost of Failure

## 1.1 Programming vs. Software Engineering

**The Software Crisis (1968 NATO Conference, Garmisch)**

The realisation that software was fundamentally different from hardware: projects routinely ran over budget and over time, systems were delivered with massive inefficiencies, and code was unmaintainable and broke frequently. The term "Software Engineering" was coined to demand a rigorous, systematic approach to code creation.

| Dimension | Programming | Software Engineering |
|---|---|---|
| Team size | Single developer | Teams of developers |
| Scale | Small ($<$ 1,000 lines) | Massive (1,000,000+ lines) |
| Lifespan | Days/weeks | Years/decades |
| Primary metric | "Does it work today?" | Maintainability, Security, Extensibility |

## 1.2 Brooks's Law & The Mythical Man-Month

**Brooks's Law (Fred Brooks, 1975)**

"Adding manpower to a late software project makes it later."

- **Training overhead:** senior engineers must stop coding to mentor new hires.
- **Communication overhead:** as team size $n$ grows, communication paths grow quadratically:

$$\text{paths} = \frac{n(n-1)}{2}$$

- Software is highly interdependent, unlike a ditch that can be perfectly partitioned among diggers.

**Exam Tip: Applying Brooks's Law**

If asked to "apply Brooks's Law" to a scenario (e.g. adding 5 developers to a late 5-person project), state explicitly: (1) short-term velocity will *drop* before it improves, due to onboarding cost and communication overhead; (2) compute the new communication path count to show quantitative reasoning, e.g. $10 \to \binom{10}{2} = 45$ paths; (3) recommend an alternative (descope, extend deadline, or add only 1–2 senior engineers already familiar with the codebase).

## 1.3 The Complexity of Discrete State

Physical structures (e.g. a bridge) have continuous, predictable stress limits and can be tested by sampling. Software state is **discrete** and exponentially vast: a single flipped bit can move a system instantly from a perfectly safe state to a catastrophic failure state. Exhaustive testing of all possible software states is mathematically impossible. This is why software risk profiles differ fundamentally from those of physical engineering: a bridge degrades gradually and visibly; software can fail with zero warning at 100% structural "integrity".

## 1.4 Moore's Law vs. Human Cognition

Hardware capability (transistor count) historically doubled roughly every two years (Moore's Law). As hardware capacity exploded, the ambition of the software built on top of it exploded too — but human cognitive ability to manage complexity did not scale at the same rate. The widening gap between what hardware can do and what humans can safely program and verify is precisely the space in which engineering processes (requirements, testing, version control) are required.

## 1.5 Systems and Emergent Behaviour

Software rarely exists in a vacuum: it is built from components (databases, front-ends, APIs, sensors, network layers) that combine into **systems**. A system can exhibit **emergent behaviour** — behaviour not present in any individual component, which can be positive (novel capability) or negative (unexpected failure mode). Critically, a bug is often not localised to one function but arises from the *interaction* between two perfectly-written components.

### 1.6 The Cost & Lifecycle of Software

**The 80/20 Rule of Lifecycle Cost**

Typically, only **20%** of a software system's total lifetime cost is spent on initial development; **80%** is spent on maintenance and evolution. **Implication:** code must be written for the engineer who reads it five years from now, not optimised purely for speed of initial writing.

**The four types of maintenance** (revisited in Lecture 5 with percentages):

- **Corrective:** fixing discovered bugs.
- **Adaptive:** making the system work in a changed environment (new OS, new hardware, cloud migration).
- **Perfective:** adding new features / improving performance demanded by users.
- **Preventive:** refactoring code to prevent future decay.

**Software Rot / Bit Rot**

Software does not physically wear out, but it suffers from **Bit Rot (Software Entropy)**: the surrounding environment changes (libraries update, security protocols deprecate, OS APIs change) so software that is never touched will eventually fail even though its own source code never changed.

### 1.7 Case Studies in Failure

**Case Study 1: Therac-25 (1985–1987)**

**Background:** A medical linear accelerator for radiation therapy, built by AECL, successor to the Therac-20.
**Fatal flaw:** The Therac-20 had *hardware interlocks* preventing the beam from firing without the targeting shield in place. The Therac-25 removed these (expensive) hardware interlocks and entrusted safety entirely to software.
**The bug (race condition):** A shared variable tracked operator input. If an operator typed "X" (X-ray mode), realised a mistake, then quickly pressed "Up" and "E" (Electron mode), a race condition configured the high-power beam *without* deploying the physical shield.
**UI failure:** The machine displayed cryptic, undocumented errors ("Malfunction 54"); operators could simply press "P" (Proceed) to bypass and fire again.
**Cost:** At least six patients received massive overdoses (up to 100× intended dose); several died. AECL initially denied any software fault.
**Lessons:** (1) *Defense in depth* — never rely on a single software check for human safety; hardware interlocks are essential. (2) *Concurrency is dangerous* — shared state in multi-threaded systems must be strictly managed. (3) *UX is safety* — error messages must be actionable and hard to blindly dismiss.

**Case Study 2: Mars Climate Orbiter (1999)**

**Background:** A $327 million NASA/JPL probe (with Lockheed Martin) to study Martian climate.
**Failure:** On 23 September 1999 the orbiter's signal was lost during Mars orbit insertion; it had disintegrated in the upper atmosphere.
**Root cause:** Lockheed Martin (Colorado) wrote thruster software using **Imperial units** (pound-seconds); NASA JPL (California) wrote navigation software expecting **Metric units** (Newton-seconds) — an unnoticed factor-of-4.45 error.
**Why not caught:** The Software Interface Specification (SIS) explicitly required metric units, but Lockheed Martin did not adhere to it; teams tested components in isolation with *no end-to-end integration testing.*
**Lessons:** (1) *Type safety* — strongly typed languages / typed units can catch such errors at compile time. (2) *Integration testing* — isolated component testing is not enough; team *boundaries* are where systems fail. (3) *Process over blame* — a systemic process breakdown, not one programmer's mistake.

**Case Study 3: London Ambulance Service CAD (1992)**

**Background:** October 1992 rollout of a new Computer Aided Dispatch (CAD) system to automate ambulance dispatch across London.

**Doomed procurement:** Contract awarded to a company with *no prior experience* building emergency CAD systems, which won by significantly underbidding; timelines were compressed by political pressure.
**Hostile user base:** Ambulance locations were tracked via imperfect radio systems; paramedics felt micro-managed, distrusted the system, and (due to lack of training) often pressed wrong status buttons, feeding it corrupt availability data.
**System overload:** A memory leak caused rapid slowdown; ambulances were dispatched multiple times to the same address while other calls were ignored; citizens re-called repeatedly, flooding the system with duplicate entries.
**Collapse:** Within 36 hours the system locked up entirely; the control room reverted to paper slips and whiteboards. Patients died waiting for ambulances that never arrived.
**Lessons:** This is a **socio-technical system** — software does not exist in isolation from human psychology, organisational culture, and hardware. (1) *Phased rollouts* — never do a "Big Bang" deployment for a critical system. (2) *User buy-in* — users must be involved in design so the software fits real workflows. (3) *Load testing* — test under extreme, anomalous load, not just the "happy path".

**Case Study 4: UK Post Office Horizon Scandal (from 1999)**

**Background:** The Horizon IT system (Fujitsu) automated accounting for thousands of local post offices. Subpostmasters began reporting inexplicable financial shortfalls; the Post Office insisted Horizon was "robust" and "effectively infallible", blaming subpostmasters.
**Reality:** The system had multiple critical bugs that could silently alter financial records, and Fujitsu's own developers were aware of these bugs.
**Cost:** Over 700 subpostmasters were prosecuted for theft and false accounting; many were imprisoned, went bankrupt, or took their own lives.
**Lessons:** (1) *Audit trails* — in financial/legal systems every automated action must be transparent and auditable by a human. (2) *Engineering integrity* — developers who identified bugs were ignored by management; technical truth must override business convenience. (3) *Presumption of guilt* — the law at the time presumed computer evidence correct unless proven otherwise; engineers must challenge this "myth of infallibility".

**Case Study 5: Boeing 737 MAX — MCAS**

**Background:** To fit larger, more fuel-efficient engines, Boeing moved them forward and higher on the airframe, changing the aerodynamics and making the plane prone to pitching up.
**The "fix":** MCAS (Manoeuvring Characteristics Augmentation System) automatically pushed the nose down on detecting a high angle of attack — but relied on a **single** Angle-of-Attack sensor (a single point of failure).
**The failure:** If that one sensor failed, MCAS would repeatedly force the plane into a dive. Boeing also "hid" MCAS's existence to avoid costly pilot retraining, so pilots did not know the system existed or how to disable it.
**Lessons:** (1) *Single point of failure* — never design a critical safety system relying on data from only one sensor; use redundancy. (2) *Complexity hiding* — software must not act autonomously without the operator's knowledge. (3) *The "software fix" trap* — don't use software to patch a fundamental hardware/design flaw unless built with absolute rigour.

**Exam Tip: Using the Case Studies**

These case studies are a toolkit, not trivia. When asked to justify an engineering principle (defense in depth, integration testing, phased rollout, single points of failure, audit trails), **name the matching case study** and briefly explain the causal mechanism — examiners reward concrete grounding over generic statements.

### 1.8 Role of the Engineer & Technical Debt

**Code as a liability:** managers often view lines of code as an asset; engineers must view them as a liability — every line written must be tested, maintained, and secured. The best code is the code you didn't have to write.

**Technical Debt (Ward Cunningham, 1992)**

A metaphor comparing the trade-off of writing quick, messy code (to meet a deadline) to taking out a financial loan.

- **Principal:** the effort required to eventually rewrite the code properly.
- **Interest:** the extra time it takes to add features or fix bugs because messy code slows you down.

| | **Deliberate** | **Inadvertent** |
|---|---|---|
| **Reckless** | "We must ship now and fix later" | "What's pytest?" (no design skill) |
| **Prudent** | "No time for design" (strategic, informed) | "Now we know how we should have done it" (learning) |

- **Intentional debt** is a strategic decision (e.g. hardcoding a DB connection to hit Friday's launch, planning to fix next sprint). **Prudent/Deliberate** debt is often a valid business decision.
- **Unintentional debt** arises from lack of skill or domain understanding. **Inadvertent** debt is where the most dangerous complexity grows silently.
- As debt accumulates, "interest payments" consume developer time; every new feature breaks old ones; morale drops; eventually the company may declare "technical bankruptcy" and rewrite from scratch.
- **Refactoring** (restructuring code without changing external behaviour) pays down debt, but is *impossible without automated tests* to prove behaviour was preserved.

**Professional Responsibility**

Engineers hold specialised knowledge that management and users do not. Engineers have a professional responsibility to push back against unrealistic deadlines that compromise safety or core quality. "Management told me to do it" is not an acceptable excuse for catastrophic failure (cf. Therac-25, Horizon).

**The Iron Triangle:** Fast, Good, Cheap — pick two. Software engineering is a continuous negotiation of trade-offs: perfect code shipped two years late is a failure; terrible code shipped tomorrow that bankrupts the company in technical debt is also a failure. Balancing this is what makes you an engineer, not just a programmer.

# 2 Requirements and Specifications

## 2.1 Problem Space vs. Solution Space

**The Danger of Perfection**

"There is nothing quite so useless as doing with great efficiency something that should not be done at all." — Peter Drucker. Engineers must separate the **Problem Space** (the "Why") from the **Solution Space** (the "What").

- **The Why (Problem Space):** technology-agnostic; business value, user needs, market realities. E.g. "Our warehouse packing process is too slow, causing delayed shipments."
- **The What (Solution Space):** introduces technology, architecture, engineering constraints. E.g. "We will build a tablet-based barcode scanning app that calculates optimal walking routes."
- **The Translation Gap:** stakeholders know the Why but speak vaguely; engineers know the How but lack domain expertise. Requirements Engineering formally bridges this gap.

## 2.2 Requirements Elicitation

| Method | Strength | Weakness |
|---|---|---|
| Interviews | Good for initial scoping, high-level business goals | Users often describe a solution, not their problem ("faster horses") |
| Ethnography / Observation | Reveals invisible workarounds users consider "normal" and would never mention | Time-consuming; requires access to the real environment |

**Worked Example: The Nurse and the Sticky Note**

An interview may never reveal that a nurse keeps a sticky note of passwords on her monitor. Only **ethnographic observation** would reveal that the current login system logs her out too fast, forcing this insecure workaround — because she considers it "normal" and wouldn't think to mention it.

## 2.3 The Problem with Natural Language

Human language is ambiguous, context-dependent, and imprecise; code is binary, deterministic, and merciless. If a requirement is ambiguous, the programmer will unconsciously make an assumption to fill the gap — and that assumption is usually wrong.

**Bad Requirement**

"The search system should be fast and easy to use." What does "fast" mean — 100ms? 2s? What does "easy to use" mean, and how would you test it? If the client rejects the software as "too slow", you have no contractual defense.

**Good Requirement**

"The system shall return search results for a database of 1 million records in under 500 milliseconds for 95% of queries under normal load." It is quantifiable, testable, and sets a clear boundary for "done".

## 2.4 Functional vs. Non-Functional Requirements

**Definitions**

**Functional Requirement (FR):** a statement of a specific action or behaviour the system must perform — the *verbs* of the system (inputs, processing, outputs). If the system doesn't do this, it is fundamentally broken.
**Non-Functional Requirement (NFR):** a constraint on the operation of the system, describing how well it performs its functions — the *adjectives/adverbs*. NFRs often dictate the entire system architecture.

**Example FRs (e-commerce site):** "allow a user to add an item to a cart"; "calculate total cost including VAT"; "send a confirmation email on successful payment."

**The "-ilities" (NFR categories):** Scalability, Reliability, Maintainability, Portability, Usability.

- **Latency vs. Throughput:** latency = time for one operation ("page load < 2s"); throughput = operations handled concurrently ("5,000 transactions/sec"). You often must design differently to optimise one over the other.
- **Security as an NFR:** generates FRs. NFR: "must comply with GDPR". Resulting FR: "system shall provide a button for the user to delete their account and all associated data."
- **Fit criteria:** every NFR needs a measurable target, e.g. "99.99% uptime per month (≈4.3 minutes downtime allowed)"; "WCAG 2.1 AA compliance."

**NFR Trade-offs**

NFRs constantly fight each other — engineering is the art of balancing them. **Security vs. Usability:** a 24-character password + 2FA is very secure but terrible usability. **Performance vs. Maintainability:** hand-optimised Assembly improves speed but destroys maintainability.

**Class Exercise: FR or NFR?**

1. "The system shall allow a student to reserve a book that is currently checked out." → **FR** (a specific action).
2. "The search functionality must return results in under 2 seconds, even with 500 concurrent users." → **NFR** (performance/scalability constraint).
3. "Database passwords must be stored using bcrypt hashing with a salt factor of 12." → **NFR** (implementation/security constraint).

### 2.5 Modelling for Communication: UML

Models abstract away complexity so stakeholders can understand a system without reading 10,000 lines of code. **UML (Unified Modeling Language)**, from the 1990s, defines 14 diagram types. Historically companies tried to generate entire codebases from highly-detailed UML "Blueprints" — this largely failed. **Modern usage is as "Sketches":** informal whiteboard diagrams that ignore strict syntax in favour of speed and clarity.

**Class Diagrams: Static Structure**

Shows classes, attributes, operations (methods), and relationships — *what exists*, not *when things happen*. Anatomy: **top** = class name; **middle** = attributes (e.g. `+ email: String`); **bottom** = methods (e.g. `+ verifyPassword(pwd): Boolean`). Visibility: `+` public, `-` private, `#` protected.

**Relationships:**

- **Association** — a general connection (plain line).
- **Inheritance** ("is-a") — arrow with a *hollow triangle*. E.g. `Undergrad` inherits from `Student`.
- **Aggregation** ("has-a", independent lifecycle) — *hollow diamond*. E.g. `Department` has `Professors`.
- **Composition** ("has-a", dependent lifecycle — child dies with parent) — *solid diamond*. E.g. `Building` has `Rooms`.

**Sequence Diagrams: Dynamic Behaviour**

Shows how processes interact and *in what order* — message flow over time; excellent for network calls, API interactions, authentication flows.

- **Lifelines:** vertical dashed lines representing an object/actor's lifespan.
- **Messages:** horizontal arrows between lifelines (e.g. `HTTP GET /login`); time flows top to bottom.
- **Activation boxes:** thin rectangles showing an object is actively processing.
- **Return messages:** dashed arrows for responses (e.g. `200 OK`).

**Choosing the Right Diagram**

Database schema / entity relationships → **Class Diagram**. How OAuth2 login flows between user/server/-Google → **Sequence Diagram**. What states a shopping cart can be in (Empty, Active, CheckedOut) → **State Machine Diagram**.

**Anti-pattern: Over-engineering Models (Analysis Paralysis)**

Spending 3 weeks perfecting a UML diagram in Visio is usually wasted effort — by the time it's finished, requirements have changed. **Rule of thumb:** if it fits on a whiteboard and the team understands it, it's good enough; start coding.

### 2.6 Roles & Deliverables

| Role | Responsibility |
|---|---|
| Product Manager (PM) | Owns the "Why"/"What"; conducts user research, analyses markets, shields engineers from changing whims, prioritises for business value |
| Engineering Manager (EM) | Owns technical execution: decides who builds what, oversees architecture, manages developer career growth |
| Technical Program Manager (TPM) | Manages complex cross-team dependencies (common at large tech companies) |

**The MoSCoW Prioritisation Method**

- **Must have:** product is completely unviable without this (e.g. login system).
- **Should have:** highly important, but a V1 can launch without it if necessary (e.g. password reset email).
- **Could have:** nice to have if time permits (e.g. dark mode).
- **Won't have:** explicitly out of scope for this release.

**Product Requirements Document (PRD):** the "source of truth", authored by the PM, containing background, business goals, personas, and FRs/NFRs.

**Requirement Traceability:** the ability to trace a requirement from origin (PRD) → implementation (code/PR) → verification (test), e.g. Requirement #42 → PR #105 → `test_req_42()`. Highly regulated industries (aviation, medical) require strict traceability matrices.

# 3 Engineering Process

## 3.1 What Is a Software Process?

**Software Process**

A structured set of activities required to develop a software system: **Specification** (define what the system should do), **Design and Implementation** (organise and build it), **Validation** (check it does what the customer wants), **Evolution** (change in response to changing needs).

Ad-hoc "code-and-fix" development works fine for one person and 500 lines, but is disastrous at scale: without process, developers overwrite each other's code, duplicate effort, test inconsistently, and miss deadlines.

## 3.2 The Waterfall Model

Origin: Winston Royce (1970), borrowing heavily from civil/manufacturing engineering (you don't build the roof before the foundation is inspected). Software is built in strict, sequential phases; you cannot move to the next phase until the current one is 100% complete and signed off.

**The Five Phases of Waterfall**

1. **Requirements Definition** — write an exhaustive specification document.
2. **System and Software Design** — exhaustive UML diagrams and schemas.
3. **Implementation and Unit Testing** — programmers write the code.
4. **Integration and System Testing** — put the pieces together.
5. **Operation and Maintenance** — deploy to the customer.

Strictly sequential: no going back.

**Why management loved it:** looks great on a Gantt chart; provides clear milestones; creates a paper trail that shifts accountability ("developers built exactly what was in the spec").

**The Fatal Flaw of Waterfall**

Software is not a bridge: if you realise halfway through building a bridge it needs to move 10 miles left, you restart; in software, clients change their minds constantly because they don't know what they want until they see it. Waterfall assumes requirements are frozen in Step 1 — by Step 5 (years later) business needs have changed and the software is obsolete on arrival. Integration testing doesn't happen until Phase 4, so a fundamental architectural flaw found there forces a full rewrite.

## 3.3 The Spiral Model

Barry Boehm (1986): an early fix for Waterfall. **Risk-driven:** instead of doing everything once, develop in a series of "spirals". Each loop incorporates prototyping and risk evaluation *before* committing to full development. The four quadrants of each loop are: **Objective Setting** (define objectives, constraints, alternatives), **Risk Assessment** (identify risks, build prototypes, simulate), **Development** (design, code, verify, unit test), **Planning** (review results, plan next phase, commit). Distance from the centre of the spiral represents cumulative cost.

## 3.4 Iterative and Incremental Development

- **Iterative:** build a rough version of the whole system, then refine it (drafting an essay).
- **Incremental:** build the system piece by piece, fully finishing one piece before the next (building Lego).

Modern processes combine both — building incrementally (feature by feature) and iteratively (improving each feature over time).

## 3.5 The Agile Revolution

**The Agile Manifesto (Snowbird, Utah, 2001)**

17 pioneers representing lightweight methodologies (XP, Scrum, DSDM) drafted:

1. Individuals and interactions **over** processes and tools
2. Working software **over** comprehensive documentation
3. Customer collaboration **over** contract negotiation
4. Responding to change **over** following a plan

| Value | Meaning |
|---|---|
| Individuals & interactions | Process should serve the team; a fool with a tool is still a fool |
| Working software | The primary measure of progress; a perfect spec doc with no code has zero value |
| Customer collaboration | Client is part of the team; show working software every ∼2 weeks and adapt |
| Responding to change | A plan is useless if the market has moved on; architecture must stay modular |

**What Agile IS NOT**

Agile is *not* an excuse to write zero documentation, *not* an excuse to skip planning (Agile teams plan *more* frequently than Waterfall teams), and *not* chaos — it requires immense discipline.

**Premature optimisation** (Donald Knuth: "the root of all evil ... in programming"): don't build a hyper-scalable distributed architecture for a product with zero users. **YAGNI** (You Ain't Gonna Need It): only build what is necessary for the current requirement; get correctness first, optimise only if performance becomes a bottleneck.

*Extreme Programming (XP)*

Kent Beck (1990s): take recognised good practices and turn them up to "extreme" levels; focused on technical execution.

- **Pair Programming:** two developers, one keyboard — driver types, navigator reviews line-by-line.
- **Test-Driven Development (TDD):** write the test before the code.
- **Continuous Integration:** merge to main multiple times a day.

Few companies do pure XP today, but its technical practices are universally adopted.

### 3.6 The Scrum Framework

Scrum is the dominant Agile framework today. Unlike XP (technical), Scrum is a **management** framework: it doesn't tell you how to write code, it tells you how to organise the work. Based on **Empiricism** — knowledge from experience, decisions from what is known.

**The Three Pillars of Scrum**

**Transparency:** the process and work must be visible to all involved. **Inspection:** artifacts must be frequently inspected to detect undesirable variance. **Adaptation:** if a process deviates outside acceptable limits, it must be adjusted as soon as possible.

**Artifacts:**

- **Product Backlog:** the master, ever-evolving to-do list; replaces the 500-page Waterfall spec. Items near the top are highly detailed; near the bottom, vague ideas.
- **User Story format:** "As a [user], I want [action/feature], so that [value/reason]." E.g. "As a student, I want to filter my timetable by term, so that I only see current lectures."
- **Acceptance Criteria:** specific conditions a story must meet. **Definition of Done (DoD):** a universal team checklist (code reviewed, tests pass, docs updated, deployed to staging). If it doesn't meet the DoD, it is not done — no exceptions.
- **Sprint Backlog:** the chosen Product Backlog items for this Sprint plus a delivery plan. **Increment:** the sum of completed work in a Sprint — must be potentially releasable.

**Events:**

- **Sprint:** a fixed timebox (usually 2 weeks) producing a Done, useable, potentially releasable Increment. No changes that endanger the Sprint Goal during the Sprint; urgent new work goes into the *next* Sprint's backlog.
- **Sprint Planning:** defines the Sprint Goal, selects backlog items, and breaks them into technical tasks.
- **Daily Scrum (Standup):** 15-minute, time-boxed sync for developers: what did I do yesterday? what will I do today? any blockers? **Not** a status report for management.

- **Sprint Review:** at Sprint end, the team presents results to stakeholders; a working session, not a formal presentation; feedback updates the Product Backlog.
- **Sprint Retrospective:** after the Review, before next Planning; discusses what went well/wrong and how to improve the *process itself*.

**Estimation: Story Points**

Humans are terrible at estimating absolute time but much better at *relative sizing*. Agile uses Story Points (often Fibonacci: 1, 2, 3, 5, 8, 13) to measure complexity/effort/risk, not time. **Planning Poker:** the team votes simultaneously; large discrepancies (e.g. an 8 vs. a 1) trigger discussion to surface hidden complexity.

### 3.7 Process Roles

| Role | Responsibility |
|---|---|
| Scrum Master | Servant-leader; facilitates Scrum events, removes blockers, protects the team from external interruptions — does *not* assign tasks |
| Product Owner (PO) | Represents business/customer; sole owner of the Product Backlog; decides priority by business value |
| Developers | Cross-functional, self-organising execution team; nobody (not even the Scrum Master) tells them *how* to turn backlog items into Done increments |

**"Scrum-but": Fake Agile**

"We do Scrum, but our Sprints are 6 weeks." "...but management injects urgent tasks daily." "...but we skip retrospectives because we're too busy." Result: Waterfall disguised with Agile terminology, leading to burnout and failed projects.

### 3.8 Kanban

Originates from Toyota manufacturing. Visualises work on a board (To Do, In Progress, In Review, Done); **no time-boxed sprints** — items are pulled continuously. Best for operations/maintenance teams handling continuous incoming tickets.

**WIP Limits**

Kanban enforces strict Work-In-Progress limits per column (e.g. "only 3 items in 'In Progress' at once"). If the limit is reached, developers must stop pulling new work and help clear the bottleneck. Optimises for *flow* over batched features.

### 3.9 Scaling Agile

Scrum works for teams of 5–9. To scale to 1,000s of developers: **SAFe** (Scaled Agile Framework) — highly structured, popular in enterprise/government, often criticised as "Waterfall in disguise"; **LeSS** (Large-Scale Scrum) — keeps Scrum principles across multiple teams sharing one product backlog; **Spotify Model** — Guilds, Tribes, Squads (Spotify themselves no longer strictly use it).

# 4 Quality Assurance & Reliable Delivery

## 4.1 Version Control Systems

Agile teams move fast; without a technical coordinator, two developers editing the same file will inevitably overwrite each other's changes. VCS provides the single source of truth and history (auditing, compliance, rollback).

| | Centralized (SVN, CVS) | Distributed (Git, Mercurial) |
|---|---|---|
| Model | Single central server holds history | Every developer has full history locally |
| Pros | Simple; easy access control; single source of truth | Fast (local ops); works offline; no single point of failure |
| Cons | Single point of failure; slow (network needed every op) | Steeper learning curve; local storage grows |

**Git (Linus Torvalds, 2005)**

**Commits:** a snapshot of the entire project; each has a unique SHA-1 hash and a message. **Staging area:** a "loading zone" to select changes for the next commit. **Safety:** once committed, data is nearly impossible to lose — you can experiment locally, fail, and revert to a known good state.

**Branching & Merging:** `main`/`master` is the stable, production-ready branch; developers create temporary **feature branches**. A **merge** combines a feature branch back into main. A **merge conflict** occurs when two branches changed the same line — Git pauses and forces a human to resolve it.

| | Git Flow | Trunk-Based Development |
|---|---|---|
| Style | Long-lived branches (`develop`, `release`, `master`) | Everyone works on `main`; features $<$ 1 day |
| Good for | Traditional versioned releases (v1.0, v2.0) | Agile/CI-heavy teams |
| Pain point | "Merge Hell" merging long-lived branches | Requires excellent automated tests to keep main stable |

**Pull Request (PR) workflow:** a dedicated space for **code review** — knowledge sharing, bug detection (four eyes $>$ two), style consistency. This is how Agile teams maintain quality without a central "Design Authority".

**Bug Lifecycle**

New → Triaged (confirmed & prioritised) → In Progress → Resolved (code written, tests pass) → Verified (QA/reporter confirms) → Closed. **Traceability:** linking commits to Issue IDs ("Fixes #42") connects the How to the Why.

**Coding standards & linters:** since 80% of cost is maintenance, uniform code (enforced by linters like ESLint/Pylint) reduces cognitive load, prevents review "nitpicking", and enables safe automated refactoring.

## 4.2 Testing

Tests provide **confidence** (no regressions), **documentation** (executable specs of how a function is meant to be used), enable safe **refactoring** (you cannot clean up spaghetti code without a suite proving external behaviour is preserved), and improve **design** (untestable code often reveals bad architecture).

| | Manual Testing | Automated Testing |
|---|---|---|
| Pros | Exploratory; catches visual glitches; human intuition | Fast; repeatable; runs on every commit; scales infinitely |
| Cons | Slow; unrepeatable; error-prone; doesn't scale | High setup cost; tests need maintenance; can't judge if UI is "ugly" |

### The Testing Pyramid

**Base — Unit Tests (many):** test a single function in isolation, fast (<10ms). E.g. `sum(2,2) == 4`.
**Middle — Integration Tests (some):** test boundaries between modules, e.g. the User Service saving to a real PostgreSQL DB.
**Top — End-to-End (E2E) Tests (few):** slow, brittle, expensive, e.g. a Selenium script opening Chrome and logging in as "Bob".
Moving up the pyramid: cost and slowness increase; speed and isolation decrease.

### Unit Testing & Mocking

**Unit tests** must be fast and isolated. If a function calls an external Payment API, don't call the real API (slow/expensive) — use a **Mock** or **Stub**.
- **Stub:** returns a hardcoded value (e.g. always "Success").
- **Mock:** verifies interaction (e.g. "ensure the email service was called exactly once").

Python's `MagicMock` is dynamic (auto-creates attributes/methods), supports stubbing (`.return_value`), spying (records all calls/arguments), and verification (`.assert_called_*`).

### Code Coverage ≠ Correctness

**Code coverage** measures the % of source code executed by the test suite. It identifies dead code and untested logic paths, but **100% coverage does not mean every logical edge case is correct**. **Classic trap:** `average(scores) = sum(scores)/len(scores)` with only the test `average([10,20]) == 15` gets 100% coverage, but `average([])` crashes with `ZeroDivisionError`. Coverage tracks *which lines ran*, not *which edge cases were verified*.

### Mutation Testing: "Test your Tests"

High coverage only proves code was *executed*, not *verified*. Mutation testing automatically injects small bugs ("mutants") into source code (e.g. `>` → `>=`, `+` → `-`, delete a call).
- If tests still pass, the mutant **survived** → your tests are weak at that location.
- A strong suite should **kill** (fail against) all mutants.

**Rarely run on every CI build for huge codebases** because generating and running tests against thousands of mutants across millions of lines is computationally very expensive.

### Test-Driven Development (TDD): Red-Green-Refactor

1. **Red:** write a failing test for the new feature.
2. **Green:** write the minimum code to pass it (no matter how ugly).
3. **Refactor:** clean up the code; the test ensures the feature isn't broken while cleaning.

Result: 100% test coverage by default, and simpler code (since you design for testability first).

## 4.3 Continuous Integration, Delivery & Deployment

| Term | Meaning |
|---|---|
| **CI** | Developers merge to main ≥ once/day; every merge triggers an automated build + full test suite; fast feedback ("breaking the build") within minutes; eliminates Merge Hell from long-lived branches |
| **CD (Delivery)** | Any version of the code is ready to ship at any time; tests pass → auto-deploy to Staging; a *human* decides when to push to Production (a business decision) |
| **CD (Deployment)** | Fully automated path from code to customer; passing all pipeline tests → auto-deployed to real users with *no* human oversight; relies on automated rollback on error spikes |

### Anatomy of a CI/CD Pipeline

Commit → Build → Unit Tests → Lint/Scan → Integration Tests → Deploy Staging → Deploy Prod. **Failing any step rejects the build.** Quality Gates: every step must pass to advance. The Build step creates a versioned artifact (e.g. a Docker image) used identically across all later environments.

**Promotion pipeline (environments of increasing production-likeness):** Development (local, "wild west") → CI/Test (clean, ephemeral) → Staging/UAT (identical to Production; the "final rehearsal") → Production (real users, real money).

**Google's "xFood" Progressive Internal Testing**

**Fishfood** (20–30 immediate devs, unstable, fast iteration) → **Teamfood** (50–100, wider dept, more stable) → **Dogfood** (all employees, testing the Release Candidate closest to real users). Goal: catch "annoyance" bugs and UX friction in increasing concentric circles before public launch.

### 4.4 Deployment Strategies

**Blue-Green Deployment**

Two identical environments. **Blue** runs the current version (100% traffic); **Green** receives the new version. The load balancer switches traffic entirely from Blue to Green. **Benefit:** instant rollback — just switch back. Zero downtime.

**Canary Deployment**

Deploy the new version to a small subset of servers/users (e.g. 5%). Monitor error rate/latency; if healthy, progressively expand to 100%; if errors spike, auto-rollback. Better suited than Blue-Green for **high-risk changes like a database migration**, since it limits the blast radius of an unforeseen issue to a small fraction of real traffic rather than switching everyone at once.

**Feature Flags (Toggles) & Progressive Delivery**

**Deployment** (moving code to production servers) is decoupled from **Release** (making the feature visible to users). New code sits behind an `if` controlled by a database toggle: turn ON for developers, then Beta testers, then everyone. Enables an "emergency shutoff" without a code rollback — this is the mechanism of **Progressive Delivery**.

**A/B Testing vs. Canary**

**A/B Testing** measures *business/product* value (e.g. does a red "Buy" button increase sales versus a blue control?) using statistical significance testing. **Canary** measures *technical* health (error rates, latency) of a new release. Don't confuse the two in an exam answer.

**SLO, SLI & Error Budget**

**SLI (Service Level Indicator):** what you measure, e.g. latency.
**SLO (Service Level Objective):** the target, e.g. 99.9% uptime.
**Error Budget** = 100% − SLO. If SLO is 99.9%, the error budget is 0.1%. If budget remains, the team may ship risky features; **if the budget is exhausted, all feature releases freeze** to focus purely on stability.

**Blameless post-mortems:** "Why did the system allow the human to make this mistake?" rather than "who do we fire?" — focuses on systemic causes and produces new automated safeguards.

**What is DevOps?**

Coined by Patrick Debois (2009, first "DevOpsDays"), addressing the toxic silo between developers (rewarded for shipping change) and operations (rewarded for stability/uptime). The solution: merge the teams — "you build it, you run it"; whoever builds a feature is also responsible for its production stability.

### 4.5 Cloud Systems

**On-Premises** (CapEx: own the hardware, power, cooling) vs. **Cloud** (OpEx: pay-as-you-go, elastic scaling from 1 to 1,000,000 users in seconds).

| Model | You get | Example | Trade-off |
|---|---|---|---|
| IaaS | Raw VMs, storage, networks | AWS EC2, GCE, Azure VMs | Max control; high management overhead (patch the OS yourself) |
| PaaS | Upload code; provider manages OS/runtime/scaling | Heroku, Elastic Beanstalk, App Engine | Fast time-to-market; vendor lock-in, less control |
| SaaS | Fully functional application over the web | Salesforce, Slack, Gmail | Zero maintenance; zero control over features/data |

**Shared Responsibility Model**

**Provider** is responsible for security *OF* the cloud (physical data centres, hardware, base networking). **Customer** is responsible for security *IN* the cloud (application code, data, firewall settings). Analogy: the landlord supplies the lock; you must remember to turn the key.

**Containers (Docker) vs. Virtual Machines**

The problem: "it works on my machine but not in production." **Docker** packages code and all dependencies (libraries, config, OS files) into a single image.
**VMs** each run a full **Guest OS** on top of a Hypervisor → heavyweight, slow to boot.
**Containers** share the **Host OS kernel** via the Docker Engine → lightweight, boot in milliseconds, ∼1/10th the resource use of a VM.

**Kubernetes (K8s): Container Orchestration**

Docker runs a single container; K8s manages fleets of them. Originally built by Google.
- **Self-healing:** restarts failed containers.
- **Auto-scaling:** adds containers under traffic spikes.
- **Load balancing:** distributes traffic across healthy containers.

Architecture: a **Control Plane** (API, Scheduler, DB) manages many **Worker Nodes**, each running multiple **Pods** (the smallest deployable unit). A **Cluster** is a single logical computer made of many physical ones.

### 4.6 Observability

**Metrics** (quantitative: CPU usage, 500-error counts), **Logs** (qualitative textual history of events), **Traces** (following a single request across microservices), and **Dashboards** (real-time visualisation, e.g. Grafana). "You cannot manage what you cannot measure."

# 5 Software Evolution

## 5.1 Lehman's Laws of Software Evolution

Software is never "finished" — most of its life is spent in maintenance/evolution, and systems must evolve or become progressively less useful.

**Key Lehman's Laws (E-type systems)**

- **Continuing Change:** a system must be continually adapted or it becomes progressively less satisfactory.
- **Increasing Complexity:** as a system evolves, its complexity increases unless active work is done to maintain/reduce it.
- **Self-Regulation:** global evolution processes are self-regulating with statistically determinable trends and invariants.

**The Four Types of Maintenance (with typical effort share)**

- **Perfective (50%):** improving performance, maintainability, or readability without changing behaviour.
- **Adaptive (25%):** updating the system for a new environment (new OS, cloud migration).
- **Corrective (20%):** fixing bugs discovered by users.
- **Preventive (5%):** finding and fixing latent problems before they become actual bugs.

Most maintenance is *not* about fixing bugs — it is about keeping the system relevant.

## 5.2 Legacy Code

**Defining Legacy Code**

Colloquially "code inherited from someone else", but more practically: **code without tests, or code we are afraid to modify**. Legacy code is the foundation of almost all profitable businesses; the original authors may have long since left.

**Case Study: COBOL & the Pandemic (2020)**

Unemployment systems in several US states (NJ, KS) crashed under COVID-19 load. The systems were written in **COBOL in the 1970s**; original authors were retired or deceased. Governors went on TV to recruit volunteers who knew the language. **Lesson:** legacy code is often the most business-critical code a company has.

**The high cost of reading:** understanding the side-effects of a 500-line function takes significant cognitive load; engineers must build a mental map of data flow they didn't design; legacy systems often mix architectural styles ("architectural drift"). Investing in readability today saves thousands of future reading hours.

**Software Archaeology**

The source code is the only ultimate source of truth. Tools: `git blame` (who changed each line, and when), `git log -S` (search history for when a string/function was added or removed). **Goal:** find the *intent* behind the code, not just what it does. Linking commits to issue trackers (e.g. `FIX-123`) preserves business context that comments miss. Beware "tribal knowledge" — implicit understanding never captured in the system.

## 5.3 API Design & Versioning

APIs are **promises**: the interaction boundary between components/systems. Once published and used by others, changing an API becomes extremely expensive. **The Contract:** if the caller provides input $X$, you promise output $Y$.

**Semantic Versioning (SemVer): MAJOR.MINOR.PATCH**

- **MAJOR:** breaking / incompatible API changes.
- **MINOR:** new, backwards-compatible functionality.
- **PATCH:** backwards-compatible bug fixes.

Example: 2.1.4 → 2.2.0 (new feature) → 3.0.0 (breaking change). **Rule:** when in doubt, bump MAJOR.

**What counts as breaking?** Structural: removing/renaming a function/class/field; changing a parameter type;

adding a required parameter. Behavioural: changing a default value; throwing a new checked exception; changing the order of returned results.

### The "Zero-Ver" Problem (0.y.z)

SemVer states $0.y.z$ is for initial development — anything may change at any time. Many projects (early React, Terraform) stay on $0.x$ for years, so users cannot rely on MAJOR/MINOR/PATCH logic to protect them. **Best practice:** bump to 1.0.0 as soon as software is used in production, as a signal of maturity and commitment to stability.

### Dependency Constraints

- **Exact** (`1.2.3`): only that version — safest, but no bug fixes.
- **Tilde** (`~1.2.3`): allows PATCH updates only (up to `1.2.x`).
- **Caret** (`^1.2.3`): allows MINOR and PATCH updates (up to `1.x.x`) — the industry default, assuming authors follow SemVer strictly.

**Lockfiles** (`package-lock.json`, `Cargo.lock`) record the exact version used in the last successful build, for reproducibility across the team.

### Postel's Law (The Robustness Principle)

"Be conservative in what you send, be liberal in what you accept." — Jon Postel.

- **Conservative (Writer):** stick strictly to the contract; don't send extra fields.
- **Liberal (Reader):** ignore unrecognised extra fields instead of crashing → **Forward Compatibility**, letting newer systems add fields without breaking older ones.

This simple rule is why the Internet (TCP/IP, HTTP, HTML) has survived 40+ years.

- **Backward compatibility:** newer code can read data / handle requests from older code (crucial during rolling deployments).
- **Forward compatibility:** older code can gracefully handle data from newer code (e.g. by ignoring unknown fields).
- In a distributed system, you cannot upgrade every node at the exact same instant — a mix of versions is always running simultaneously.

### Hyrum's Law

"With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody."

### Hyrum's Law in Action: The SimCity/Windows 95 Hack

Windows 95 developers found SimCity crashed on the new OS due to a memory bug *in the game*. Rather than let SimCity break, Microsoft added a special check to the Windows 95 kernel: "if the app is SimCity, use a special, slower memory allocator that masks the game's bug." **Takeaway:** once you have enough users, every bug becomes a "feature" someone depends on — this is why breaking changes/migrations of long-standing legacy systems are so complicated.

### The Deprecation Path (4 Steps)

1. **Announce:** warn users a feature will be removed.
2. **Deprecate:** mark the API `@Deprecated` (generates compiler/runtime warnings); keep the old endpoint running *alongside* the new one.
3. **Wait:** give users 6–12 months (or several minor versions) to migrate.
4. **Remove:** finally remove the code in a new MAJOR version.

This is why an old endpoint (e.g. an XML API) cannot simply be turned off tomorrow — doing so would be an uncommunicated breaking change with no migration window, instantly breaking every downstream customer's CI/CD pipeline and production system.

### 5.4 Refactoring & Technical Debt Remediation

Refactoring is *restructuring existing code without changing external behaviour*; it is a disciplined engineering technique, not just "cleaning up". **Prerequisite:** a comprehensive automated test suite — if tests pass before and after, behaviour was (likely) preserved.

**Debt in Action: The Y2K Bug ($300 Billion)**

**The debt (1960s):** years were stored as two digits (`98` vs. `1998`) to save 2 bytes of expensive RAM, under the assumption "this code will be replaced by 2000". **The reality:** Lehman's Law (Continuing Change) meant the systems stayed in use for 40+ years. **The interest:** in 1999 the world spent an estimated $300 billion to prevent global financial/infrastructure collapse. **Lesson:** short-term technical debt can carry multi-billion-dollar long-term interest.

**Characterization ("Golden Master") Tests**

How do you refactor code with no tests and no understood requirements? Record the *current* output for a given input and assert it stays identical during refactoring. You are testing for **consistency**, not **correctness**.

**The Strangler Fig Pattern**

Named after the vine that grows around a tree, eventually replacing it entirely.
1. Create a **Proxy/Facade** in front of the legacy system.
2. Build new functionality as a separate service.
3. Route specific calls to the new service; leave the rest for legacy.
4. Gradually migrate all functionality until legacy has no traffic left.

**Benefit:** low risk, continuous delivery, avoids the dangerous "Big Bang" rewrite.

**The Big Bang Rewrite: Netscape's Disaster (1997)**

Netscape Navigator dominated the browser market but had "messy" code, so they threw it away and rewrote from scratch (Netscape 5.0). The rewrite took **3 years** with *no new features shipped*; Internet Explorer took 90% of the market; Netscape went bankrupt. **Rule:** never do a Big Bang rewrite of a successful system — it is high-risk because it stops all feature delivery for an extended, uncertain period while competitors continue to ship.

**Code Smells (When to Refactor)**

- **Long Method:** if you can't describe what a function does in one sentence, it's too long.
- **God Object:** a class that knows/does too much (violates Single Responsibility).
- **Feature Envy:** a method more interested in another class's data than its own.
- **Data Clumps:** groups of variables always passed together (e.g. $x, y, z$ coordinates).

A smell is not a bug — it is a signal that the design is decaying.

**Extract Method refactoring example:** a monolithic `printOwing()` that both calculates an outstanding balance and prints details is split into `getOutstanding()` and `printDetails(outstanding)`, called from a slim `printOwing()`. Behaviour is unchanged; readability and testability improve.

### 5.5 Observability & Site Reliability Engineering (SRE)

**Monitoring vs. Observability**

**Monitoring** tells you *that* something is wrong ("CPU is at 99%") — it addresses **known unknowns**. **Observability** tells you *why* something is wrong: the ability to understand a system's internal state from its external outputs ("which specific request caused the memory leak?") — it addresses **unknown unknowns**.

**The Telemetry Triad: Logs** (discrete, textual events), **Metrics** (aggregatable numbers: counters, gauges), **Tracing** (end-to-end request flow through microservices) → together they give *Observability*.

**The Four Golden Signals of Monitoring**

**Latency** (time to service a request), **Traffic** (demand, e.g. req/sec), **Errors** (rate of failed requests, e.g. 500s/timeouts), **Saturation** (how "full" the service is, e.g. CPU/Memory/Disk I/O). **Saturation is often harder to measure than Latency** because it requires knowing a resource's true maximum capacity in advance (which shifts under different workloads), whereas latency is a direct, unambiguous per-request timing that requires no such prior model.

**Site Reliability Engineering (SRE)**

"SRE is what happens when you ask a software engineer to design an operations team." — Ben Treynor Sloss (Google). Treat Operations as a Software Problem: automate repetitive tasks instead of doing them by hand.

- **The 50% Rule:** Google SREs spend max 50% of time on "toil" (manual ops); the rest on engineering projects that improve the system.
- **Reliability is a feature, not an absolute:** 100% reliability is the wrong target for almost everything — too expensive and it slows innovation.
- **SLI** = what you measure (e.g. latency). **SLO** = the target (e.g. <200ms for 99.9% of requests).

**Resilience Engineering:** traditional Ops tries to prevent failure (maximise **MTBF** — Mean Time Between Failures); modern SRE assumes failure is inevitable and focuses on **MTTR** — Mean Time To Recovery. The goal is a system that survives the loss of a disk, server, or entire data centre without user impact.

**Chaos Monkey (Netflix, 2010)**

As Netflix migrated from stable data centres to the "unreliable" cloud (AWS), they knew instances would disappear randomly. **The solution:** a tool that randomly killed production servers *during business hours*, forcing engineers to build services that recovered automatically. If your server might die at 2pm on a Tuesday, you make sure it doesn't matter. Expanded into the "Simian Army": **Chaos Gorilla** (kills availability zones), **Chaos Kong** (kills entire regions).

**Chaos Engineering: The Method**

"The discipline of experimenting on a software system to build confidence in its capability to withstand turbulent conditions in production."

1. Define the **Steady State** (e.g. error rate $< 0.1\%$).
2. Form a **Hypothesis** (e.g. "if we stop the database, the cache keeps the site alive").
3. Introduce a real **Experiment** (e.g. actually stop the DB).
4. **Verify** whether the steady state was maintained.

### 5.6 Software Sustainability & Supply Chain

Modern apps rely on hundreds of third-party libraries (npm, PyPI, Maven); a **transitive dependency** is your dependency's own dependency. A vulnerability in a small, obscure library can compromise your entire system.

**The Left-Pad Incident (2016)**

A developer deleted a tiny 11-line npm package called `left-pad`. Within hours, thousands of projects (including React and Babel) failed to build — many developers didn't even know they transitively depended on it. **Lesson:** you are responsible for the security and availability of every library in your dependency tree.

**Log4Shell (2021)**

A bug in `log4j`, a standard Java logging library, allowed attackers to take over a server just by sending a crafted string to be logged (e.g. via a chat box or search bar). Companies spent weeks manually searching their systems to see if they even used `log4j`. **The solution:** an **SBOM (Software Bill of Materials)** — a machine-readable list of all components, so a newly-discovered library bug can be checked against your systems instantly.

**CVE (Common Vulnerabilities and Exposures):** a list of publicly disclosed cybersecurity vulnerabilities. Engineers must regularly update dependencies even absent new feature needs; automated tools (Dependabot,

Snyk) flag outdated/vulnerable libraries.

# 6 Software Engineering in the AI Era

**Disclaimer**

This is a high-velocity area: breakthroughs occur weekly. Exam answers should focus on **enduring engineering principles** (verification, ownership, architecture) rather than which specific tool is currently fashionable.

## 6.1 AI as a Coder

- **Autocomplete vs. semantic understanding:** traditional IntelliSense (LSP) knows `user.` should be followed by a `User` member; LLMs understand *intent* across multiple files — a comment `// Calculate the Haversine distance` triggers understanding of the mathematical intent, not just syntax.
- **Semantic (project) awareness:** modern tools use "project context" (RAG for code) — reading `package.json`, existing utilities, and naming conventions to suggest code matching *your* codebase's style, not a generic solution.
- **Boilerplate elimination:** 40–60% of professional code is boilerplate (routes, schemas/migrations, DTOs/mappers, test scaffolding); AI reduces "time to Hello World" from hours to seconds.
- **Prototyping:** AI can generate an entire directory structure, Dockerfile, DB schema, and API endpoints in one pass for rapid hypothesis testing — but prototypes are often "happy path" only.
- **Language translation:** handles the "syntactic tax" of switching languages (e.g. Python → C++/Rust for performance; C → Rust for memory safety).
- **Testing:** AI can generate comprehensive suites (happy path, edge cases, mocking) but "tests what the code does, not necessarily what it should do". **AI Fuzzing:** generates plausible-but-malformed data (e.g. a JSON payload with a circular reference) rather than random bytes.
- **Self-healing CI/CD:** an agent reads a failing test's error log/stack trace, identifies the failing line, and proposes a fix commit — a "night watchman" for the codebase.
- **Debugging:** correlating disparate information (e.g. "Service A expects ISO-8601, Service B sends a Unix timestamp") and performing anomaly detection across terabytes of logs.
- **Refactoring & onboarding:** restructuring high-cyclomatic-complexity functions; semantic search across a 10-million-line codebase ("show me every place we calculate VAT").

**Risks of AI Coding**

- **Not all models are equal:** a small local model may be fast but logic-error-prone; a large model has better reasoning but higher latency/cost. **The trap:** assuming 10 correct answers guarantee an 11th correct answer.
- **Technical debt of ignorance:** if you submit code you don't understand, you cannot maintain it; AI often uses "hacky" tricks that pass tests but violate long-term architecture. **Rule: you own every line you commit.**
- **Hallucinations:** AI invents libraries/API methods that don't exist, and can confidently suggest insecure patterns common in its training data (e.g. old library versions with known CVEs).
- **Algorithmic bias:** AI reflects training-data biases (GitHub/StackOverflow), which can surface as biased logic in socio-technical systems (e.g. loan approval) or a bias toward "popular" but sub-optimal solutions.

**Mitigation: The "Critic" (Generator/Verifier) Architecture**

A **Generator** model writes the code; a separate **Verifier** model tries to find bugs, security flaws, or style violations in the generated output. This adversarial pairing significantly reduces hallucinations reaching production.

## 6.2 AI Throughout the Workflow

- **Design/brainstorming:** generating candidate features, personas, and "Jobs to be Done" as a creative partner for Product Management; generating architectural trade-off tables and Mermaid/PlantUML diagrams from a description.
- **Team coordination:** summarising long Slack/Jira threads; acting as a "Scrum Assistant" flagging blocked tasks from chat sentiment.

- **Automated code review:** beyond linting ("you missed a semicolon"), AI can catch *architectural* violations ("this function is called in a loop but performs a synchronous DB query — performance bottleneck as users grow").
- **Documentation generation:** auto-generating and *keeping in sync* docstrings, READMEs, and API references (e.g. Swagger); flags stale docs when code changes.

**AI Agents**

**Tool use:** models can call functions, run shell commands, and read files. **Agency:** breaking a complex goal ("fix this bug") into multiple steps and executing them autonomously. Example (Claude Code / Copilot Workspace, "Implement Dark Mode"): the agent identifies all CSS/theme files, modifies them, updates the toggle component, and runs tests to check for regressions — end-to-end task completion, not just single-file suggestions.

### 6.3 Building FOR AI vs. WITH AI

- **Building WITH AI:** using Copilot to write a conventional app; the final binary contains no AI logic.
- **Building FOR AI:** an LLM is a *core runtime component* of the product itself (e.g. an automated legal or medical assistant).

**The Reliability Paradox**

**Traditional code:** $f(x) \to y$ (always, deterministically). **AI code:** $f(x) \to y$ (probably). How do you build a reliable system on top of a black box that might behave differently tomorrow? **Why AI coding advances faster than, say, creative writing AI:** code has a **ground truth** — does it compile? do tests pass? — so the model can learn from its own mistakes by running what it wrote. There is *no compiler* for medical advice or legal opinions (**the "Virtual Doctor Problem"**); confidence there requires layered guardrails, large human-graded evaluation sets, and using LLMs to check other LLMs for factual consistency.

**Architectural challenges of building FOR AI:**

- **Latency:** a DB query takes ∼10ms; an LLM generation takes 2,000–10,000ms → requires streaming architectures (show text as generated) and asynchronous UX patterns.
- **Cost/scalability:** AI APIs charge per token, unlike traditional "free once hosted" APIs; a viral app can go bankrupt in a week without cost-caching and rate-limiting.
- **Provider lock-in:** building tightly around one vendor's proprietary API makes switching providers hard; best practice is to build a **model-agnostic abstraction layer**.

### 6.4 The New Engineer

- **The Junior Developer Gap:** junior tasks (small bug fixes, boilerplate) are exactly what AI is best at; if companies stop hiring juniors to do this "easy" work, the traditional training ladder to Senior Architect breaks. The new junior skill is *orchestrating* AI to do 10× more work while maintaining quality.
- **Jevons Paradox:** making something more efficient usually *increases* total consumption. Making software 10× cheaper to build won't mean 1/10th the engineers are needed — it means companies will want 100× more software.
- **Shifting skillset:** raw syntax knowledge is now a commodity; what remains valuable is **domain knowledge**, **system design**, and **human communication** (defining the right thing to build).
- **Lifelong learning:** a tech stack's lifespan is now 2–3 years; you are not "a React developer" but a **problem solver** who uses the best available tools.

**The Reasoning Ceiling**

Current LLMs are excellent at pattern-matching against vast training data and at tasks with a clear feedback loop (compile/test). They lack: genuine long-horizon strategic judgement across ambiguous, conflicting stakeholder priorities; accountable ownership of consequences; and the tacit, cross-domain organisational context a Senior Architect accumulates over years. This is why an AI cannot currently *replace* a Senior Software Architect — it can accelerate their execution, but not their judgement.

**Course Review: What AI Changes (and Doesn't)**

AI changes the **tools**, not the **goals**. The core engineering questions remain: is the code **reliable** (tested, robust)? Is it **maintainable** (can another human – or AI – understand it in 2 years)? Is it **secure** (is user data protected)? Does it deliver real **value** (does it solve a real human problem)?

"AI will not replace software engineers. Engineers who use AI will replace engineers who don't."

# 7 Security Engineering Appendix

**Why This Section Exists**

The course is titled *Software and Security Engineering*, and the 2025 Tripos paper devotes a full 14-mark part to authentication design. This appendix consolidates the security-specific vocabulary (access control, threat modelling, information flow) referenced across the lectures and needed to answer such questions with precision.

## 7.1 Access Control

**The Access Control Matrix**

A conceptual matrix with **Subjects** (users/processes) as rows and **Objects** (files/resources) as columns; each cell lists the permissions a subject has on an object. Two practical implementations store this matrix differently:

- **Access Control Lists (ACLs) — column-centric:** each *object* stores the list of subjects and their permissions ("who can access *this file*?"). Easy to audit *who can reach a given resource*; harder to answer "what can this user access?" without scanning every object.
- **Capabilities — row-centric:** each *subject* holds an unforgeable token per object it may access ("what can *this user* do?"). Easy to answer "what can this user access?" and supports delegation (passing a capability on); harder to revoke access to a specific object across all holders, and harder to audit "who can reach this resource?" centrally.

## 7.2 STRIDE Threat Modelling

**STRIDE**

A mnemonic for a systematic threat-modelling checklist, applied to each component/data-flow of a system design:

- **S**poofing — pretending to be someone/something else (e.g. a forged login).
- **T**ampering — unauthorised modification of data or code in transit or at rest.
- **R**epudiation — denying having performed an action, absent an audit trail.
- **I**nformation Disclosure — exposing data to unauthorised parties.
- **D**enial of Service — degrading or blocking legitimate access.
- **E**levation of Privilege — gaining capabilities beyond what was granted.

**Exam Tip: Using STRIDE**

If asked to analyse the security of a described system (e.g. a decentralised social network), walk each STRIDE category against the specific data flow given in the question (e.g. "a post travelling from a home server to a remote server"), rather than reciting the mnemonic generically — examiners reward concrete application.

## 7.3 Information Flow

**One-Way Information Flow**

In a system with security levels, information should only travel in directions that cannot leak higher-security information to lower-security observers:

- For **confidentiality**, information flows from **low** → **high** (a low-clearance input may be read by a high-clearance process, but high-clearance data must not flow down to low-clearance readers).
- For **integrity**, trusted (high-integrity) information flows from **high** → **low** (untrusted/low-integrity input must not silently corrupt trusted, high-integrity state).

## 7.4 Testing a Decentralised Server

For a scenario like the 2025 Tripos question (a decentralised, federated social network where anyone can run a server), useful testing strategies to cite include:

- **Unit tests** for core server logic (post validation, ranking, storage) in isolation, mocking the network layer.
- **Integration tests** against a real database and a locally-run second "federated" server instance, to catch boundary/serialisation bugs (cf. the Mars Climate Orbiter units failure).

- **Interoperability / conformance testing** against the federation protocol specification, since *other people's* servers are untrusted, independently-implemented black boxes — you must test against the *protocol*, not just your own implementation.
- **Fuzzing** of federation message parsers, since a hostile or buggy remote server can send malformed data (deliberately or otherwise); this is a classic Denial-of-Service / Tampering (STRIDE) surface.
- **Load / chaos testing** for scale and resilience, given the system is described as "rapidly growing" (cf. Chaos Monkey, London Ambulance Service overload failure).
- **Security testing** (penetration testing, STRIDE-driven test cases) specifically around authentication and inter-server trust, since a compromised or malicious server could otherwise forge posts.

### 7.5 Password vs. Public-Key Authentication

**How Password Authentication Works**

The user submits a username and password over TLS; the server compares it against a stored, **salted and hashed** credential (e.g. bcrypt, cf. Lecture 2's NFR example) and issues a session token/cookie on success.

**How Public-Key Authentication Could Work**

**Login to the home server:** the user generates a key pair; the **public key** is registered with their home server. To log in, the server sends a random **challenge (nonce)**; the client signs it with their **private key**; the server verifies the signature against the stored public key. No secret ever crosses the network.
**Authenticating posts across servers:** each post is **signed** by the author's private key. A remote server verifies the signature using the author's public key (fetched from the author's home server, analogous to fetching a certificate). This lets a receiving server cryptographically confirm "this post really came from this user's home server" without having to trust the remote server's word alone — directly mitigating STRIDE's **Spoofing** and **Repudiation** categories for federated content.

| Aspect | Password | Public-Key |
|---|---|---|
| Secret transmitted? | Yes (over TLS) — vulnerable if server or TLS is compromised, and vulnerable to phishing | No; only a signed challenge is sent — resistant to phishing and server-side leaks of the credential itself |
| Server-side breach impact | Stolen hash database can be brute-forced/cracked offline | Only public keys are stored server-side; a breach discloses no secret |
| Usability | Familiar to all users; easy recovery via email reset | Unfamiliar to most users; key management/loss recovery is hard (no "forgot password" equivalent) |
| Cross-server trust | Each server independently manages passwords; a compromised server cannot forge *other* servers' users' identities via passwords alone anyway | A signature is independently, cryptographically verifiable by any receiving server — ideal for the decentralised, federated setting described in the question |
| Revocation | Change password instantly | Revoking/rotating a compromised key requires a distribution mechanism (e.g. a revocation list) |
| Key/credential loss | Password reset via secondary channel (email) is simple | Losing the private key with no backup can mean permanent loss of account control |

**Exam Tip: Structuring the Answer**

For the 2025 Q5(b) style question, structure the 14-mark answer in three clearly labelled parts: (1) **how login works** with a challenge-response signature scheme; (2) **how cross-server post authentication works** via signed posts and public-key distribution; (3) **pros and cons vs. passwords** for each of the two sub-cases separately (login vs. federation), since the question explicitly asks you to consider both. A balanced conclusion (e.g. "a hybrid: public-key signing for inter-server post authenticity, passwords with 2FA for human login usability") scores well.

# 8 2025 Tripos Paper 2, Q5 — Full Model Answer

**The Question**

You have joined the team developing a new, rapidly growing social media service. The system is decentralised: anybody can run their own server and interact with users on other servers, e.g. by seeing posts made by users on other servers.

(a) Explain some possible strategies for testing the server software for this social network. **[6 marks]**

(b) Your server currently authenticates users via a password, but your team is discussing whether to introduce public key authentication. Discuss how this might work, and its pros and cons compared to passwords. Consider both how a user logs in to their home server, and also how posts on one server are authenticated by another server. **[14 marks]**

## 8.1 Part (a): Testing Strategies [6 marks]

- **Unit tests** on core logic (post storage, ranking, validation) in isolation, with the network/federation layer mocked or stubbed — fast feedback, per the Testing Pyramid base.
- **Integration tests** against a real database and a second, locally-hosted server instance to exercise real federation calls — catches boundary bugs of the kind seen in the Mars Climate Orbiter case (isolated components can each "pass" while disagreeing on a shared format).
- **Protocol conformance / interoperability testing:** since other servers are independently implemented and untrusted, test your server against the *published federation protocol specification* directly, not just against your own other components.
- **Fuzzing** of message parsers that accept federated input from arbitrary remote servers, since malformed or hostile payloads are a realistic threat (Denial of Service / Tampering).
- **Load / scale testing,** since the service is described as "rapidly growing" — test under anomalous, bursty load, not just the happy path (cf. the London Ambulance Service collapse under real-world load).
- **End-to-end / manual exploratory testing** for the (few) critical user journeys (posting, following a remote user), plus **security-focused testing** (e.g. attempting to forge a post from another user) given the federated trust model.

**Mark scheme guidance:** 6 marks likely reward roughly one mark per distinct, well-justified strategy (not simply named, but explained *why* it matters for a decentralised system specifically) — aim for 4–6 concise, justified strategies rather than one strategy explained at length.

## 8.2 Part (b): Password vs. Public-Key Authentication [14 marks]

**1. How password authentication currently works (baseline, briefly):** the user submits credentials over TLS to their home server, which compares a salted hash (e.g. bcrypt) and issues a session token.

**2. How public-key login to the home server could work:**

- The user generates a key pair locally; the **public key** is registered with the home server (analogous to SSH key-based login).
- To log in, the server issues a random **challenge (nonce)**.
- The client signs the challenge with the **private key** (which never leaves the client device).
- The server verifies the signature against the stored public key and, on success, issues a session.
- **Crucially, no secret is ever transmitted over the network** — unlike a password.

**3. How posts are authenticated cross-server with public keys:**

- Each post is **digitally signed** by the author's private key at creation time.
- A remote server that receives the post fetches the author's **public key** (e.g. from the author's home server, similar to fetching a TLS certificate) and verifies the signature.
- This lets any receiving server **independently and cryptographically** confirm authorship, without needing to fully trust the claims of the sending server — directly relevant to the decentralised, "anybody can run their own server" trust model of the question, since a malicious or compromised server cannot forge a post attributed to a user on another server.
- By contrast, under password authentication, cross-server post authenticity has *no natural equivalent*: server B has no cryptographic way to confirm a post claimed to be from a user of server A is genuine, and must simply trust server A's word (a **Spoofing** risk in STRIDE terms).

**4. Pros and cons — Login (public key vs. password):**

- **Pro:** resistant to phishing (nothing secret is typed into a fake login page) and to server-side credential database breaches (only public keys are stored).
- **Pro:** eliminates weak/reused password risk entirely.
- **Con:** much less familiar to ordinary users; **key management** is hard — there is no simple "forgot password" equivalent, and losing the private key with no backup can mean permanently losing the account.
- **Con:** requires client-side tooling (secure key storage, e.g. in a browser extension or OS keychain) that most users don't currently have for a social network.

**5. Pros and cons — Cross-server post authentication (public key vs. password):**

- **Pro:** enables *independent verification* by any server without a trusted third party or blind trust in the sending server — essential in a decentralised system where servers are run by arbitrary, unvetted operators.
- **Pro:** directly mitigates spoofing/repudiation of federated content (STRIDE).
- **Con:** requires a public-key distribution and revocation mechanism (how does server B know server A's claimed key for a user is genuine, and how is a compromised key revoked?) — added infrastructure complexity.
- **Con:** signature verification and key lookups add computational and latency overhead at scale ("rapidly growing" service), versus a password check that only matters at login time.

**6. Balanced conclusion:** a strong answer proposes a **hybrid**: keep password (+ 2FA) login for home-server usability, since login is a human-facing, low-frequency operation where usability dominates; but adopt **public-key signing for every post**, since inter-server content authentication is a machine-to-machine, high-frequency operation where cryptographic verifiability (not human usability) is what matters, and is the only mechanism that scales trust across a decentralised network of independently-operated servers.

**Mark scheme guidance:** 14 marks likely split roughly as: ∼4 marks for correctly describing the challenge-response login mechanism; ∼4 marks for correctly describing signed-post cross-server authentication; ∼4 marks for balanced, specific pros/cons (not generic); ∼2 marks for a coherent, justified conclusion/recommendation. Marks are lost by only discussing login *or* only discussing post authentication — the question explicitly asks for both.

# 9 Example Sheet 1: Mindset, Requirements & Process — Model Answers

**Note on Example Sheets**

Example Sheet questions are not past Tripos questions, but are excellent practice in the same style. Answers below are structured to hit the likely marking points concisely — expand with your own words and examples in a real answer.

### 9.0 Q1. The Complexity of Scale

**(a) Why Software Engineering, not just Programming, for the health system:** the health system involves a large team (communication overhead, Brooks's Law), a huge, long-lived codebase (maintainability over decades), and safety/legal stakes (cf. Therac-25, Horizon) — success requires process, testing, and traceability, not just working code. The To-Do app is small, solo, short-lived, and low-stakes: "does it work today?" suffices.

**(b) Three emergent behaviours:**

- *Positive:* cross-department analytics emerge from combining previously siloed patient records — no single module was designed to do this.
- *Negative:* a race condition between two independently-correct modules (cf. Therac-25) causes an unsafe combined state.
- *Negative:* systemic overload when many components each behave reasonably in isolation but together exceed capacity (cf. London Ambulance Service).

None of these are possible in a solo app with one component and one user.

**(c) Discrete state and risk:** a physical hospital building degrades continuously and visibly (cracks, wear) and can be inspected; software state is discrete, so a single flipped bit/bad branch can instantly flip the system from "perfectly safe" to "catastrophic failure" with zero warning, and exhaustive state-space testing is mathematically impossible — so the risk profile is qualitatively different, not just quantitatively larger.

### 9.0 Q2. Requirements Engineering: The Autonomous Drone

**(a) Critique "ultra-fast and safe":** not testable/measurable (what is "fast"? what is "safe" — zero incidents? a specific failure rate?); not verifiable by an automated test; provides no contractual boundary for "done"; ambiguous and will be interpreted differently by engineers, forcing unconscious (likely wrong) assumptions.

**(b) Three testable NFRs:**

- *Performance:* "95% of deliveries within a 3km radius shall complete in under 15 minutes."
- *Safety:* "The drone shall maintain a minimum 5-metre separation from any detected person or building, with an automatic abort-and-land manoeuvre on violation."
- *Reliability:* "The drone shall complete 99.9% of flights without a critical navigation fault, measured over a rolling 30-day window."

**(c) Trade-offs:** faster delivery may require riskier flight paths (lower safety margins) or lighter batteries (lower reliability/range); higher safety margins reduce achievable speed. Justify to a stakeholder with a **quantified trade-off**, e.g. "increasing minimum separation from 5m to 8m would reduce the incident rate by an estimated X% but increase average delivery time by Y%" — ground the negotiation in measurable NFR fit criteria, not opinion.

### 9.0 Q3. Brooks's Law & Project Management

**(a) Immediate effect:** progress will likely *worsen* in the short term. New developers require ramp-up time and mentoring from the existing team (training overhead pulls senior engineers off delivery), and communication paths roughly double from $\binom{5}{2} = 10$ to $\binom{10}{2} = 45$, increasing coordination overhead. "Adding manpower to a late software project makes it later."

**(b) Two hidden costs:** (1) *Onboarding cost* — existing developers lose productive time mentoring and answering questions instead of coding; (2) *Integration/communication cost* — more people means more merge conflicts, more meetings, and a higher chance of duplicated or conflicting work on an already partially-built, interdependent codebase.

**(c) Alternative strategy:** descope the project (negotiate a reduced feature set for the deadline, per MoSCoW "Won't have"), extend the deadline, or bring in a very small number (1–2) of senior engineers already familiar with similar systems rather than many junior hires — minimising onboarding cost while adding real capacity.

### 9.0 Q4. Technical Debt & The Legacy Trap

**(a) Technical Debt:** the trade-off of writing quick, messy code to hit a deadline, versus the "principal" cost of eventually rewriting it properly. Not always bad — *deliberate/prudent* debt (a conscious, informed strategic choice, e.g. to hit a launch date) can be a valid business decision, as long as it is tracked and repaid.
**(b) Interest:** the tangled code means every new change (like a simple button) requires understanding and safely modifying deeply coupled, undocumented logic — the "interest payment" on the original shortcut. What should have taken an hour now takes 4 weeks because the original messy structure actively resists change.
**(c) Strategy:** refactor incrementally alongside feature work rather than a Big Bang rewrite (cf. Netscape) — e.g. apply the **Strangler Fig Pattern**, extracting and cleaning one module at a time behind a facade while continuing to ship features; write **characterization tests** first to safely refactor code with no existing test suite.

### 9.0 Q5. Process Selection & The Cost of Change

**(a) Process choice:** (A) Flight control software → **Waterfall** (or a heavily gated, spiral/risk-driven process): requirements are stable and well-understood (physics of flight), the cost of a late-discovered defect is catastrophic (loss of life, cf. 737 MAX), and extensive upfront verification/certification is mandated. (B) Lecture-notes sharing app → **Agile**: requirements are unstable and best discovered by shipping to real students and iterating, and the cost of a bug is low (inconvenience, not danger).
**(b) Cost of Change curve:** for the jet, the curve rises steeply (Boehm's curve: $1\times$ design → $100\times$ + production) because a defect found after certification/deployment may require a full re-certification, physical recall, or grounding a fleet — astronomically expensive and safety-critical. For the mobile app, the curve stays relatively flat: a bug found in production can usually be patched and redeployed within hours via CI/CD, at low cost.
**(c) Sprint and the Translation Gap:** a 2-week Sprint produces a working Increment shown to real student users every fortnight, directly closing the gap between the vague "Why" (students want to share notes easily) and the precise "What" being built — feedback from each Sprint Review continuously re-calibrates the backlog, rather than betting everything on one upfront specification (as in Waterfall) that risks the Translation Gap widening unnoticed for years.

### 9.0 Q6. Requirements Elicitation: The Hospital Ward

**(a) Why ethnography might reject/modify the PIN requirement:** an interview captures what the Head of Nursing *believes* should happen; direct observation on the ward would likely reveal that entering a 10-digit PIN for every single vitals entry is impractical in a fast-paced, hands-busy clinical environment, and nurses would develop dangerous workarounds (writing PINs on stickers, sharing logged-in devices) — exactly the invisible-workaround problem seen with the nurse's sticky note in Lecture 2. Observation surfaces the real usability constraint that an interview alone would miss.
**(b) User Requirement vs. System Specification:** a **User Requirement** is a high-level, business-facing statement of need in the client's language ("nurses must be able to record vitals quickly and securely"); a **System Specification** is the precise, technical, testable elaboration of how the system will satisfy it ("the system shall accept a 4-digit PIN authenticated within 2 seconds, or a badge-tap NFC login").
**(c) FR/NFR conflict:** a security NFR (data privacy, e.g. requiring strong authentication per entry) directly conflicts with a usability NFR for emergency response time (authentication delay could cost seconds in a life-threatening situation) — this is the classic Security-vs-Usability NFR trade-off from Lecture 2, resolved by, e.g., badge-tap authentication or a time-limited "emergency override" mode with full audit logging (balancing both concerns rather than sacrificing either).

### 9.0 Q7. Scrum Ceremonies & Failure Modes

**(a) Violated principles:** a 2-hour Daily Standup violates the **time-boxed**, 15-minute synchronisation purpose of the Daily Scrum (it has become a status meeting, not a sync); "no working software at Sprint end" violates the core Scrum requirement that a Sprint produce a **Done, potentially releasable Increment** and violates the **Transparency** pillar (the Sprint Review has nothing genuine to inspect).
**(b) Sprint Retrospective:** its purpose is continuous process improvement — the team asks what went well, what went wrong, and how to improve. It could directly fix the standup problem: the team could agree a concrete action ("standups are capped at 15 minutes; detailed technical discussion moves to a separate follow-up meeting with only the people involved") and review whether it worked at the next Retrospective.
**(c) Scrum Master vs. traditional Project Manager:** the Scrum Master is a **servant-leader** who facilitates events and removes blockers but does *not* assign tasks or dictate technical decisions; here they would coach the team to self-organise a fix for the standup, not personally mandate a solution. A traditional PM would more likely directly command a fix top-down and might also be evaluated on the same broken deliverables, creating less incentive to flag the dysfunction transparently.

### 9.0 Q8. The Agile Manifesto in Conflict

**(a) Conflicting value:** "Customer collaboration over contract negotiation" (and, relatedly, "Responding to change over following a plan") directly conflicts with a 500-page fixed-price, fully-specified contract, which is the Waterfall/legal mindset of freezing requirements upfront.
**(b) Risk discussion:** following the fixed contract risks delivering software that no longer matches the government's real needs 2 years later (business/political requirements evolve; cf. the Translation Gap and Waterfall's fatal flaw), potentially wasting enormous public funds on an obsolete system. Responding to change risks scope creep, unpredictable cost, and difficulty enforcing accountability without a fixed contractual baseline — a genuine tension in public-sector procurement.
**(c) Prototyping to satisfy both:** use **Spiral-model-style prototyping** within an overall fixed-price wrapper — e.g. contract for a series of fixed, short, prototype-validated phases with formal sign-off gates (satisfying legal certainty and traceability) while using each prototype and stakeholder demo to genuinely adapt the next phase's detailed scope (satisfying Agile flexibility). This mirrors how the Spiral Model uses risk-driven prototyping loops instead of one irreversible upfront commitment.

# 10 Example Sheet 2: Quality, Evolution & AI — Model Answers

## 10.0 Q1. VCS and Conflict Resolution

**(a)** No conflict: Alice renamed a method and Bob added a new, different method to the same file. Git merges line-by-line; since their changes touch different lines, Git auto-merges cleanly (though the *codebase* may now be broken — other callers of `get_name()` won't compile/run — which Git's textual merge cannot detect; this is a **semantic** conflict, not a **textual** one).
**(b)** If Bob also modified logic *inside* `get_name()` while Alice renamed it, both branches touch the same lines: a **Distributed VCS (Git)** would flag a textual **merge conflict** for a human to resolve locally, offline, using the full history on either side. A **Centralized VCS** handles this similarly in principle (conflicting check-ins are flagged) but typically requires a network round-trip to the central server to detect and resolve, and offers less local tooling/history to reason about the conflict.
**(c)** **Branching strategy for 30 developers: Trunk-Based Development** with short-lived feature branches ($<$ 1 day) merged frequently via small Pull Requests, backed by a strong CI pipeline that runs the full test suite on every merge — this avoids the "Merge Hell" of long-lived Git-Flow-style branches accumulating divergence over a 2-week cycle, and surfaces conflicts while they are still small and easy to resolve.

## 10.0 Q2. Quality Assurance: Mutation Testing

**(a)** High coverage only proves lines were *executed*, not that the tests *verify correct behaviour* (cf. the `average([])` coverage trap). **Mutation testing** injects small artificial bugs (mutants) and checks whether the existing suite fails; if many mutants **survive** (tests still pass despite injected bugs), it proves the assertions are too weak or missing for the code paths being "covered", explaining why bugs slip through despite 95% coverage.
**(b)** A **Killed** mutant is one where the test suite failed (caught the injected bug) — good. A **Survived** mutant is one where all tests still passed despite the injected bug — bad, it means that logic path has no effective assertion. A high survival rate indicates the test suite executes code without actually checking its correctness (weak or missing assertions), i.e. coverage without verification.
**(c)** Mutation testing is computationally expensive: for every mutant, the *entire relevant test suite* must be re-run, and a large codebase can generate thousands of mutants per file — multiplying test-suite runtime by orders of magnitude, making it impractical to run on every CI build for millions of lines of code (it's typically run periodically or on a sampled subset instead).

## 10.0 Q3. Testing in the Real World: Isolation

**(a)** Real Stripe calls in unit tests are slow (network round-trip), costly (real transaction fees / rate limits), non-deterministic (network failures, sandbox downtime), and could have real-world side effects (unintended charges) — violating the core unit-test requirement of being fast, isolated, and repeatable.
**(b)** A **Stub** returns a hardcoded canned value regardless of input (e.g. `charge()` always returns `True`), used to let the test run without a real dependency. A **Mock** additionally *records and verifies interactions* (e.g. asserting `charge` was called exactly once with amount `100`), letting the test check the code under test used the dependency correctly, not just that it got a plausible return value.
**(c)** Discrete state makes it *easier* because "Success" and "Insufficient Funds" are exact, enumerable states you can stub deterministically and assert against precisely (unlike a continuous physical system). It makes it *more dangerous* because a single incorrect branch/flag can silently flip a charge from "Insufficient Funds" to "Success" (or vice-versa) with no gradual warning sign, and exhaustively testing every discrete combination of account/payment states is not feasible, so edge-case combinations are easily missed.

### 10.0 Q4. Reliable Delivery: Deployment Strategies

**(a)** **Blue-Green** runs two full identical environments and switches *all* traffic at once (instant rollback by switching back); **Canary** exposes the new version to a small traffic percentage first and expands gradually while monitoring. For a **high-risk database migration**, **Canary** is more appropriate: it limits the blast radius of an unforeseen data-corruption or performance issue to a small fraction of real users/traffic before committing everyone, whereas Blue-Green's full traffic switch exposes 100% of users immediately to any migration defect.
**(b)** Feature Flags decouple **deployment** (code reaching production servers) from **release** (the feature being visible/active). This enables Progressive Delivery: the same deployed build can be turned on for developers, then beta users, then everyone, incrementally, and instantly disabled ("emergency shutoff") without a code rollback if a problem is found — exactly the graduated exposure model Canary uses for infrastructure, applied at the feature level.
**(c)** The **Error Budget** is 100% − SLO — the amount of unreliability the team is allowed before breaching its target. If the budget is exhausted, the team must **freeze all new risky feature deployments** and redirect all effort to stability/reliability work until the error rate recovers back within budget — deployment velocity is directly throttled by reliability performance.

### 10.0 Q5. The Maintenance Reality

**(a)** Lehman's Law of **Continuing Change** states a system must be continually adapted or it becomes progressively less satisfactory; freezing the code would leave it unable to adapt to new regulations, security threats, changing tax/compliance requirements, and new hardware/OS environments (**Adaptive** maintenance), causing it to become obsolete and eventually unusable/insecure even though "nothing changed" in the code itself (Bit Rot).
**(b)** (i) fixing a rounding error → **Corrective**; (ii) adding a new government tax API → **Adaptive**; (iii) rewriting a data-fetching loop to be more efficient → **Perfective**; (iv) migrating on-premise mainframe to the cloud → **Adaptive** (arguably also has Perfective elements, but the primary driver — responding to a changed deployment environment — is Adaptive).
**(c)** The 80/20 rule states most lifetime cost is maintenance, not initial development; a high-quality automated test suite pays for itself over 20 years by making every future Corrective/Adaptive/Perfective change *safe and fast* (enabling confident refactoring, cf. Lecture 1/5) rather than risky and slow — the upfront cost is small relative to the compounding "interest" saved on every one of the thousands of future changes a 20-year-old system will undergo.

### 10.0 Q6. SRE and Chaos Engineering

**(a)** Chaos Monkey philosophy: deliberately and randomly kill production servers (during business hours, when engineers are watching) to force systems and engineers to build in redundancy and automatic recovery *before* a real, unplanned failure occurs at a worse time. It shifts focus from maximising MTBF (preventing all failure) to minimising MTTR (surviving failure gracefully) — "if your server might die at 2pm on a Tuesday, make sure it doesn't matter."
**(b)** The Four Golden Signals: **Latency, Traffic, Errors, Saturation**. Saturation is harder to measure than Latency because latency is a direct, unambiguous per-request timing measurement, whereas saturation requires knowing a resource's true maximum capacity (which is often not fixed — it varies with workload mix, caching state, and hardware) in order to express "how full" the system is as a meaningful percentage.
**(c)** **Monitoring** addresses **known unknowns** — pre-defined dashboards/alerts telling you *that* a known metric has crossed a threshold ("CPU is at 99%"). **Observability** addresses **unknown unknowns** — the ability to ask arbitrary new questions of a system's logs/metrics/traces after the fact to understand *why* something unexpected happened, without having predicted that specific failure mode in advance.

### 10.0 Q7. Legacy Evolution: The Strangler Fig

**(a)** A Big Bang rewrite is high-risk because it halts new feature delivery for an extended, uncertain period (cf. Netscape's 3-year rewrite that let Internet Explorer capture 90% market share); it also risks silently dropping "undocumented" behaviour that real users depend on (Hyrum's Law) since the new system is built from a fresh, incomplete understanding of the old one, and the "all-or-nothing" cutover concentrates all migration risk into a single high-stakes event.
**(b) Strangler Fig steps:** (1) place a proxy/facade in front of the legacy COBOL system; (2) build new functionality as a separate, modern microservice; (3) route specific calls/traffic to the new service while the rest continues to hit the legacy mainframe; (4) incrementally repeat, migrating more functionality until the legacy system receives no traffic and can be safely decommissioned.
**(c)** Hyrum's Law states that with enough users, every observable behaviour of a system — including undocumented quirks and bugs — becomes something somebody depends on. For a 30-year-old COBOL system, this means the migration cannot rely on the *documented* spec alone: even bugs and side effects (e.g. specific rounding behaviour, edge-case output ordering) may be silently relied upon by downstream consumers, so the new system must be validated against real historical behaviour (e.g. via characterization/golden-master tests) rather than a clean-room reimplementation of the "intended" spec.

### 10.0 Q8. Software Archaeology and API Design

**(a) Three steps of Software Archaeology:** (1) use `git blame`/`git log` to see who changed each line and when, building a change history; (2) use `git log -S` to search for when a specific function/string was introduced or removed, to recover intent; (3) trace linked issue-tracker IDs in commit messages (e.g. `FIX-123`) back to the original business/bug context, and write **characterization tests** to lock in current behaviour before touching anything.
**(b)** Adding a required parameter (`process(data)` → `process(data, options)`) is a **breaking, structural change** under SemVer (existing callers using the old single-argument signature will no longer compile/run), so the new version should be **3.0.0** (a MAJOR bump from 2.4.1), *unless* `options` is given a default value making the call backward-compatible, in which case it would only be a MINOR bump to `2.5.0`. State the assumption explicitly: as written (a new required parameter), it is MAJOR.
**(c)** An API is a **contract**: it promises that given input $X$, the system returns output $Y$, and every consumer builds on that promise. When the contract is broken (a breaking change without warning), every downstream consumer's code may stop compiling or behaving correctly, breaking their CI/CD pipelines and production systems — multiplying the cost of one change across the entire ecosystem of dependents (cf. Hyrum's Law and the Left-Pad incident).

### 10.0 Q9. API Evolution & Deprecation

**(a)** Turning off the XML endpoint tomorrow is an uncommunicated **breaking change**: existing customer integrations built against XML would fail instantly with no warning, breaking their production systems and CI/CD pipelines (Hyrum's Law — however officially "deprecated" XML may be, real customers depend on it continuing to work until they choose to migrate).
**(b) 3-step Deprecation Path:** (1) **Announce** the new JSON endpoint and the planned XML removal date, running both formats in parallel (a MINOR version bump, backward-compatible since nothing existing breaks yet); (2) **Deprecate** the XML endpoint with clear warnings (e.g. a `Deprecation` HTTP header) while both still function, giving customers 6–12 months to migrate; (3) **Remove** the XML endpoint in a new **MAJOR** version release, once usage has dropped and the announced window has passed.
**(c)** A breaking change (e.g. abrupt removal of the XML endpoint) will cause any customer CI/CD pipeline whose integration/contract tests call the old format to start **failing the build** the moment the change ships, per the "fail any step → reject build" principle of a CI/CD pipeline — effectively propagating your internal API change into an involuntary, urgent incident for every customer's own release process.

### 10.0 Q10. AI-Augmented Engineering

**(a)** The AI likely produced an insecure library version because its training data reflects historically common code (including outdated tutorials/StackOverflow snippets using an older, vulnerable package version) — it pattern-matches to statistically frequent solutions, not necessarily the most current or secure one (an AI hallucination/bias risk). **Responsibility for the bug lies with the developer who committed it**: "you are the owner of every line you commit" — code review, dependency scanning (e.g. Dependabot/Snyk), and understanding what was shipped are the developer's responsibility, not the AI's.
**(b)** A 2010 junior engineer spent significant time on manual boilerplate, syntax lookup, and low-level implementation, gradually building system-level understanding through repetition. A 2026 junior engineer instead **orchestrates AI** to generate that boilerplate rapidly, so the required cognitive skillset shifts from *writing* syntax to *critically reviewing, verifying, and directing* AI output — requiring earlier, faster development of architectural judgement and domain knowledge, precisely the skills the "training ladder" used to build gradually over years of manual practice.
**(c)** The **Reasoning Ceiling**: LLMs excel at tasks with a clear feedback loop (does it compile? do tests pass?) and pattern-matching against vast training data, but lack long-horizon strategic judgement across ambiguous, conflicting stakeholder priorities, accountable ownership of consequences, and the tacit organisational/domain context a Senior Architect accumulates over years. An AI can accelerate execution but cannot currently replace the judgement, accountability, and cross-domain synthesis a Senior Architect provides.

### 10.0 Q11. The Human in the AI Loop

**(a) Context Window:** the finite amount of text (code, files, conversation) an LLM can consider at once. If the database migration scripts lived outside the portion of the codebase included in the AI's context for this task (e.g. in a separate, unreferenced directory), the model had no way to know they needed updating too — it correctly updated everything *within* its context but had no visibility of the dependency, illustrating that context window limits are a real architectural constraint, not just a minor inconvenience.
**(b) AI Review Checklist (Senior Engineer, before merge):**

- Does the change compile/pass the full test suite, including integration tests that span the files the AI touched?
- Were *all* related artefacts updated (migrations, config, docs, IaC) — explicitly search for cross-cutting concerns the diff might have missed?
- Does the reviewer *understand* every line well enough to maintain it ("you own every line you commit")?
- Are there hallucinated dependencies/APIs, or newly introduced libraries with known CVEs?
- Does the change match existing architectural conventions, or does it introduce a "clever hack" that passes tests but violates long-term design goals?

**(c)** Software Archaeology remains essential because summarising *what* a file currently does is not the same as understanding *why* it is the way it is — the historical intent, prior incidents, and business context behind a design decision (captured in `git blame`, linked issue trackers, and tribal knowledge) are not recoverable from the file's current text alone, however fast an AI can read it; this context is exactly what determines whether a proposed AI change is safe.

# 11 Quick Reference: Definitions & Frameworks

## 11.0 Lecture 1: Mindset & Failure

**Core Definitions**

**Brooks's Law:** adding manpower to a late project makes it later; communication paths $= \frac{n(n-1)}{2}$.
**80/20 Rule:** 20% development cost, 80% maintenance cost over the lifecycle.
**Technical Debt:** Principal (cost to fix properly) + Interest (extra cost of future changes while unfixed); quadrant = Reckless/Prudent × Deliberate/Inadvertent.
**Iron Triangle:** Fast, Good, Cheap — pick two.
**Key case studies:** Therac-25 (race condition, no hardware interlock), Mars Climate Orbiter (unit mismatch, no integration testing), London Ambulance Service (socio-technical, no phased rollout), Post Office Horizon (no audit trail, ignored engineers), Boeing 737 MAX (single point of failure, hidden complexity).

## 11.0 Lecture 2: Requirements

**Core Definitions**

**FR:** an action the system performs (verb). **NFR:** a quality constraint (adjective/adverb); the "-ilities".
**Elicitation:** Interviews (scoping, but users describe solutions) vs. Ethnography (reveals invisible workarounds).
**MoSCoW:** Must / Should / Could / Won't have.
**UML relationships:** Association (line), Inheritance (hollow triangle, "is-a"), Aggregation (hollow diamond, "has-a", independent), Composition (solid diamond, "has-a", dependent lifecycle).
**Boehm's Cost Curve:** Design 1×, Coding 5–10×, Testing 10×, Production 100 × +.

## 11.0 Lecture 3: Process

**Core Definitions**

**Waterfall phases:** Requirements → Design → Implementation & Unit Testing → Integration & System Testing → Operation & Maintenance.
**Spiral quadrants:** Objectives → Risk Assessment → Development → Planning (repeat).
**Agile Manifesto:** Individuals/interactions > processes/tools; Working software > documentation; Customer collaboration > contract negotiation; Responding to change > following a plan.
**Scrum artifacts:** Product Backlog, Sprint Backlog, Increment. **Events:** Sprint, Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective. **Roles:** Scrum Master, Product Owner, Developers.
**Kanban:** continuous flow, WIP limits, no fixed sprints.

## 11.0 Lecture 4: QA & Delivery

**Core Definitions**

**Testing Pyramid:** Unit (many, fast) → Integration (some) → E2E (few, slow).
**Stub** = hardcoded return value. **Mock** = verifies interaction occurred.
**Coverage** = % lines executed (≠ correctness). **Mutation testing** = inject bugs, check tests catch them (survived vs. killed mutants).
**TDD:** Red (failing test) → Green (minimal pass) → Refactor.
**CI** = merge + auto-build/test ≥ daily. **CDE (Delivery)** = auto-deploy to staging, human decides prod release. **CDP (Deployment)** = fully automated to production.
**Blue-Green** = instant full-traffic switch. **Canary** = gradual traffic ramp-up with monitoring.
**Error Budget** = 100% − SLO.
**Cloud:** IaaS (VMs) < PaaS (upload code) < SaaS (full app). **Containers** share host OS kernel (light); **VMs** each run a full guest OS (heavy). **Kubernetes:** self-healing, auto-scaling, load balancing across Pods/Nodes.

### 11.0 Lecture 5: Evolution

**Core Definitions**

**Maintenance split:** Perfective 50%, Adaptive 25%, Corrective 20%, Preventive 5%.
**SemVer:** MAJOR.MINOR.PATCH — breaking / new feature / bug fix. Caret ^ allows minor+patch; tilde ~ allows patch only.
**Postel's Law:** be conservative in what you send, liberal in what you accept.
**Hyrum's Law:** with enough users, every observable behaviour becomes depended upon.
**Deprecation path:** Announce → Deprecate → Wait → Remove.
**Strangler Fig:** proxy/facade → build new service → route traffic gradually → decommission legacy.
**Code smells:** Long Method, God Object, Feature Envy, Data Clumps.
**Monitoring** = known unknowns. **Observability** = unknown unknowns (Logs + Metrics + Tracing).
**Golden Signals:** Latency, Traffic, Errors, Saturation.
**SRE:** SLI (measured) vs. SLO (target); MTBF (prevent failure) vs. MTTR (recover fast); Chaos Engineering = steady state → hypothesis → experiment → verify.
**Supply chain:** transitive dependency, CVE, SBOM (cf. Left-Pad, Log4Shell).

### 11.0 Lecture 6: AI Era

**Core Definitions**

**Building WITH AI** (tool during development) vs. **Building FOR AI** (LLM is a runtime component).
**Reliability Paradox:** traditional code is deterministic ($f(x) \to y$ always); AI code is probabilistic ($f(x) \to y$ probably).
**Why coding AI advances fast:** code has ground truth (compiler/tests); no equivalent "compiler" for subjective domains (Virtual Doctor Problem).
**Generator/Verifier (Critic) architecture:** adversarial pairing reduces hallucination.
**Jevons Paradox:** efficiency gains increase total demand, not reduce total headcount need.
**Reasoning Ceiling:** AI lacks long-horizon judgement, accountability, and tacit cross-domain context.

### 11.0 Security Appendix

**Core Definitions**

**ACLs** (column-centric, per-object) vs. **Capabilities** (row-centric, per-subject).
**STRIDE:** Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege.
**One-way flow:** confidentiality flows low→high; integrity flows high→low.
**Password auth:** secret transmitted, familiar, easy recovery. **Public-key auth:** challenge-response, no secret transmitted, phishing-resistant, but harder key management/recovery; ideal for cross-server signed-content verification in decentralised systems.

# 11 Tripos Exam Strategy & Common Pitfalls

### 11.0 Paper Structure & High-Value Topics

| Aspect | Detail |
|---|---|
| Format | Paper 2; multi-part questions (a)–(c), each part building on the last |
| Question style | Scenario-based: a fictional company/system, then several sub-questions applying course concepts to it |
| High-yield topics | Case studies & failure lessons, FR/NFR, Agile/Scrum mechanics, testing (coverage/mutation/TDD), CI/CD & deployment strategies, SemVer/API evolution, SRE & observability |
| Also examinable | Security concepts (authentication design, STRIDE, access control) as shown in 2025 Q5; AI-era engineering judgement questions |

### 11.0 General Exam Technique

1. **Read all parts of a question before starting.** Later parts often hint at what earlier parts expect, and marks are frequently split roughly evenly across sub-parts.
2. **Ground every abstract principle in a named case study or framework.** "Never rely on a single sensor" is weaker than "Never rely on a single sensor (cf. Boeing 737 MAX's single Angle-of-Attack sensor)".
3. **Answer scenario questions in the scenario's own terms.** If the question describes a decentralised social network, don't just define STRIDE generically — apply each relevant threat to *that* system's specific data flows.
4. **For "pros and cons" or "discuss trade-offs" questions, use a clearly labelled two-sided structure** (a short table or explicit "Pro:"/"Con:" bullets) rather than a single blended paragraph — this is easier for an examiner to mark and for you to keep balanced.
5. **Quantify wherever the scenario permits it** (communication paths via Brooks's Law, error budgets, SemVer version bumps) — numeric, worked reasoning demonstrates precise understanding beyond prose.
6. **For "is this approach appropriate?" questions,** give both supporting and opposing arguments, then a clear, justified conclusion — don't just pick a side.
7. **Don't forget the conclusion/recommendation.** Many high-mark parts (e.g. 2025 Q5b) explicitly reward a final, justified synthesis, not just a list of pros and cons.
8. **Keep answers proportional to the marks available.** A 6-mark part wants 4–6 distinct, justified points; a 14-mark part wants full structured coverage of every sub-clause in the question (as in the worked 2025 Q5 answer above).

**Top 10 Software & Security Engineering Tripos Traps**

1. **Confusing FR and NFR:** an NFR is a *quality/constraint*, not an action — "the system shall calculate VAT" is functional even though tax calculation sounds like a "quality".
2. **Treating Agile as "no planning/no docs":** Agile teams plan *more* frequently than Waterfall teams; this is a common misconception examiners test directly.
3. **Confusing A/B Testing (business/product validation) with Canary Deployment (technical validation).**
4. **Confusing Blue-Green (instant full switch) with Canary (gradual, monitored ramp-up)** — know which is more appropriate for a high-risk change.
5. **Assuming 100% code coverage means correct code.** Always mention the coverage $\neq$ correctness trap (e.g. the empty-list division bug) if discussing testing quality.
6. **Getting SemVer bumps wrong:** a required new parameter or removed field is MAJOR, not MINOR, even if it "adds" functionality.
7. **Forgetting Hyrum's Law when discussing legacy migration or breaking changes:** undocumented behaviour is still depended upon by somebody.
8. **Conflating Monitoring (known unknowns) with Observability (unknown unknowns).**
9. **Applying Brooks's Law only qualitatively:** show the $\frac{n(n-1)}{2}$ communication-path calculation for full marks on quantitative parts.
10. **Discussing AI risk questions generically instead of naming the specific failure mode** (hallucination, technical debt of ignorance, bias, reasoning ceiling) that the scenario is actually illustrating.

## 11 Final Exam Checklist

Before submitting your answers, verify these things:

1. **Named a case study or framework** to ground every abstract engineering principle you invoked.
2. **Answered every sub-part of every question**, in proportion to its marks — did you cover both "how login works" *and* "how cross-server posts are authenticated" if both were asked (2025 Q5b), not just one?
3. **Distinguished lookalike pairs correctly:** FR/NFR; Mock/Stub; Monitoring/Observability; Blue-Green/Canary; A/B Testing/Canary; MTBF/MTTR; Backward/Forward compatibility.
4. **Used quantitative reasoning** where the scenario allows it (communication paths, SemVer version numbers, error budgets, coverage percentages).
5. **Gave a balanced pros/cons discussion** for any "discuss" or "evaluate" question, clearly labelled, followed by a justified conclusion.
6. **Applied concepts to the scenario's specific details**, not generic textbook definitions — e.g. STRIDE applied to the actual data flow described, not just listed.
7. **Checked SemVer/versioning logic**: does the described change break existing callers? If in doubt, MAJOR.
8. **For security questions,** covered confidentiality/integrity/availability implications as relevant, and distinguished access control mechanisms (ACL vs. capability) if asked.

**Final Advice**

This paper rewards **concrete, well-grounded reasoning** over generic definitions. The case studies (Therac-25, Mars Climate Orbiter, London Ambulance Service, Post Office Horizon, Boeing 737 MAX, Y2K, Netscape, Left-Pad, Log4Shell, SimCity/Windows 95) are not decoration — they are the evidence base examiners expect you to draw on to justify engineering principles. Learn the mechanism behind each one (not just the headline), and you will be able to construct a strong, specific answer to almost any scenario-based question on this paper.

**Good luck in your Tripos!**