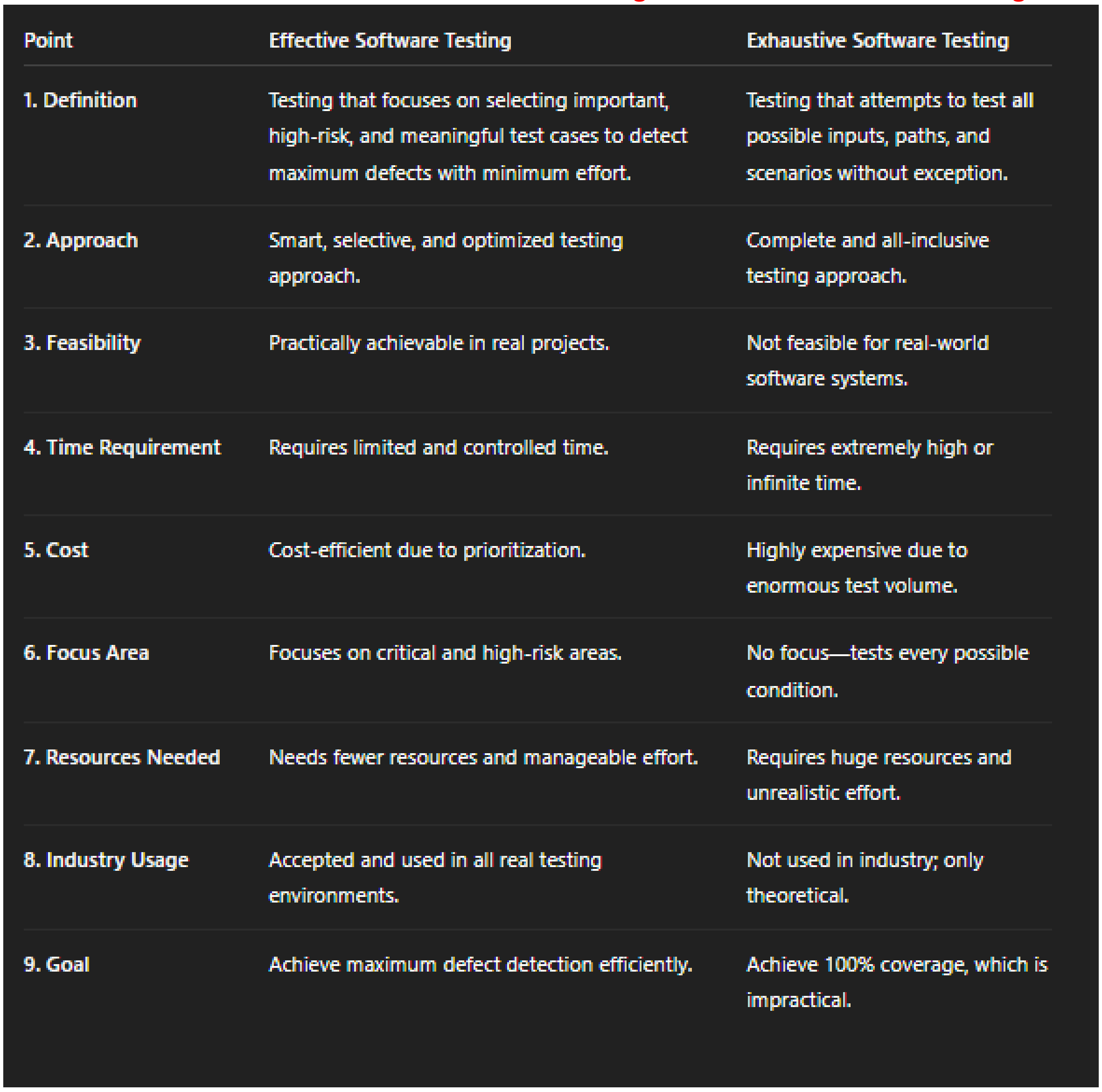
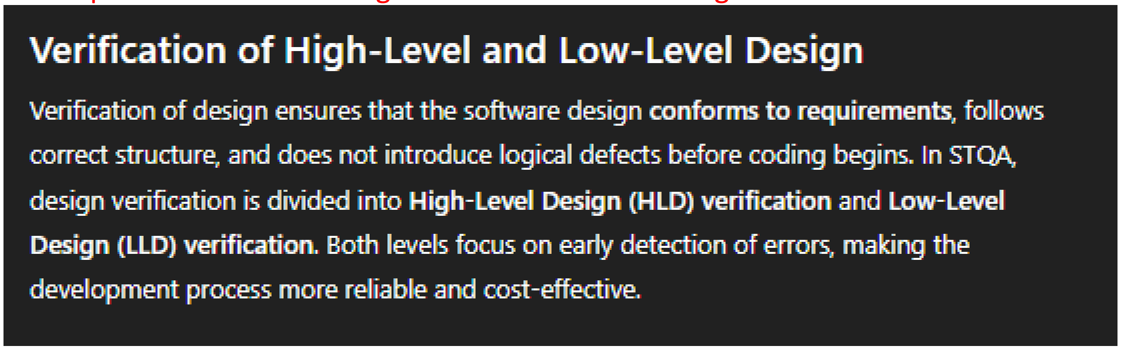
UNIT–I : Testing Methodology

1. Differentiate between Effective Software Testing vs Exhaustive Software Testing.



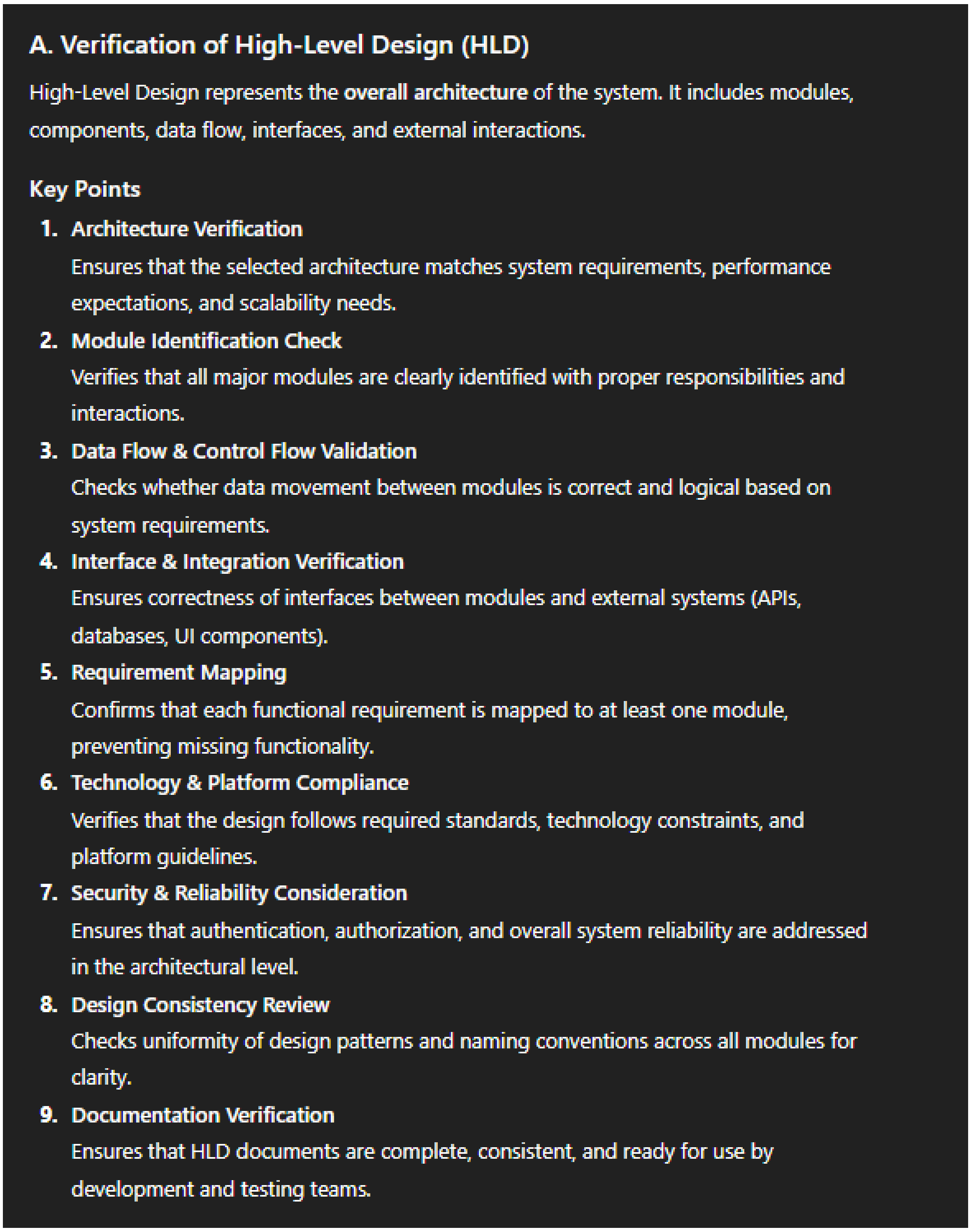
Point	Effective Software Testing	Exhaustive Software Testing
1. Definition	Testing that focuses on selecting important, high-risk, and meaningful test cases to detect maximum defects with minimum effort.	Testing that attempts to test all possible inputs, paths, and scenarios without exception.
2. Approach	Smart, selective, and optimized testing approach.	Complete and all-inclusive testing approach.
3. Feasibility	Practically achievable in real projects.	Not feasible for real-world software systems.
4. Time Requirement	Requires limited and controlled time.	Requires extremely high or infinite time.
5. Cost	Cost-efficient due to prioritization.	Highly expensive due to enormous test volume.
6. Focus Area	Focuses on critical and high-risk areas.	No focus—tests every possible condition.
7. Resources Needed	Needs fewer resources and manageable effort.	Requires huge resources and unrealistic effort.
8. Industry Usage	Accepted and used in all real testing environments.	Not used in industry; only theoretical.
9. Goal	Achieve maximum defect detection efficiently.	Achieve 100% coverage, which is impractical.

2. Explain Verification of high level and low level design.



Verification of High-Level and Low-Level Design

Verification of design ensures that the software design **conforms to requirements**, follows correct structure, and does not introduce logical defects before coding begins. In STQA, design verification is divided into **High-Level Design (HLD) verification** and **Low-Level Design (LLD) verification**. Both levels focus on early detection of errors, making the development process more reliable and cost-effective.

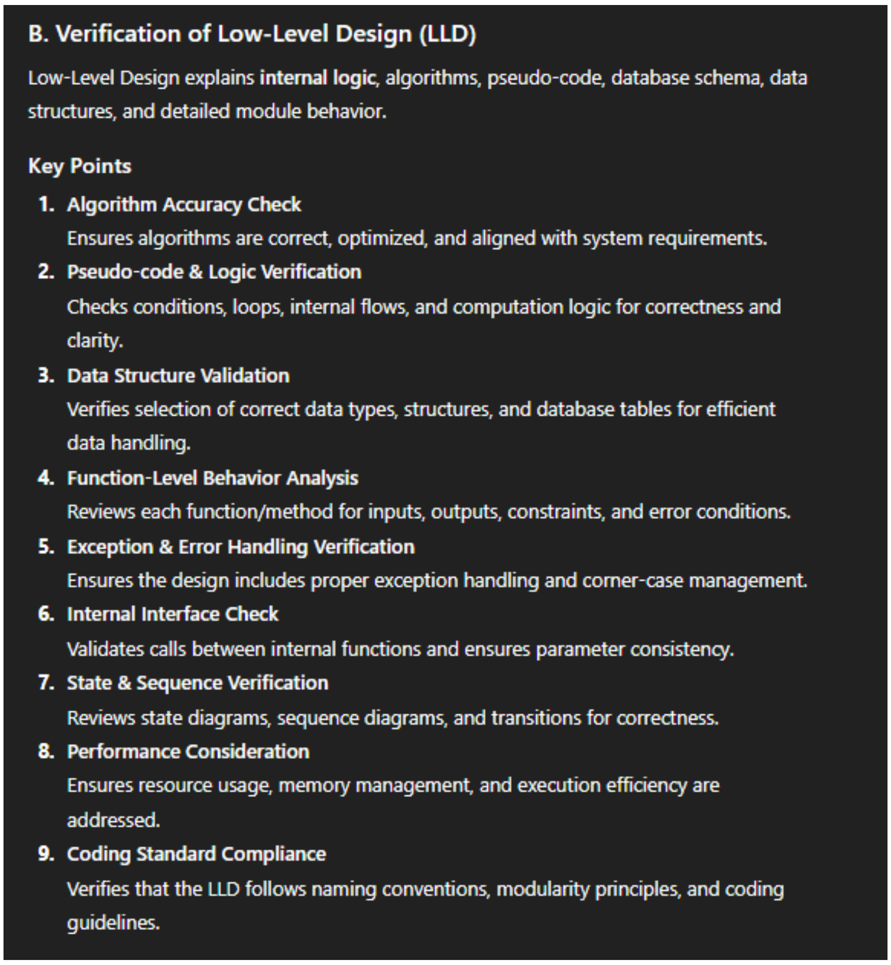


A. Verification of High-Level Design (HLD)

High-Level Design represents the **overall architecture** of the system. It includes modules, components, data flow, interfaces, and external interactions.

Key Points

1. **Architecture Verification**
   Ensures that the selected architecture matches system requirements, performance expectations, and scalability needs.
2. **Module Identification Check**
   Verifies that all major modules are clearly identified with proper responsibilities and interactions.
3. **Data Flow & Control Flow Validation**
   Checks whether data movement between modules is correct and logical based on system requirements.
4. **Interface & Integration Verification**
   Ensures correctness of interfaces between modules and external systems (APIs, databases, UI components).
5. **Requirement Mapping**
   Confirms that each functional requirement is mapped to at least one module, preventing missing functionality.
6. **Technology & Platform Compliance**
   Verifies that the design follows required standards, technology constraints, and platform guidelines.
7. **Security & Reliability Consideration**
   Ensures that authentication, authorization, and overall system reliability are addressed in the architectural level.
8. **Design Consistency Review**
   Checks uniformity of design patterns and naming conventions across all modules for clarity.
9. **Documentation Verification**
   Ensures that HLD documents are complete, consistent, and ready for use by development and testing teams.



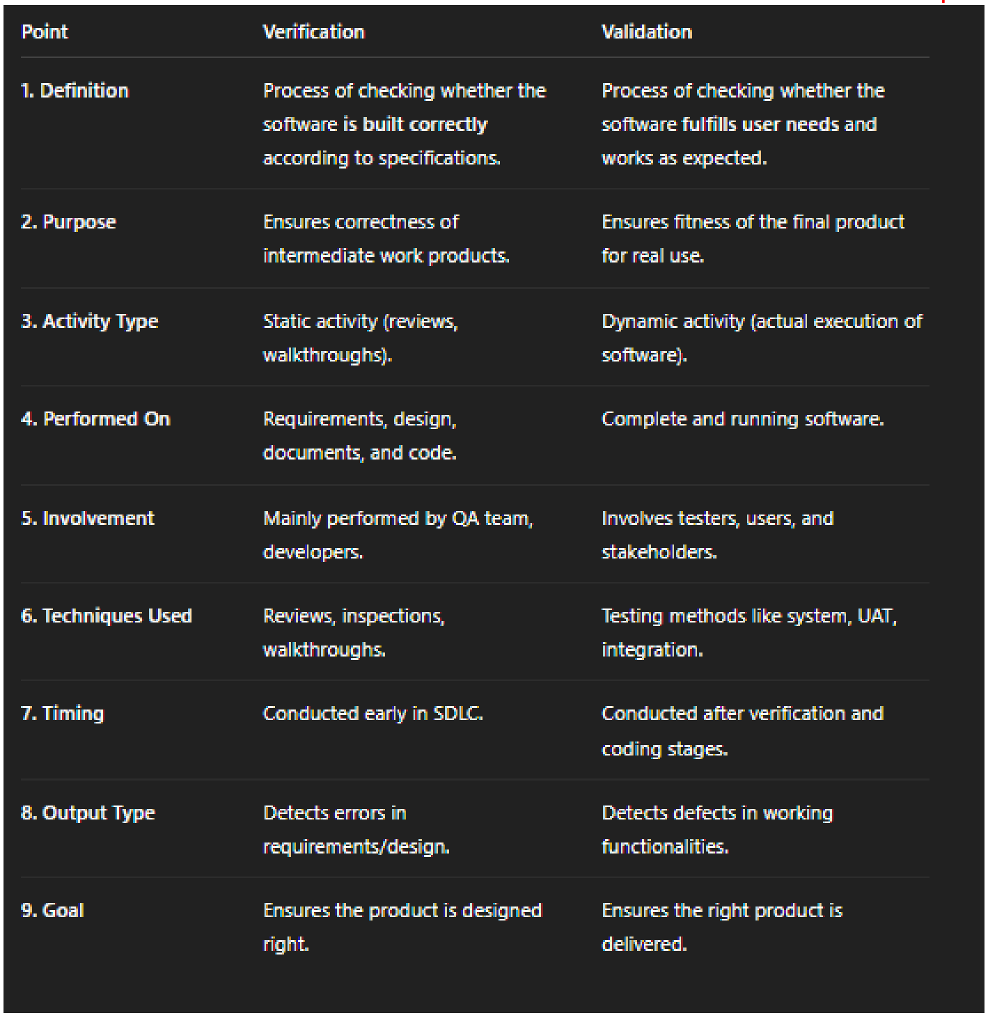
B. Verification of Low-Level Design (LLD)

Low-Level Design explains **internal logic**, algorithms, pseudo-code, database schema, data structures, and detailed module behavior.

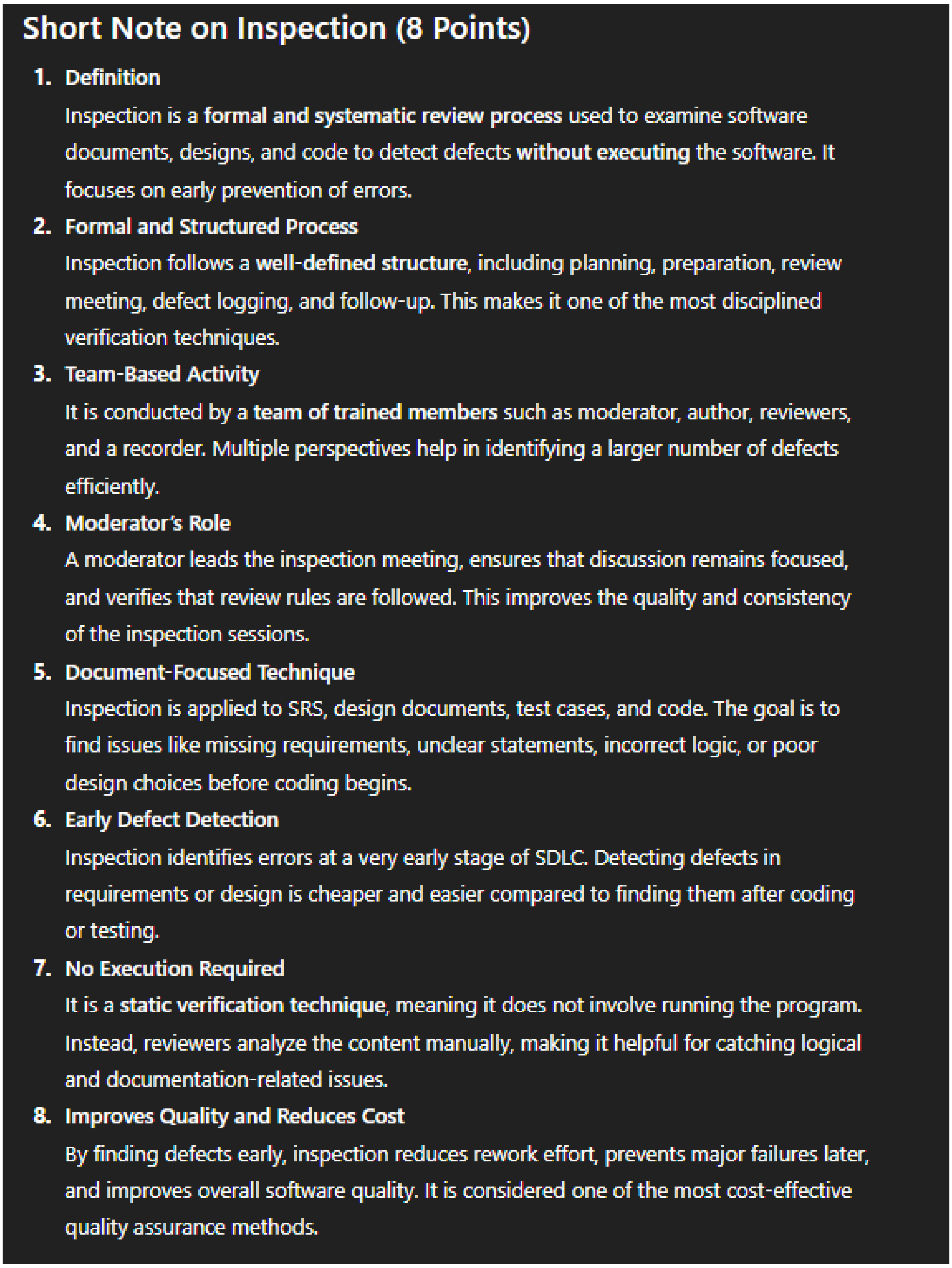
Key Points

1. **Algorithm Accuracy Check**
   Ensures algorithms are correct, optimized, and aligned with system requirements.
2. **Pseudo-code & Logic Verification**
   Checks conditions, loops, internal flows, and computation logic for correctness and clarity.
3. **Data Structure Validation**
   Verifies selection of correct data types, structures, and database tables for efficient data handling.
4. **Function-Level Behavior Analysis**
   Reviews each function/method for inputs, outputs, constraints, and error conditions.
5. **Exception & Error Handling Verification**
   Ensures the design includes proper exception handling and corner-case management.
6. **Internal Interface Check**
   Validates calls between internal functions and ensures parameter consistency.
7. **State & Sequence Verification**
   Reviews state diagrams, sequence diagrams, and transitions for correctness.
8. **Performance Consideration**
   Ensures resource usage, memory management, and execution efficiency are addressed.
9. **Coding Standard Compliance**
   Verifies that the LLD follows naming conventions, modularity principles, and coding guidelines.

3. Differentiate between Verification and Validation. Short note on Inspection.



Point	Verification	Validation
1. Definition	Process of checking whether the software **is built correctly** according to specifications.	Process of checking whether the software **fulfills user needs** and works as expected.
2. Purpose	Ensures correctness of intermediate work products.	Ensures fitness of the final product for real use.
3. Activity Type	Static activity (reviews, walkthroughs).	Dynamic activity (actual execution of software).
4. Performed On	Requirements, design, documents, and code.	Complete and running software.
5. Involvement	Mainly performed by QA team, developers.	Involves testers, users, and stakeholders.
6. Techniques Used	Reviews, inspections, walkthroughs.	Testing methods like system, UAT, integration.
7. Timing	Conducted early in SDLC.	Conducted after verification and coding stages.
8. Output Type	Detects errors in requirements/design.	Detects defects in working functionalities.
9. Goal	Ensures the product is designed right.	Ensures the right product is delivered.



Short Note on Inspection (8 Points)

1. **Definition**
   Inspection is a **formal and systematic review process** used to examine software documents, designs, and code to detect defects **without executing** the software. It focuses on early prevention of errors.
2. **Formal and Structured Process**
   Inspection follows a **well-defined structure**, including planning, preparation, review meeting, defect logging, and follow-up. This makes it one of the most disciplined verification techniques.
3. **Team-Based Activity**
   It is conducted by a **team of trained members** such as moderator, author, reviewers, and a recorder. Multiple perspectives help in identifying a larger number of defects efficiently.
4. **Moderator's Role**
   A moderator leads the inspection meeting, ensures that discussion remains focused, and verifies that review rules are followed. This improves the quality and consistency of the inspection sessions.
5. **Document-Focused Technique**
   Inspection is applied to SRS, design documents, test cases, and code. The goal is to find issues like missing requirements, unclear statements, incorrect logic, or poor design choices before coding begins.
6. **Early Defect Detection**
   Inspection identifies errors at a very early stage of SDLC. Detecting defects in requirements or design is cheaper and easier compared to finding them after coding or testing.
7. **No Execution Required**
   It is a **static verification technique**, meaning it does not involve running the program. Instead, reviewers analyze the content manually, making it helpful for catching logical and documentation-related issues.
8. **Improves Quality and Reduces Cost**
   By finding defects early, inspection reduces rework effort, prevents major failures later, and improves overall software quality. It is considered one of the most cost-effective quality assurance methods.

4. What are the different types of bugs depending upon stages of SDLC.

4. Types of Bugs Depending Upon Stages of SDLC

Bugs can occur at any phase of the Software Development Life Cycle. STQA classifies defects based on the stage in which they commonly arise. Identifying bugs early helps reduce cost, improves quality, and prevents major failures during later phases. Below are the major bug types classified according to SDLC stages.

---

1. Requirement Bugs

These bugs arise during the **Requirement Analysis phase** when requirements are unclear, incomplete, contradictory, or misunderstood. Examples include wrong assumptions, missing requirements, ambiguity in statements, and inconsistent functionality descriptions. These defects propagate into design and development if not detected early.

---

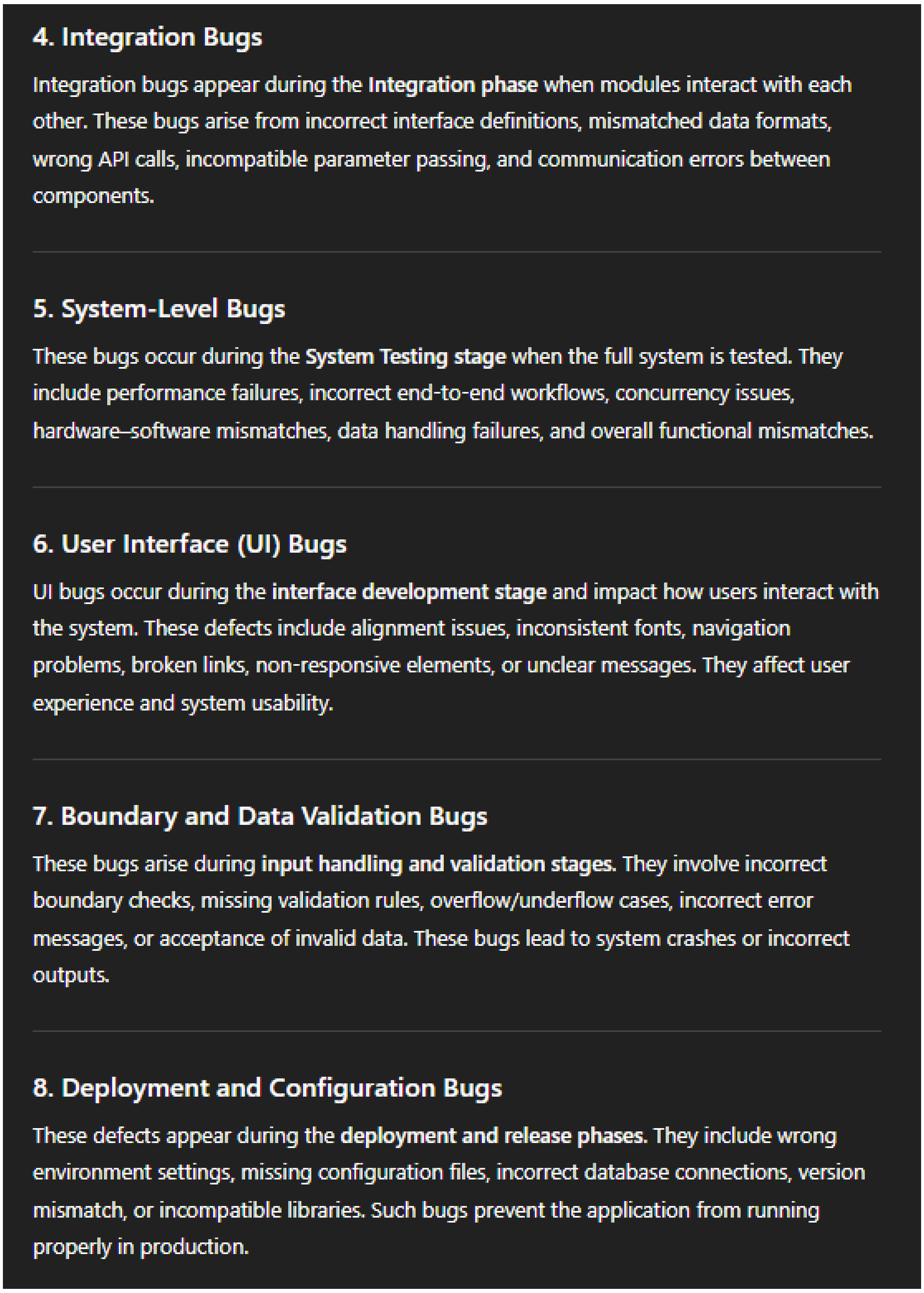
2. Design Bugs

Design bugs appear during the **High-Level Design and Low-Level Design stages.** They occur when the architecture, module division, data flow, algorithms, or interface definitions are incorrect. Such bugs lead to performance issues, integration failures, incorrect logic flow, or scalability problems during implementation.

---

3. Coding Bugs

These bugs occur during the **Implementation/Coding phase.** They include syntax errors, logical errors, incorrect loops, wrong conditions, memory mismanagement, incorrect data types, and undefined variables. Coding bugs are the most frequently encountered defects because they directly affect execution.



4. Integration Bugs

Integration bugs appear during the **Integration phase** when modules interact with each other. These bugs arise from incorrect interface definitions, mismatched data formats, wrong API calls, incompatible parameter passing, and communication errors between components.

5. System-Level Bugs

These bugs occur during the **System Testing stage** when the full system is tested. They include performance failures, incorrect end-to-end workflows, concurrency issues, hardware–software mismatches, data handling failures, and overall functional mismatches.

6. User Interface (UI) Bugs

UI bugs occur during the **interface development stage** and impact how users interact with the system. These defects include alignment issues, inconsistent fonts, navigation problems, broken links, non-responsive elements, or unclear messages. They affect user experience and system usability.

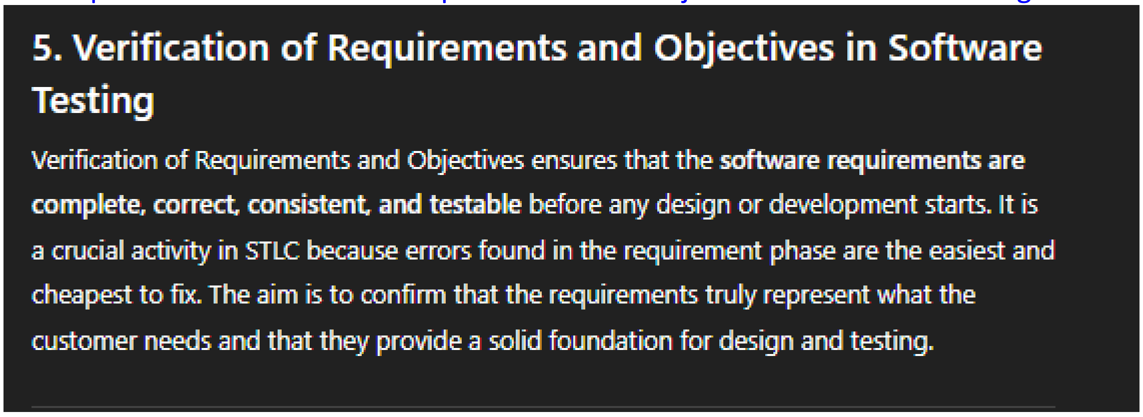
7. Boundary and Data Validation Bugs

These bugs arise during **input handling and validation stages.** They involve incorrect boundary checks, missing validation rules, overflow/underflow cases, incorrect error messages, or acceptance of invalid data. These bugs lead to system crashes or incorrect outputs.

8. Deployment and Configuration Bugs

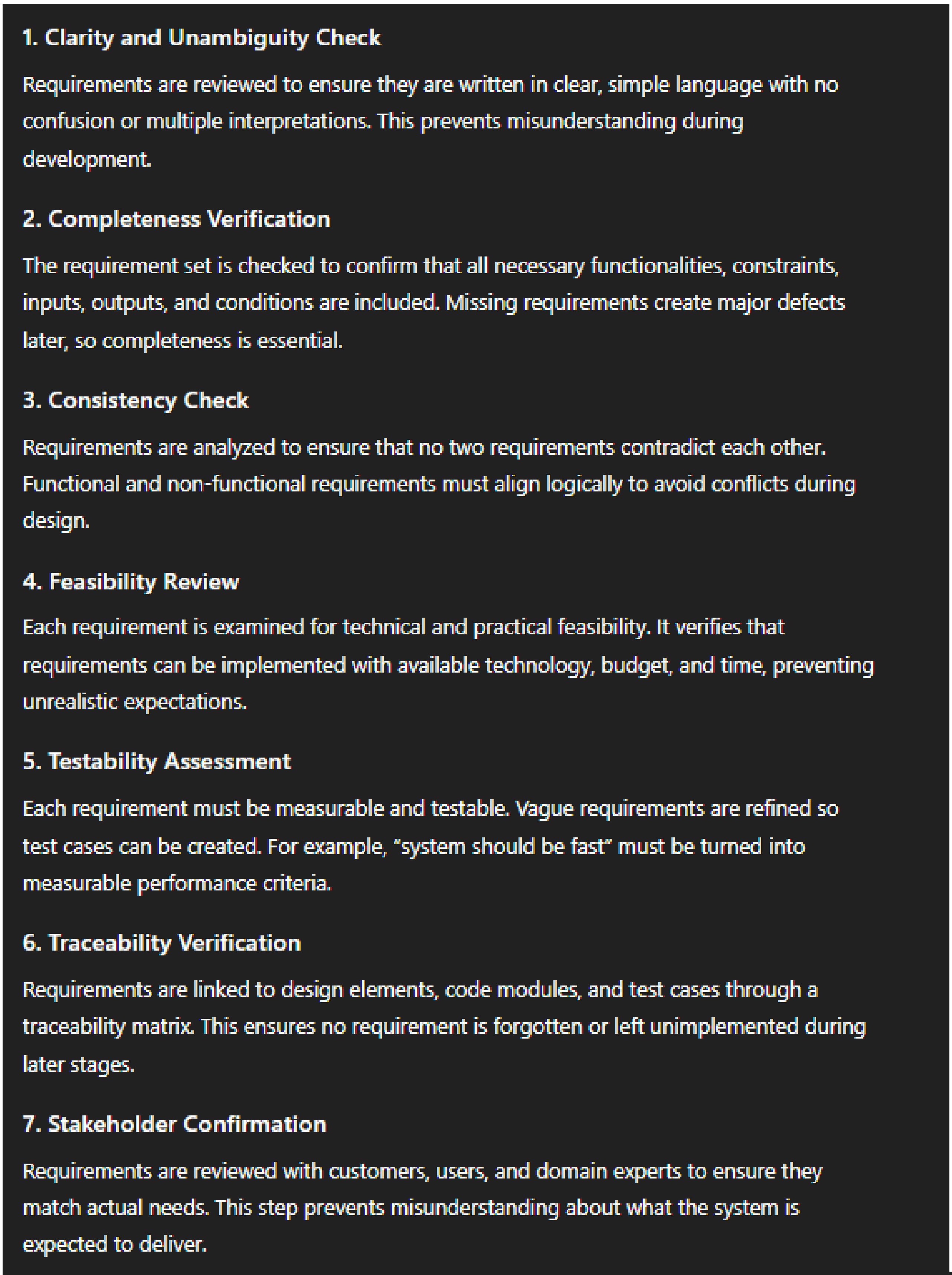
These defects appear during the **deployment and release phases.** They include wrong environment settings, missing configuration files, incorrect database connections, version mismatch, or incompatible libraries. Such bugs prevent the application from running properly in production.

5. Explain the verification of Requirements and Objectives in Software testing.



5. Verification of Requirements and Objectives in Software Testing

Verification of Requirements and Objectives ensures that the **software requirements are complete, correct, consistent, and testable** before any design or development starts. It is a crucial activity in STLC because errors found in the requirement phase are the easiest and cheapest to fix. The aim is to confirm that the requirements truly represent what the customer needs and that they provide a solid foundation for design and testing.



1. Clarity and Unambiguity Check

Requirements are reviewed to ensure they are written in clear, simple language with no confusion or multiple interpretations. This prevents misunderstanding during development.

2. Completeness Verification

The requirement set is checked to confirm that all necessary functionalities, constraints, inputs, outputs, and conditions are included. Missing requirements create major defects later, so completeness is essential.

3. Consistency Check

Requirements are analyzed to ensure that no two requirements contradict each other. Functional and non-functional requirements must align logically to avoid conflicts during design.

4. Feasibility Review

Each requirement is examined for technical and practical feasibility. It verifies that requirements can be implemented with available technology, budget, and time, preventing unrealistic expectations.

5. Testability Assessment

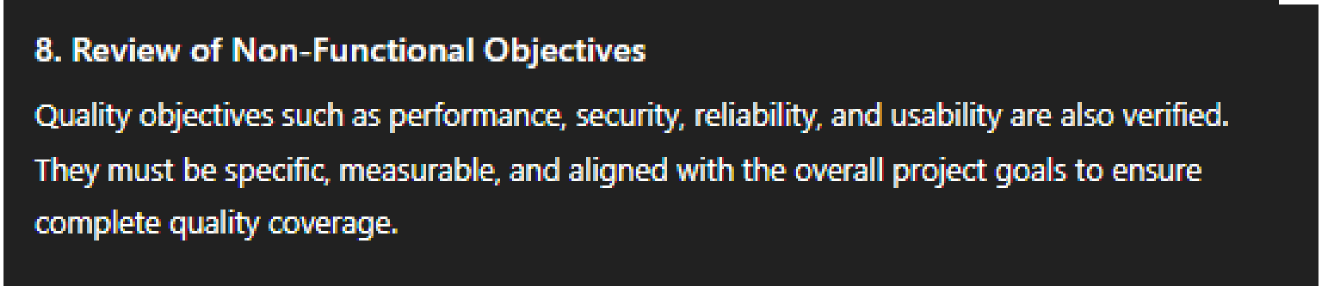
Each requirement must be measurable and testable. Vague requirements are refined so test cases can be created. For example, "system should be fast" must be turned into measurable performance criteria.

6. Traceability Verification

Requirements are linked to design elements, code modules, and test cases through a traceability matrix. This ensures no requirement is forgotten or left unimplemented during later stages.

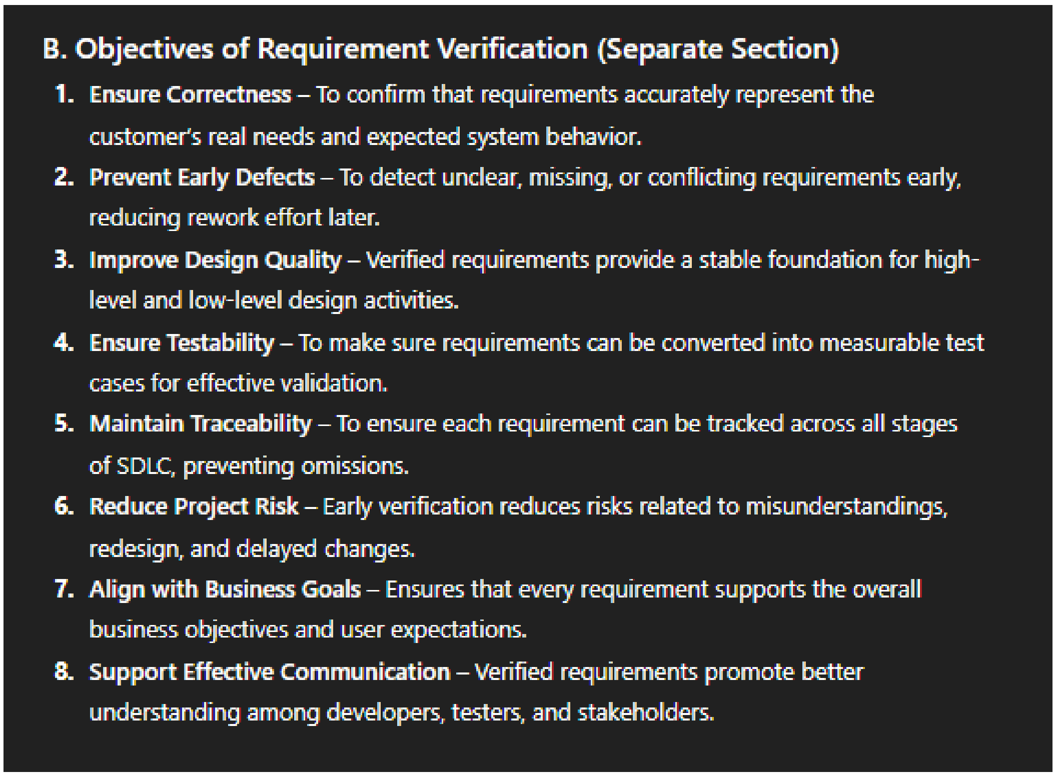
7. Stakeholder Confirmation

Requirements are reviewed with customers, users, and domain experts to ensure they match actual needs. This step prevents misunderstanding about what the system is expected to deliver.



8. Review of Non-Functional Objectives

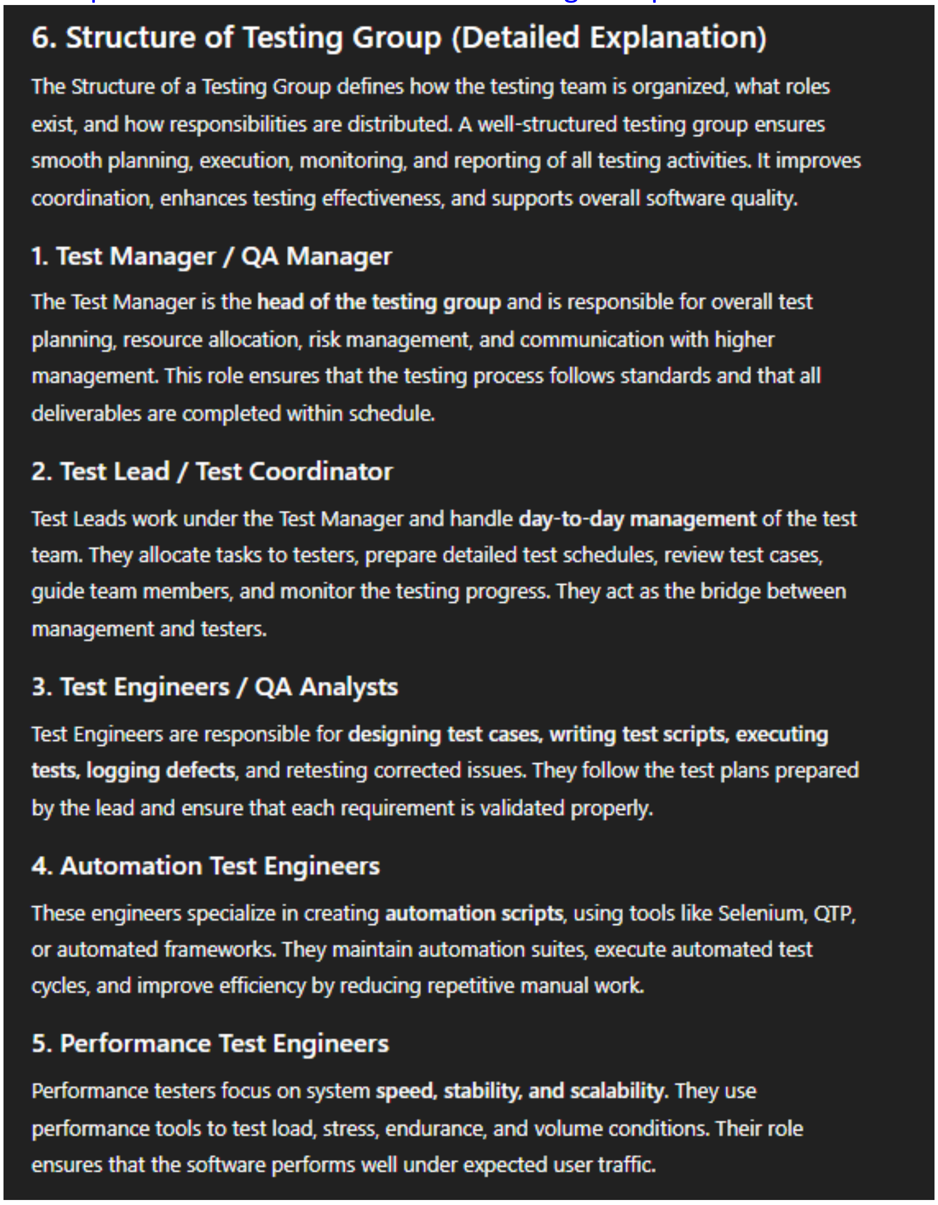
Quality objectives such as performance, security, reliability, and usability are also verified. They must be specific, measurable, and aligned with the overall project goals to ensure complete quality coverage.



B. Objectives of Requirement Verification (Separate Section)

1. **Ensure Correctness** – To confirm that requirements accurately represent the customer's real needs and expected system behavior.
2. **Prevent Early Defects** – To detect unclear, missing, or conflicting requirements early, reducing rework effort later.
3. **Improve Design Quality** – Verified requirements provide a stable foundation for high-level and low-level design activities.
4. **Ensure Testability** – To make sure requirements can be converted into measurable test cases for effective validation.
5. **Maintain Traceability** – To ensure each requirement can be tracked across all stages of SDLC, preventing omissions.
6. **Reduce Project Risk** – Early verification reduces risks related to misunderstandings, redesign, and delayed changes.
7. **Align with Business Goals** – Ensures that every requirement supports the overall business objectives and user expectations.
8. **Support Effective Communication** – Verified requirements promote better understanding among developers, testers, and stakeholders.

6. Explain in detail the Structure of Testing Group.



6. Structure of Testing Group (Detailed Explanation)

The Structure of a Testing Group defines how the testing team is organized, what roles exist, and how responsibilities are distributed. A well-structured testing group ensures smooth planning, execution, monitoring, and reporting of all testing activities. It improves coordination, enhances testing effectiveness, and supports overall software quality.

1. Test Manager / QA Manager

The Test Manager is the **head of the testing group** and is responsible for overall test planning, resource allocation, risk management, and communication with higher management. This role ensures that the testing process follows standards and that all deliverables are completed within schedule.

2. Test Lead / Test Coordinator

Test Leads work under the Test Manager and handle **day-to-day management** of the test team. They allocate tasks to testers, prepare detailed test schedules, review test cases, guide team members, and monitor the testing progress. They act as the bridge between management and testers.

3. Test Engineers / QA Analysts

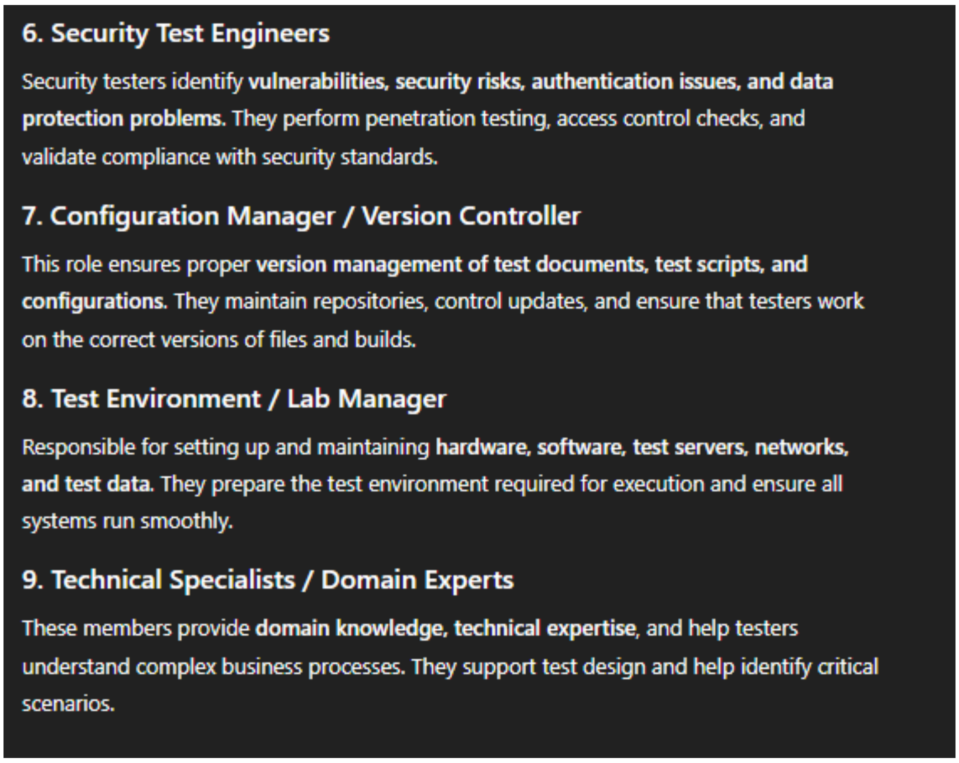
Test Engineers are responsible for **designing test cases, writing test scripts, executing tests, logging defects**, and retesting corrected issues. They follow the test plans prepared by the lead and ensure that each requirement is validated properly.

4. Automation Test Engineers

These engineers specialize in creating **automation scripts**, using tools like Selenium, QTP, or automated frameworks. They maintain automation suites, execute automated test cycles, and improve efficiency by reducing repetitive manual work.

5. Performance Test Engineers

Performance testers focus on system **speed, stability, and scalability.** They use performance tools to test load, stress, endurance, and volume conditions. Their role ensures that the software performs well under expected user traffic.



6. Security Test Engineers

Security testers identify **vulnerabilities, security risks, authentication issues, and data protection problems.** They perform penetration testing, access control checks, and validate compliance with security standards.

7. Configuration Manager / Version Controller

This role ensures proper **version management of test documents, test scripts, and configurations.** They maintain repositories, control updates, and ensure that testers work on the correct versions of files and builds.

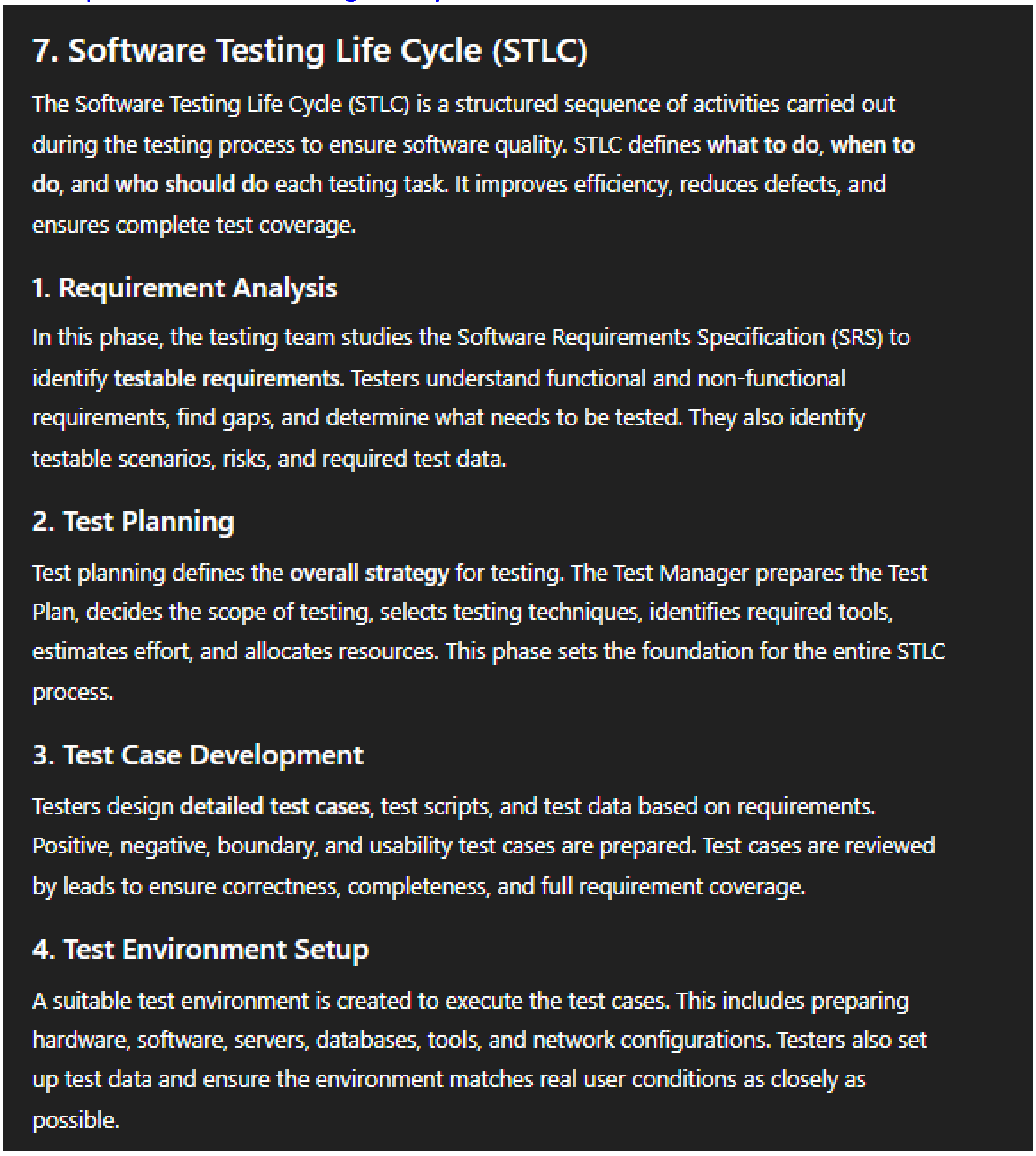
8. Test Environment / Lab Manager

Responsible for setting up and maintaining **hardware, software, test servers, networks, and test data.** They prepare the test environment required for execution and ensure all systems run smoothly.

9. Technical Specialists / Domain Experts

These members provide **domain knowledge, technical expertise**, and help testers understand complex business processes. They support test design and help identify critical scenarios.

7. Explain Software Testing Life Cycle.



7. Software Testing Life Cycle (STLC)

The Software Testing Life Cycle (STLC) is a structured sequence of activities carried out during the testing process to ensure software quality. STLC defines **what to do, when to do**, and **who should do** each testing task. It improves efficiency, reduces defects, and ensures complete test coverage.

1. Requirement Analysis

In this phase, the testing team studies the Software Requirements Specification (SRS) to identify **testable requirements.** Testers understand functional and non-functional requirements, find gaps, and determine what needs to be tested. They also identify testable scenarios, risks, and required test data.

2. Test Planning

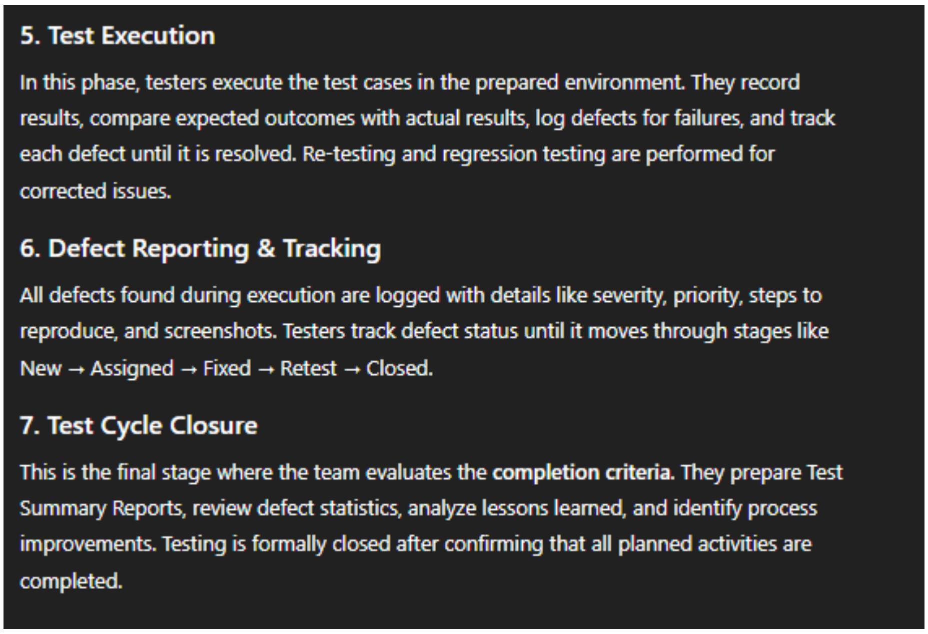
Test planning defines the **overall strategy** for testing. The Test Manager prepares the Test Plan, decides the scope of testing, selects testing techniques, identifies required tools, estimates effort, and allocates resources. This phase sets the foundation for the entire STLC process.

3. Test Case Development

Testers design **detailed test cases**, test scripts, and test data based on requirements. Positive, negative, boundary, and usability test cases are prepared. Test cases are reviewed by leads to ensure correctness, completeness, and full requirement coverage.

4. Test Environment Setup

A suitable test environment is created to execute the test cases. This includes preparing hardware, software, servers, databases, tools, and network configurations. Testers also set up test data and ensure the environment matches real user conditions as closely as possible.



5. Test Execution

In this phase, testers execute the test cases in the prepared environment. They record results, compare expected outcomes with actual results, log defects for failures, and track each defect until it is resolved. Re-testing and regression testing are performed for corrected issues.

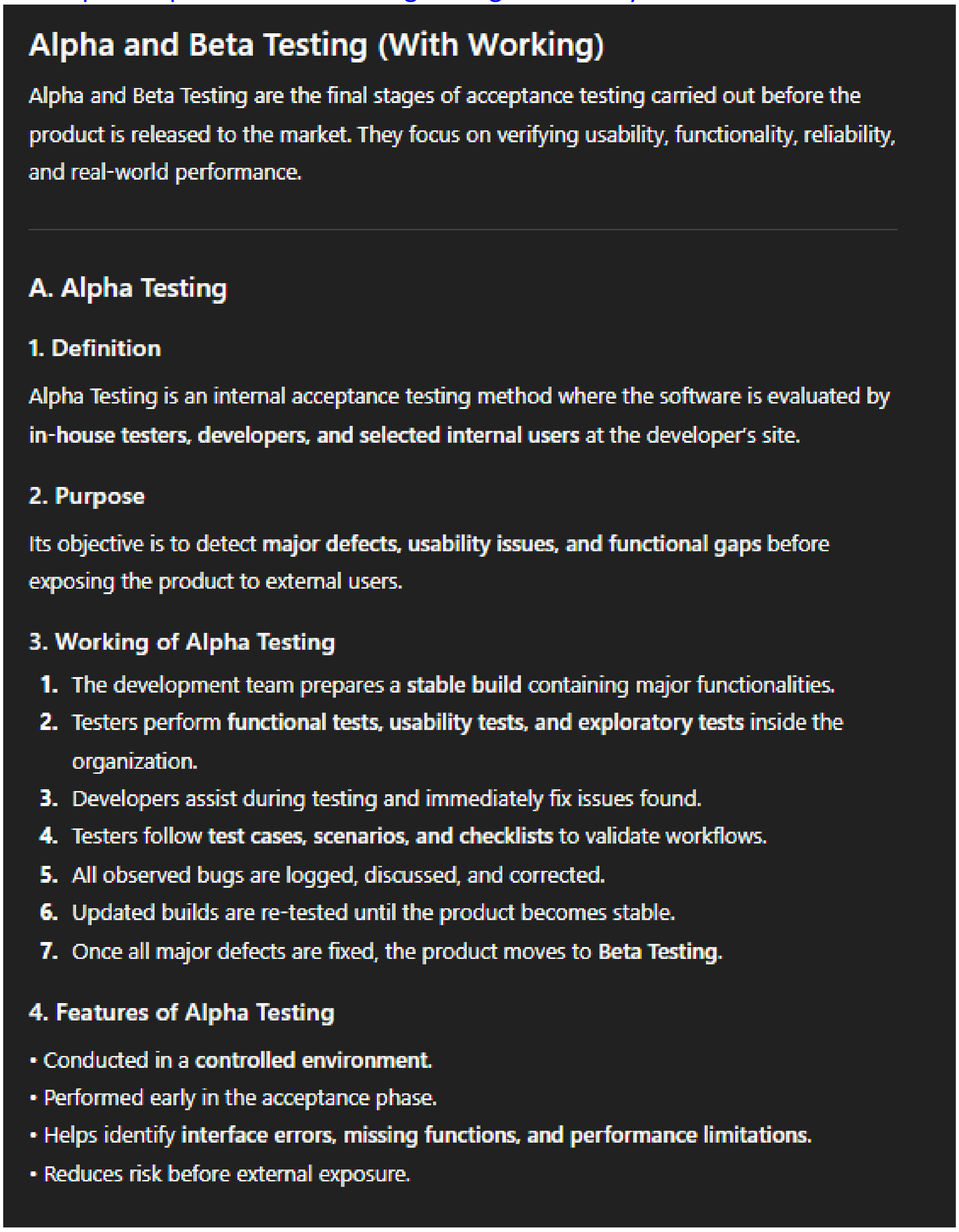
6. Defect Reporting & Tracking

All defects found during execution are logged with details like severity, priority, steps to reproduce, and screenshots. Testers track defect status until it moves through stages like New → Assigned → Fixed → Retest → Closed.

7. Test Cycle Closure

This is the final stage where the team evaluates the **completion criteria.** They prepare Test Summary Reports, review defect statistics, analyze lessons learned, and identify process improvements. Testing is formally closed after confirming that all planned activities are completed.

8. Explain Alpha and Beta Testing. Also give its entry and exit criteria.



Alpha and Beta Testing (With Working)

Alpha and Beta Testing are the final stages of acceptance testing carried out before the product is released to the market. They focus on verifying usability, functionality, reliability, and real-world performance.

A. Alpha Testing

1. Definition

Alpha Testing is an internal acceptance testing method where the software is evaluated by **in-house testers, developers, and selected internal users** at the developer's site.

2. Purpose

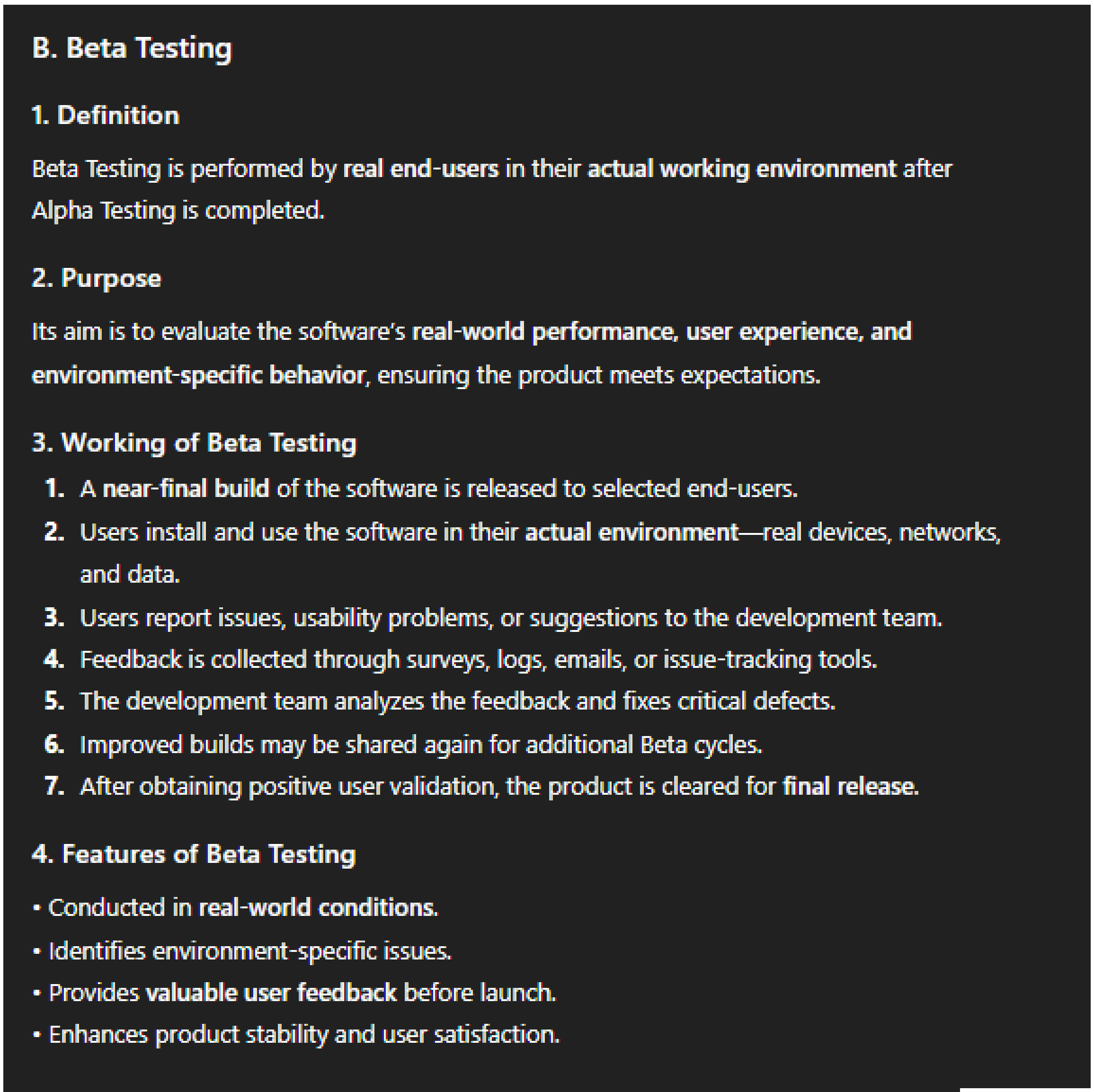
Its objective is to detect **major defects, usability issues, and functional gaps** before exposing the product to external users.

3. Working of Alpha Testing

1. The development team prepares a **stable build** containing major functionalities.
2. Testers perform **functional tests, usability tests, and exploratory tests** inside the organization.
3. Developers assist during testing and immediately fix issues found.
4. Testers follow **test cases, scenarios, and checklists** to validate workflows.
5. All observed bugs are logged, discussed, and corrected.
6. Updated builds are re-tested until the product becomes stable.
7. Once all major defects are fixed, the product moves to **Beta Testing.**

4. Features of Alpha Testing

• Conducted in a **controlled environment.**
• Performed early in the acceptance phase.
• Helps identify **interface errors, missing functions, and performance limitations.**
• Reduces risk before external exposure.



B. Beta Testing

1. Definition

Beta Testing is performed by **real end-users** in their **actual working environment** after Alpha Testing is completed.

2. Purpose

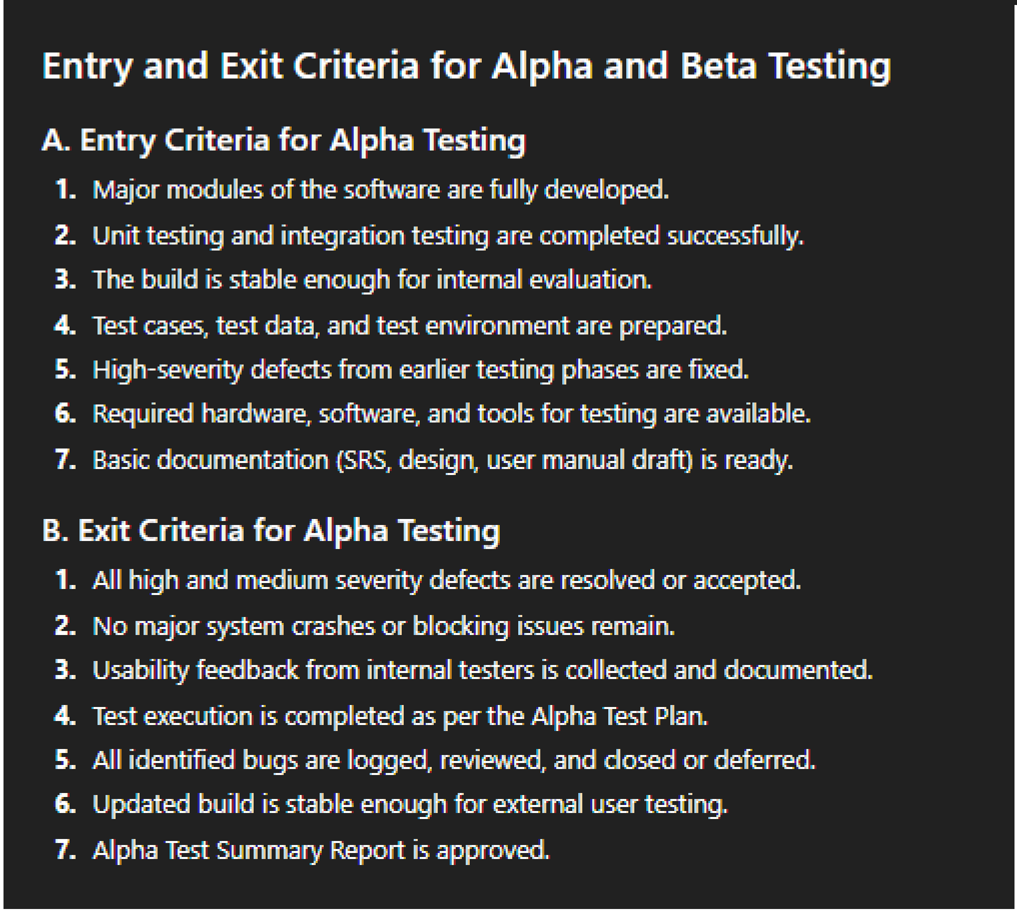
Its aim is to evaluate the software's **real-world performance, user experience, and environment-specific behavior**, ensuring the product meets expectations.

3. Working of Beta Testing

1. A **near-final build** of the software is released to selected end-users.
2. Users install and use the software in their **actual environment**—real devices, networks, and data.
3. Users report issues, usability problems, or suggestions to the development team.
4. Feedback is collected through surveys, logs, emails, or issue-tracking tools.
5. The development team analyzes the feedback and fixes critical defects.
6. Improved builds may be shared again for additional Beta cycles.
7. After obtaining positive user validation, the product is cleared for **final release.**

4. Features of Beta Testing

- Conducted in **real-world conditions.**
- Identifies environment-specific issues.
- Provides **valuable user feedback** before launch.
- Enhances product stability and user satisfaction.



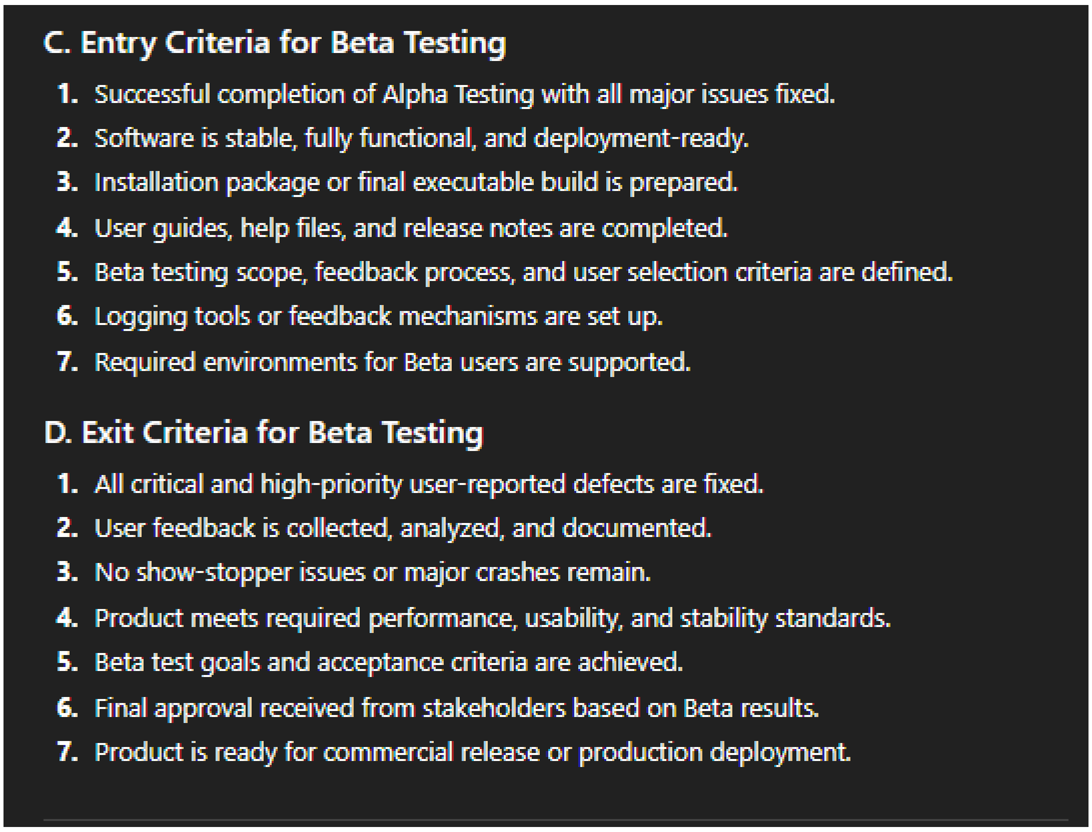
Entry and Exit Criteria for Alpha and Beta Testing

A. Entry Criteria for Alpha Testing

1. Major modules of the software are fully developed.
2. Unit testing and integration testing are completed successfully.
3. The build is stable enough for internal evaluation.
4. Test cases, test data, and test environment are prepared.
5. High-severity defects from earlier testing phases are fixed.
6. Required hardware, software, and tools for testing are available.
7. Basic documentation (SRS, design, user manual draft) is ready.

B. Exit Criteria for Alpha Testing

1. All high and medium severity defects are resolved or accepted.
2. No major system crashes or blocking issues remain.
3. Usability feedback from internal testers is collected and documented.
4. Test execution is completed as per the Alpha Test Plan.
5. All identified bugs are logged, reviewed, and closed or deferred.
6. Updated build is stable enough for external user testing.
7. Alpha Test Summary Report is approved.



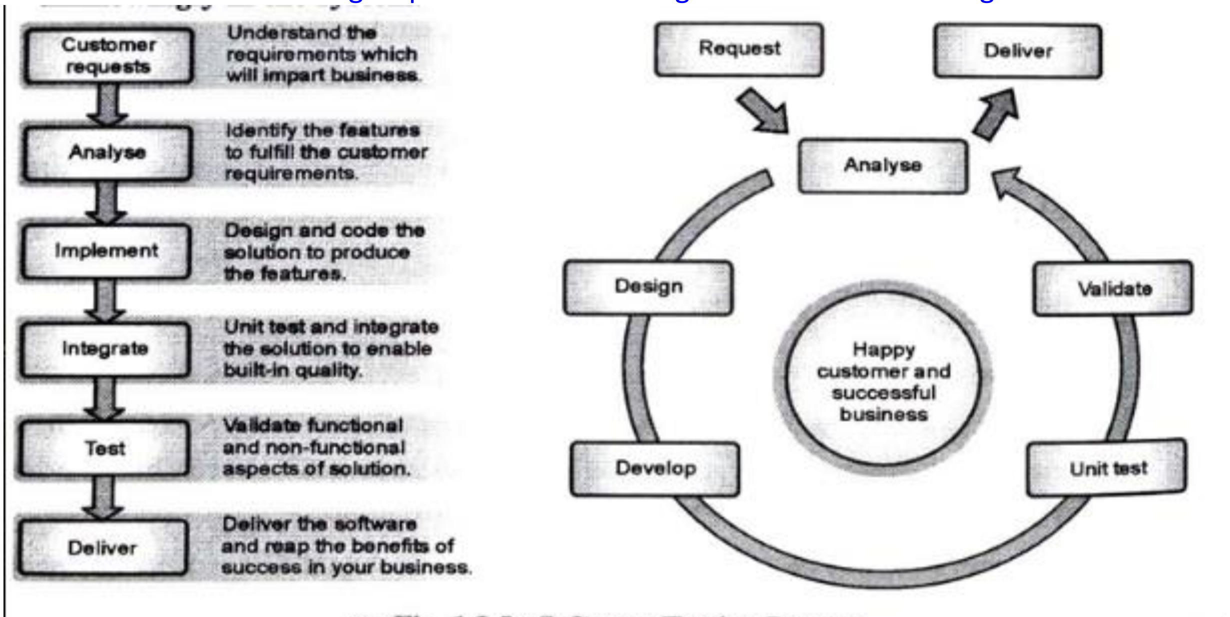
C. Entry Criteria for Beta Testing

1. Successful completion of Alpha Testing with all major issues fixed.
2. Software is stable, fully functional, and deployment-ready.
3. Installation package or final executable build is prepared.
4. User guides, help files, and release notes are completed.
5. Beta testing scope, feedback process, and user selection criteria are defined.
6. Logging tools or feedback mechanisms are set up.
7. Required environments for Beta users are supported.

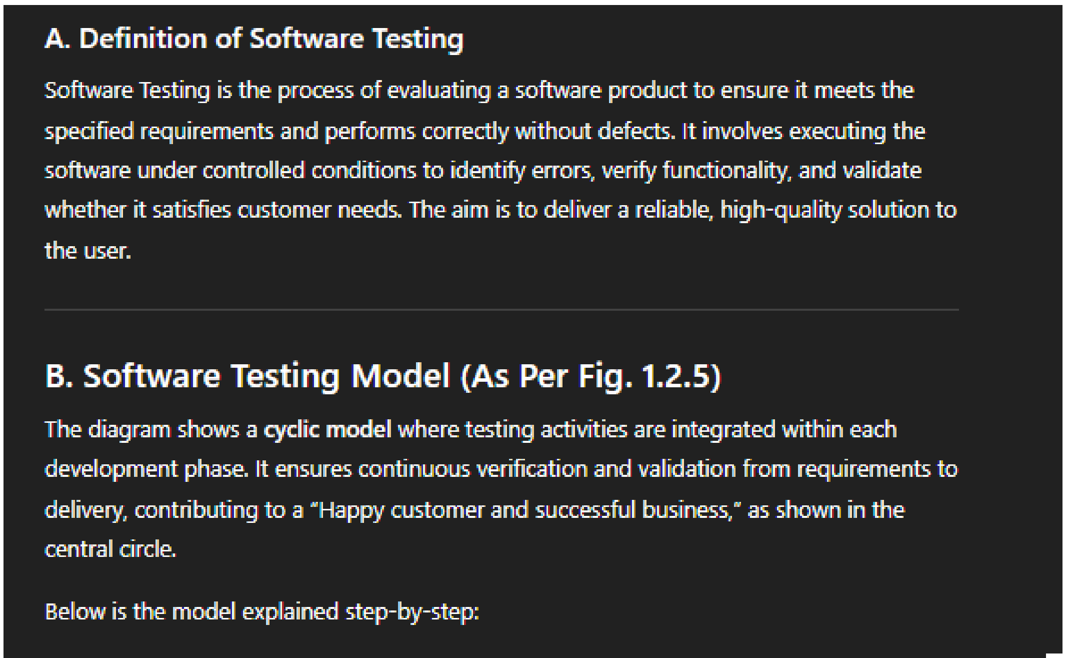
D. Exit Criteria for Beta Testing

1. All critical and high-priority user-reported defects are fixed.
2. User feedback is collected, analyzed, and documented.
3. No show-stopper issues or major crashes remain.
4. Product meets required performance, usability, and stability standards.
5. Beta test goals and acceptance criteria are achieved.
6. Final approval received from stakeholders based on Beta results.
7. Product is ready for commercial release or production deployment.

9. Define software testing. Explain software testing model with a neat diagram.


Customer requests
Understand the requirements which will impart business.
Analyse
Identify the features to fulfill the customer requirements.
Implement
Design and code the solution to produce the features.
Integrate
Unit test and integrate the solution to enable built-in quality.
Test
Validate functional and non-functional aspects of solution.
Deliver
Deliver the software and reap the benefits of success in your business.
Request
Deliver
Analyse
Design
Validate
Happy customer and successful business
Develop
Unit test

(1A5)Fig. 1.2.5 : Software Testing Process



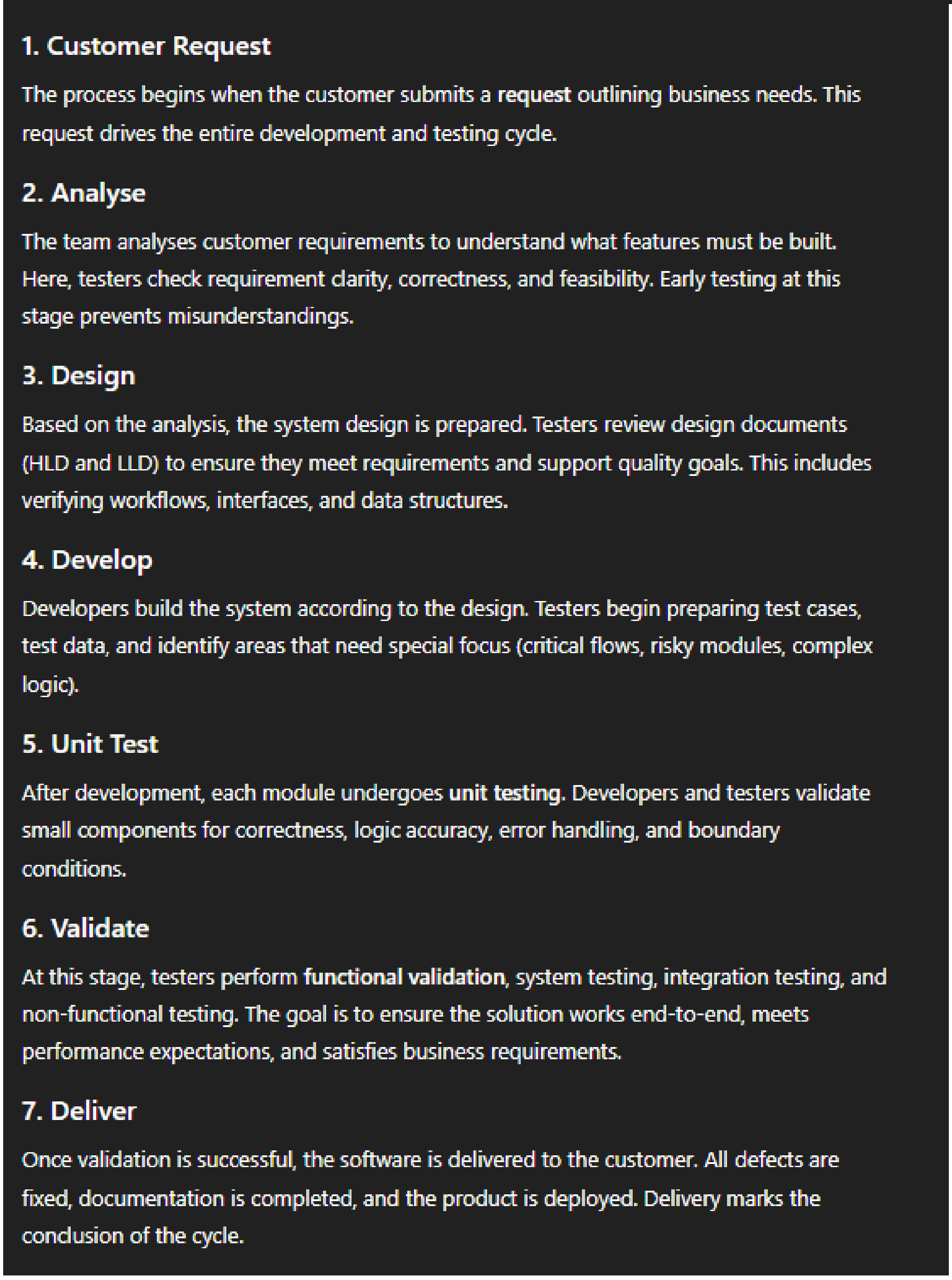
A. Definition of Software Testing

Software Testing is the process of evaluating a software product to ensure it meets the specified requirements and performs correctly without defects. It involves executing the software under controlled conditions to identify errors, verify functionality, and validate whether it satisfies customer needs. The aim is to deliver a reliable, high-quality solution to the user.

B. Software Testing Model (As Per Fig. 1.2.5)

The diagram shows a **cyclic model** where testing activities are integrated within each development phase. It ensures continuous verification and validation from requirements to delivery, contributing to a "Happy customer and successful business," as shown in the central circle.

Below is the model explained step-by-step:



1. Customer Request

The process begins when the customer submits a **request** outlining business needs. This request drives the entire development and testing cycle.

2. Analyse

The team analyses customer requirements to understand what features must be built. Here, testers check requirement clarity, correctness, and feasibility. Early testing at this stage prevents misunderstandings.

3. Design

Based on the analysis, the system design is prepared. Testers review design documents (HLD and LLD) to ensure they meet requirements and support quality goals. This includes verifying workflows, interfaces, and data structures.

4. Develop

Developers build the system according to the design. Testers begin preparing test cases, test data, and identify areas that need special focus (critical flows, risky modules, complex logic).

5. Unit Test

After development, each module undergoes **unit testing**. Developers and testers validate small components for correctness, logic accuracy, error handling, and boundary conditions.

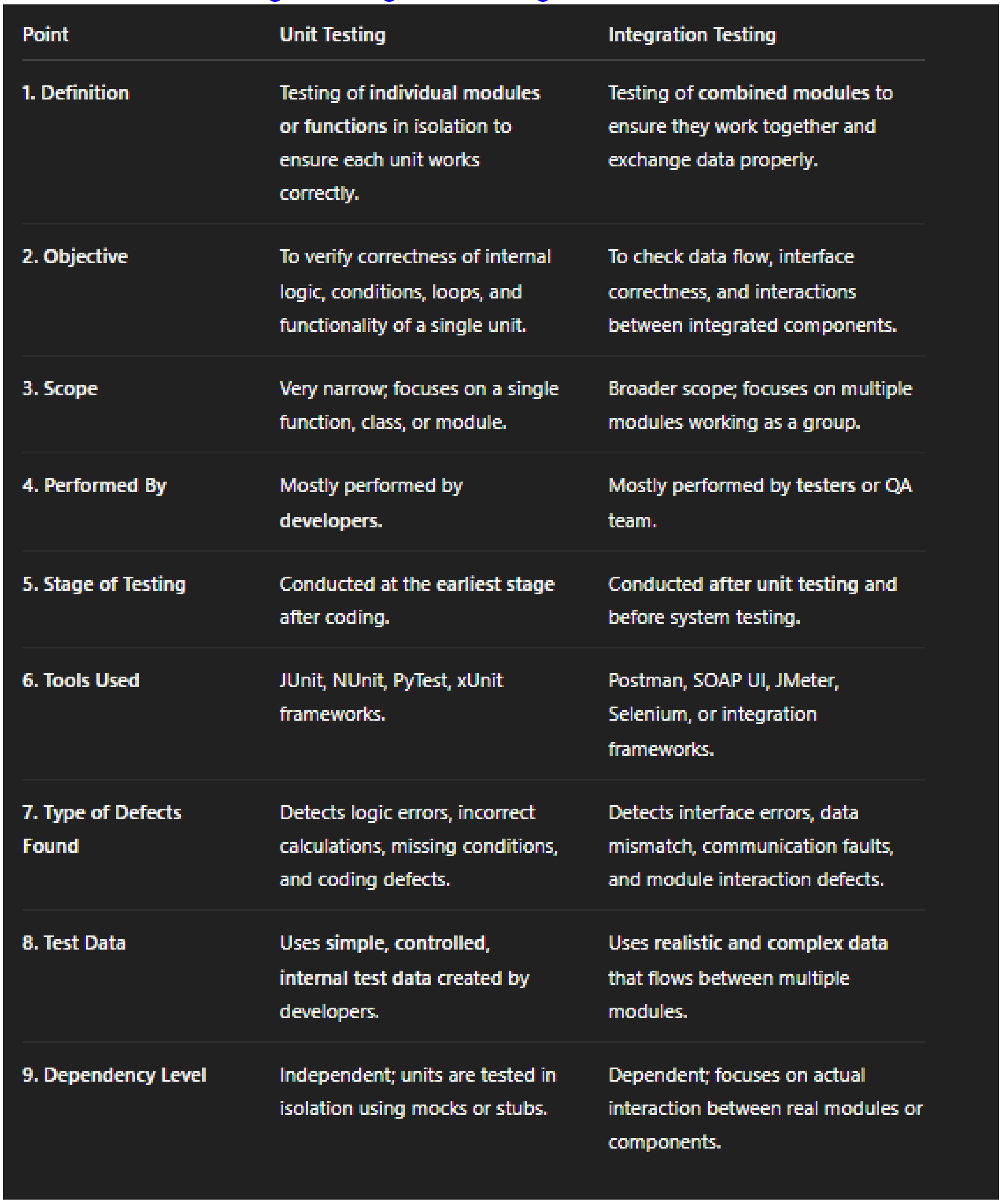
6. Validate

At this stage, testers perform **functional validation**, system testing, integration testing, and non-functional testing. The goal is to ensure the solution works end-to-end, meets performance expectations, and satisfies business requirements.

7. Deliver

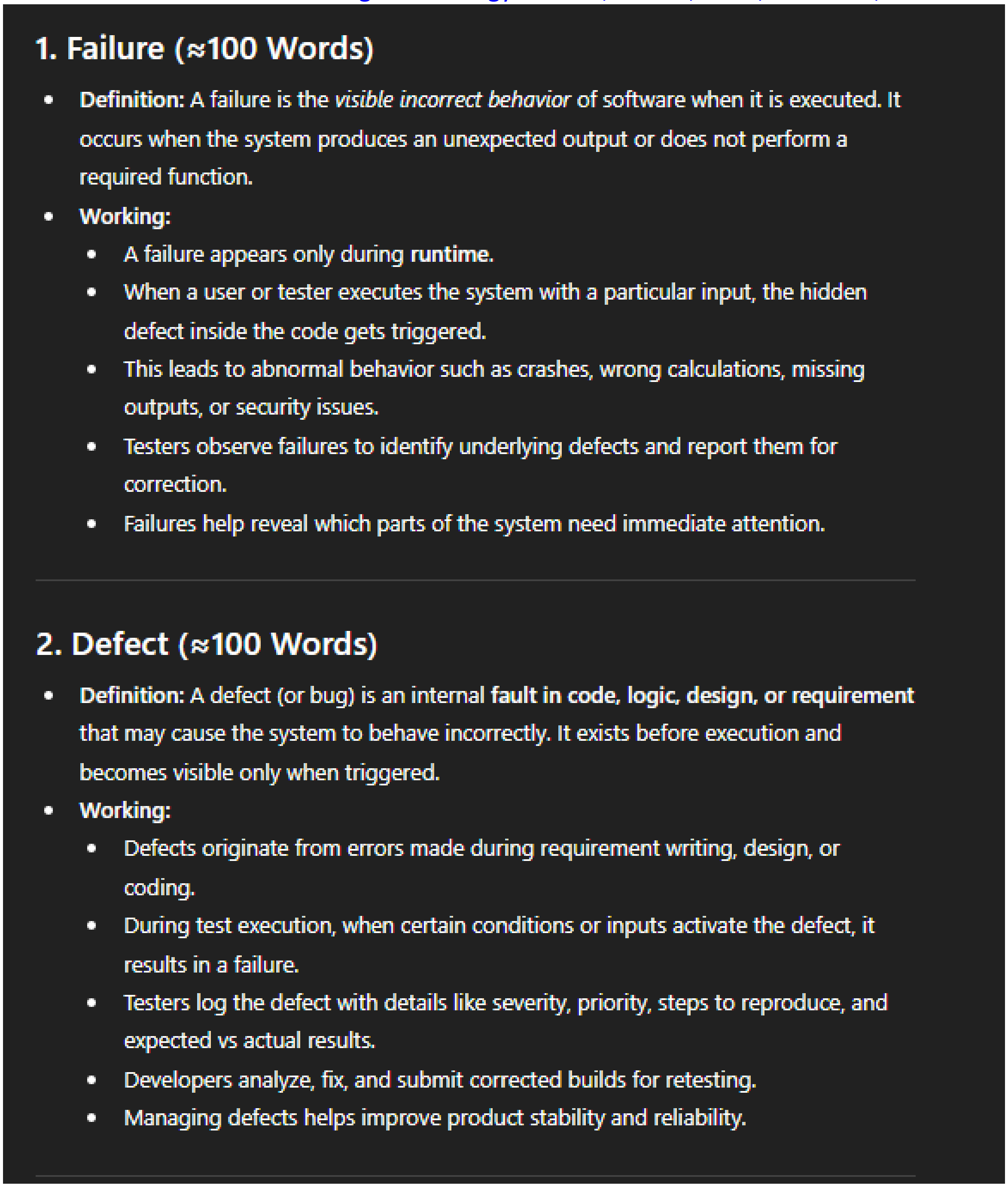
Once validation is successful, the software is delivered to the customer. All defects are fixed, documentation is completed, and the product is deployed. Delivery marks the conclusion of the cycle.

10. How do unit Testing and Integration testing differ from each other?



Point	Unit Testing	Integration Testing
1. Definition	Testing of **individual modules or functions** in isolation to ensure each unit works correctly.	Testing of **combined modules** to ensure they work together and exchange data properly.
2. Objective	To verify correctness of internal logic, conditions, loops, and functionality of a single unit.	To check data flow, interface correctness, and interactions between integrated components.
3. Scope	Very narrow; focuses on a single function, class, or module.	Broader scope; focuses on multiple modules working as a group.
4. Performed By	Mostly performed by **developers**.	Mostly performed by **testers or QA team**.
5. Stage of Testing	Conducted at the **earliest stage** after coding.	Conducted **after unit testing** and **before system testing**.
6. Tools Used	JUnit, NUnit, PyTest, xUnit frameworks.	Postman, SOAP UI, JMeter, Selenium, or integration frameworks.
7. Type of Defects Found	Detects logic errors, incorrect calculations, missing conditions, and coding defects.	Detects interface errors, data mismatch, communication faults, and module interaction defects.
8. Test Data	Uses simple, controlled, internal test data created by developers.	Uses **realistic and complex data** that flows between multiple modules.
9. Dependency Level	Independent; units are tested in isolation using mocks or stubs.	Dependent; focuses on actual interaction between real modules or components.

11. Define each software testing terminology: Failure, Defect, Error, Testware, Test oracle.



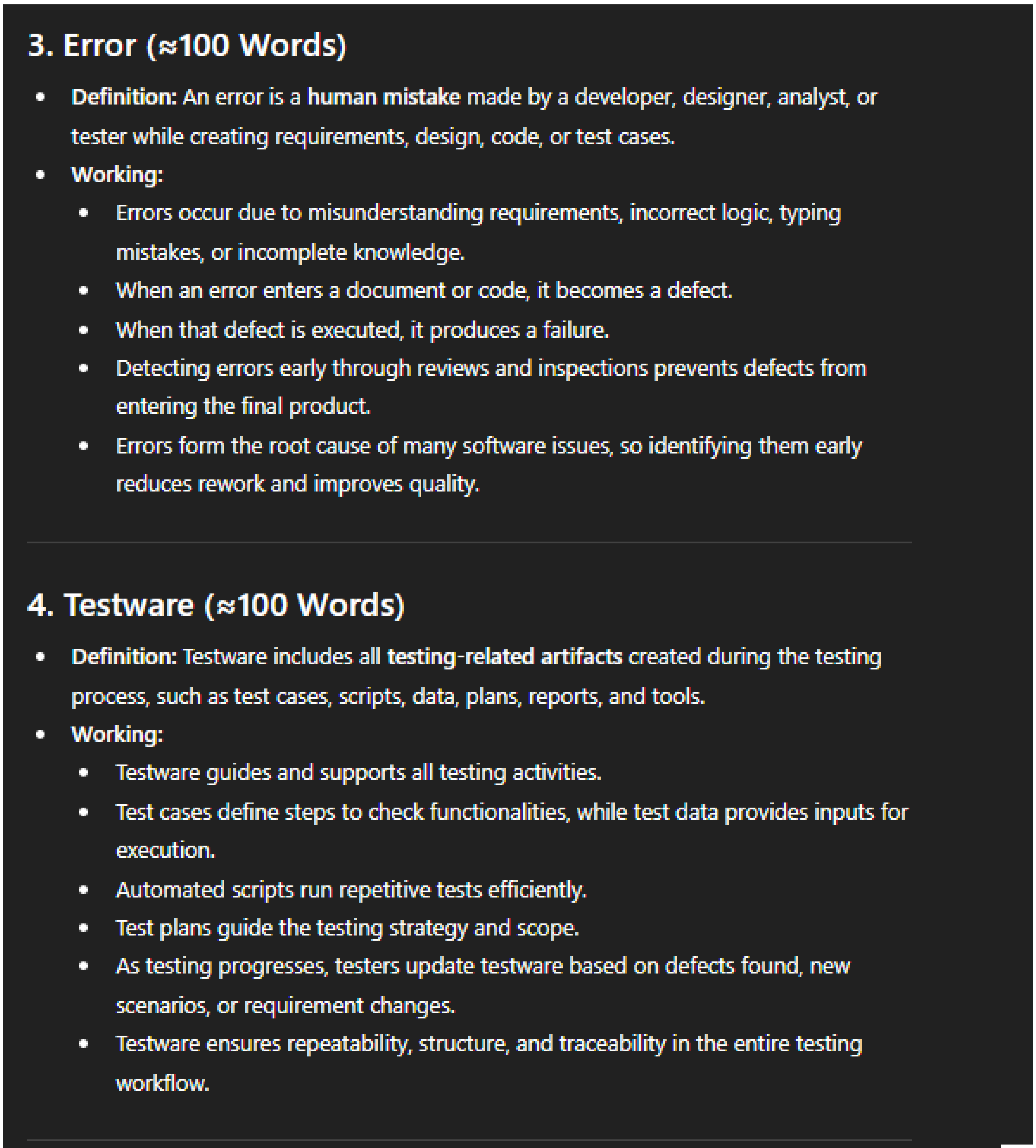
1. Failure (≈100 Words)

- **Definition:** A failure is the *visible incorrect behavior* of software when it is executed. It occurs when the system produces an unexpected output or does not perform a required function.
- **Working:**
  - A failure appears only during **runtime.**
  - When a user or tester executes the system with a particular input, the hidden defect inside the code gets triggered.
  - This leads to abnormal behavior such as crashes, wrong calculations, missing outputs, or security issues.
  - Testers observe failures to identify underlying defects and report them for correction.
  - Failures help reveal which parts of the system need immediate attention.

---

2. Defect (≈100 Words)

- **Definition:** A defect (or bug) is an internal **fault in code, logic, design, or requirement** that may cause the system to behave incorrectly. It exists before execution and becomes visible only when triggered.
- **Working:**
  - Defects originate from errors made during requirement writing, design, or coding.
  - During test execution, when certain conditions or inputs activate the defect, it results in a failure.
  - Testers log the defect with details like severity, priority, steps to reproduce, and expected vs actual results.
  - Developers analyze, fix, and submit corrected builds for retesting.
  - Managing defects helps improve product stability and reliability.



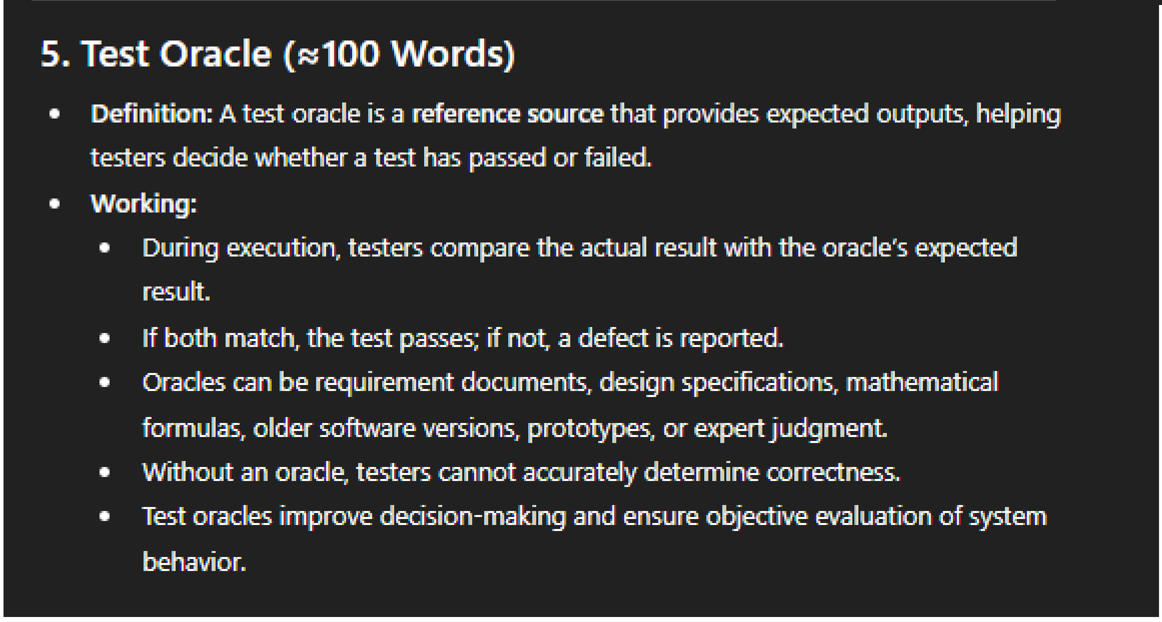
3. Error (≈100 Words)

- **Definition:** An error is a **human mistake** made by a developer, designer, analyst, or tester while creating requirements, design, code, or test cases.
- **Working:**
  - Errors occur due to misunderstanding requirements, incorrect logic, typing mistakes, or incomplete knowledge.
  - When an error enters a document or code, it becomes a defect.
  - When that defect is executed, it produces a failure.
  - Detecting errors early through reviews and inspections prevents defects from entering the final product.
  - Errors form the root cause of many software issues, so identifying them early reduces rework and improves quality.

---

4. Testware (≈100 Words)

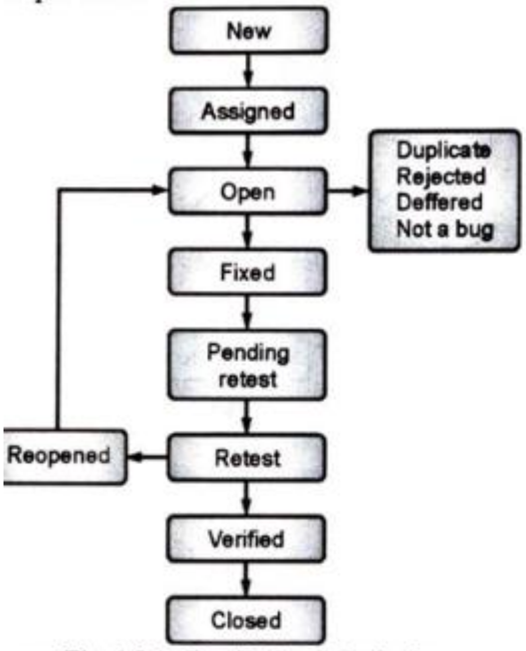
- **Definition:** Testware includes all **testing-related artifacts** created during the testing process, such as test cases, scripts, data, plans, reports, and tools.
- **Working:**
  - Testware guides and supports all testing activities.
  - Test cases define steps to check functionalities, while test data provides inputs for execution.
  - Automated scripts run repetitive tests efficiently.
  - Test plans guide the testing strategy and scope.
  - As testing progresses, testers update testware based on defects found, new scenarios, or requirement changes.
  - Testware ensures repeatability, structure, and traceability in the entire testing workflow.



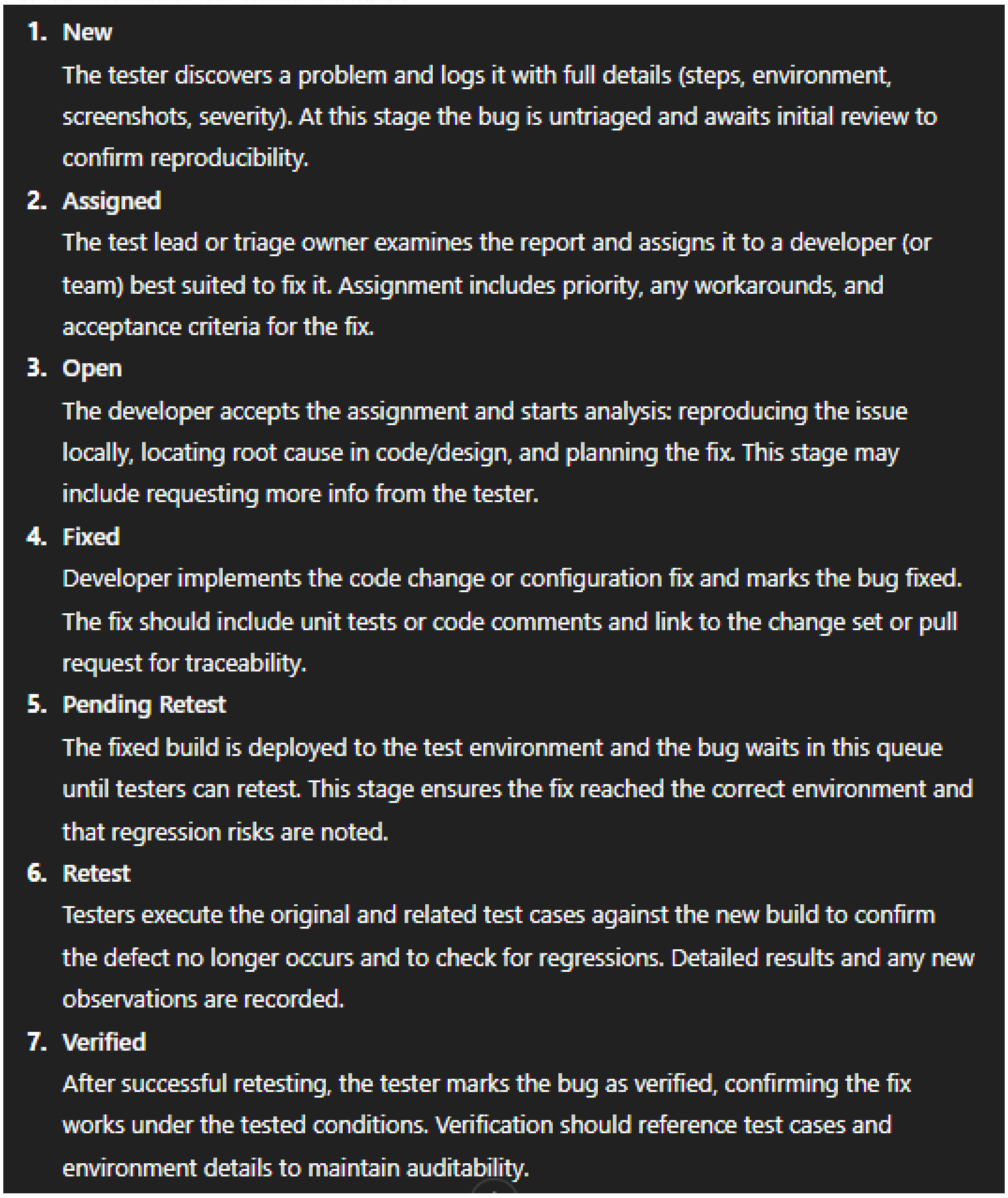
5. Test Oracle (≈100 Words)

- **Definition:** A test oracle is a **reference source** that provides expected outputs, helping testers decide whether a test has passed or failed.
- **Working:**
  - During execution, testers compare the actual result with the oracle's expected result.
  - If both match, the test passes; if not, a defect is reported.
  - Oracles can be requirement documents, design specifications, mathematical formulas, older software versions, prototypes, or expert judgment.
  - Without an oracle, testers cannot accurately determine correctness.
  - Test oracles improve decision-making and ensure objective evaluation of system behavior.

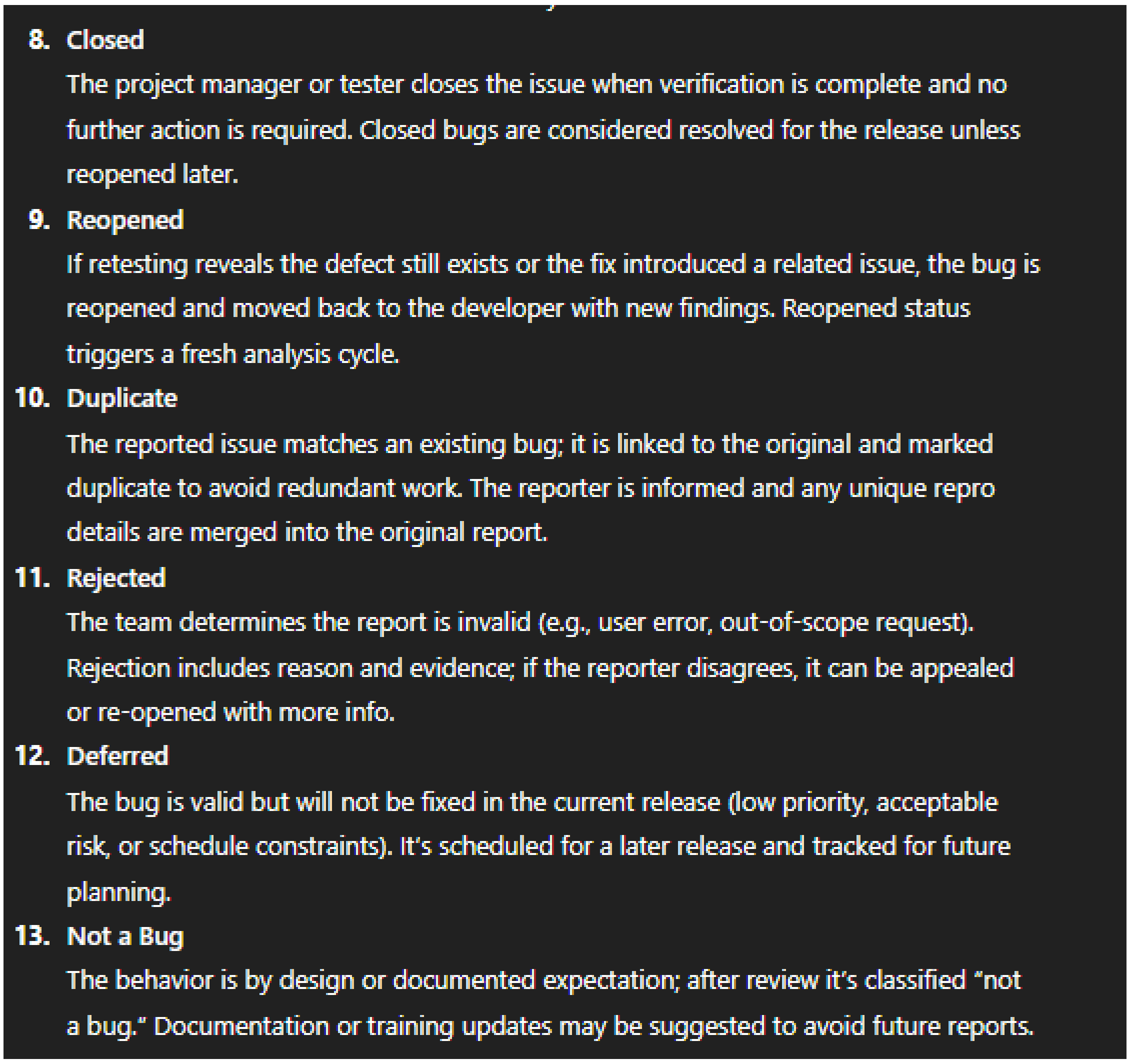
12. Explain a bug life cycle with a neat diagram in detail. List down the states of a bug.



(1A10)Fig. 1.8.1 : Bug/Defect Life Cycle



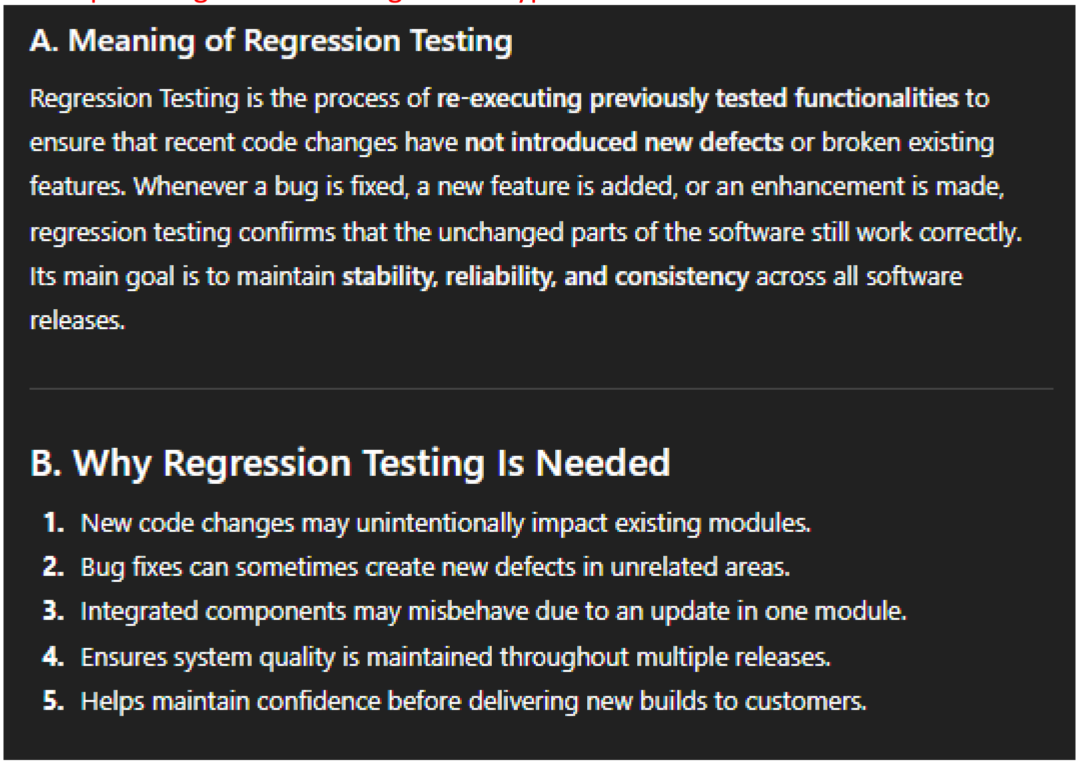
1. **New**
   The tester discovers a problem and logs it with full details (steps, environment, screenshots, severity). At this stage the bug is untriaged and awaits initial review to confirm reproducibility.
2. **Assigned**
   The test lead or triage owner examines the report and assigns it to a developer (or team) best suited to fix it. Assignment includes priority, any workarounds, and acceptance criteria for the fix.
3. **Open**
   The developer accepts the assignment and starts analysis: reproducing the issue locally, locating root cause in code/design, and planning the fix. This stage may include requesting more info from the tester.
4. **Fixed**
   Developer implements the code change or configuration fix and marks the bug fixed. The fix should include unit tests or code comments and link to the change set or pull request for traceability.
5. **Pending Retest**
   The fixed build is deployed to the test environment and the bug waits in this queue until testers can retest. This stage ensures the fix reached the correct environment and that regression risks are noted.
6. **Retest**
   Testers execute the original and related test cases against the new build to confirm the defect no longer occurs and to check for regressions. Detailed results and any new observations are recorded.
7. **Verified**
   After successful retesting, the tester marks the bug as verified, confirming the fix works under the tested conditions. Verification should reference test cases and environment details to maintain auditability.



8. **Closed**
   The project manager or tester closes the issue when verification is complete and no further action is required. Closed bugs are considered resolved for the release unless reopened later.
9. **Reopened**
   If retesting reveals the defect still exists or the fix introduced a related issue, the bug is reopened and moved back to the developer with new findings. Reopened status triggers a fresh analysis cycle.
10. **Duplicate**
    The reported issue matches an existing bug; it is linked to the original and marked duplicate to avoid redundant work. The reporter is informed and any unique repro details are merged into the original report.
11. **Rejected**
    The team determines the report is invalid (e.g., user error, out-of-scope request). Rejection includes reason and evidence; if the reporter disagrees, it can be appealed or re-opened with more info.
12. **Deferred**
    The bug is valid but will not be fixed in the current release (low priority, acceptable risk, or schedule constraints). It's scheduled for a later release and tracked for future planning.
13. **Not a Bug**
    The behavior is by design or documented expectation; after review it's classified "not a bug." Documentation or training updates may be suggested to avoid future reports.

UNIT–II : Testing Techniques

13. Explain Regression Testing and its types in detail.

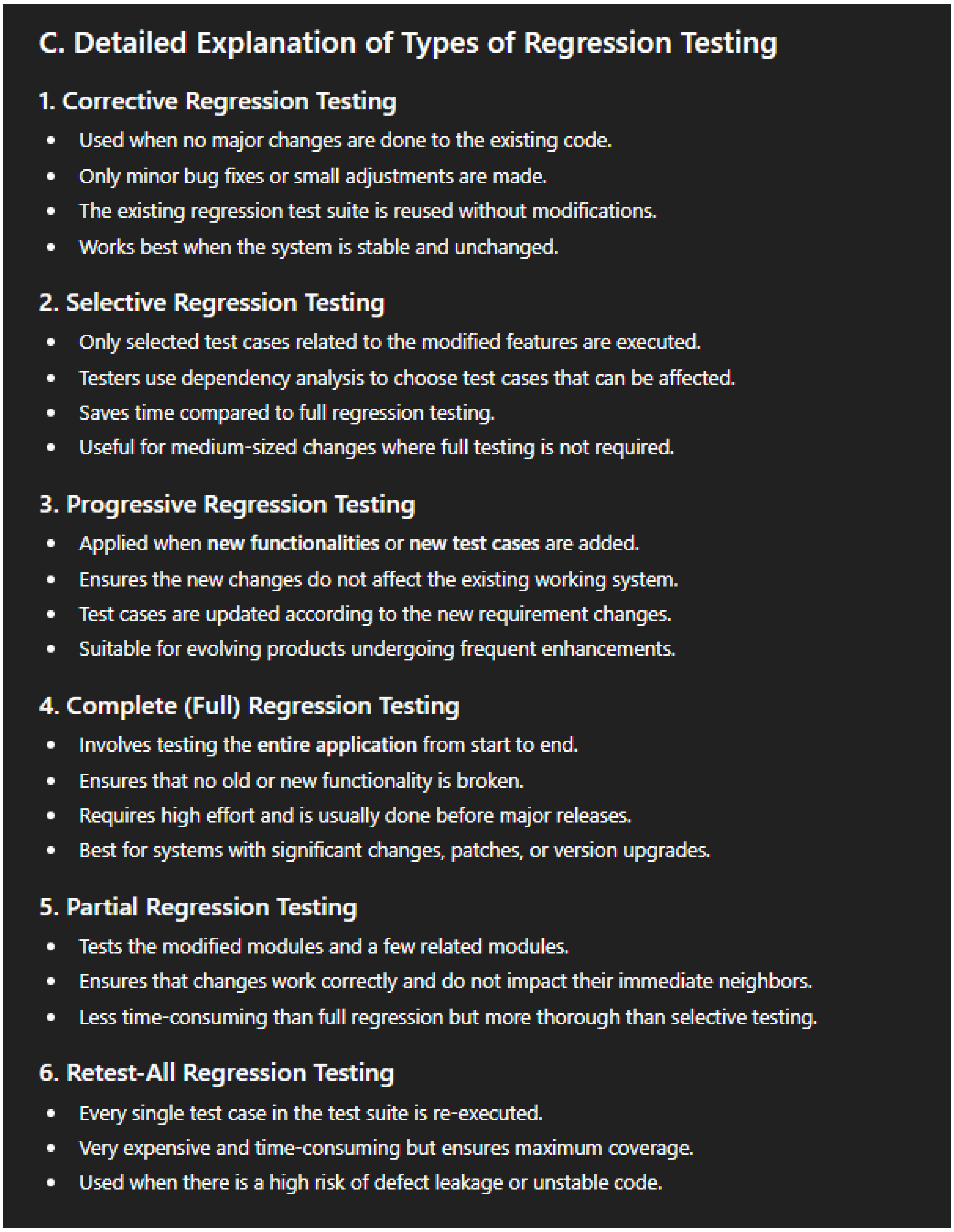


A. Meaning of Regression Testing

Regression Testing is the process of **re-executing previously tested functionalities** to ensure that recent code changes have **not introduced new defects** or broken existing features. Whenever a bug is fixed, a new feature is added, or an enhancement is made, regression testing confirms that the unchanged parts of the software still work correctly. Its main goal is to maintain **stability, reliability, and consistency** across all software releases.

B. Why Regression Testing Is Needed

1. New code changes may unintentionally impact existing modules.
2. Bug fixes can sometimes create new defects in unrelated areas.
3. Integrated components may misbehave due to an update in one module.
4. Ensures system quality is maintained throughout multiple releases.
5. Helps maintain confidence before delivering new builds to customers.



C. Detailed Explanation of Types of Regression Testing

1. Corrective Regression Testing

- Used when no major changes are done to the existing code.
- Only minor bug fixes or small adjustments are made.
- The existing regression test suite is reused without modifications.
- Works best when the system is stable and unchanged.

2. Selective Regression Testing

- Only selected test cases related to the modified features are executed.
- Testers use dependency analysis to choose test cases that can be affected.
- Saves time compared to full regression testing.
- Useful for medium-sized changes where full testing is not required.

3. Progressive Regression Testing

- Applied when **new functionalities** or **new test cases** are added.
- Ensures the new changes do not affect the existing working system.
- Test cases are updated according to the new requirement changes.
- Suitable for evolving products undergoing frequent enhancements.

4. Complete (Full) Regression Testing

- Involves testing the **entire application** from start to end.
- Ensures that no old or new functionality is broken.
- Requires high effort and is usually done before major releases.
- Best for systems with significant changes, patches, or version upgrades.

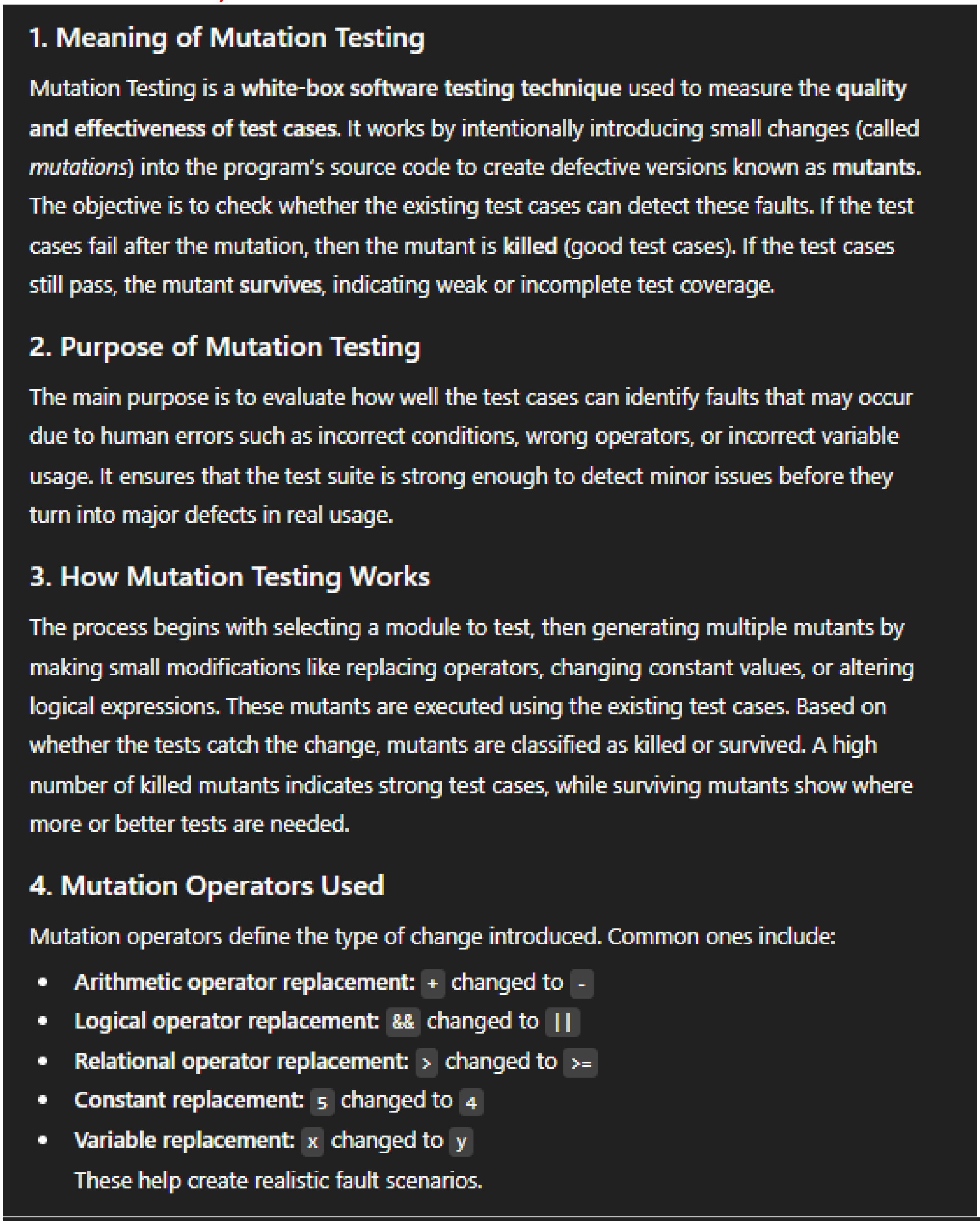
5. Partial Regression Testing

- Tests the modified modules and a few related modules.
- Ensures that changes work correctly and do not impact their immediate neighbors.
- Less time-consuming than full regression but more thorough than selective testing.

6. Retest-All Regression Testing

- Every single test case in the test suite is re-executed.
- Very expensive and time-consuming but ensures maximum coverage.
- Used when there is a high risk of defect leakage or unstable code.

14. What is Mutation Testing? Explain primary mutant with an example.* Differentiate between primary and secondary mutants.*



1. Meaning of Mutation Testing

Mutation Testing is a **white-box software testing technique** used to measure the **quality and effectiveness of test cases**. It works by intentionally introducing small changes (called *mutations*) into the program's source code to create defective versions known as **mutants**. The objective is to check whether the existing test cases can detect these faults. If the test cases fail after the mutation, then the mutant is **killed** (good test cases). If the test cases still pass, the mutant **survives**, indicating weak or incomplete test coverage.

2. Purpose of Mutation Testing

The main purpose is to evaluate how well the test cases can identify faults that may occur due to human errors such as incorrect conditions, wrong operators, or incorrect variable usage. It ensures that the test suite is strong enough to detect minor issues before they turn into major defects in real usage.

3. How Mutation Testing Works

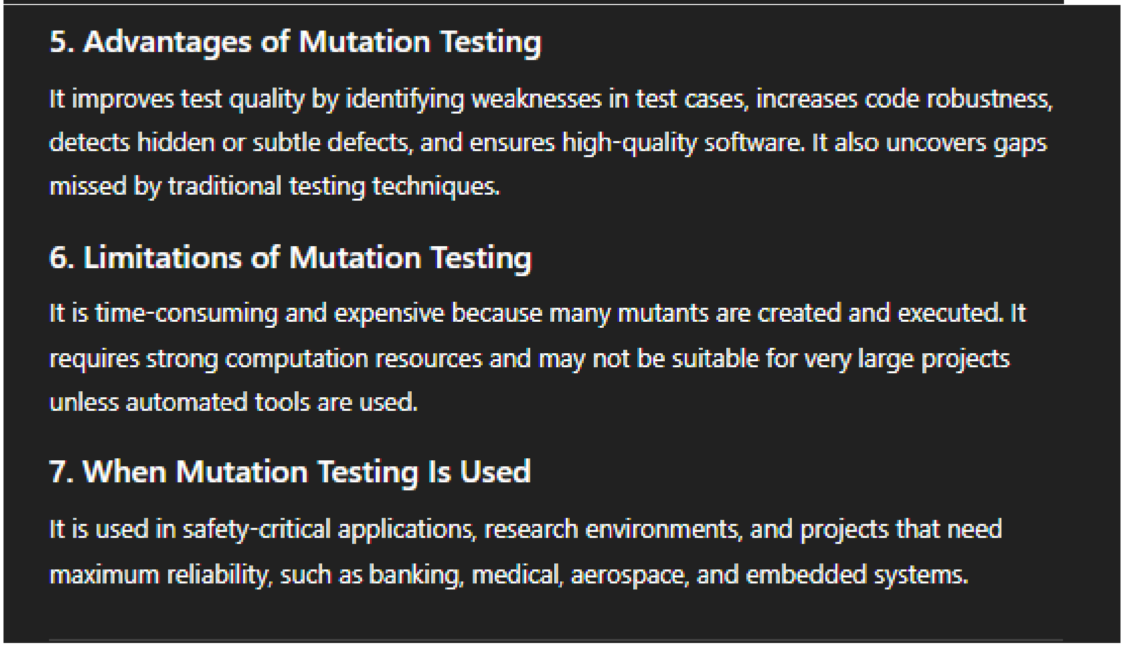
The process begins with selecting a module to test, then generating multiple mutants by making small modifications like replacing operators, changing constant values, or altering logical expressions. These mutants are executed using the existing test cases. Based on whether the tests catch the change, mutants are classified as killed or survived. A high number of killed mutants indicates strong test cases, while surviving mutants show where more or better tests are needed.

4. Mutation Operators Used

Mutation operators define the type of change introduced. Common ones include:

- **Arithmetic operator replacement:** `+` changed to `-`
- **Logical operator replacement:** `&&` changed to `||`
- **Relational operator replacement:** `>` changed to `>=`
- **Constant replacement:** `5` changed to `4`
- **Variable replacement:** `x` changed to `y`

  These help create realistic fault scenarios.



5. Advantages of Mutation Testing

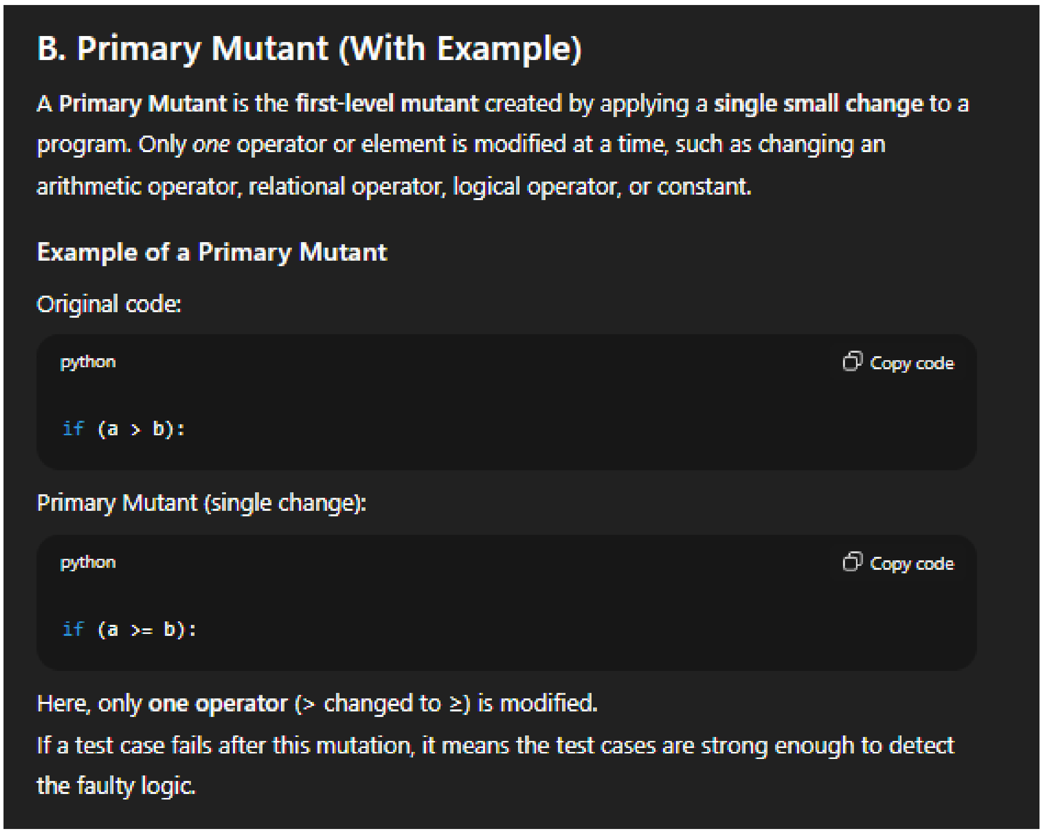
It improves test quality by identifying weaknesses in test cases, increases code robustness, detects hidden or subtle defects, and ensures high-quality software. It also uncovers gaps missed by traditional testing techniques.

6. Limitations of Mutation Testing

It is time-consuming and expensive because many mutants are created and executed. It requires strong computation resources and may not be suitable for very large projects unless automated tools are used.

7. When Mutation Testing Is Used

It is used in safety-critical applications, research environments, and projects that need maximum reliability, such as banking, medical, aerospace, and embedded systems.



B. Primary Mutant (With Example)

A **Primary Mutant** is the **first-level mutant** created by applying a **single small change** to a program. Only *one* operator or element is modified at a time, such as changing an arithmetic operator, relational operator, logical operator, or constant.

Example of a Primary Mutant

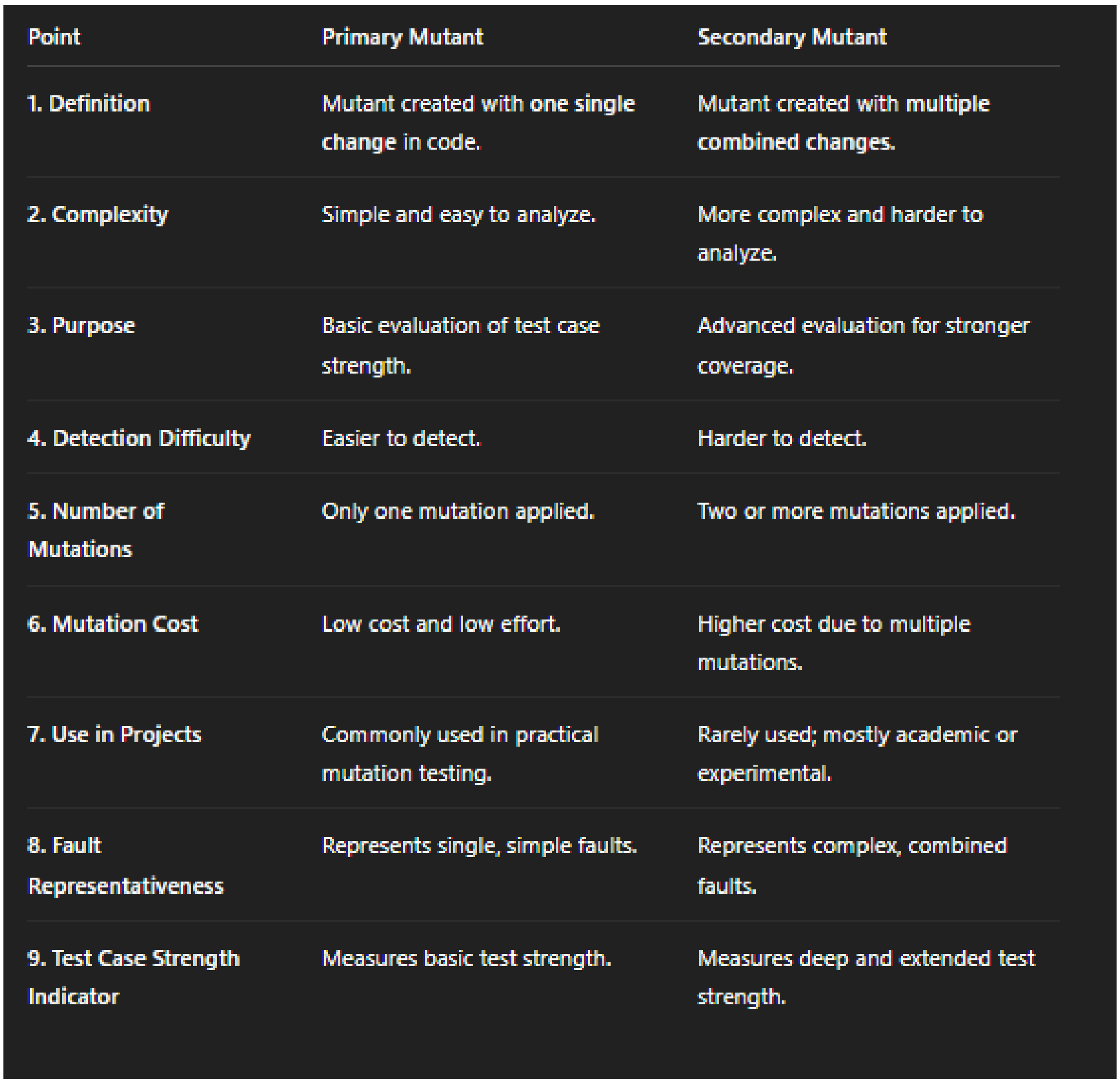
Original code:

```python
if (a > b):
```

Primary Mutant (single change):

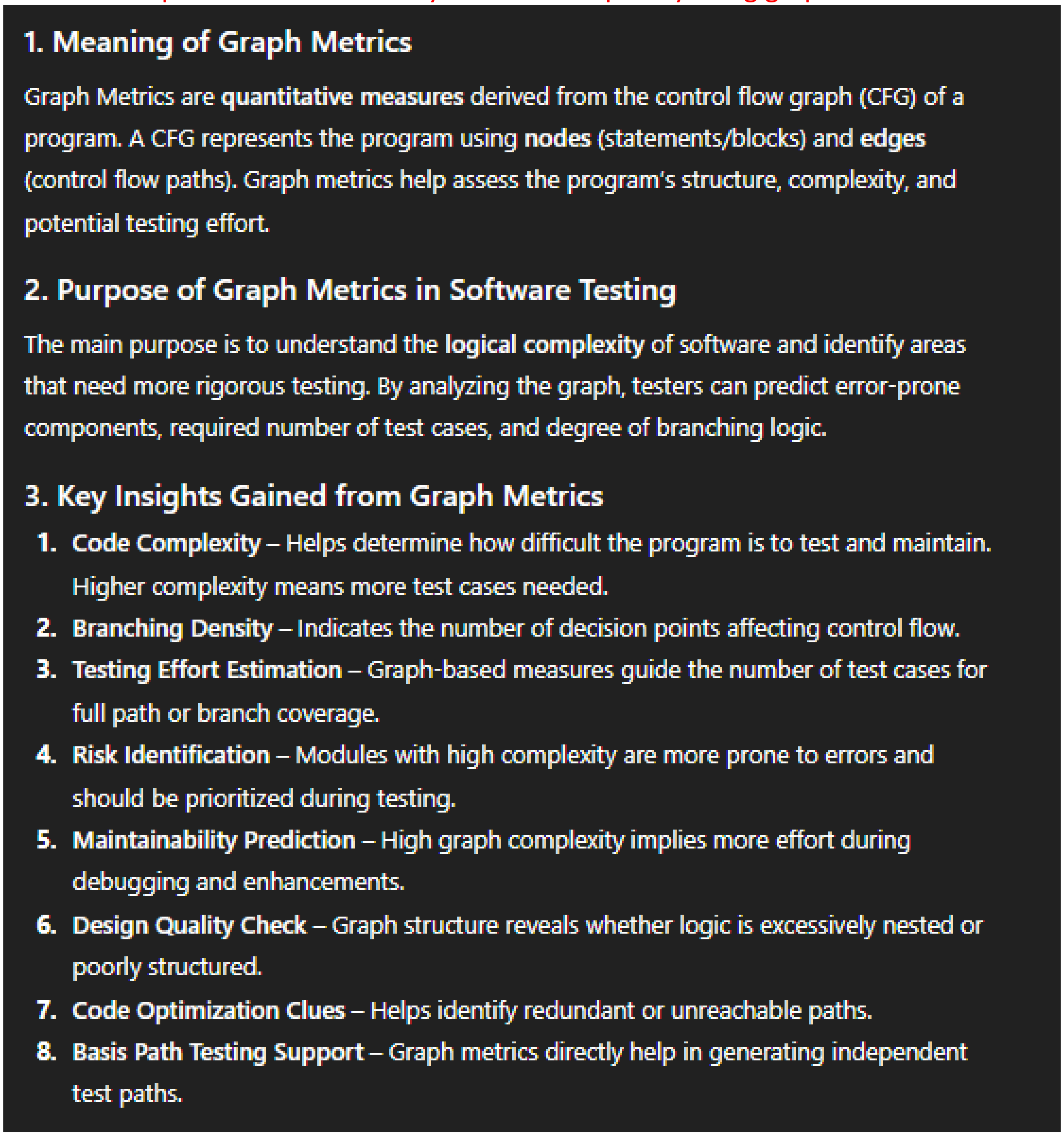
```python
if (a >= b):
```

Here, only **one operator** (> changed to ≥) is modified.
If a test case fails after this mutation, it means the test cases are strong enough to detect the faulty logic.



Point	Primary Mutant	Secondary Mutant
1. Definition	Mutant created with **one single change** in code.	Mutant created with **multiple combined changes.**
2. Complexity	Simple and easy to analyze.	More complex and harder to analyze.
3. Purpose	Basic evaluation of test case strength.	Advanced evaluation for stronger coverage.
4. Detection Difficulty	Easier to detect.	Harder to detect.
5. Number of Mutations	Only one mutation applied.	Two or more mutations applied.
6. Mutation Cost	Low cost and low effort.	Higher cost due to multiple mutations.
7. Use in Projects	Commonly used in practical mutation testing.	Rarely used; mostly academic or experimental.
8. Fault Representativeness	Represents single, simple faults.	Represents complex, combined faults.
9. Test Case Strength Indicator	Measures basic test strength.	Measures deep and extended test strength.

15. What are graph metrics and what insights can be gained from exploring Graph Metrics? Explain with an example how to calculate cyclomatic complexity using graph metrics.



1. Meaning of Graph Metrics

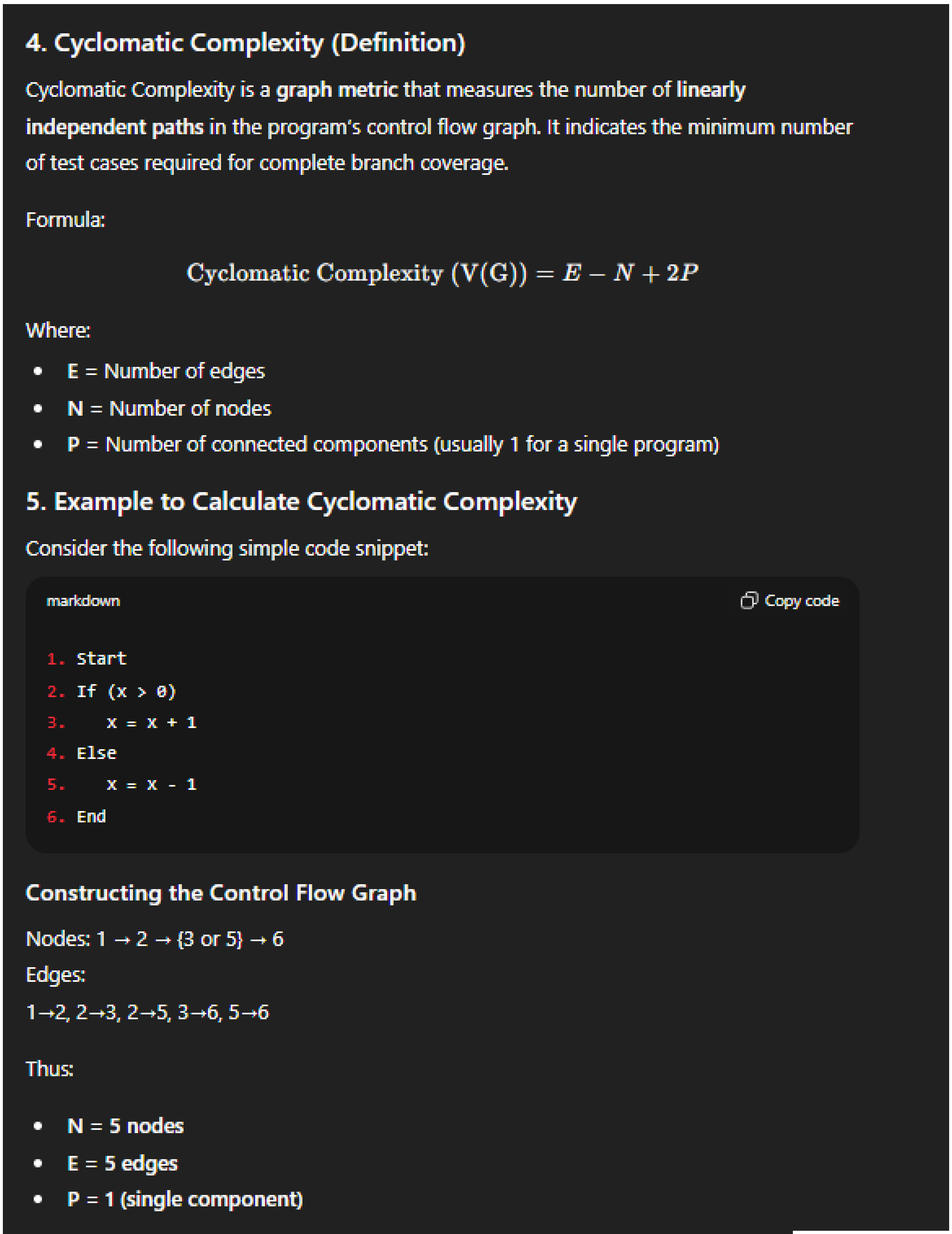
Graph Metrics are **quantitative measures** derived from the control flow graph (CFG) of a program. A CFG represents the program using **nodes** (statements/blocks) and **edges** (control flow paths). Graph metrics help assess the program's structure, complexity, and potential testing effort.

2. Purpose of Graph Metrics in Software Testing

The main purpose is to understand the **logical complexity** of software and identify areas that need more rigorous testing. By analyzing the graph, testers can predict error-prone components, required number of test cases, and degree of branching logic.

3. Key Insights Gained from Graph Metrics

1. **Code Complexity** – Helps determine how difficult the program is to test and maintain. Higher complexity means more test cases needed.
2. **Branching Density** – Indicates the number of decision points affecting control flow.
3. **Testing Effort Estimation** – Graph-based measures guide the number of test cases for full path or branch coverage.
4. **Risk Identification** – Modules with high complexity are more prone to errors and should be prioritized during testing.
5. **Maintainability Prediction** – High graph complexity implies more effort during debugging and enhancements.
6. **Design Quality Check** – Graph structure reveals whether logic is excessively nested or poorly structured.
7. **Code Optimization Clues** – Helps identify redundant or unreachable paths.
8. **Basis Path Testing Support** – Graph metrics directly help in generating independent test paths.



4. Cyclomatic Complexity (Definition)

Cyclomatic Complexity is a **graph metric** that measures the number of **linearly independent paths** in the program's control flow graph. It indicates the minimum number of test cases required for complete branch coverage.

Formula:

$$\text{Cyclomatic Complexity (V(G))} = E - N + 2P$$

Where:

- E = Number of edges
- N = Number of nodes
- P = Number of connected components (usually 1 for a single program)

5. Example to Calculate Cyclomatic Complexity

Consider the following simple code snippet:

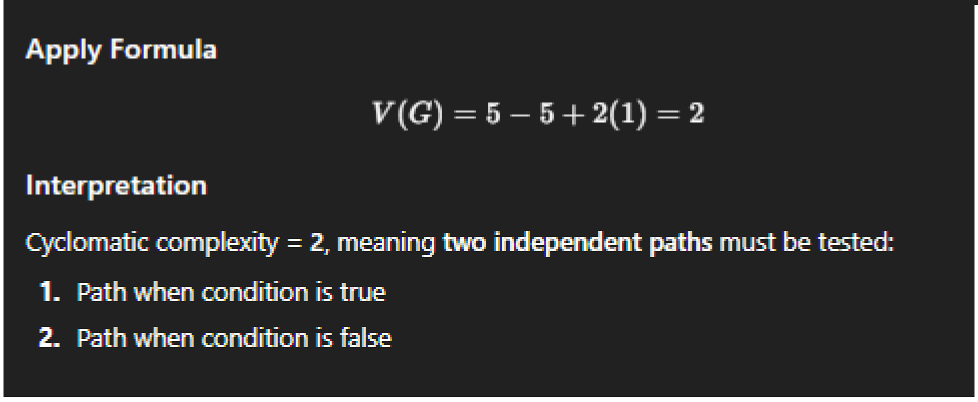
```markdown
1. Start
2. If (x > 0)
3.    x = x + 1
4. Else
5.    x = x - 1
6. End
```

Constructing the Control Flow Graph

Nodes: 1 → 2 → {3 or 5} → 6
Edges:
1→2, 2→3, 2→5, 3→6, 5→6

Thus:

- **N = 5 nodes**
- **E = 5 edges**
- **P = 1 (single component)**



Apply Formula

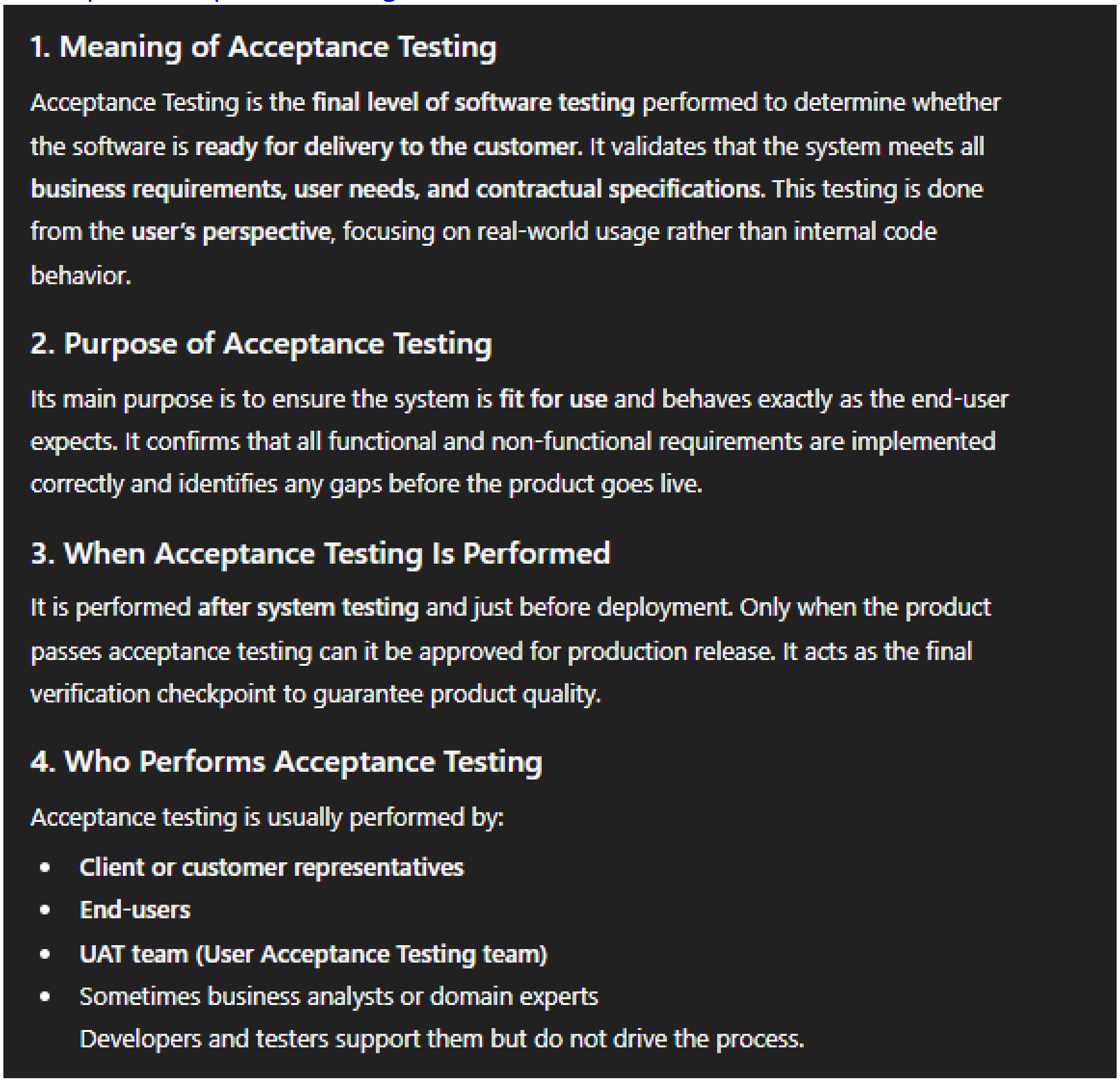
$$V(G) = 5 - 5 + 2(1) = 2$$

Interpretation

Cyclomatic complexity = 2, meaning **two independent paths** must be tested:

1. Path when condition is true
2. Path when condition is false

16. Explain Acceptance Testing.



1. Meaning of Acceptance Testing

Acceptance Testing is the **final level of software testing** performed to determine whether the software is **ready for delivery to the customer.** It validates that the system meets all **business requirements, user needs, and contractual specifications.** This testing is done from the **user's perspective**, focusing on real-world usage rather than internal code behavior.

2. Purpose of Acceptance Testing

Its main purpose is to ensure the system is **fit for use** and behaves exactly as the end-user expects. It confirms that all functional and non-functional requirements are implemented correctly and identifies any gaps before the product goes live.

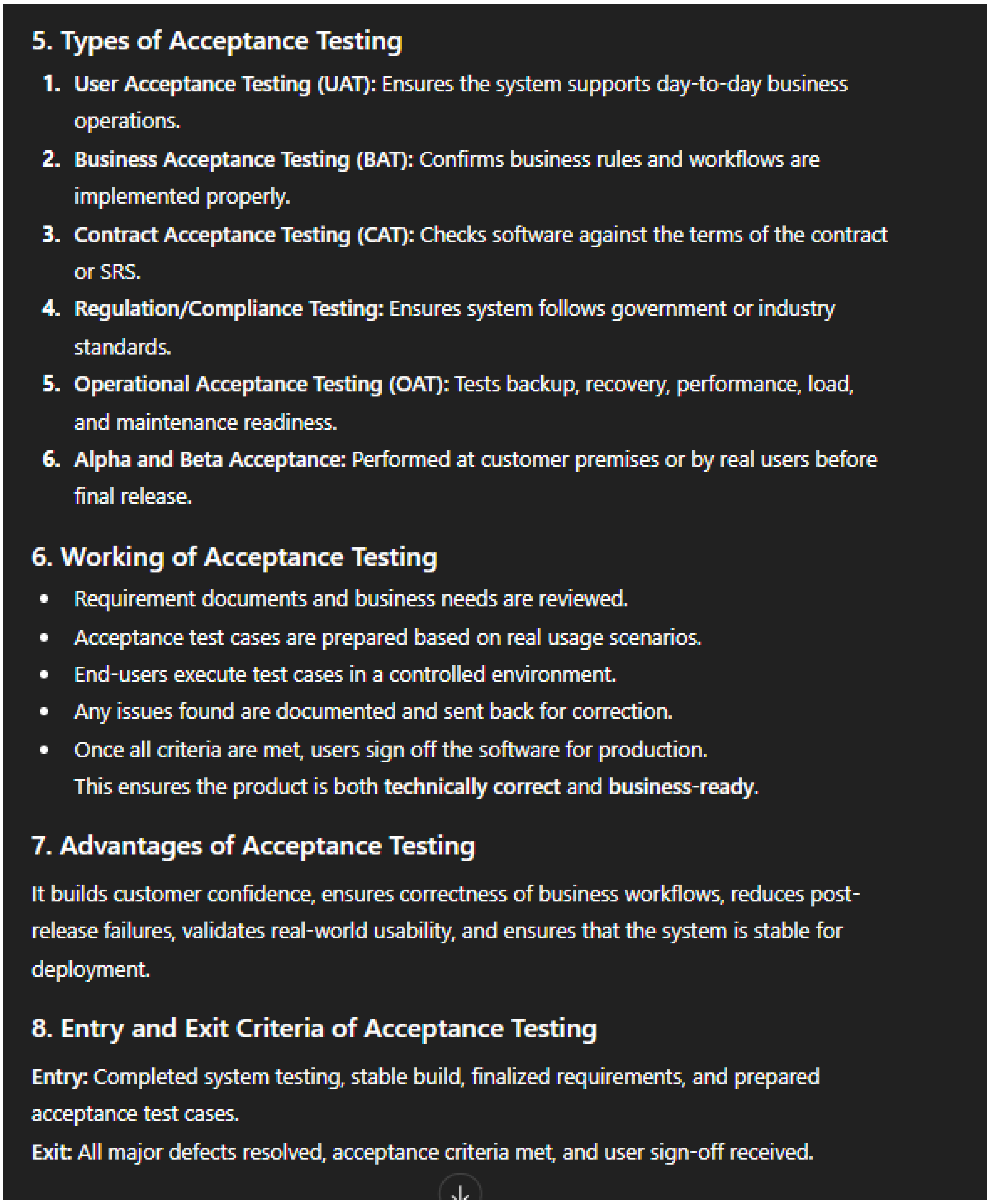
3. When Acceptance Testing Is Performed

It is performed **after system testing** and just before deployment. Only when the product passes acceptance testing can it be approved for production release. It acts as the final verification checkpoint to guarantee product quality.

4. Who Performs Acceptance Testing

Acceptance testing is usually performed by:

- **Client or customer representatives**
- **End-users**
- **UAT team (User Acceptance Testing team)**
- Sometimes business analysts or domain experts
  Developers and testers support them but do not drive the process.



5. Types of Acceptance Testing

1. **User Acceptance Testing (UAT):** Ensures the system supports day-to-day business operations.
2. **Business Acceptance Testing (BAT):** Confirms business rules and workflows are implemented properly.
3. **Contract Acceptance Testing (CAT):** Checks software against the terms of the contract or SRS.
4. **Regulation/Compliance Testing:** Ensures system follows government or industry standards.
5. **Operational Acceptance Testing (OAT):** Tests backup, recovery, performance, load, and maintenance readiness.
6. **Alpha and Beta Acceptance:** Performed at customer premises or by real users before final release.

6. Working of Acceptance Testing

- Requirement documents and business needs are reviewed.
- Acceptance test cases are prepared based on real usage scenarios.
- End-users execute test cases in a controlled environment.
- Any issues found are documented and sent back for correction.
- Once all criteria are met, users sign off the software for production.
  This ensures the product is both **technically correct** and **business-ready**.

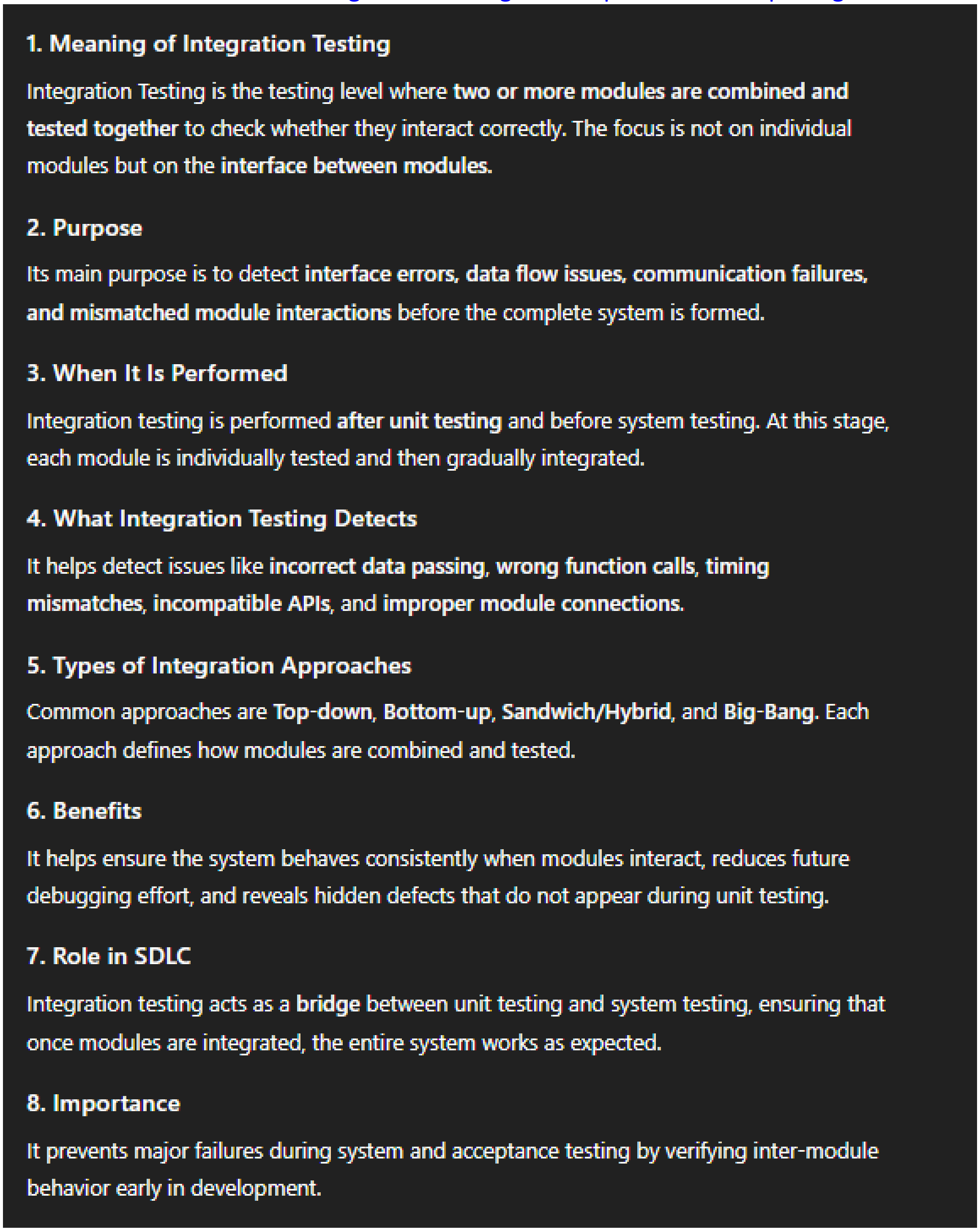
7. Advantages of Acceptance Testing

It builds customer confidence, ensures correctness of business workflows, reduces post-release failures, validates real-world usability, and ensures that the system is stable for deployment.

8. Entry and Exit Criteria of Acceptance Testing

Entry: Completed system testing, stable build, finalized requirements, and prepared acceptance test cases.
Exit: All major defects resolved, acceptance criteria met, and user sign-off received.

17. Write short note on — Integration Testing. Also explain Bottom-up Integration Testing with example.



1. Meaning of Integration Testing

Integration Testing is the testing level where **two or more modules are combined and tested together** to check whether they interact correctly. The focus is not on individual modules but on the **interface between modules.**

2. Purpose

Its main purpose is to detect **interface errors, data flow issues, communication failures, and mismatched module interactions** before the complete system is formed.

3. When It Is Performed

Integration testing is performed **after unit testing** and before system testing. At this stage, each module is individually tested and then gradually integrated.

4. What Integration Testing Detects

It helps detect issues like **incorrect data passing**, **wrong function calls**, **timing mismatches**, **incompatible APIs**, and **improper module connections.**

5. Types of Integration Approaches

Common approaches are **Top-down**, **Bottom-up**, **Sandwich/Hybrid**, and **Big-Bang.** Each approach defines how modules are combined and tested.

6. Benefits

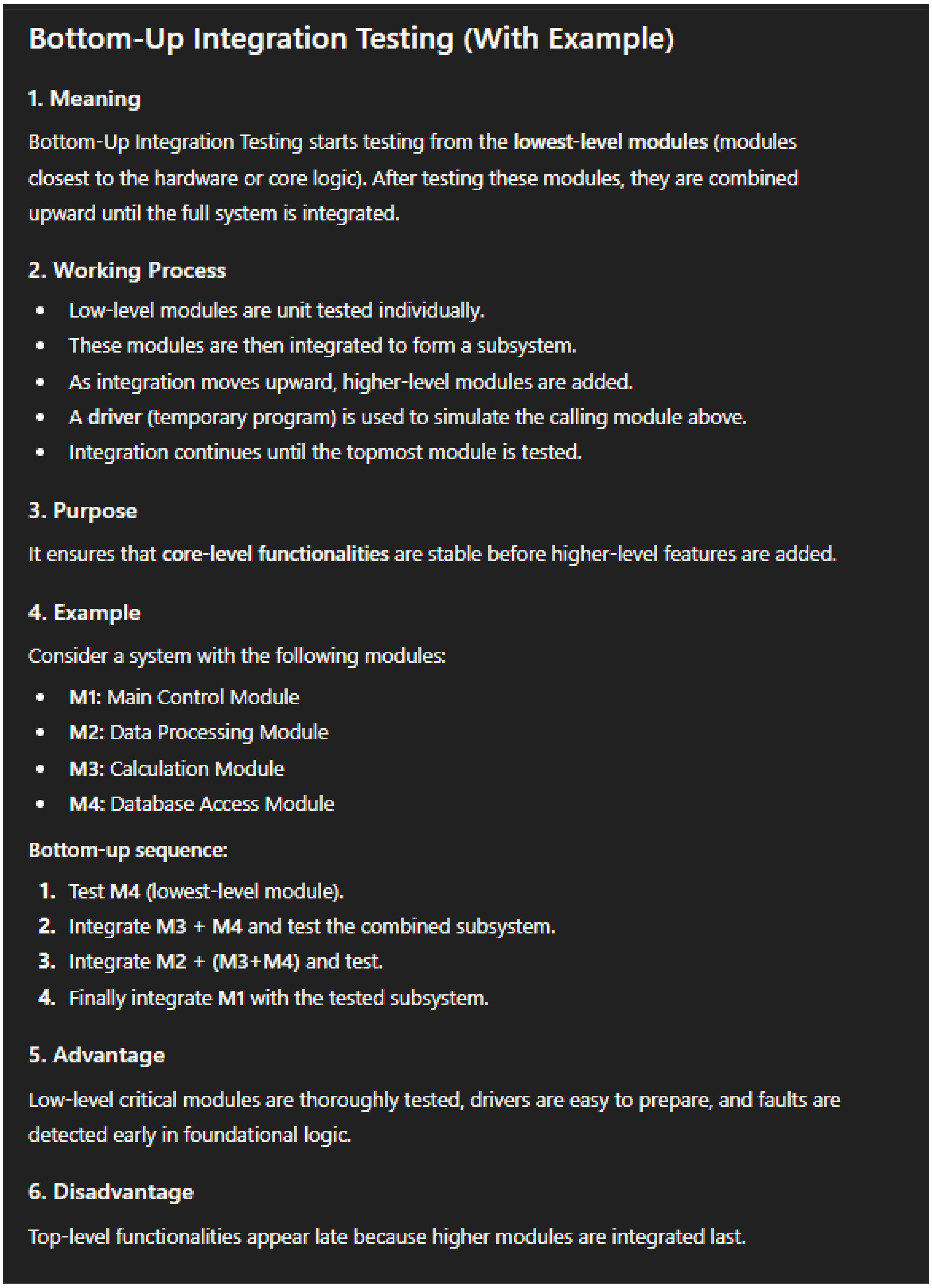
It helps ensure the system behaves consistently when modules interact, reduces future debugging effort, and reveals hidden defects that do not appear during unit testing.

7. Role in SDLC

Integration testing acts as a **bridge** between unit testing and system testing, ensuring that once modules are integrated, the entire system works as expected.

8. Importance

It prevents major failures during system and acceptance testing by verifying inter-module behavior early in development.



Bottom-Up Integration Testing (With Example)

1. Meaning

Bottom-Up Integration Testing starts testing from the **lowest-level modules** (modules closest to the hardware or core logic). After testing these modules, they are combined upward until the full system is integrated.

2. Working Process

- Low-level modules are unit tested individually.
- These modules are then integrated to form a subsystem.
- As integration moves upward, higher-level modules are added.
- A **driver** (temporary program) is used to simulate the calling module above.
- Integration continues until the topmost module is tested.

3. Purpose

It ensures that **core-level functionalities** are stable before higher-level features are added.

4. Example

Consider a system with the following modules:

- **M1:** Main Control Module
- **M2:** Data Processing Module
- **M3:** Calculation Module
- **M4:** Database Access Module

Bottom-up sequence:

1. Test **M4** (lowest-level module).
2. Integrate **M3** + **M4** and test the combined subsystem.
3. Integrate **M2** + **(M3+M4)** and test.
4. Finally integrate **M1** with the tested subsystem.

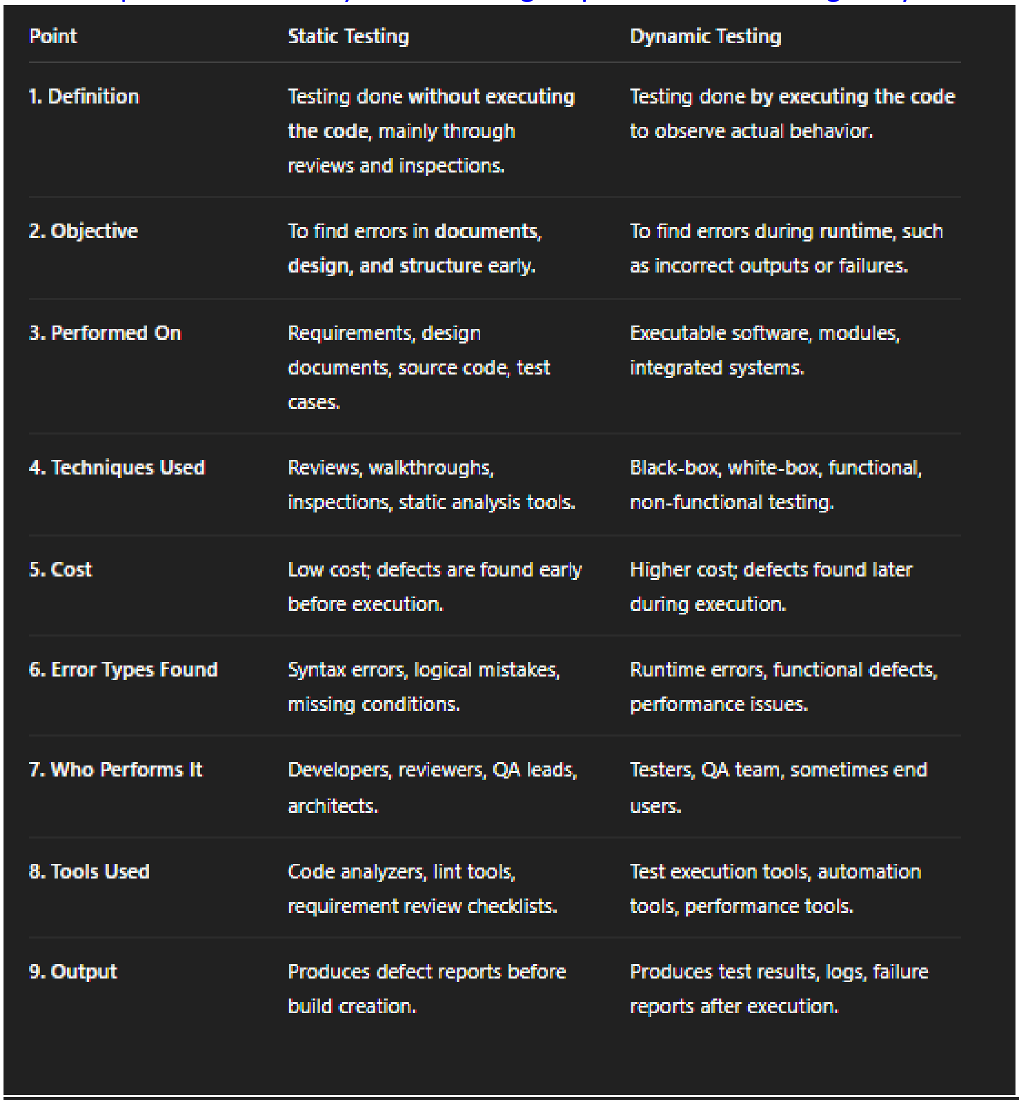
5. Advantage

Low-level critical modules are thoroughly tested, drivers are easy to prepare, and faults are detected early in foundational logic.

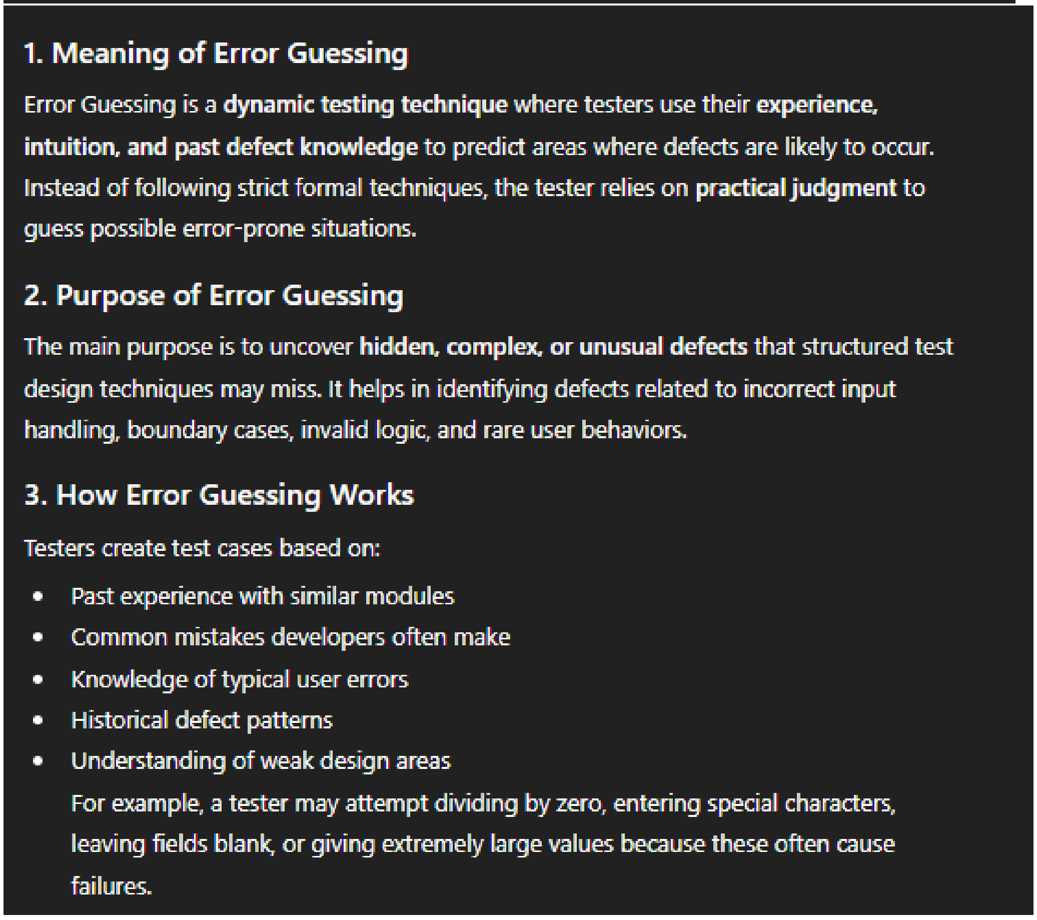
6. Disadvantage

Top-level functionalities appear late because higher modules are integrated last.

18. Compare Static and Dynamic Testing. Explain Error Guessing in Dynamic Testing.



Point	Static Testing	Dynamic Testing
1. Definition	Testing done **without executing the code**, mainly through reviews and inspections.	Testing done **by executing the code** to observe actual behavior.
2. Objective	To find errors in **documents, design, and structure** early.	To find errors during **runtime**, such as incorrect outputs or failures.
3. Performed On	Requirements, design documents, source code, test cases.	Executable software, modules, integrated systems.
4. Techniques Used	Reviews, walkthroughs, inspections, static analysis tools.	Black-box, white-box, functional, non-functional testing.
5. Cost	Low cost; defects are found early before execution.	Higher cost; defects found later during execution.
6. Error Types Found	Syntax errors, logical mistakes, missing conditions.	Runtime errors, functional defects, performance issues.
7. Who Performs It	Developers, reviewers, QA leads, architects.	Testers, QA team, sometimes end users.
8. Tools Used	Code analyzers, lint tools, requirement review checklists.	Test execution tools, automation tools, performance tools.
9. Output	Produces defect reports before build creation.	Produces test results, logs, failure reports after execution.



1. Meaning of Error Guessing

Error Guessing is a **dynamic testing technique** where testers use their **experience, intuition, and past defect knowledge** to predict areas where defects are likely to occur. Instead of following strict formal techniques, the tester relies on **practical judgment** to guess possible error-prone situations.

2. Purpose of Error Guessing

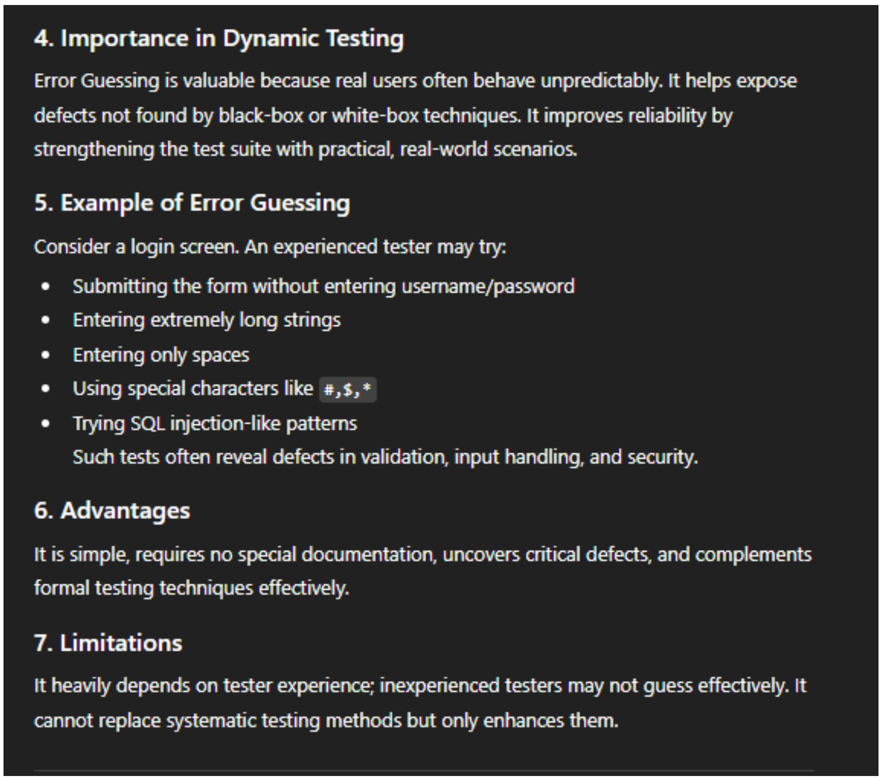
The main purpose is to uncover **hidden, complex, or unusual defects** that structured test design techniques may miss. It helps in identifying defects related to incorrect input handling, boundary cases, invalid logic, and rare user behaviors.

3. How Error Guessing Works

Testers create test cases based on:

- Past experience with similar modules
- Common mistakes developers often make
- Knowledge of typical user errors
- Historical defect patterns
- Understanding of weak design areas

  For example, a tester may attempt dividing by zero, entering special characters, leaving fields blank, or giving extremely large values because these often cause failures.



4. Importance in Dynamic Testing

Error Guessing is valuable because real users often behave unpredictably. It helps expose defects not found by black-box or white-box techniques. It improves reliability by strengthening the test suite with practical, real-world scenarios.

5. Example of Error Guessing

Consider a login screen. An experienced tester may try:

- Submitting the form without entering username/password
- Entering extremely long strings
- Entering only spaces
- Using special characters like `#,$,*`
- Trying SQL injection-like patterns
  Such tests often reveal defects in validation, input handling, and security.

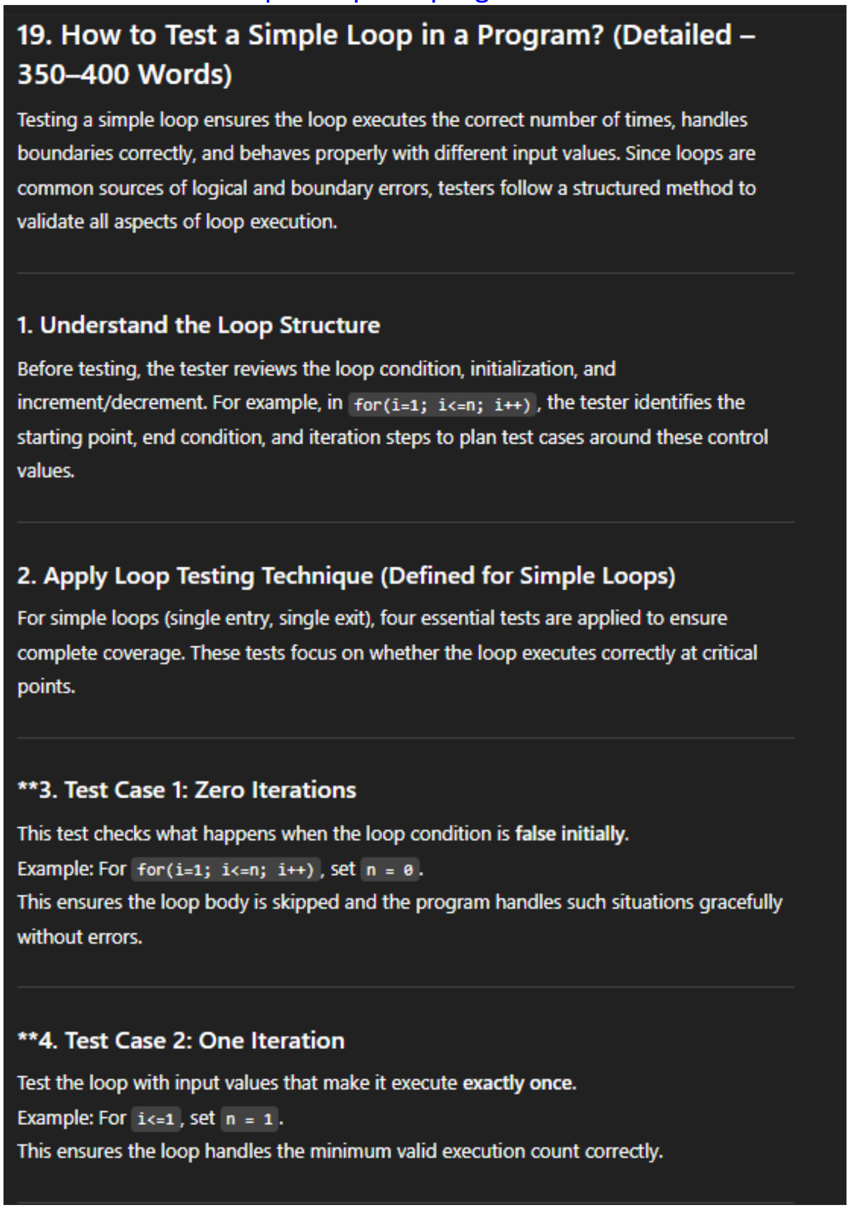
6. Advantages

It is simple, requires no special documentation, uncovers critical defects, and complements formal testing techniques effectively.

7. Limitations

It heavily depends on tester experience; inexperienced testers may not guess effectively. It cannot replace systematic testing methods but only enhances them.

19. How to test simple loop in a program?



19. How to Test a Simple Loop in a Program? (Detailed – 350–400 Words)

Testing a simple loop ensures the loop executes the correct number of times, handles boundaries correctly, and behaves properly with different input values. Since loops are common sources of logical and boundary errors, testers follow a structured method to validate all aspects of loop execution.

---

1. Understand the Loop Structure

Before testing, the tester reviews the loop condition, initialization, and increment/decrement. For example, in `for(i=1; i<=n; i++)`, the tester identifies the starting point, end condition, and iteration steps to plan test cases around these control values.

---

2. Apply Loop Testing Technique (Defined for Simple Loops)

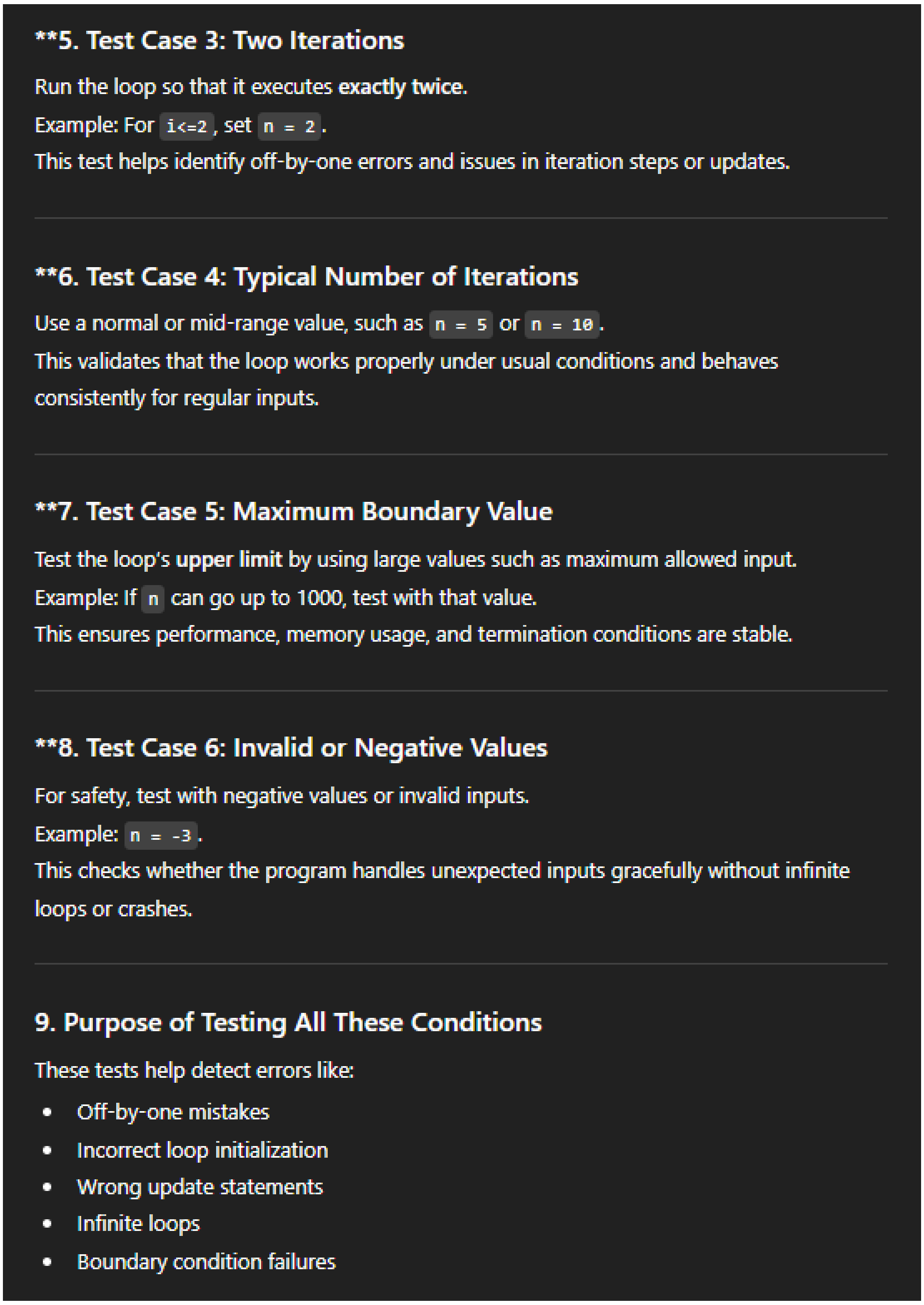
For simple loops (single entry, single exit), four essential tests are applied to ensure complete coverage. These tests focus on whether the loop executes correctly at critical points.

**3. Test Case 1: Zero Iterations

This test checks what happens when the loop condition is **false initially**.
Example: For `for(i=1; i<=n; i++)`, set `n = 0`.
This ensures the loop body is skipped and the program handles such situations gracefully without errors.

**4. Test Case 2: One Iteration

Test the loop with input values that make it execute **exactly once**.
Example: For `i<=1`, set `n = 1`.
This ensures the loop handles the minimum valid execution count correctly.



**5. Test Case 3: Two Iterations

Run the loop so that it executes **exactly twice.**

Example: For `i<=2`, set `n = 2`.

This test helps identify off-by-one errors and issues in iteration steps or updates.

**6. Test Case 4: Typical Number of Iterations

Use a normal or mid-range value, such as `n = 5` or `n = 10`.

This validates that the loop works properly under usual conditions and behaves consistently for regular inputs.

**7. Test Case 5: Maximum Boundary Value

Test the loop's **upper limit** by using large values such as maximum allowed input.

Example: If `n` can go up to 1000, test with that value.

This ensures performance, memory usage, and termination conditions are stable.

**8. Test Case 6: Invalid or Negative Values

For safety, test with negative values or invalid inputs.

Example: `n = -3`.

This checks whether the program handles unexpected inputs gracefully without infinite loops or crashes.

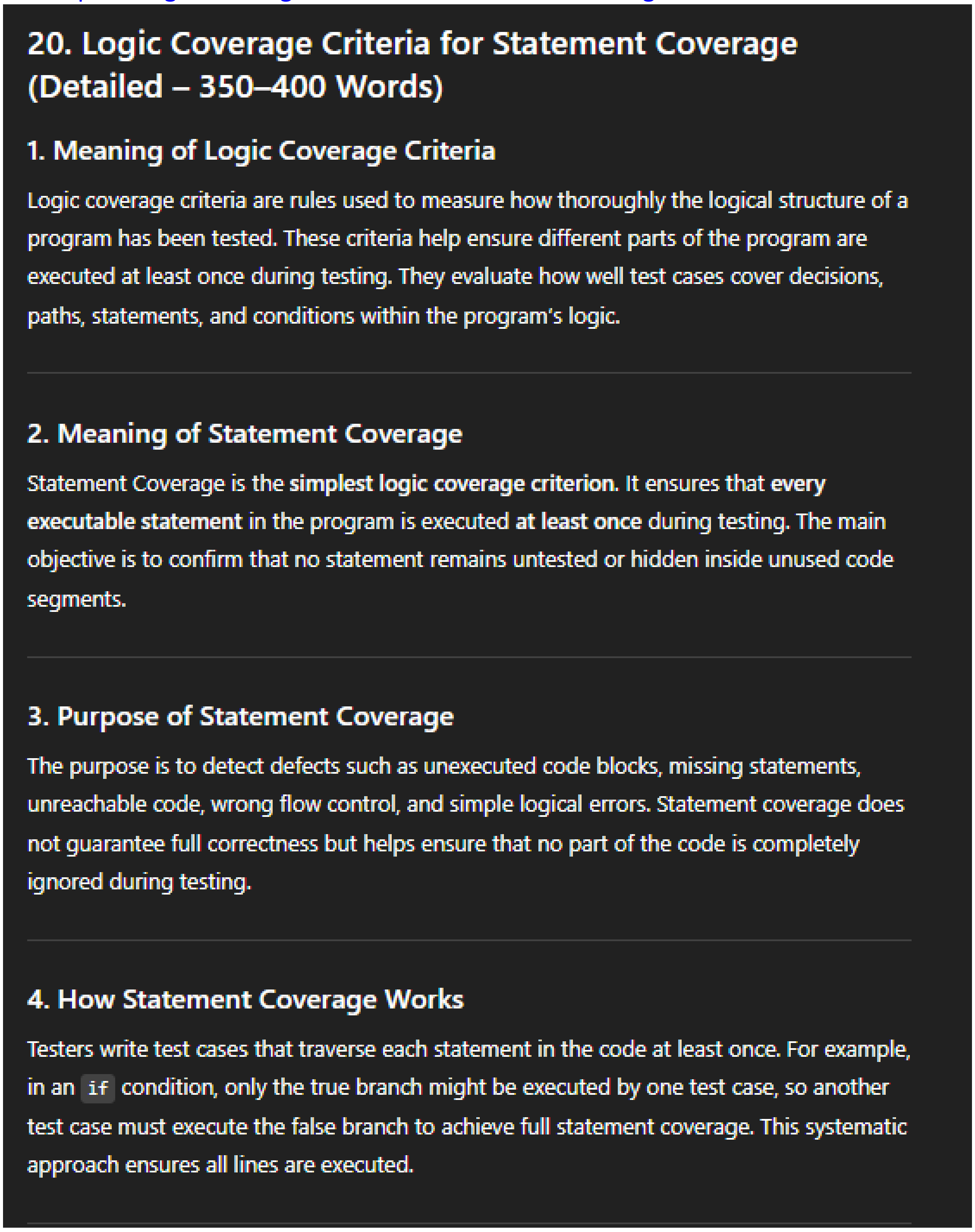
---

9. Purpose of Testing All These Conditions

These tests help detect errors like:

- Off-by-one mistakes
- Incorrect loop initialization
- Wrong update statements
- Infinite loops
- Boundary condition failures

20. Explain Logic Coverage Criteria for Statement Coverage.



20. Logic Coverage Criteria for Statement Coverage (Detailed – 350–400 Words)

1. Meaning of Logic Coverage Criteria

Logic coverage criteria are rules used to measure how thoroughly the logical structure of a program has been tested. These criteria help ensure different parts of the program are executed at least once during testing. They evaluate how well test cases cover decisions, paths, statements, and conditions within the program's logic.

---

2. Meaning of Statement Coverage

Statement Coverage is the **simplest logic coverage criterion**. It ensures that **every executable statement** in the program is executed **at least once** during testing. The main objective is to confirm that no statement remains untested or hidden inside unused code segments.

---

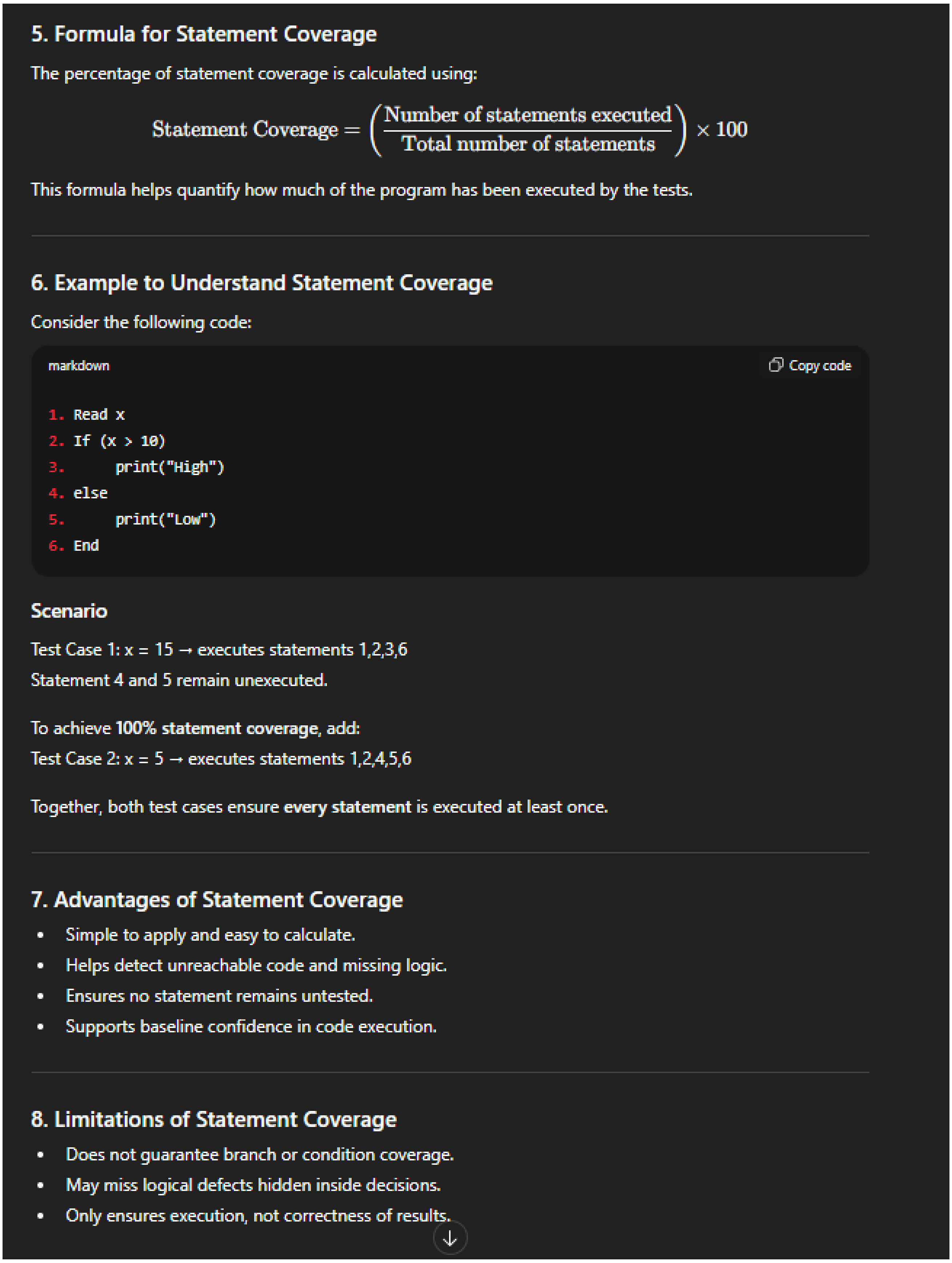
3. Purpose of Statement Coverage

The purpose is to detect defects such as unexecuted code blocks, missing statements, unreachable code, wrong flow control, and simple logical errors. Statement coverage does not guarantee full correctness but helps ensure that no part of the code is completely ignored during testing.

---

4. How Statement Coverage Works

Testers write test cases that traverse each statement in the code at least once. For example, in an `if` condition, only the true branch might be executed by one test case, so another test case must execute the false branch to achieve full statement coverage. This systematic approach ensures all lines are executed.



5. Formula for Statement Coverage

The percentage of statement coverage is calculated using:

$$\text{Statement Coverage} = \left(\frac{\text{Number of statements executed}}{\text{Total number of statements}} \right) \times 100$$

This formula helps quantify how much of the program has been executed by the tests.

---

6. Example to Understand Statement Coverage

Consider the following code:

```markdown
1. Read x
2. If (x > 10)
3.      print("High")
4. else
5.      print("Low")
6. End
```

Scenario

Test Case 1: x = 15 → executes statements 1,2,3,6
Statement 4 and 5 remain unexecuted.

To achieve **100% statement coverage**, add:
Test Case 2: x = 5 → executes statements 1,2,4,5,6

Together, both test cases ensure **every statement** is executed at least once.

---

7. Advantages of Statement Coverage

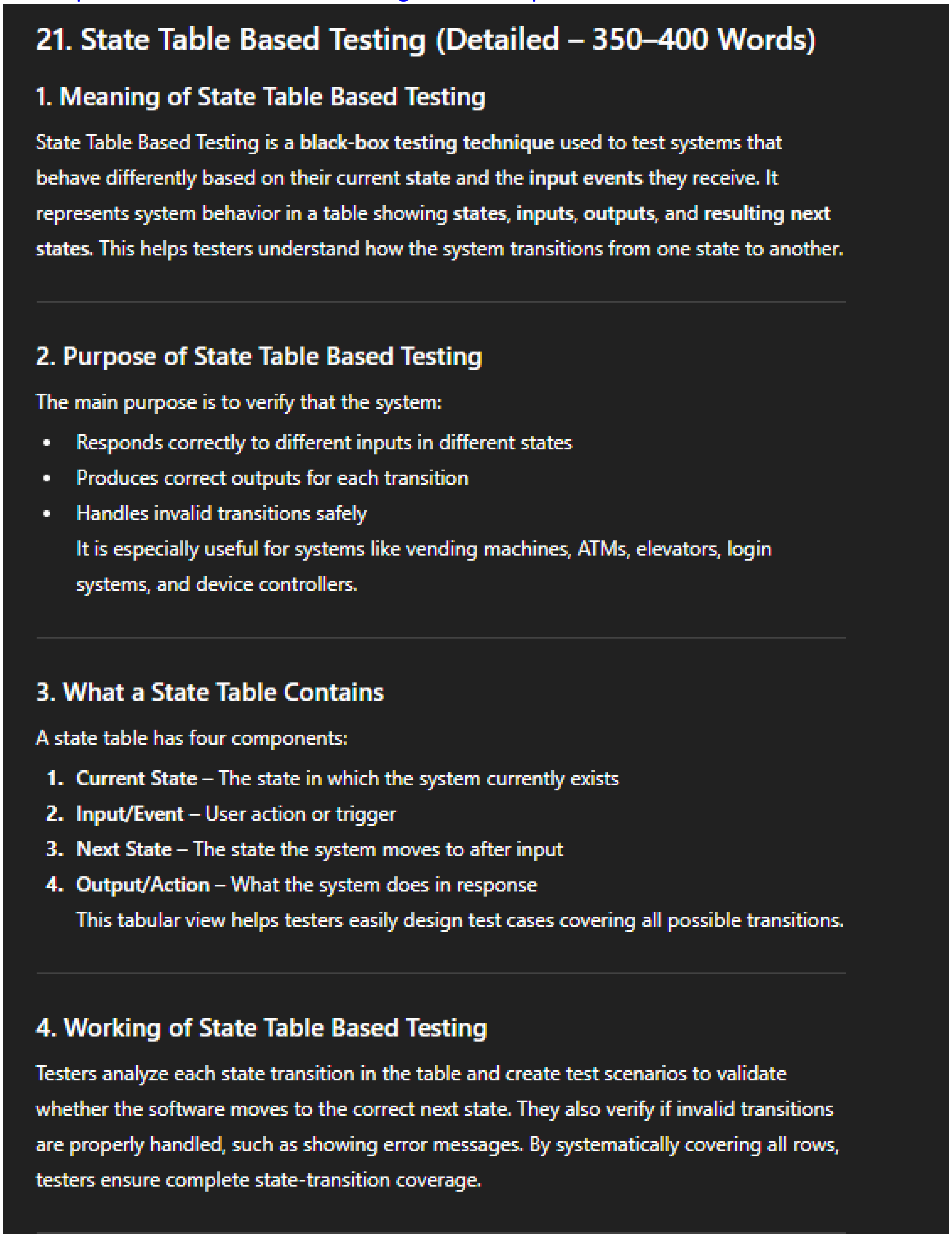
- Simple to apply and easy to calculate.
- Helps detect unreachable code and missing logic.
- Ensures no statement remains untested.
- Supports baseline confidence in code execution.

---

8. Limitations of Statement Coverage

- Does not guarantee branch or condition coverage.
- May miss logical defects hidden inside decisions.
- Only ensures execution, not correctness of results.

21. Explain State Table Based testing with example.



21. State Table Based Testing (Detailed – 350–400 Words)

1. Meaning of State Table Based Testing

State Table Based Testing is a **black-box testing technique** used to test systems that behave differently based on their current **state** and the **input events** they receive. It represents system behavior in a table showing **states**, **inputs**, **outputs**, and **resulting next states**. This helps testers understand how the system transitions from one state to another.

2. Purpose of State Table Based Testing

The main purpose is to verify that the system:

- Responds correctly to different inputs in different states
- Produces correct outputs for each transition
- Handles invalid transitions safely
  It is especially useful for systems like vending machines, ATMs, elevators, login systems, and device controllers.

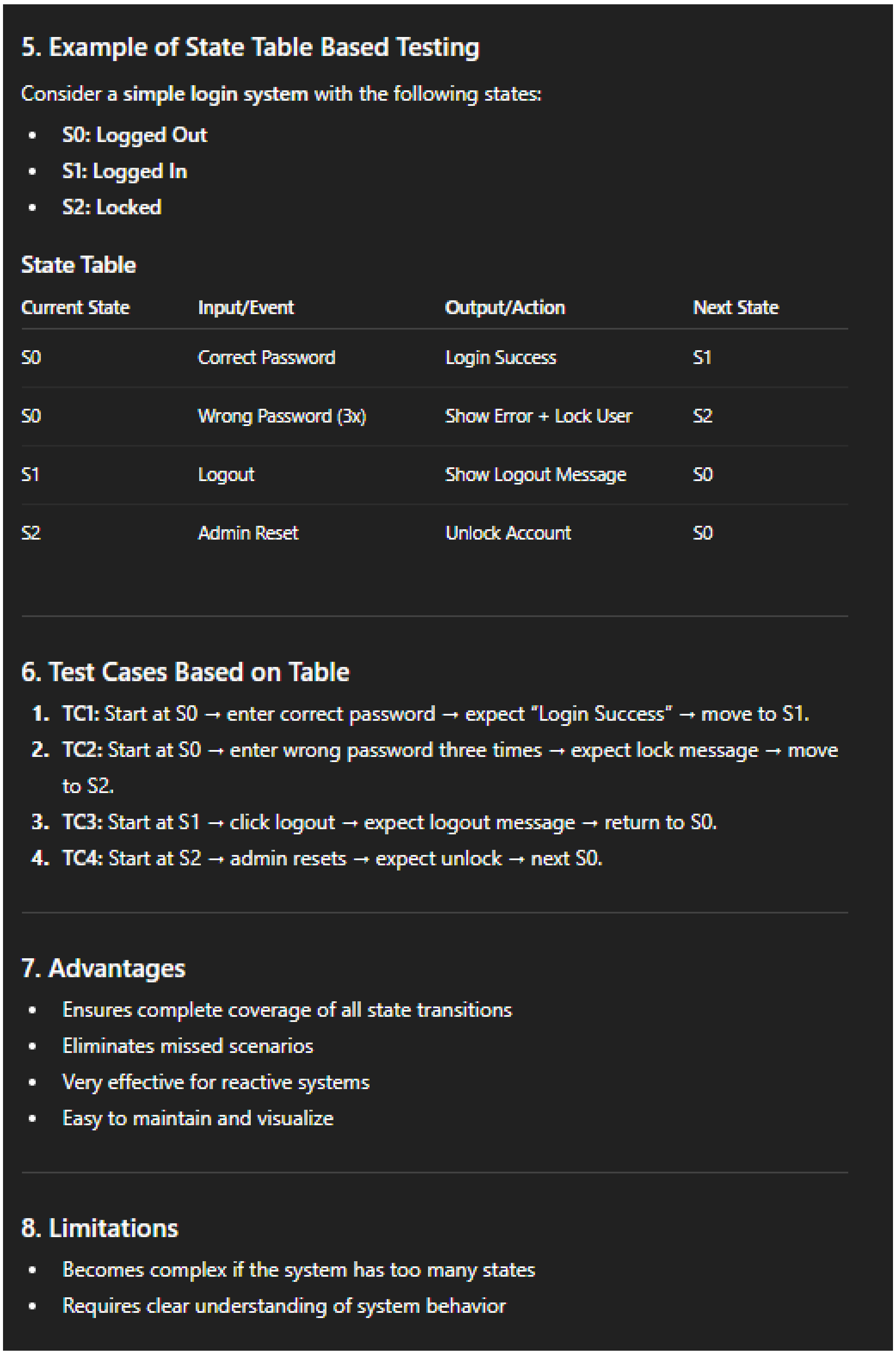
3. What a State Table Contains

A state table has four components:

1. **Current State** – The state in which the system currently exists
2. **Input/Event** – User action or trigger
3. **Next State** – The state the system moves to after input
4. **Output/Action** – What the system does in response
   This tabular view helps testers easily design test cases covering all possible transitions.

4. Working of State Table Based Testing

Testers analyze each state transition in the table and create test scenarios to validate whether the software moves to the correct next state. They also verify if invalid transitions are properly handled, such as showing error messages. By systematically covering all rows, testers ensure complete state-transition coverage.



5. Example of State Table Based Testing

Consider a **simple login system** with the following states:

- **S0: Logged Out**
- **S1: Logged In**
- **S2: Locked**

State Table

Current State	Input/Event	Output/Action	Next State
S0	Correct Password	Login Success	S1
S0	Wrong Password (3x)	Show Error + Lock User	S2
S1	Logout	Show Logout Message	S0
S2	Admin Reset	Unlock Account	S0

---

6. Test Cases Based on Table

1. **TC1:** Start at S0 → enter correct password → expect "Login Success" → move to S1.
2. **TC2:** Start at S0 → enter wrong password three times → expect lock message → move to S2.
3. **TC3:** Start at S1 → click logout → expect logout message → return to S0.
4. **TC4:** Start at S2 → admin resets → expect unlock → next S0.

---

7. Advantages

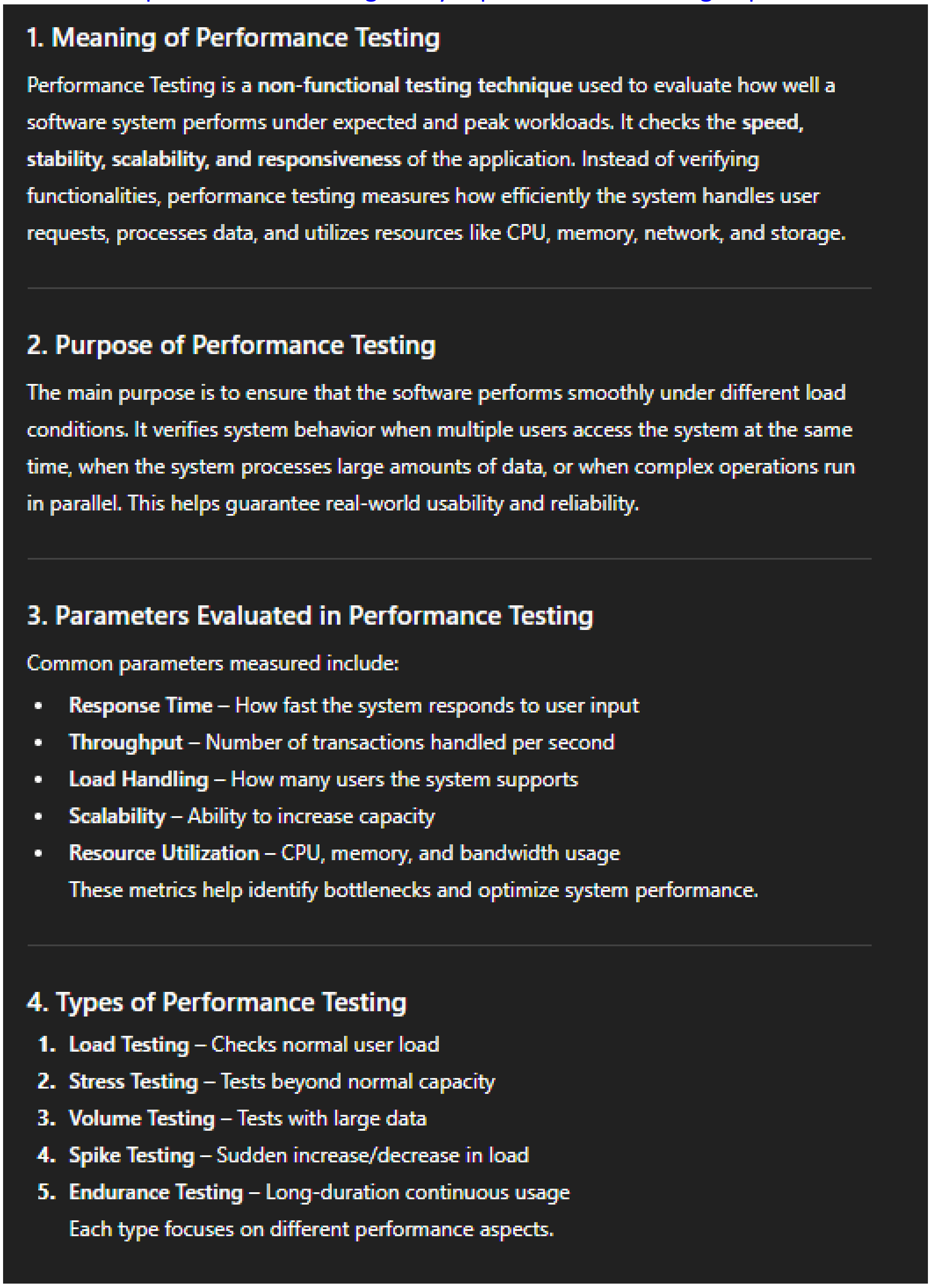
- Ensures complete coverage of all state transitions
- Eliminates missed scenarios
- Very effective for reactive systems
- Easy to maintain and visualize

---

8. Limitations

- Becomes complex if the system has too many states
- Requires clear understanding of system behavior

22. What is performance testing? Why is performance testing important?



1. Meaning of Performance Testing

Performance Testing is a **non-functional testing technique** used to evaluate how well a software system performs under expected and peak workloads. It checks the **speed, stability, scalability, and responsiveness** of the application. Instead of verifying functionalities, performance testing measures how efficiently the system handles user requests, processes data, and utilizes resources like CPU, memory, network, and storage.

---

2. Purpose of Performance Testing

The main purpose is to ensure that the software performs smoothly under different load conditions. It verifies system behavior when multiple users access the system at the same time, when the system processes large amounts of data, or when complex operations run in parallel. This helps guarantee real-world usability and reliability.

---

3. Parameters Evaluated in Performance Testing

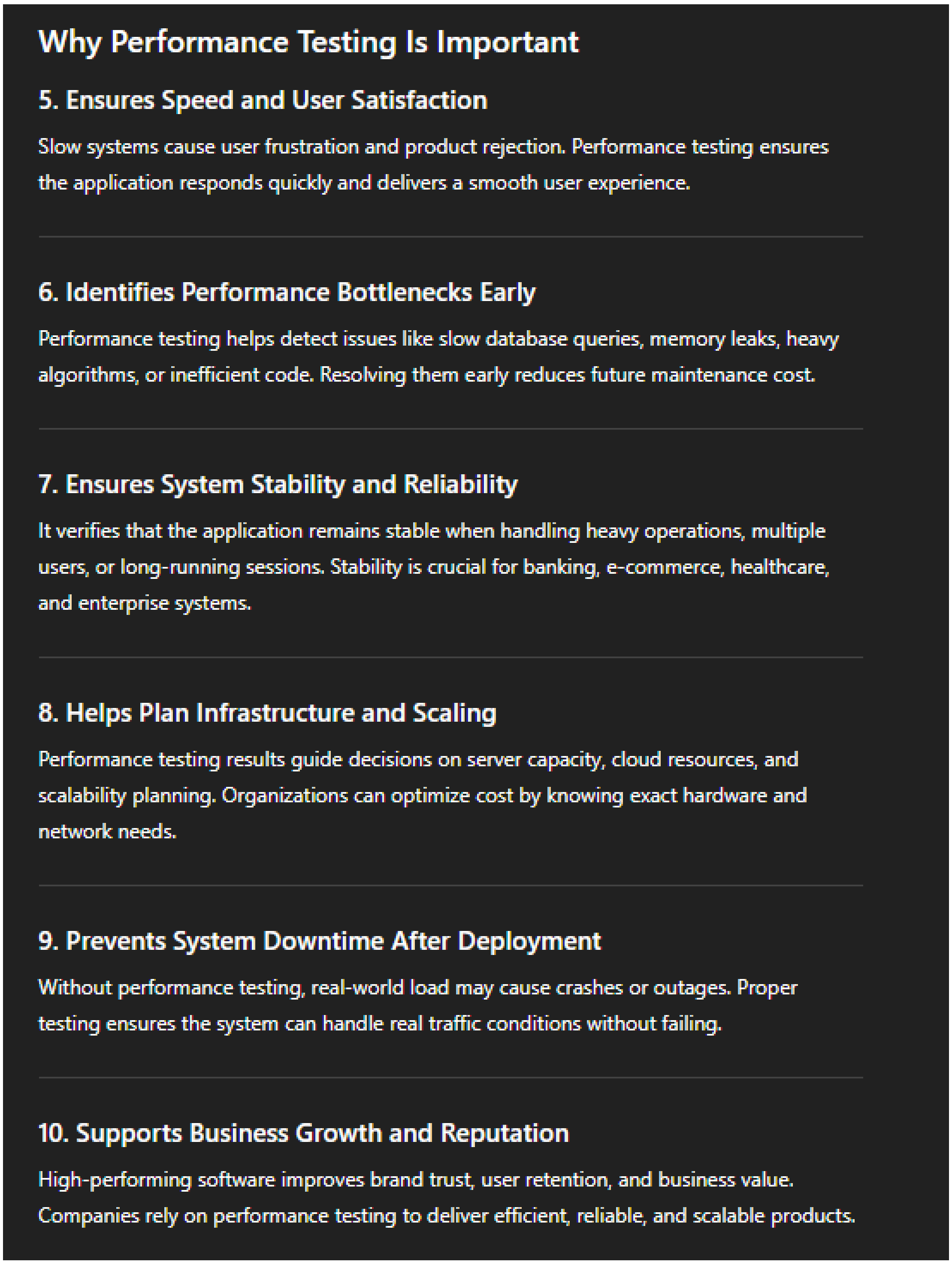
Common parameters measured include:

- **Response Time** – How fast the system responds to user input
- **Throughput** – Number of transactions handled per second
- **Load Handling** – How many users the system supports
- **Scalability** – Ability to increase capacity
- **Resource Utilization** – CPU, memory, and bandwidth usage
  These metrics help identify bottlenecks and optimize system performance.

---

4. Types of Performance Testing

1. **Load Testing** – Checks normal user load
2. **Stress Testing** – Tests beyond normal capacity
3. **Volume Testing** – Tests with large data
4. **Spike Testing** – Sudden increase/decrease in load
5. **Endurance Testing** – Long-duration continuous usage
   Each type focuses on different performance aspects.



Why Performance Testing Is Important

5. Ensures Speed and User Satisfaction

Slow systems cause user frustration and product rejection. Performance testing ensures the application responds quickly and delivers a smooth user experience.

---

6. Identifies Performance Bottlenecks Early

Performance testing helps detect issues like slow database queries, memory leaks, heavy algorithms, or inefficient code. Resolving them early reduces future maintenance cost.

---

7. Ensures System Stability and Reliability

It verifies that the application remains stable when handling heavy operations, multiple users, or long-running sessions. Stability is crucial for banking, e-commerce, healthcare, and enterprise systems.

---

8. Helps Plan Infrastructure and Scaling

Performance testing results guide decisions on server capacity, cloud resources, and scalability planning. Organizations can optimize cost by knowing exact hardware and network needs.

---

9. Prevents System Downtime After Deployment

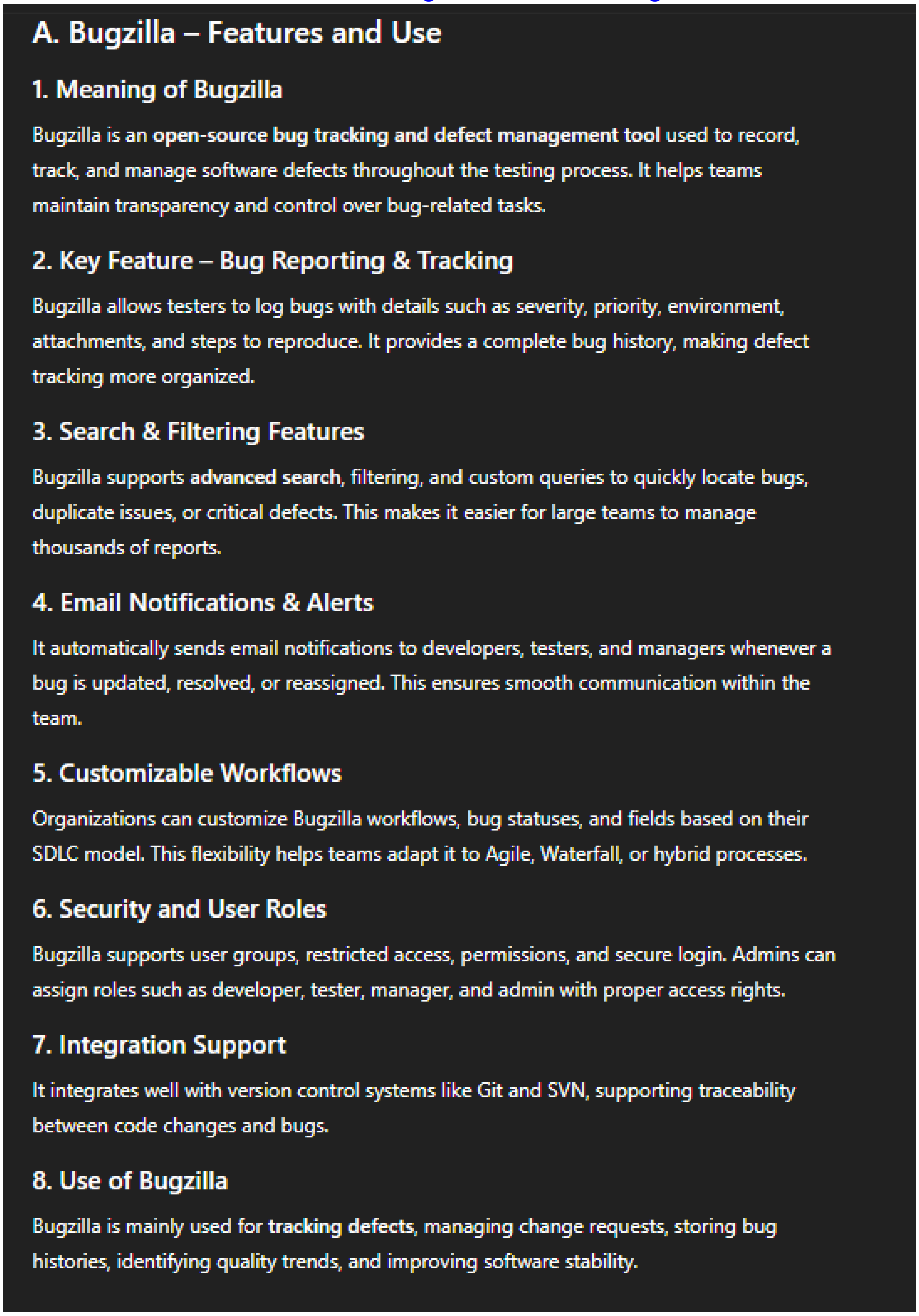
Without performance testing, real-world load may cause crashes or outages. Proper testing ensures the system can handle real traffic conditions without failing.

---

10. Supports Business Growth and Reputation

High-performing software improves brand trust, user retention, and business value. Companies rely on performance testing to deliver efficient, reliable, and scalable products.

23. Discuss the features and use of Bugzilla and JIRA Testing Tool.



A. Bugzilla – Features and Use

1. Meaning of Bugzilla

Bugzilla is an **open-source bug tracking and defect management tool** used to record, track, and manage software defects throughout the testing process. It helps teams maintain transparency and control over bug-related tasks.

2. Key Feature – Bug Reporting & Tracking

Bugzilla allows testers to log bugs with details such as severity, priority, environment, attachments, and steps to reproduce. It provides a complete bug history, making defect tracking more organized.

3. Search & Filtering Features

Bugzilla supports **advanced search**, filtering, and custom queries to quickly locate bugs, duplicate issues, or critical defects. This makes it easier for large teams to manage thousands of reports.

4. Email Notifications & Alerts

It automatically sends email notifications to developers, testers, and managers whenever a bug is updated, resolved, or reassigned. This ensures smooth communication within the team.

5. Customizable Workflows

Organizations can customize Bugzilla workflows, bug statuses, and fields based on their SDLC model. This flexibility helps teams adapt it to Agile, Waterfall, or hybrid processes.

6. Security and User Roles

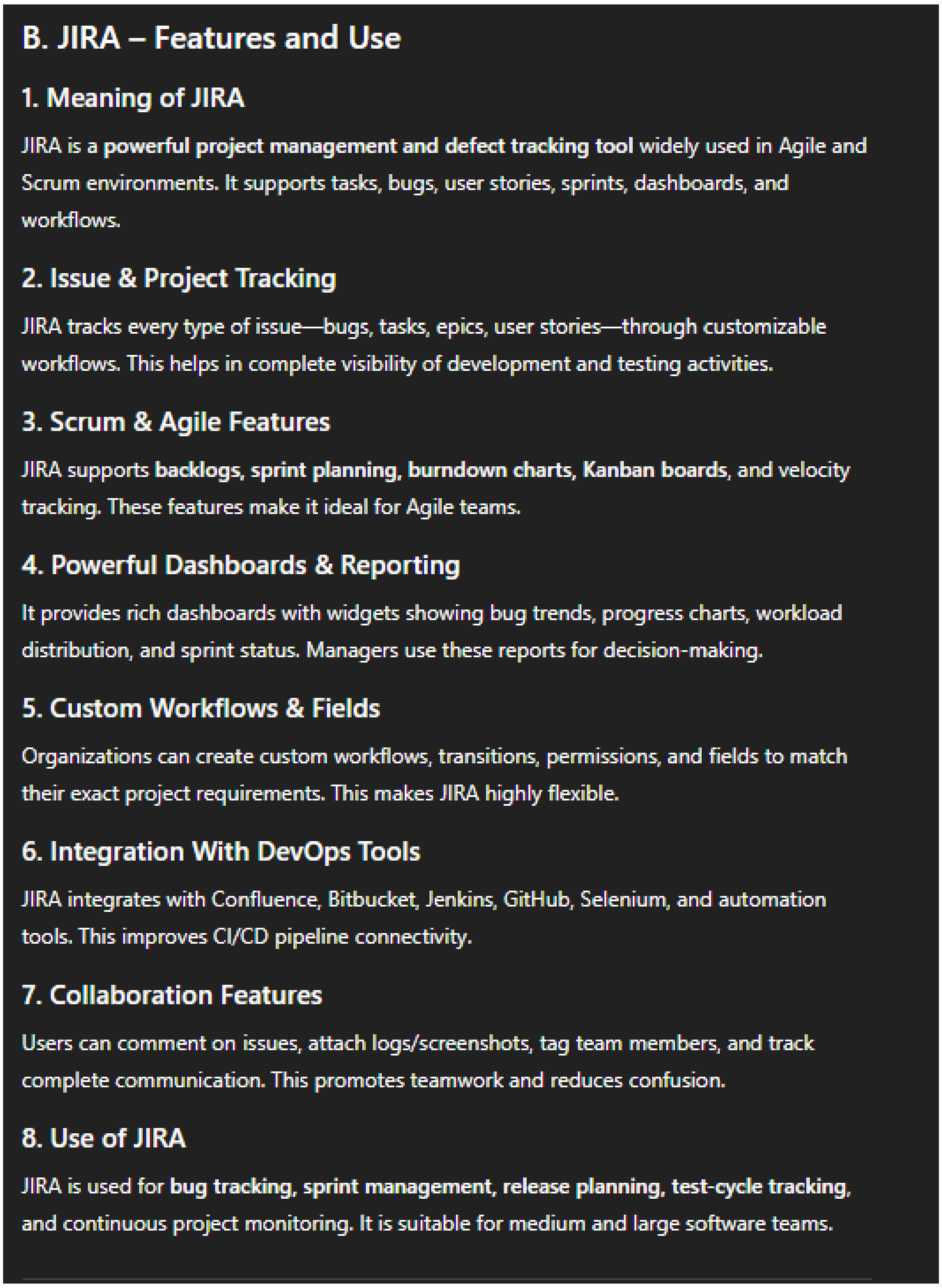
Bugzilla supports user groups, restricted access, permissions, and secure login. Admins can assign roles such as developer, tester, manager, and admin with proper access rights.

7. Integration Support

It integrates well with version control systems like Git and SVN, supporting traceability between code changes and bugs.

8. Use of Bugzilla

Bugzilla is mainly used for **tracking defects**, managing change requests, storing bug histories, identifying quality trends, and improving software stability.



B. JIRA – Features and Use

1. Meaning of JIRA

JIRA is a **powerful project management and defect tracking tool** widely used in Agile and Scrum environments. It supports tasks, bugs, user stories, sprints, dashboards, and workflows.

2. Issue & Project Tracking

JIRA tracks every type of issue—bugs, tasks, epics, user stories—through customizable workflows. This helps in complete visibility of development and testing activities.

3. Scrum & Agile Features

JIRA supports **backlogs, sprint planning, burndown charts, Kanban boards**, and velocity tracking. These features make it ideal for Agile teams.

4. Powerful Dashboards & Reporting

It provides rich dashboards with widgets showing bug trends, progress charts, workload distribution, and sprint status. Managers use these reports for decision-making.

5. Custom Workflows & Fields

Organizations can create custom workflows, transitions, permissions, and fields to match their exact project requirements. This makes JIRA highly flexible.

6. Integration With DevOps Tools

JIRA integrates with Confluence, Bitbucket, Jenkins, GitHub, Selenium, and automation tools. This improves CI/CD pipeline connectivity.

7. Collaboration Features

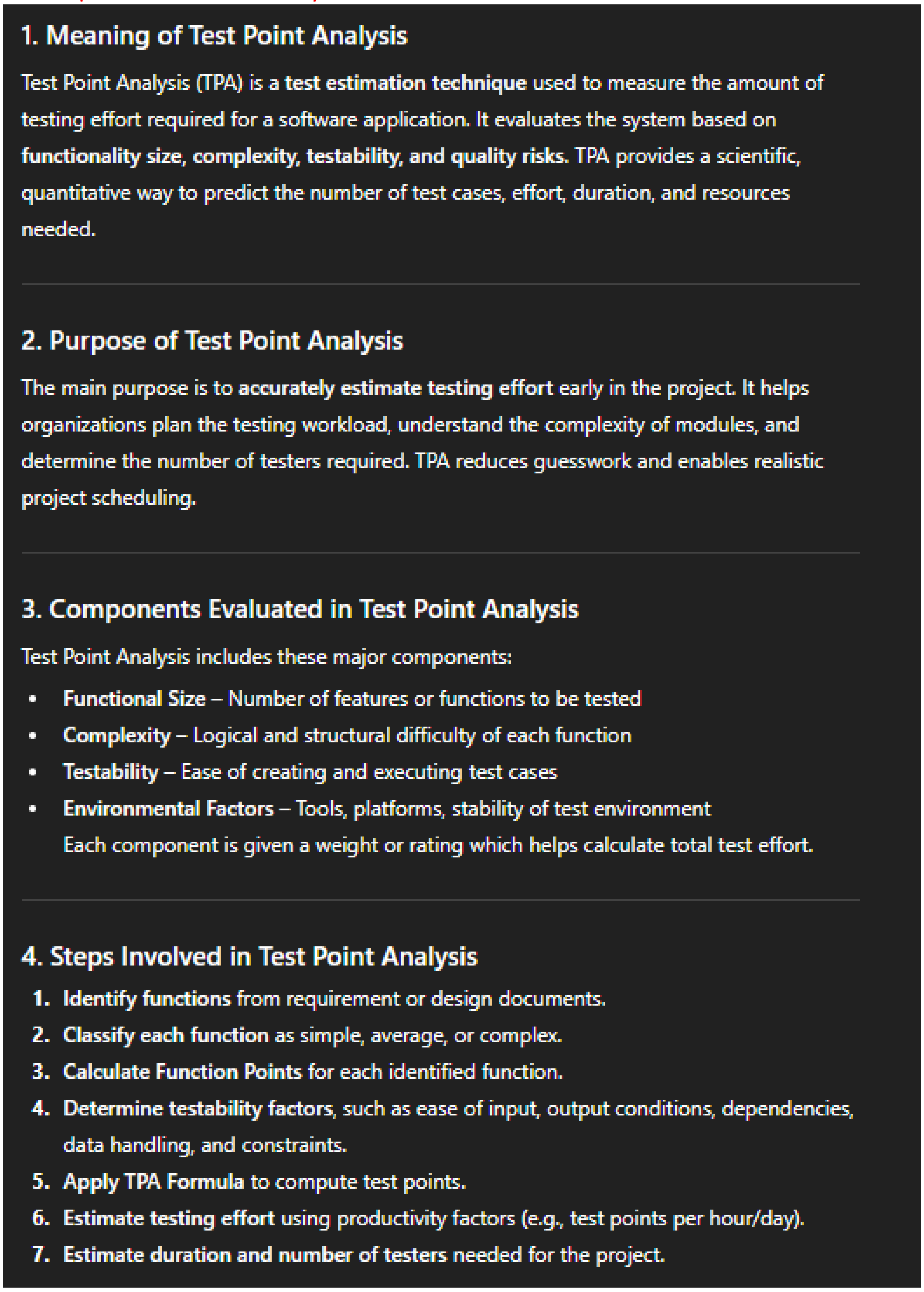
Users can comment on issues, attach logs/screenshots, tag team members, and track complete communication. This promotes teamwork and reduces confusion.

8. Use of JIRA

JIRA is used for **bug tracking, sprint management, release planning, test-cycle tracking**, and continuous project monitoring. It is suitable for medium and large software teams.

UNIT–III : Managing the Test Process

24. Explain Test Point Analysis.*



1. Meaning of Test Point Analysis

Test Point Analysis (TPA) is a **test estimation technique** used to measure the amount of testing effort required for a software application. It evaluates the system based on **functionality size, complexity, testability, and quality risks.** TPA provides a scientific, quantitative way to predict the number of test cases, effort, duration, and resources needed.

2. Purpose of Test Point Analysis

The main purpose is to **accurately estimate testing effort** early in the project. It helps organizations plan the testing workload, understand the complexity of modules, and determine the number of testers required. TPA reduces guesswork and enables realistic project scheduling.

3. Components Evaluated in Test Point Analysis

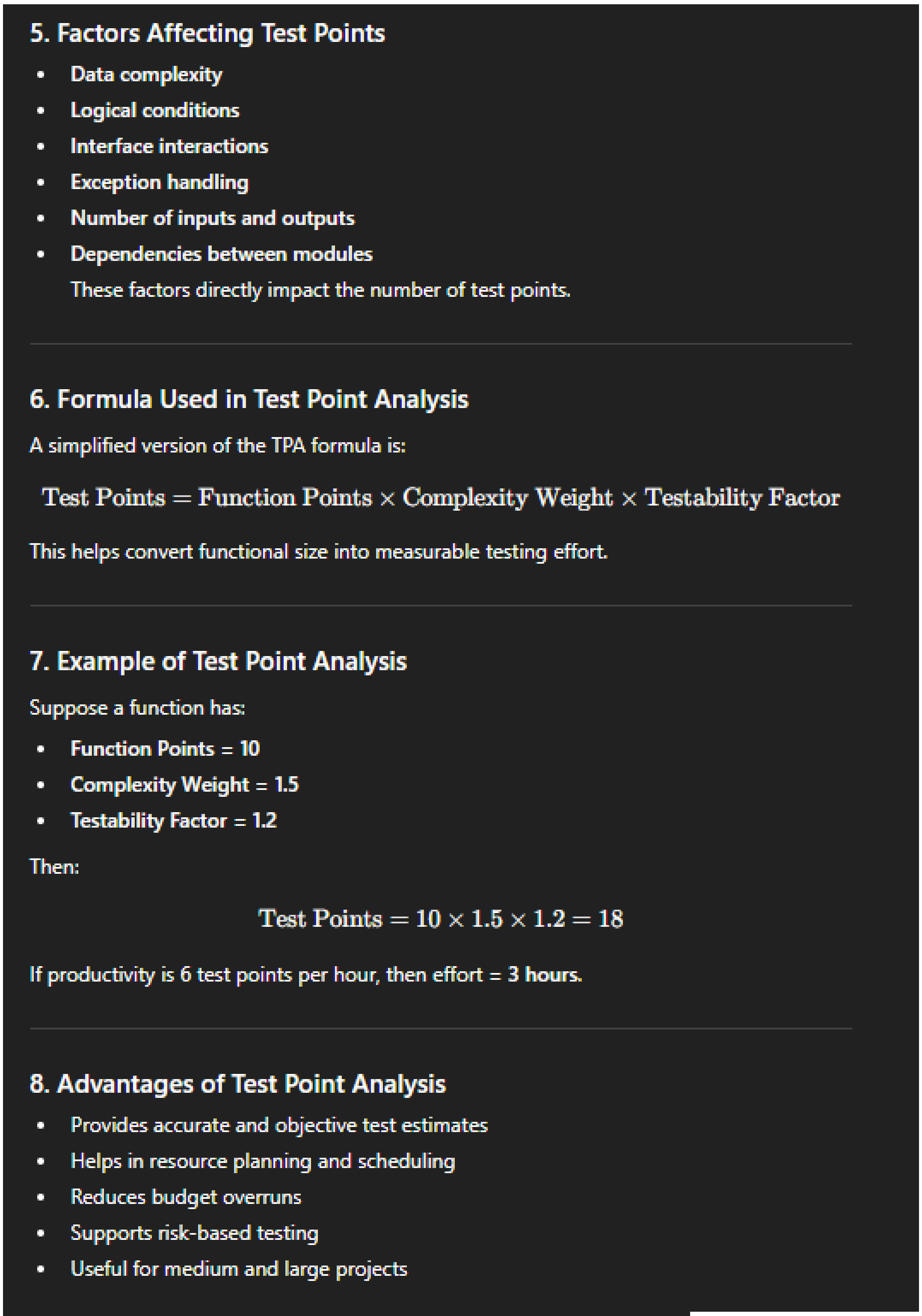
Test Point Analysis includes these major components:

- **Functional Size** – Number of features or functions to be tested
- **Complexity** – Logical and structural difficulty of each function
- **Testability** – Ease of creating and executing test cases
- **Environmental Factors** – Tools, platforms, stability of test environment

  Each component is given a weight or rating which helps calculate total test effort.

4. Steps Involved in Test Point Analysis

1. **Identify functions** from requirement or design documents.
2. **Classify each function** as simple, average, or complex.
3. **Calculate Function Points** for each identified function.
4. **Determine testability factors**, such as ease of input, output conditions, dependencies, data handling, and constraints.
5. **Apply TPA Formula** to compute test points.
6. **Estimate testing effort** using productivity factors (e.g., test points per hour/day).
7. **Estimate duration and number of testers** needed for the project.



5. Factors Affecting Test Points

- **Data complexity**
- **Logical conditions**
- **Interface interactions**
- **Exception handling**
- **Number of inputs and outputs**
- **Dependencies between modules**

  These factors directly impact the number of test points.

---

6. Formula Used in Test Point Analysis

A simplified version of the TPA formula is:

$$\text{Test Points} = \text{Function Points} \times \text{Complexity Weight} \times \text{Testability Factor}$$

This helps convert functional size into measurable testing effort.

---

7. Example of Test Point Analysis

Suppose a function has:

- **Function Points** = 10
- **Complexity Weight** = 1.5
- **Testability Factor** = 1.2

Then:

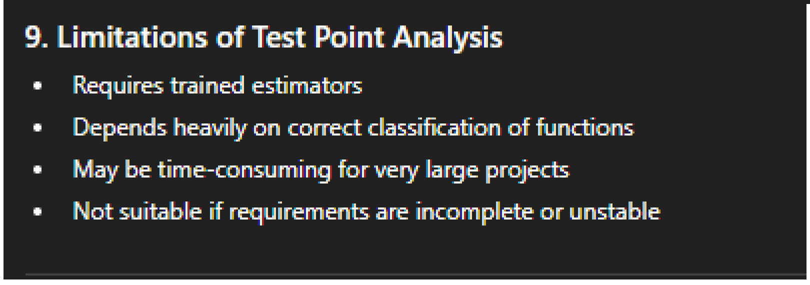
$$\text{Test Points} = 10 \times 1.5 \times 1.2 = 18$$

If productivity is 6 test points per hour, then effort = **3 hours.**

---

8. Advantages of Test Point Analysis

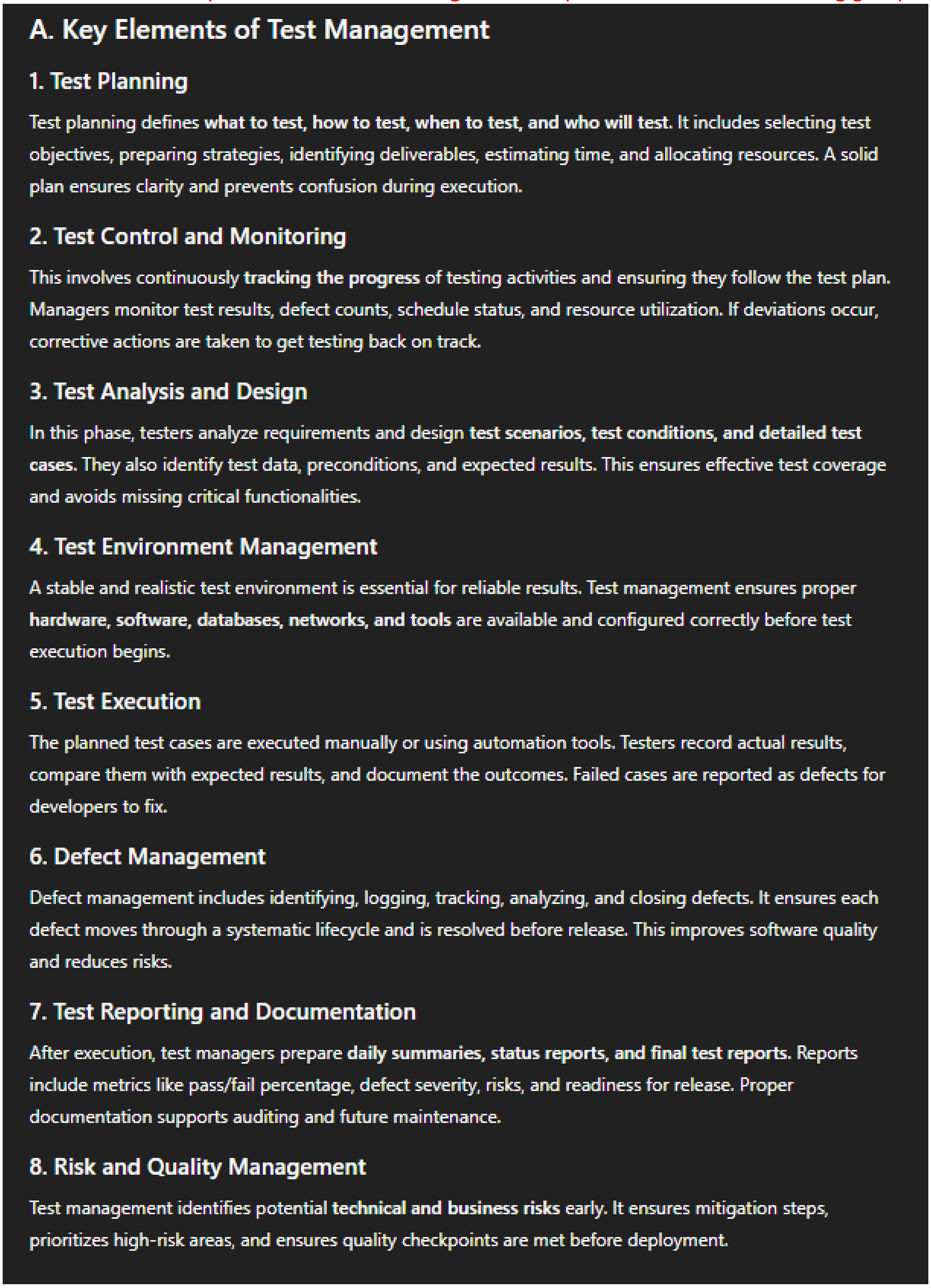
- Provides accurate and objective test estimates
- Helps in resource planning and scheduling
- Reduces budget overruns
- Supports risk-based testing
- Useful for medium and large projects



9. Limitations of Test Point Analysis

- Requires trained estimators
- Depends heavily on correct classification of functions
- May be time-consuming for very large projects
- Not suitable if requirements are incomplete or unstable

25. What are the Key elements of Test Management? Explain the structure of testing group.*



A. Key Elements of Test Management

1. Test Planning

Test planning defines **what to test, how to test, when to test, and who will test.** It includes selecting test objectives, preparing strategies, identifying deliverables, estimating time, and allocating resources. A solid plan ensures clarity and prevents confusion during execution.

2. Test Control and Monitoring

This involves continuously **tracking the progress** of testing activities and ensuring they follow the test plan. Managers monitor test results, defect counts, schedule status, and resource utilization. If deviations occur, corrective actions are taken to get testing back on track.

3. Test Analysis and Design

In this phase, testers analyze requirements and design **test scenarios, test conditions, and detailed test cases.** They also identify test data, preconditions, and expected results. This ensures effective test coverage and avoids missing critical functionalities.

4. Test Environment Management

A stable and realistic test environment is essential for reliable results. Test management ensures proper **hardware, software, databases, networks, and tools** are available and configured correctly before test execution begins.

5. Test Execution

The planned test cases are executed manually or using automation tools. Testers record actual results, compare them with expected results, and document the outcomes. Failed cases are reported as defects for developers to fix.

6. Defect Management

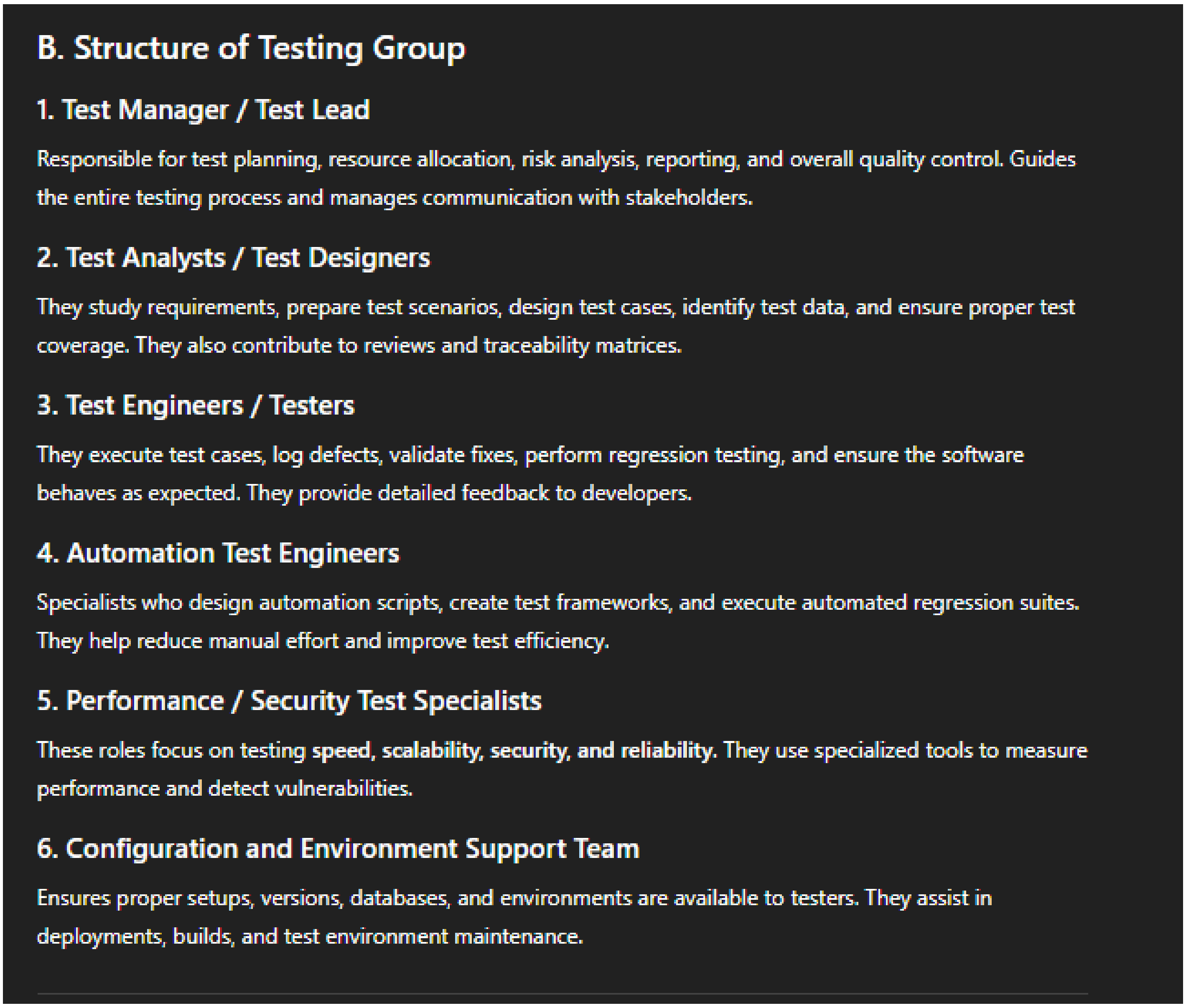
Defect management includes identifying, logging, tracking, analyzing, and closing defects. It ensures each defect moves through a systematic lifecycle and is resolved before release. This improves software quality and reduces risks.

7. Test Reporting and Documentation

After execution, test managers prepare **daily summaries, status reports, and final test reports.** Reports include metrics like pass/fail percentage, defect severity, risks, and readiness for release. Proper documentation supports auditing and future maintenance.

8. Risk and Quality Management

Test management identifies potential **technical and business risks** early. It ensures mitigation steps, prioritizes high-risk areas, and ensures quality checkpoints are met before deployment.



B. Structure of Testing Group

1. Test Manager / Test Lead

Responsible for test planning, resource allocation, risk analysis, reporting, and overall quality control. Guides the entire testing process and manages communication with stakeholders.

2. Test Analysts / Test Designers

They study requirements, prepare test scenarios, design test cases, identify test data, and ensure proper test coverage. They also contribute to reviews and traceability matrices.

3. Test Engineers / Testers

They execute test cases, log defects, validate fixes, perform regression testing, and ensure the software behaves as expected. They provide detailed feedback to developers.

4. Automation Test Engineers

Specialists who design automation scripts, create test frameworks, and execute automated regression suites. They help reduce manual effort and improve test efficiency.

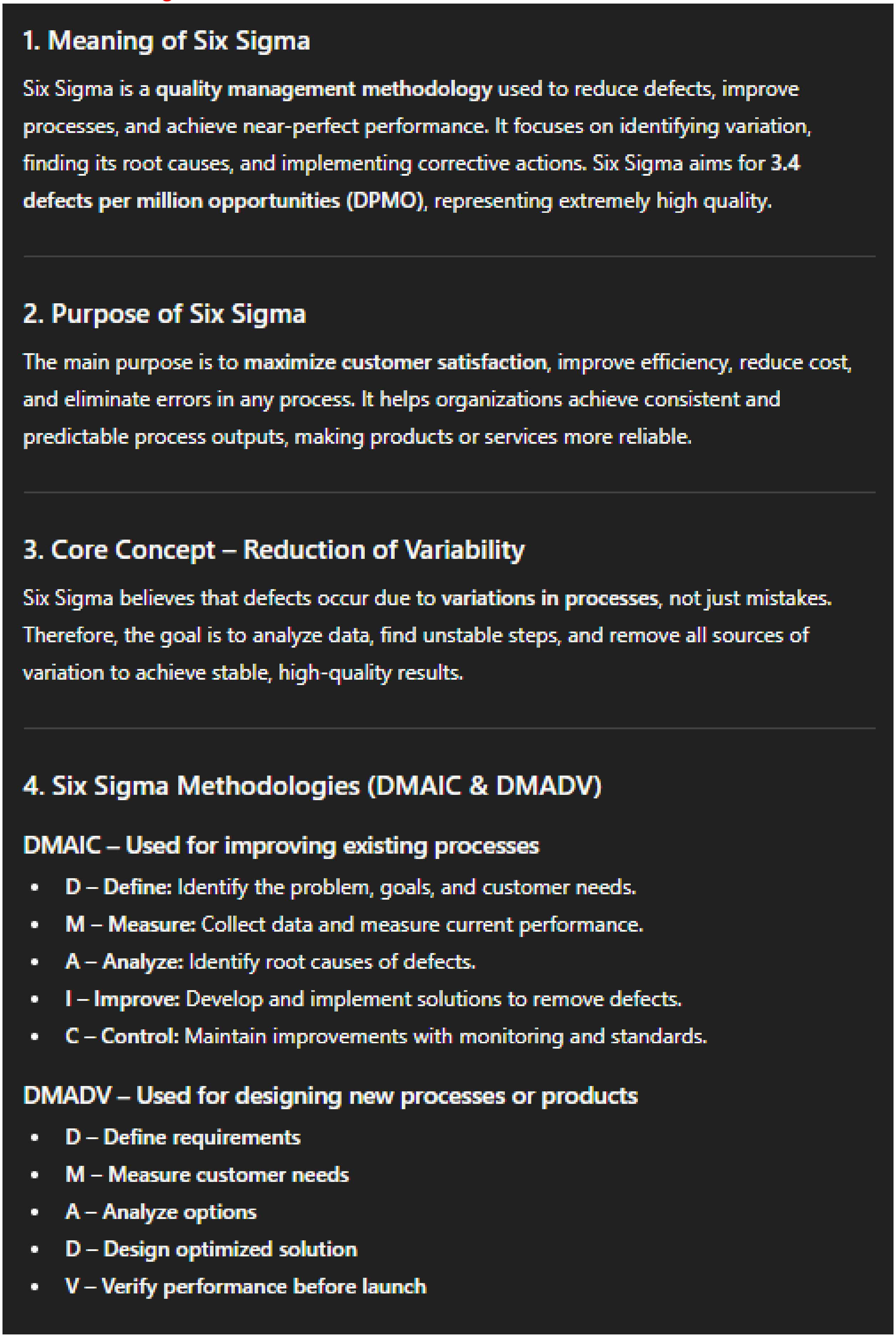
5. Performance / Security Test Specialists

These roles focus on testing **speed, scalability, security, and reliability.** They use specialized tools to measure performance and detect vulnerabilities.

6. Configuration and Environment Support Team

Ensures proper setups, versions, databases, and environments are available to testers. They assist in deployments, builds, and test environment maintenance.

26. Discuss Six Sigma in detail.



1. Meaning of Six Sigma

Six Sigma is a **quality management methodology** used to reduce defects, improve processes, and achieve near-perfect performance. It focuses on identifying variation, finding its root causes, and implementing corrective actions. Six Sigma aims for **3.4 defects per million opportunities (DPMO)**, representing extremely high quality.

---

2. Purpose of Six Sigma

The main purpose is to **maximize customer satisfaction**, improve efficiency, reduce cost, and eliminate errors in any process. It helps organizations achieve consistent and predictable process outputs, making products or services more reliable.

---

3. Core Concept – Reduction of Variability

Six Sigma believes that defects occur due to **variations in processes**, not just mistakes. Therefore, the goal is to analyze data, find unstable steps, and remove all sources of variation to achieve stable, high-quality results.

---

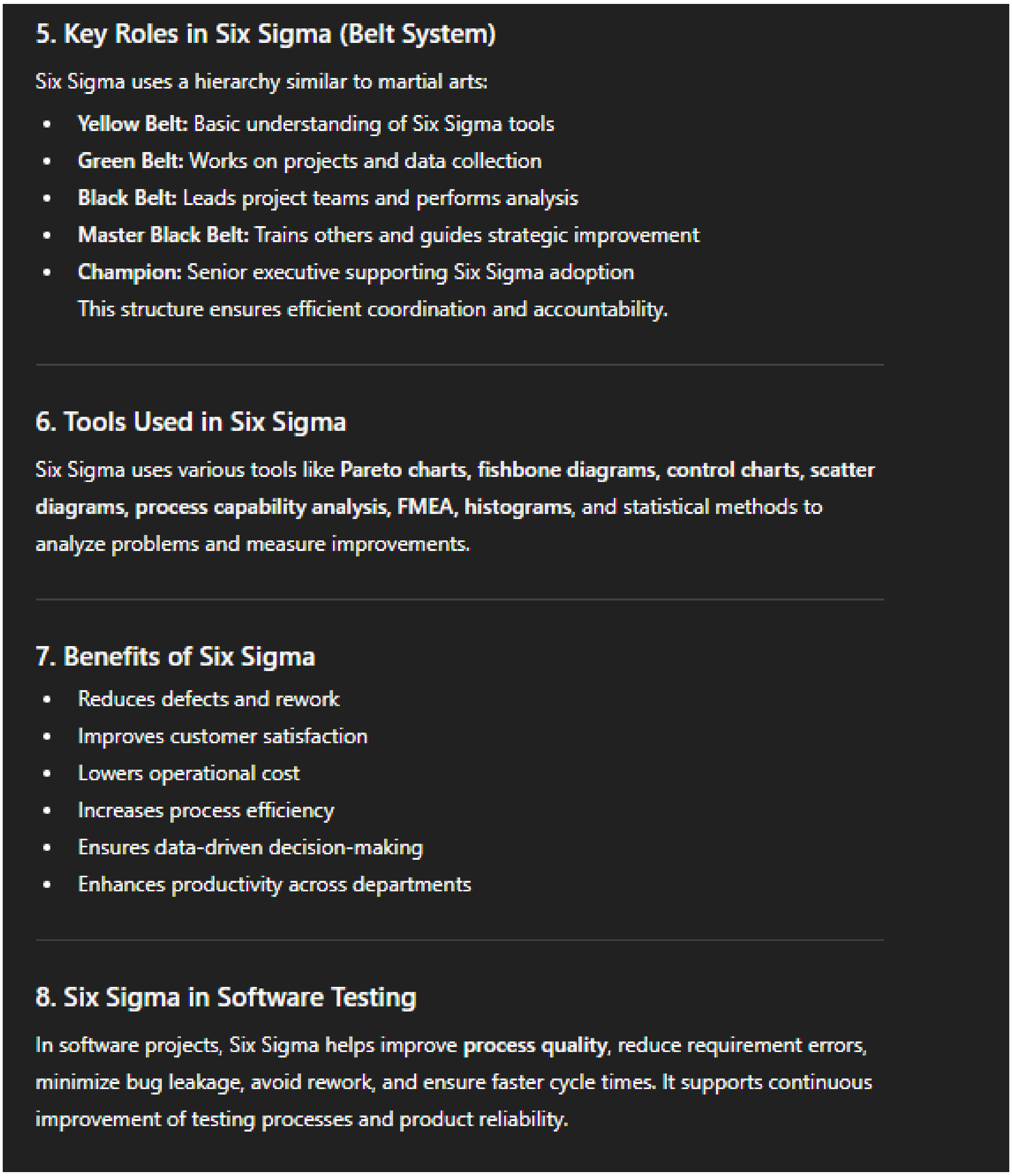
4. Six Sigma Methodologies (DMAIC & DMADV)

DMAIC – Used for improving existing processes

- **D – Define:** Identify the problem, goals, and customer needs.
- **M – Measure:** Collect data and measure current performance.
- **A – Analyze:** Identify root causes of defects.
- **I – Improve:** Develop and implement solutions to remove defects.
- **C – Control:** Maintain improvements with monitoring and standards.

DMADV – Used for designing new processes or products

- **D – Define requirements**
- **M – Measure customer needs**
- **A – Analyze options**
- **D – Design optimized solution**
- **V – Verify performance before launch**



5. Key Roles in Six Sigma (Belt System)

Six Sigma uses a hierarchy similar to martial arts:

- **Yellow Belt:** Basic understanding of Six Sigma tools
- **Green Belt:** Works on projects and data collection
- **Black Belt:** Leads project teams and performs analysis
- **Master Black Belt:** Trains others and guides strategic improvement
- **Champion:** Senior executive supporting Six Sigma adoption
  This structure ensures efficient coordination and accountability.

---

6. Tools Used in Six Sigma

Six Sigma uses various tools like **Pareto charts, fishbone diagrams, control charts, scatter diagrams, process capability analysis, FMEA, histograms**, and statistical methods to analyze problems and measure improvements.

---

7. Benefits of Six Sigma

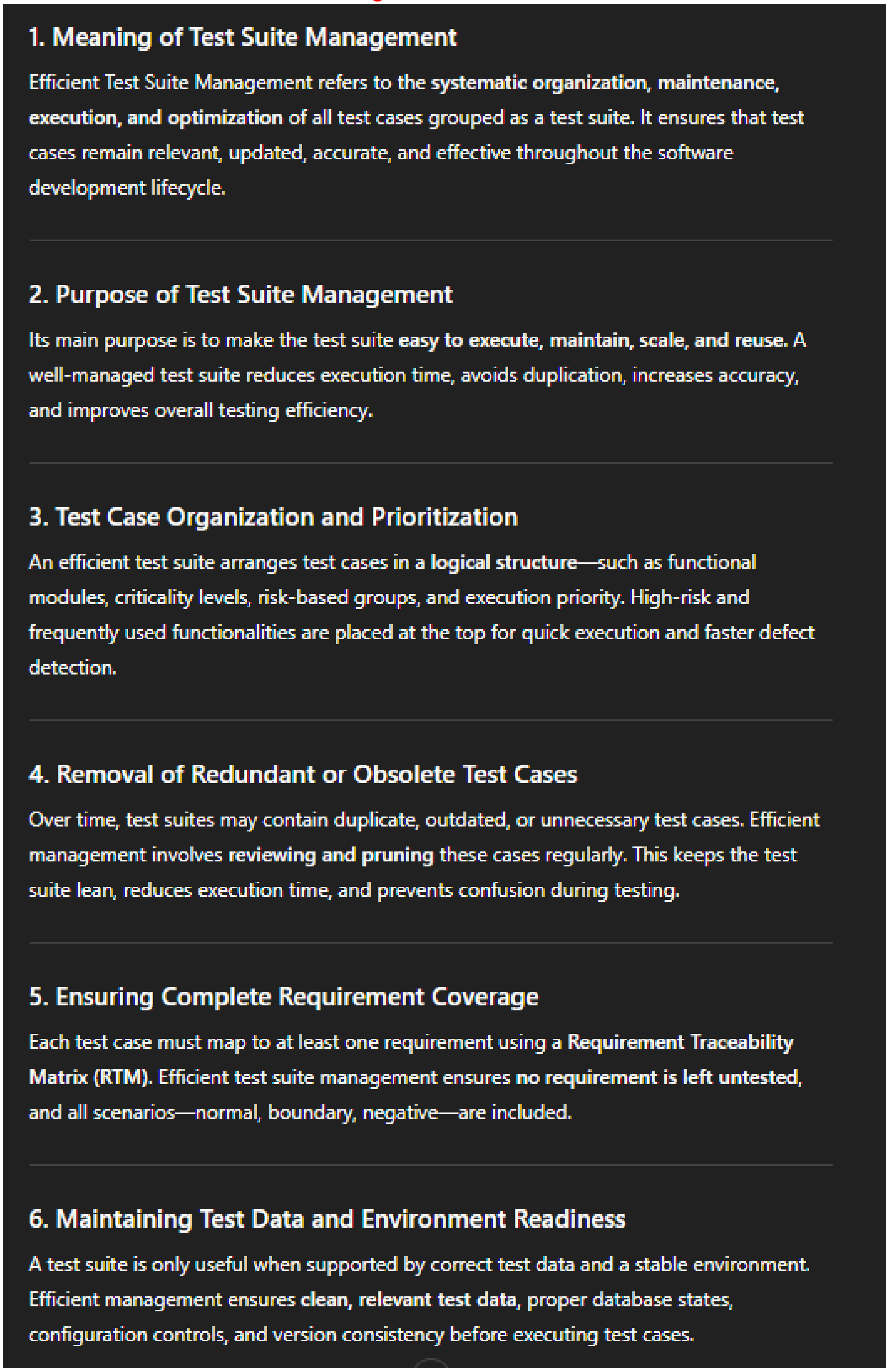
- Reduces defects and rework
- Improves customer satisfaction
- Lowers operational cost
- Increases process efficiency
- Ensures data-driven decision-making
- Enhances productivity across departments

---

8. Six Sigma in Software Testing

In software projects, Six Sigma helps improve **process quality**, reduce requirement errors, minimize bug leakage, avoid rework, and ensure faster cycle times. It supports continuous improvement of testing processes and product reliability.

27. Discuss Efficient Test Suite Management.



1. Meaning of Test Suite Management

Efficient Test Suite Management refers to the **systematic organization, maintenance, execution, and optimization** of all test cases grouped as a test suite. It ensures that test cases remain relevant, updated, accurate, and effective throughout the software development lifecycle.

---

2. Purpose of Test Suite Management

Its main purpose is to make the test suite **easy to execute, maintain, scale, and reuse.** A well-managed test suite reduces execution time, avoids duplication, increases accuracy, and improves overall testing efficiency.

---

3. Test Case Organization and Prioritization

An efficient test suite arranges test cases in a **logical structure**—such as functional modules, criticality levels, risk-based groups, and execution priority. High-risk and frequently used functionalities are placed at the top for quick execution and faster defect detection.

---

4. Removal of Redundant or Obsolete Test Cases

Over time, test suites may contain duplicate, outdated, or unnecessary test cases. Efficient management involves **reviewing and pruning** these cases regularly. This keeps the test suite lean, reduces execution time, and prevents confusion during testing.

---

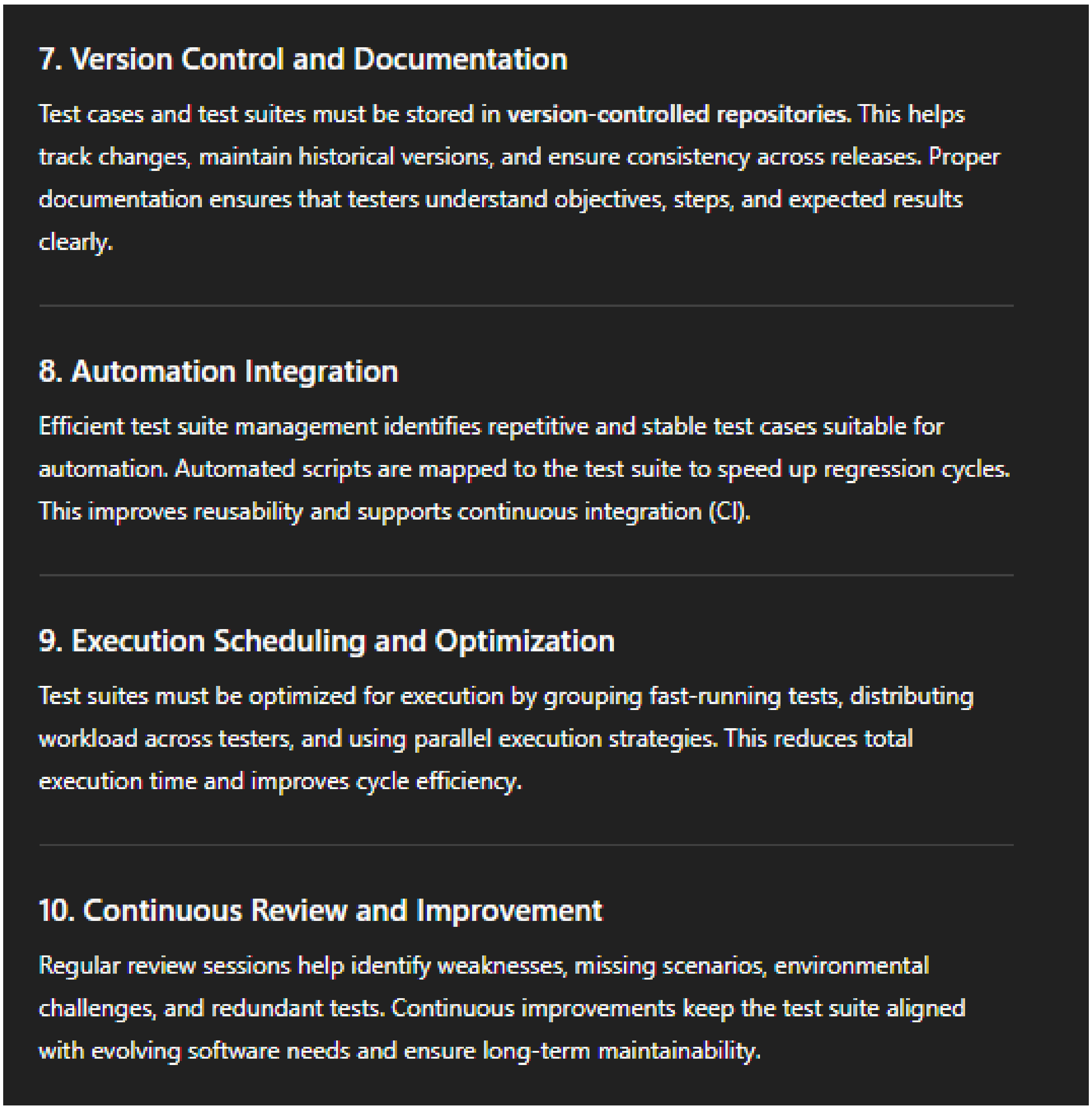
5. Ensuring Complete Requirement Coverage

Each test case must map to at least one requirement using a **Requirement Traceability Matrix (RTM)**. Efficient test suite management ensures **no requirement is left untested**, and all scenarios—normal, boundary, negative—are included.

---

6. Maintaining Test Data and Environment Readiness

A test suite is only useful when supported by correct test data and a stable environment. Efficient management ensures **clean, relevant test data**, proper database states, configuration controls, and version consistency before executing test cases.



7. Version Control and Documentation

Test cases and test suites must be stored in **version-controlled repositories.** This helps track changes, maintain historical versions, and ensure consistency across releases. Proper documentation ensures that testers understand objectives, steps, and expected results clearly.

---

8. Automation Integration

Efficient test suite management identifies repetitive and stable test cases suitable for automation. Automated scripts are mapped to the test suite to speed up regression cycles. This improves reusability and supports continuous integration (CI).

---

9. Execution Scheduling and Optimization

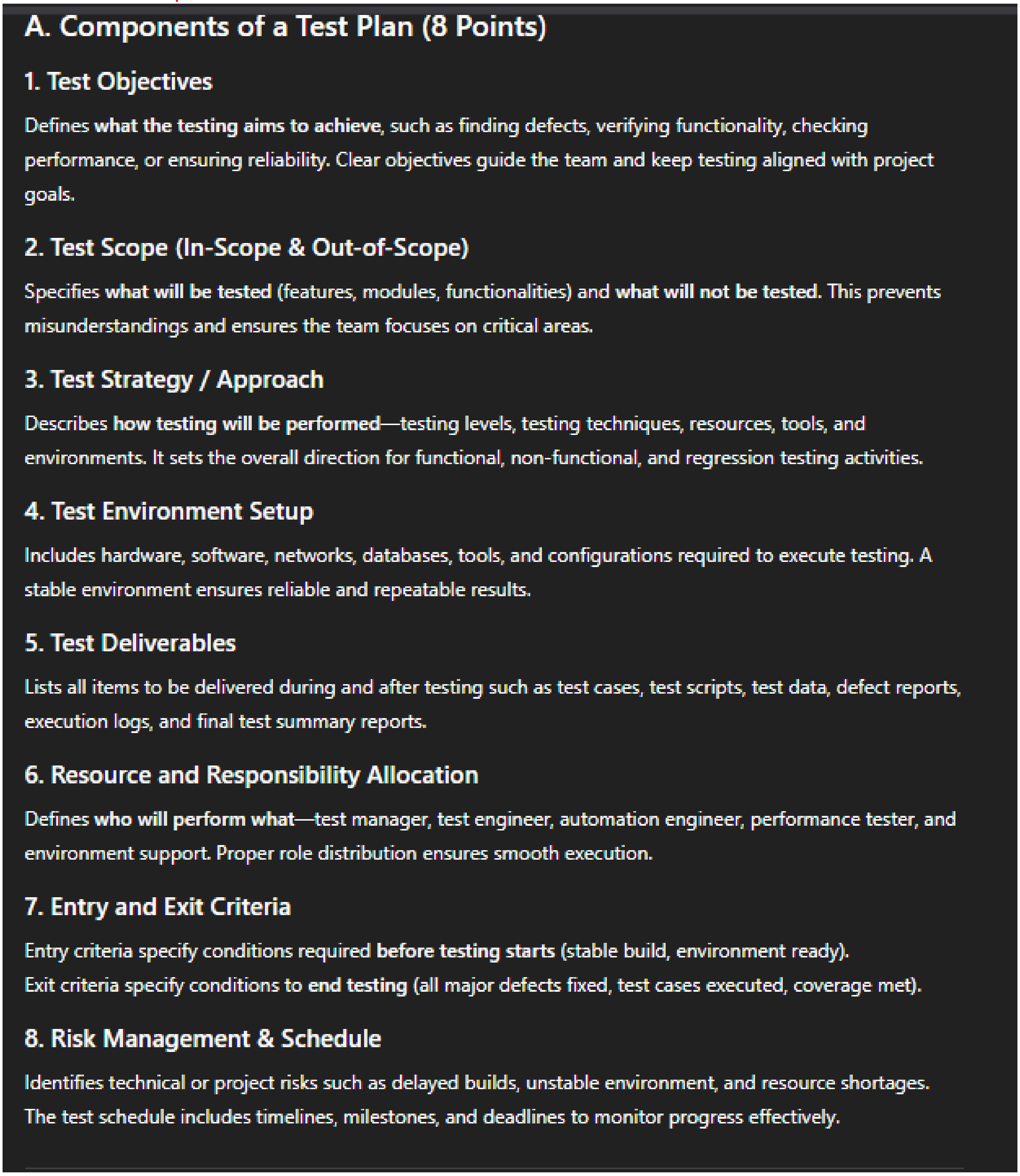
Test suites must be optimized for execution by grouping fast-running tests, distributing workload across testers, and using parallel execution strategies. This reduces total execution time and improves cycle efficiency.

---

10. Continuous Review and Improvement

Regular review sessions help identify weaknesses, missing scenarios, environmental challenges, and redundant tests. Continuous improvements keep the test suite aligned with evolving software needs and ensure long-term maintainability.

28. What are the components of a test plan? Illustrate test plan hierarchy with a neat diagram. Also give different test plan document.



A. Components of a Test Plan (8 Points)

1. Test Objectives

Defines **what the testing aims to achieve**, such as finding defects, verifying functionality, checking performance, or ensuring reliability. Clear objectives guide the team and keep testing aligned with project goals.

2. Test Scope (In-Scope & Out-of-Scope)

Specifies **what will be tested** (features, modules, functionalities) and **what will not be tested.** This prevents misunderstandings and ensures the team focuses on critical areas.

3. Test Strategy / Approach

Describes **how testing will be performed**—testing levels, testing techniques, resources, tools, and environments. It sets the overall direction for functional, non-functional, and regression testing activities.

4. Test Environment Setup

Includes hardware, software, networks, databases, tools, and configurations required to execute testing. A stable environment ensures reliable and repeatable results.

5. Test Deliverables

Lists all items to be delivered during and after testing such as test cases, test scripts, test data, defect reports, execution logs, and final test summary reports.

6. Resource and Responsibility Allocation

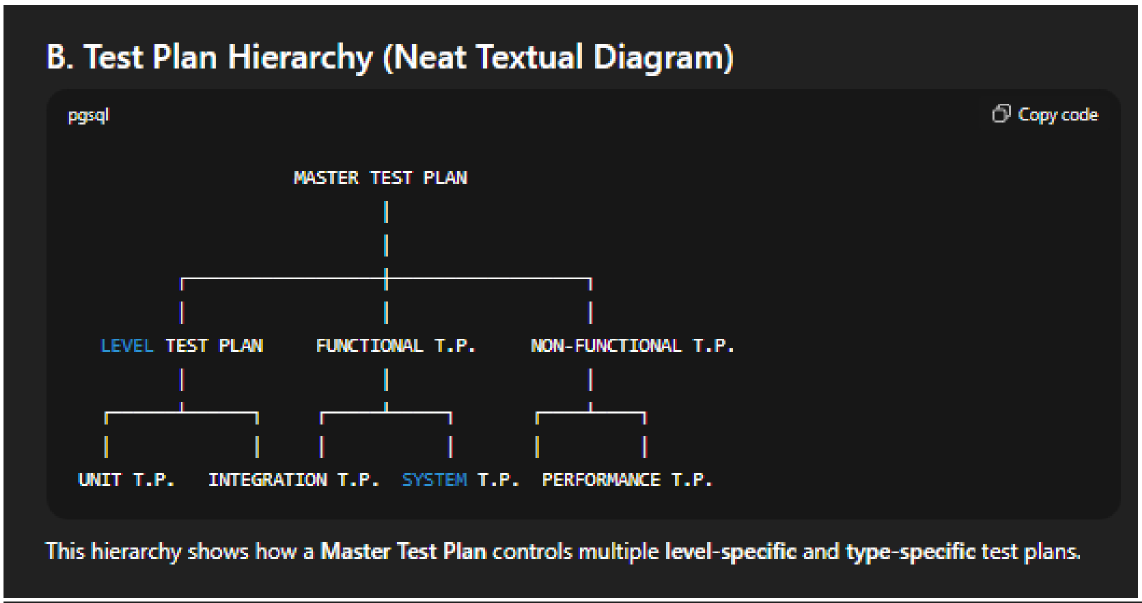
Defines **who will perform what**—test manager, test engineer, automation engineer, performance tester, and environment support. Proper role distribution ensures smooth execution.

7. Entry and Exit Criteria

Entry criteria specify conditions required **before testing starts** (stable build, environment ready).
Exit criteria specify conditions to **end testing** (all major defects fixed, test cases executed, coverage met).

8. Risk Management & Schedule

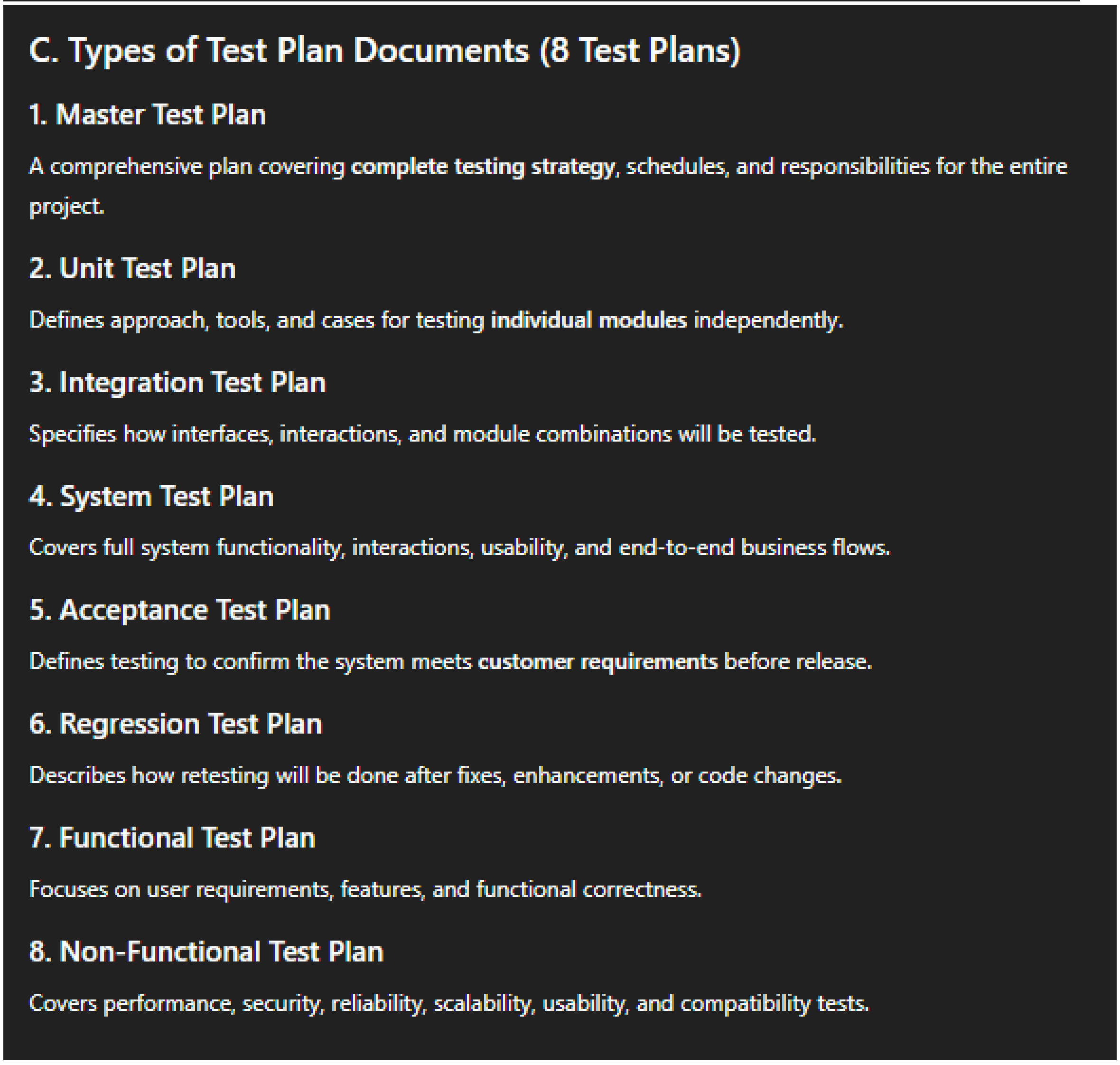
Identifies technical or project risks such as delayed builds, unstable environment, and resource shortages.
The test schedule includes timelines, milestones, and deadlines to monitor progress effectively.



B. Test Plan Hierarchy (Neat Textual Diagram)

```
                     MASTER TEST PLAN
                            |
                            |
             ┌──────────────┼──────────────┐
             |              |              |
      LEVEL TEST PLAN    FUNCTIONAL T.P.    NON-FUNCTIONAL T.P.
             |              |              |
       ┌─────┴─────┐     ┌──┴───────┐    ┌──┴───┐
       |           |     |          |    |      |
   UNIT T.P.  INTEGRATION T.P.  SYSTEM T.P.  PERFORMANCE T.P.
```

This hierarchy shows how a **Master Test Plan** controls multiple **level-specific** and **type-specific** test plans.



C. Types of Test Plan Documents (8 Test Plans)

1. Master Test Plan

A comprehensive plan covering **complete testing strategy**, schedules, and responsibilities for the entire project.

2. Unit Test Plan

Defines approach, tools, and cases for testing **individual modules** independently.

3. Integration Test Plan

Specifies how interfaces, interactions, and module combinations will be tested.

4. System Test Plan

Covers full system functionality, interactions, usability, and end-to-end business flows.

5. Acceptance Test Plan

Defines testing to confirm the system meets **customer requirements** before release.

6. Regression Test Plan

Describes how retesting will be done after fixes, enhancements, or code changes.

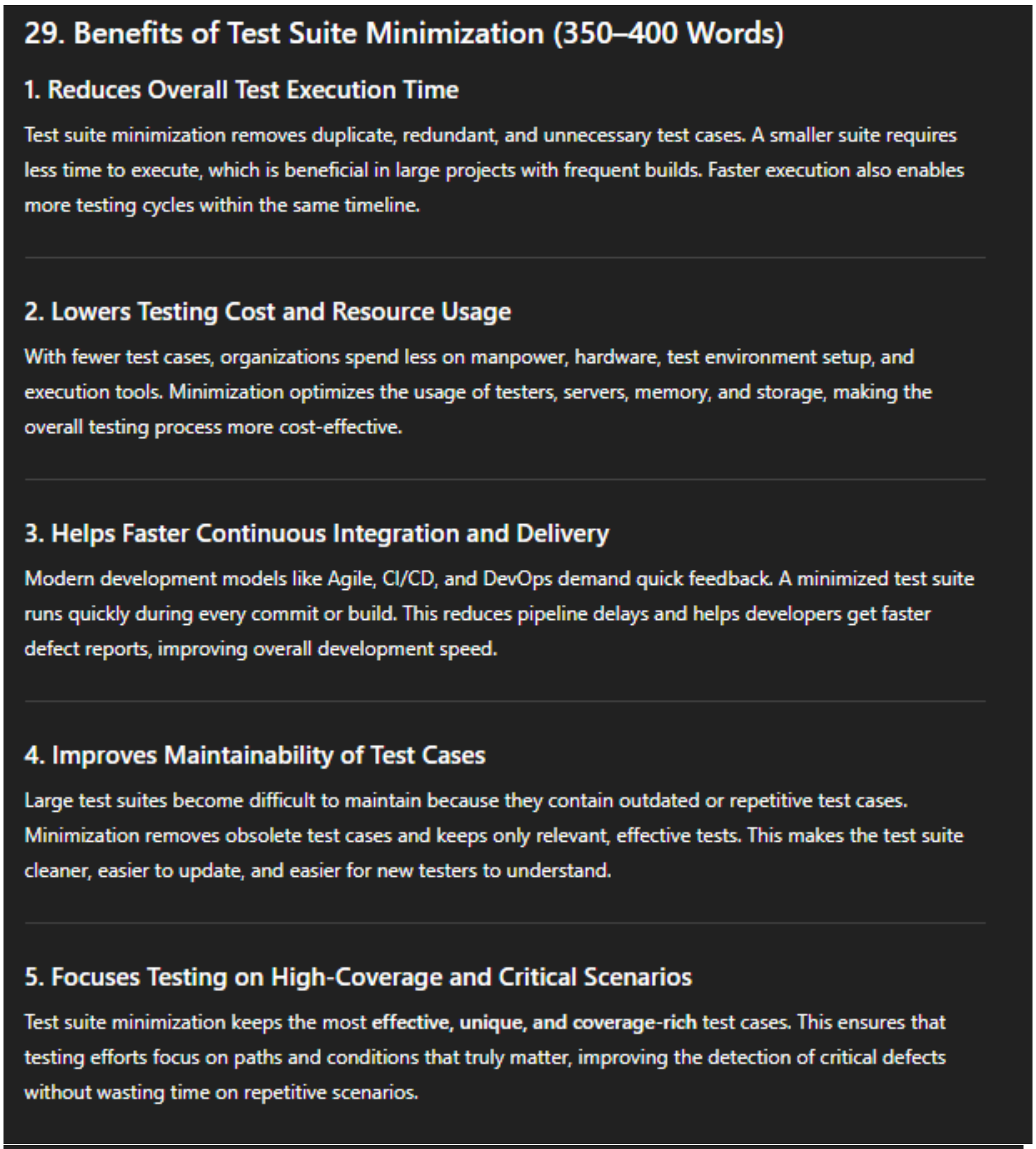
7. Functional Test Plan

Focuses on user requirements, features, and functional correctness.

8. Non-Functional Test Plan

Covers performance, security, reliability, scalability, usability, and compatibility tests.

29. What are the benefits of Test Suite Minimization?



29. Benefits of Test Suite Minimization (350–400 Words)

1. Reduces Overall Test Execution Time

Test suite minimization removes duplicate, redundant, and unnecessary test cases. A smaller suite requires less time to execute, which is beneficial in large projects with frequent builds. Faster execution also enables more testing cycles within the same timeline.

2. Lowers Testing Cost and Resource Usage

With fewer test cases, organizations spend less on manpower, hardware, test environment setup, and execution tools. Minimization optimizes the usage of testers, servers, memory, and storage, making the overall testing process more cost-effective.

3. Helps Faster Continuous Integration and Delivery

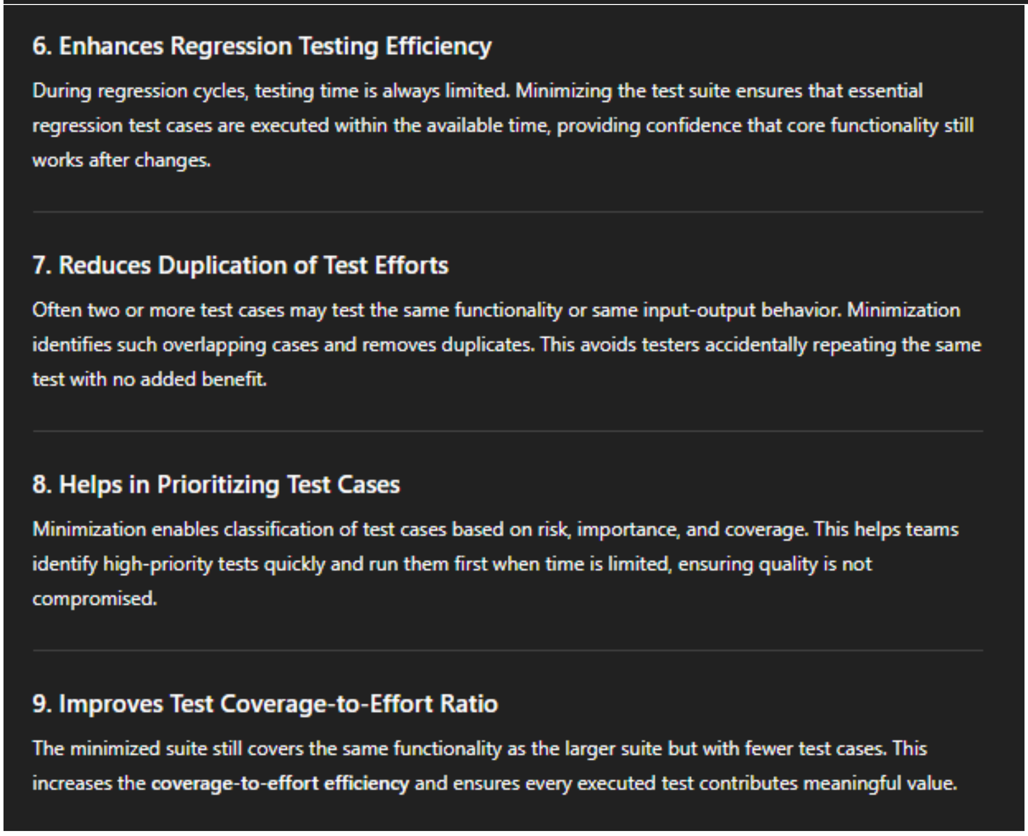
Modern development models like Agile, CI/CD, and DevOps demand quick feedback. A minimized test suite runs quickly during every commit or build. This reduces pipeline delays and helps developers get faster defect reports, improving overall development speed.

4. Improves Maintainability of Test Cases

Large test suites become difficult to maintain because they contain outdated or repetitive test cases. Minimization removes obsolete test cases and keeps only relevant, effective tests. This makes the test suite cleaner, easier to update, and easier for new testers to understand.

5. Focuses Testing on High-Coverage and Critical Scenarios

Test suite minimization keeps the most **effective, unique, and coverage-rich** test cases. This ensures that testing efforts focus on paths and conditions that truly matter, improving the detection of critical defects without wasting time on repetitive scenarios.



6. Enhances Regression Testing Efficiency

During regression cycles, testing time is always limited. Minimizing the test suite ensures that essential regression test cases are executed within the available time, providing confidence that core functionality still works after changes.

7. Reduces Duplication of Test Efforts

Often two or more test cases may test the same functionality or same input-output behavior. Minimization identifies such overlapping cases and removes duplicates. This avoids testers accidentally repeating the same test with no added benefit.

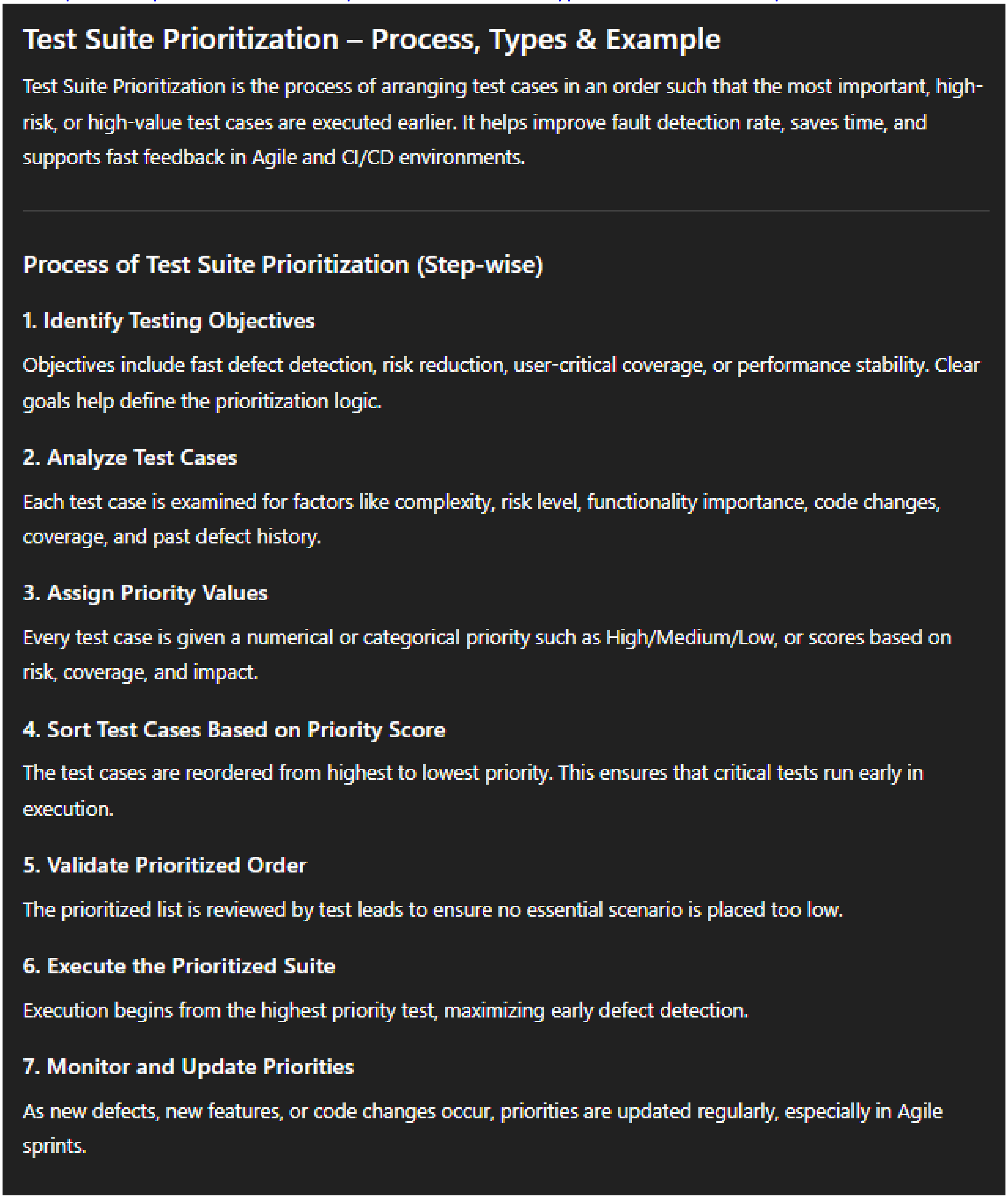
8. Helps in Prioritizing Test Cases

Minimization enables classification of test cases based on risk, importance, and coverage. This helps teams identify high-priority tests quickly and run them first when time is limited, ensuring quality is not compromised.

9. Improves Test Coverage-to-Effort Ratio

The minimized suite still covers the same functionality as the larger suite but with fewer test cases. This increases the **coverage-to-effort efficiency** and ensures every executed test contributes meaningful value.

30. Explain the process of test suite prioritization with its type in detail with example.



Test Suite Prioritization – Process, Types & Example

Test Suite Prioritization is the process of arranging test cases in an order such that the most important, high-risk, or high-value test cases are executed earlier. It helps improve fault detection rate, saves time, and supports fast feedback in Agile and CI/CD environments.

Process of Test Suite Prioritization (Step-wise)

1. Identify Testing Objectives

Objectives include fast defect detection, risk reduction, user-critical coverage, or performance stability. Clear goals help define the prioritization logic.

2. Analyze Test Cases

Each test case is examined for factors like complexity, risk level, functionality importance, code changes, coverage, and past defect history.

3. Assign Priority Values

Every test case is given a numerical or categorical priority such as High/Medium/Low, or scores based on risk, coverage, and impact.

4. Sort Test Cases Based on Priority Score

The test cases are reordered from highest to lowest priority. This ensures that critical tests run early in execution.

5. Validate Prioritized Order

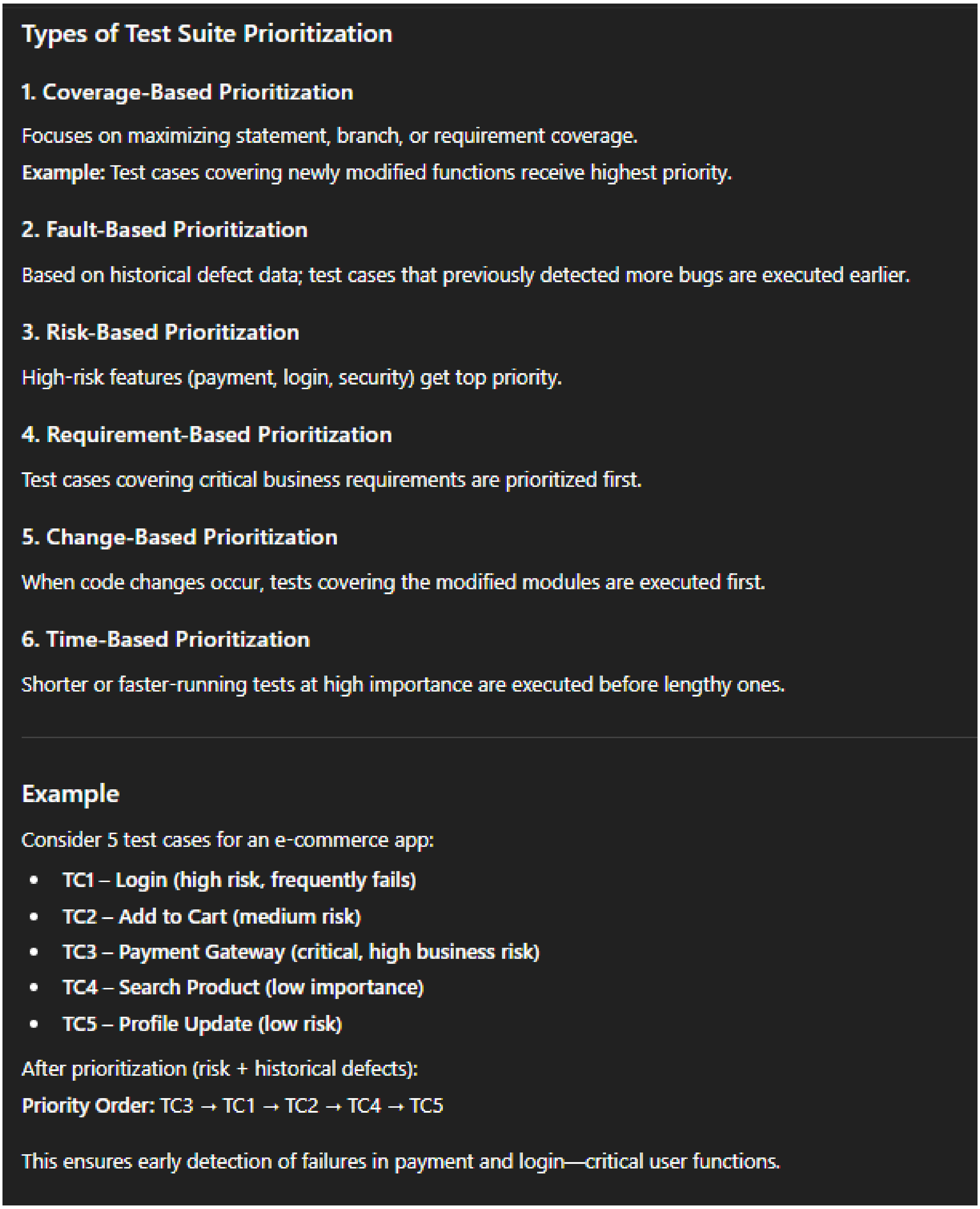
The prioritized list is reviewed by test leads to ensure no essential scenario is placed too low.

6. Execute the Prioritized Suite

Execution begins from the highest priority test, maximizing early defect detection.

7. Monitor and Update Priorities

As new defects, new features, or code changes occur, priorities are updated regularly, especially in Agile sprints.



Types of Test Suite Prioritization

1. Coverage-Based Prioritization

Focuses on maximizing statement, branch, or requirement coverage.
Example: Test cases covering newly modified functions receive highest priority.

2. Fault-Based Prioritization

Based on historical defect data; test cases that previously detected more bugs are executed earlier.

3. Risk-Based Prioritization

High-risk features (payment, login, security) get top priority.

4. Requirement-Based Prioritization

Test cases covering critical business requirements are prioritized first.

5. Change-Based Prioritization

When code changes occur, tests covering the modified modules are executed first.

6. Time-Based Prioritization

Shorter or faster-running tests at high importance are executed before lengthy ones.

Example

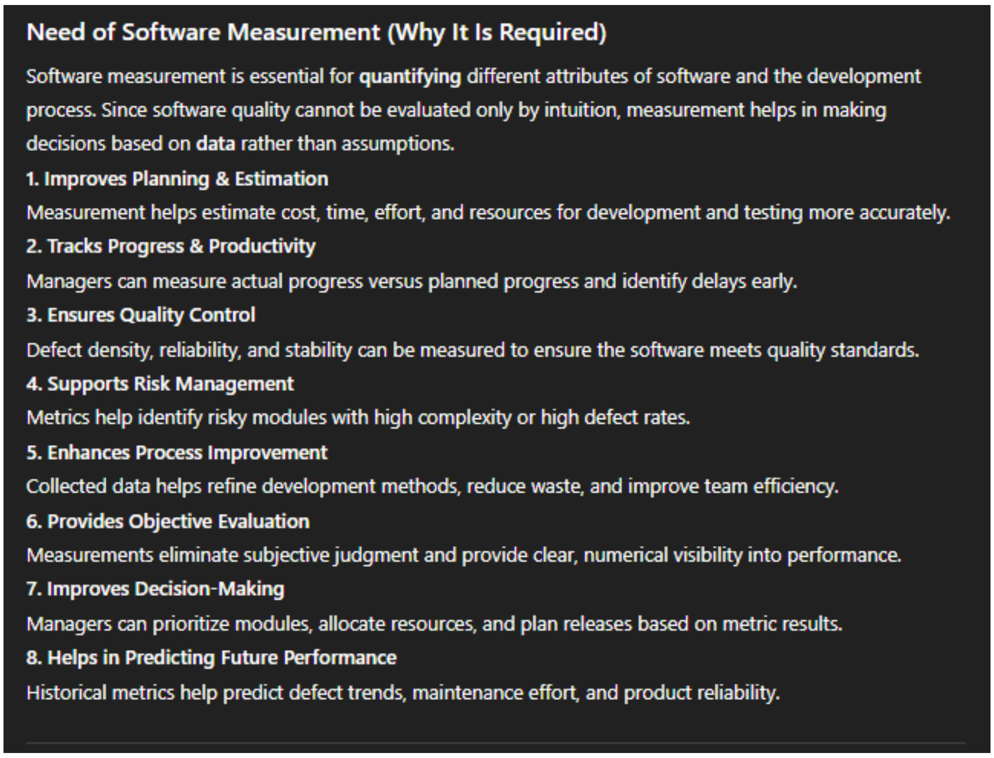
Consider 5 test cases for an e-commerce app:

- **TC1 – Login (high risk, frequently fails)**
- **TC2 – Add to Cart (medium risk)**
- **TC3 – Payment Gateway (critical, high business risk)**
- **TC4 – Search Product (low importance)**
- **TC5 – Profile Update (low risk)**

After prioritization (risk + historical defects):
Priority Order: TC3 → TC1 → TC2 → TC4 → TC5

This ensures early detection of failures in payment and login—critical user functions.

31. What is the need of software measurement?



Need of Software Measurement (Why It Is Required)

Software measurement is essential for **quantifying** different attributes of software and the development process. Since software quality cannot be evaluated only by intuition, measurement helps in making decisions based on **data** rather than assumptions.

1. Improves Planning & Estimation

Measurement helps estimate cost, time, effort, and resources for development and testing more accurately.

2. Tracks Progress & Productivity

Managers can measure actual progress versus planned progress and identify delays early.

3. Ensures Quality Control

Defect density, reliability, and stability can be measured to ensure the software meets quality standards.

4. Supports Risk Management

Metrics help identify risky modules with high complexity or high defect rates.

5. Enhances Process Improvement

Collected data helps refine development methods, reduce waste, and improve team efficiency.

6. Provides Objective Evaluation

Measurements eliminate subjective judgment and provide clear, numerical visibility into performance.

7. Improves Decision-Making

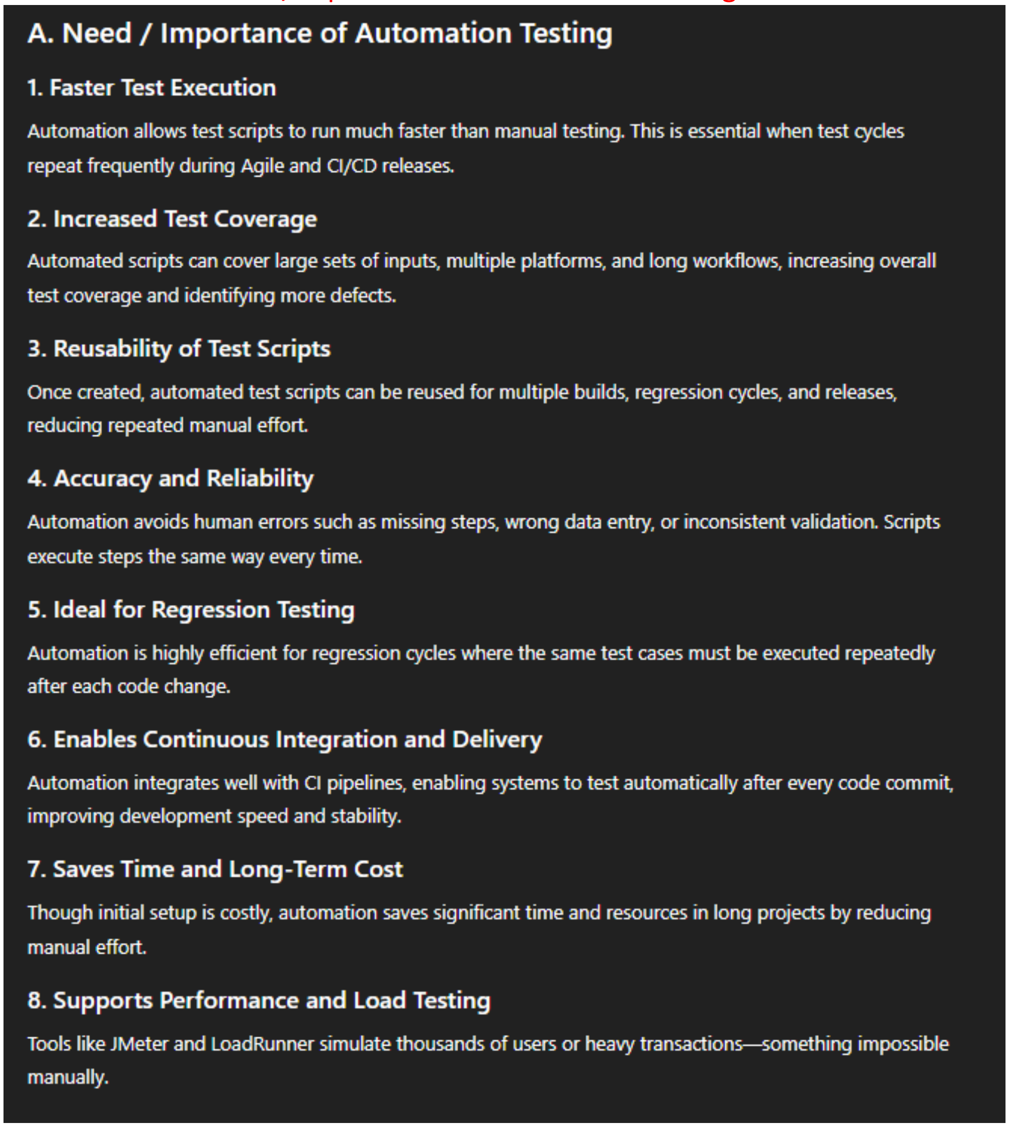
Managers can prioritize modules, allocate resources, and plan releases based on metric results.

8. Helps in Predicting Future Performance

Historical metrics help predict defect trends, maintenance effort, and product reliability.

UNIT–IV : Test Automation

32. What is the need/importance of Automation testing. Differentiate between static and dynamic tools?



A. Need / Importance of Automation Testing

1. Faster Test Execution

Automation allows test scripts to run much faster than manual testing. This is essential when test cycles repeat frequently during Agile and CI/CD releases.

2. Increased Test Coverage

Automated scripts can cover large sets of inputs, multiple platforms, and long workflows, increasing overall test coverage and identifying more defects.

3. Reusability of Test Scripts

Once created, automated test scripts can be reused for multiple builds, regression cycles, and releases, reducing repeated manual effort.

4. Accuracy and Reliability

Automation avoids human errors such as missing steps, wrong data entry, or inconsistent validation. Scripts execute steps the same way every time.

5. Ideal for Regression Testing

Automation is highly efficient for regression cycles where the same test cases must be executed repeatedly after each code change.

6. Enables Continuous Integration and Delivery

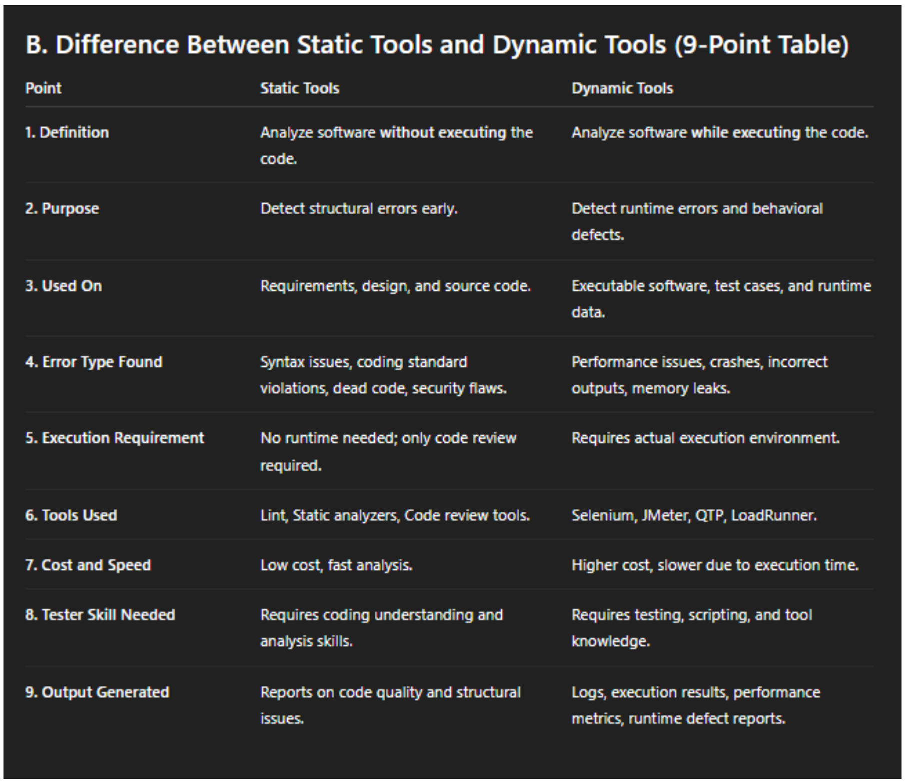
Automation integrates well with CI pipelines, enabling systems to test automatically after every code commit, improving development speed and stability.

7. Saves Time and Long-Term Cost

Though initial setup is costly, automation saves significant time and resources in long projects by reducing manual effort.

8. Supports Performance and Load Testing

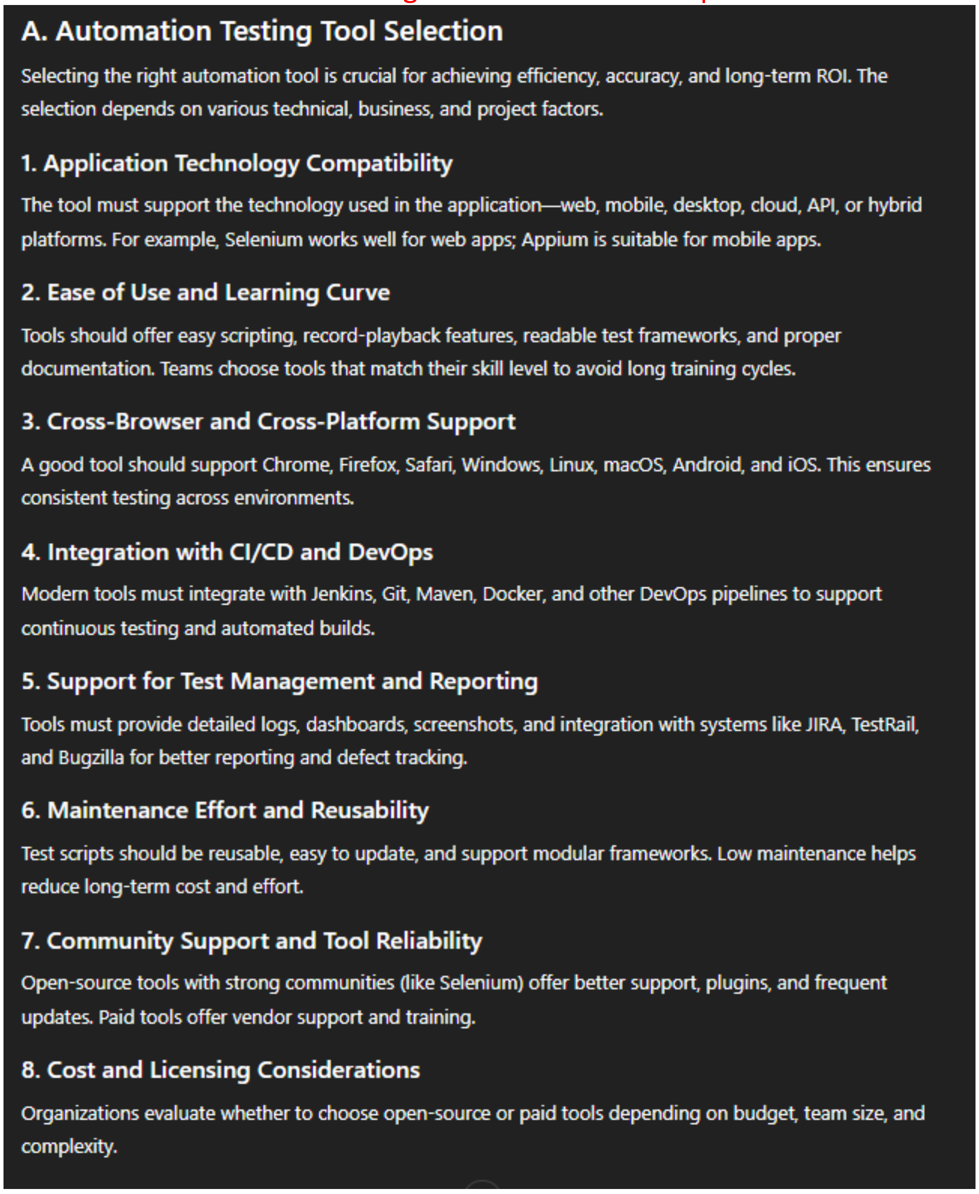
Tools like JMeter and LoadRunner simulate thousands of users or heavy transactions—something impossible manually.



B. Difference Between Static Tools and Dynamic Tools (9-Point Table)

Point	Static Tools	Dynamic Tools
1. Definition	Analyze software **without executing** the code.	Analyze software **while executing** the code.
2. Purpose	Detect structural errors early.	Detect runtime errors and behavioral defects.
3. Used On	Requirements, design, and source code.	Executable software, test cases, and runtime data.
4. Error Type Found	Syntax issues, coding standard violations, dead code, security flaws.	Performance issues, crashes, incorrect outputs, memory leaks.
5. Execution Requirement	No runtime needed; only code review required.	Requires actual execution environment.
6. Tools Used	Lint, Static analyzers, Code review tools.	Selenium, JMeter, QTP, LoadRunner.
7. Cost and Speed	Low cost, fast analysis.	Higher cost, slower due to execution time.
8. Tester Skill Needed	Requires coding understanding and analysis skills.	Requires testing, scripting, and tool knowledge.
9. Output Generated	Reports on code quality and structural issues.	Logs, execution results, performance metrics, runtime defect reports.

33. Discuss Automation Testing Tool selection and explain the cost incurred in Automation testing tools.



A. Automation Testing Tool Selection

Selecting the right automation tool is crucial for achieving efficiency, accuracy, and long-term ROI. The selection depends on various technical, business, and project factors.

1. Application Technology Compatibility

The tool must support the technology used in the application—web, mobile, desktop, cloud, API, or hybrid platforms. For example, Selenium works well for web apps; Appium is suitable for mobile apps.

2. Ease of Use and Learning Curve

Tools should offer easy scripting, record-playback features, readable test frameworks, and proper documentation. Teams choose tools that match their skill level to avoid long training cycles.

3. Cross-Browser and Cross-Platform Support

A good tool should support Chrome, Firefox, Safari, Windows, Linux, macOS, Android, and iOS. This ensures consistent testing across environments.

4. Integration with CI/CD and DevOps

Modern tools must integrate with Jenkins, Git, Maven, Docker, and other DevOps pipelines to support continuous testing and automated builds.

5. Support for Test Management and Reporting

Tools must provide detailed logs, dashboards, screenshots, and integration with systems like JIRA, TestRail, and Bugzilla for better reporting and defect tracking.

6. Maintenance Effort and Reusability

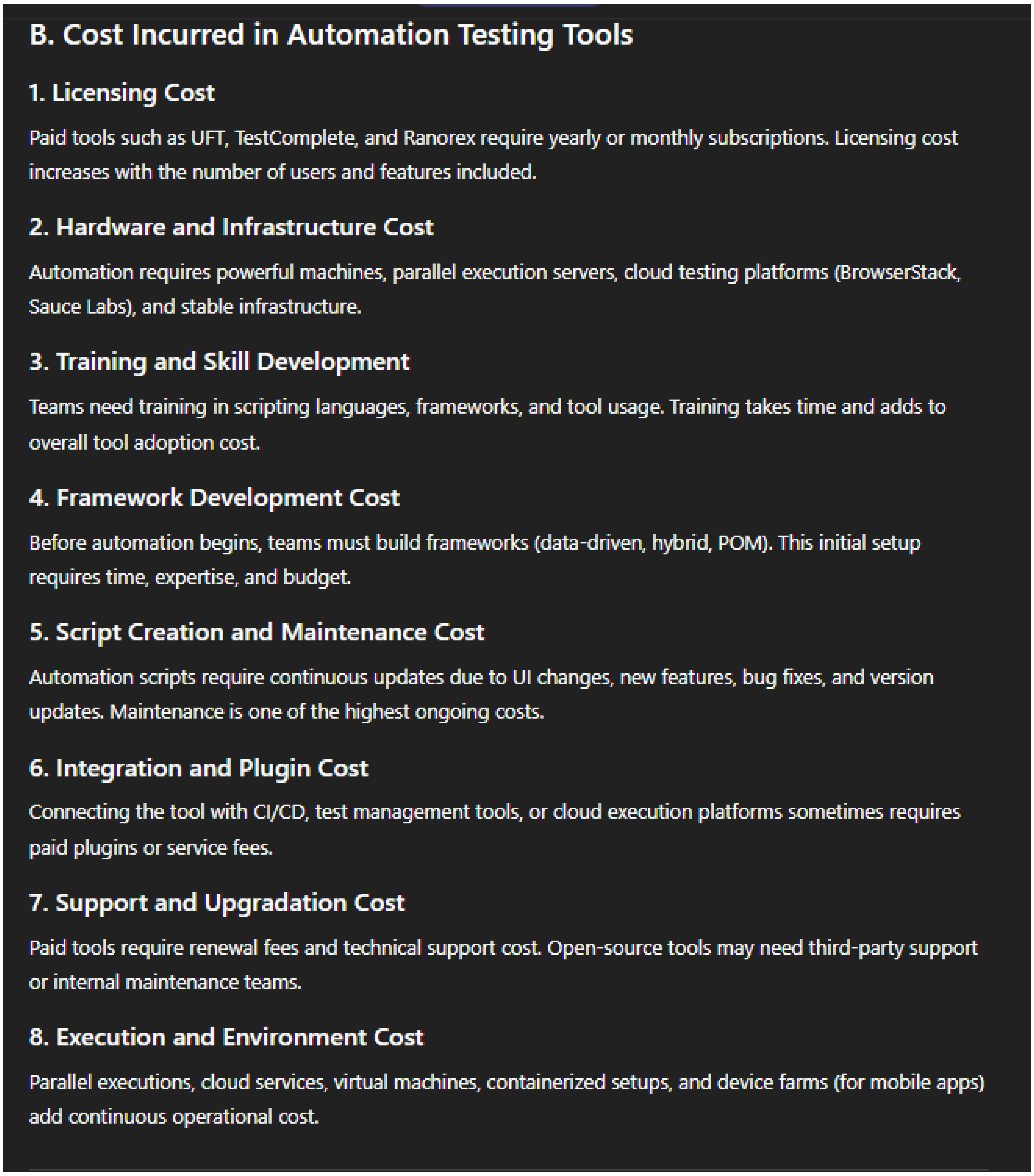
Test scripts should be reusable, easy to update, and support modular frameworks. Low maintenance helps reduce long-term cost and effort.

7. Community Support and Tool Reliability

Open-source tools with strong communities (like Selenium) offer better support, plugins, and frequent updates. Paid tools offer vendor support and training.

8. Cost and Licensing Considerations

Organizations evaluate whether to choose open-source or paid tools depending on budget, team size, and complexity.



B. Cost Incurred in Automation Testing Tools

1. Licensing Cost

Paid tools such as UFT, TestComplete, and Ranorex require yearly or monthly subscriptions. Licensing cost increases with the number of users and features included.

2. Hardware and Infrastructure Cost

Automation requires powerful machines, parallel execution servers, cloud testing platforms (BrowserStack, Sauce Labs), and stable infrastructure.

3. Training and Skill Development

Teams need training in scripting languages, frameworks, and tool usage. Training takes time and adds to overall tool adoption cost.

4. Framework Development Cost

Before automation begins, teams must build frameworks (data-driven, hybrid, POM). This initial setup requires time, expertise, and budget.

5. Script Creation and Maintenance Cost

Automation scripts require continuous updates due to UI changes, new features, bug fixes, and version updates. Maintenance is one of the highest ongoing costs.

6. Integration and Plugin Cost

Connecting the tool with CI/CD, test management tools, or cloud execution platforms sometimes requires paid plugins or service fees.

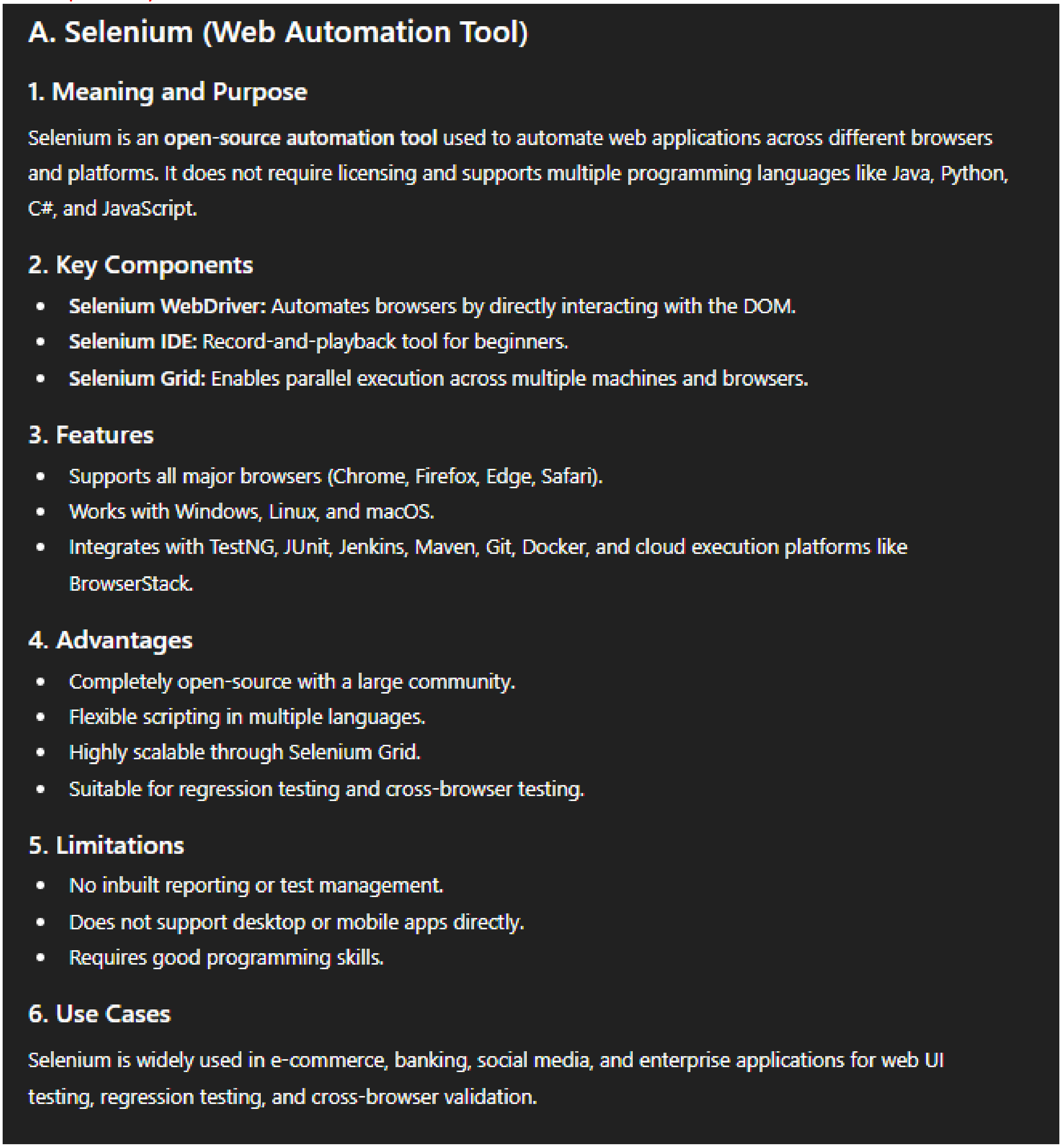
7. Support and Upgradation Cost

Paid tools require renewal fees and technical support cost. Open-source tools may need third-party support or internal maintenance teams.

8. Execution and Environment Cost

Parallel executions, cloud services, virtual machines, containerized setups, and device farms (for mobile apps) add continuous operational cost.

34. Explain any two Automation tools in detail.



A. Selenium (Web Automation Tool)

1. Meaning and Purpose

Selenium is an **open-source automation tool** used to automate web applications across different browsers and platforms. It does not require licensing and supports multiple programming languages like Java, Python, C#, and JavaScript.

2. Key Components

- **Selenium WebDriver:** Automates browsers by directly interacting with the DOM.
- **Selenium IDE:** Record-and-playback tool for beginners.
- **Selenium Grid:** Enables parallel execution across multiple machines and browsers.

3. Features

- Supports all major browsers (Chrome, Firefox, Edge, Safari).
- Works with Windows, Linux, and macOS.
- Integrates with TestNG, JUnit, Jenkins, Maven, Git, Docker, and cloud execution platforms like BrowserStack.

4. Advantages

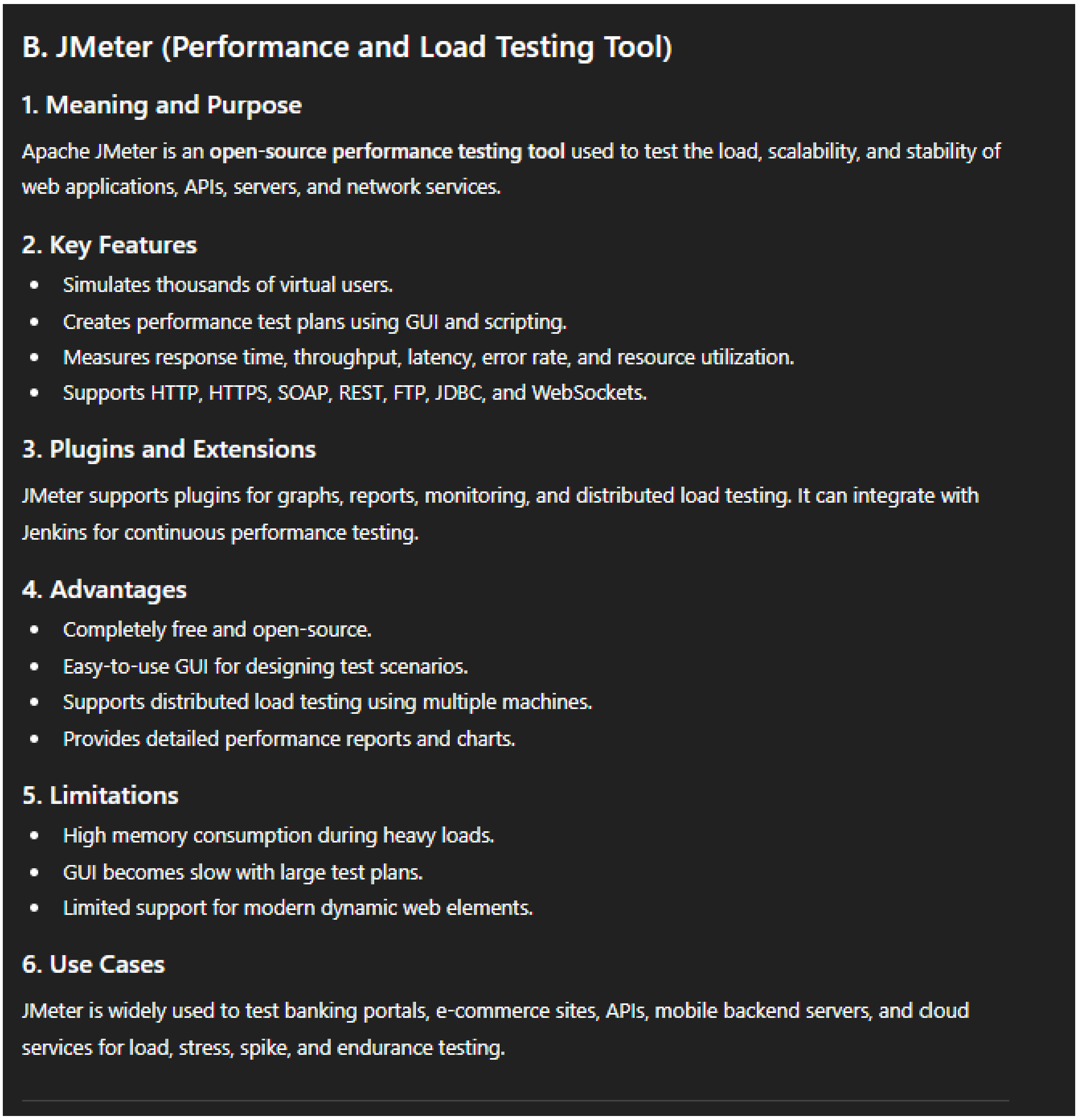
- Completely open-source with a large community.
- Flexible scripting in multiple languages.
- Highly scalable through Selenium Grid.
- Suitable for regression testing and cross-browser testing.

5. Limitations

- No inbuilt reporting or test management.
- Does not support desktop or mobile apps directly.
- Requires good programming skills.

6. Use Cases

Selenium is widely used in e-commerce, banking, social media, and enterprise applications for web UI testing, regression testing, and cross-browser validation.



B. JMeter (Performance and Load Testing Tool)

1. Meaning and Purpose

Apache JMeter is an **open-source performance testing tool** used to test the load, scalability, and stability of web applications, APIs, servers, and network services.

2. Key Features

- Simulates thousands of virtual users.
- Creates performance test plans using GUI and scripting.
- Measures response time, throughput, latency, error rate, and resource utilization.
- Supports HTTP, HTTPS, SOAP, REST, FTP, JDBC, and WebSockets.

3. Plugins and Extensions

JMeter supports plugins for graphs, reports, monitoring, and distributed load testing. It can integrate with Jenkins for continuous performance testing.

4. Advantages

- Completely free and open-source.
- Easy-to-use GUI for designing test scenarios.
- Supports distributed load testing using multiple machines.
- Provides detailed performance reports and charts.

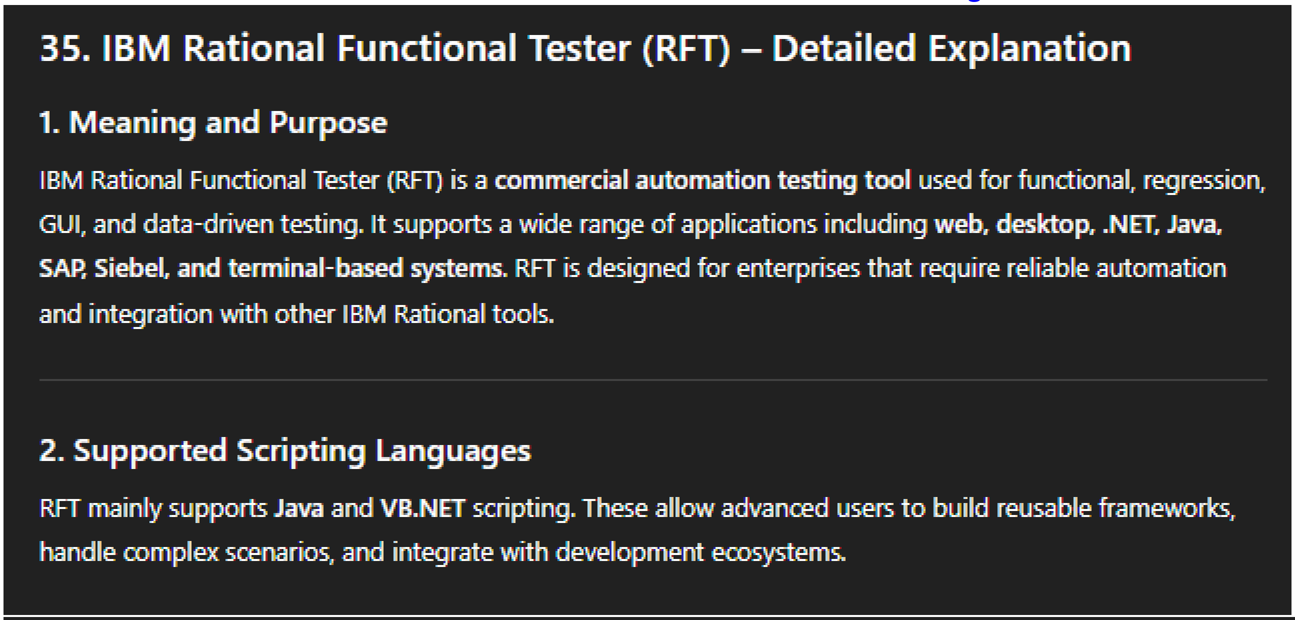
5. Limitations

- High memory consumption during heavy loads.
- GUI becomes slow with large test plans.
- Limited support for modern dynamic web elements.

6. Use Cases

JMeter is widely used to test banking portals, e-commerce sites, APIs, mobile backend servers, and cloud services for load, stress, spike, and endurance testing.

35. Discuss about IBM Rational Functional Tester as an Automation Testing Tool.



35. IBM Rational Functional Tester (RFT) – Detailed Explanation

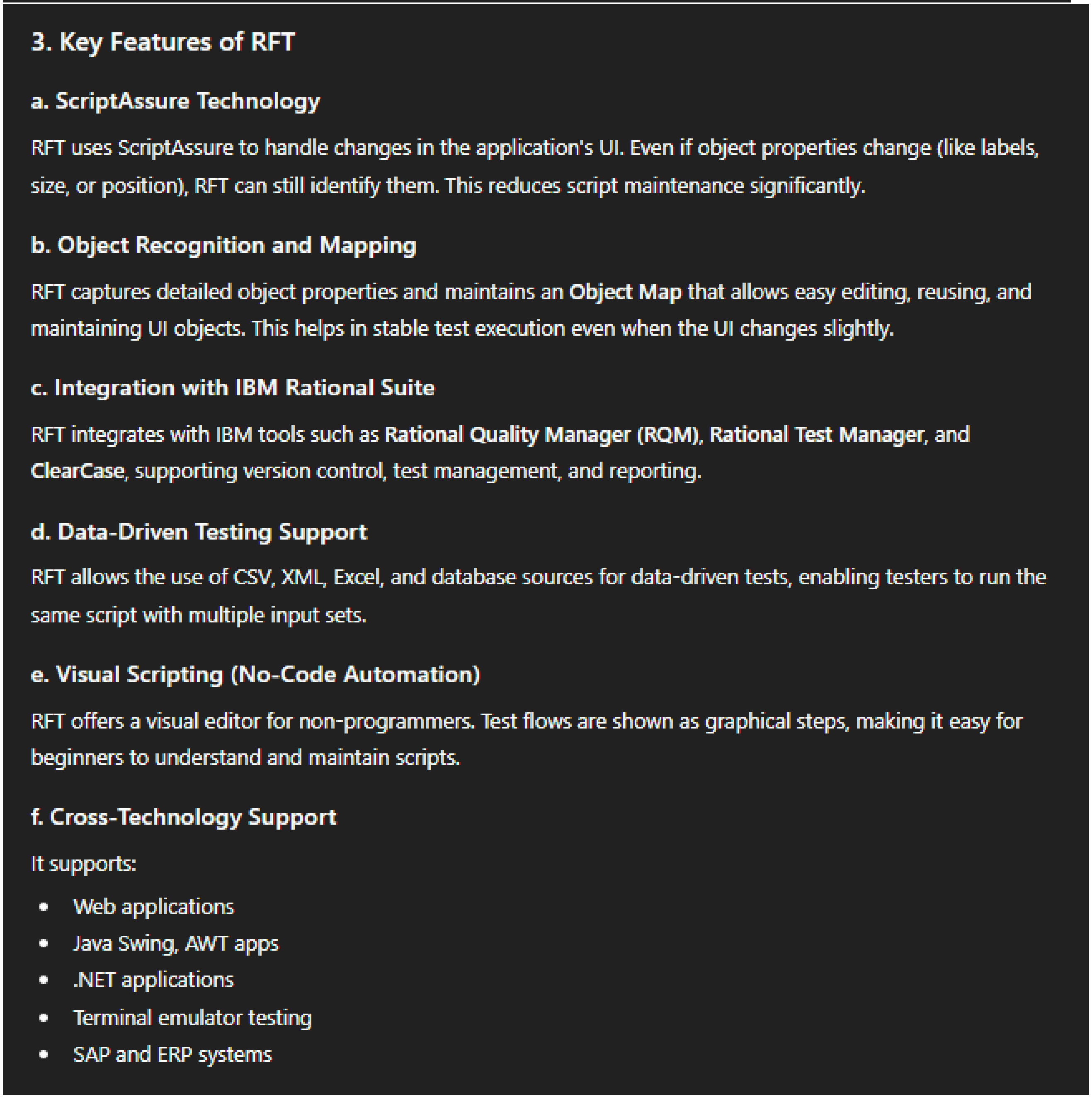
1. Meaning and Purpose

IBM Rational Functional Tester (RFT) is a **commercial automation testing tool** used for functional, regression, GUI, and data-driven testing. It supports a wide range of applications including **web, desktop, .NET, Java, SAP, Siebel, and terminal-based systems.** RFT is designed for enterprises that require reliable automation and integration with other IBM Rational tools.

---

2. Supported Scripting Languages

RFT mainly supports **Java** and **VB.NET** scripting. These allow advanced users to build reusable frameworks, handle complex scenarios, and integrate with development ecosystems.



3. Key Features of RFT

a. ScriptAssure Technology

RFT uses ScriptAssure to handle changes in the application's UI. Even if object properties change (like labels, size, or position), RFT can still identify them. This reduces script maintenance significantly.

b. Object Recognition and Mapping

RFT captures detailed object properties and maintains an **Object Map** that allows easy editing, reusing, and maintaining UI objects. This helps in stable test execution even when the UI changes slightly.

c. Integration with IBM Rational Suite

RFT integrates with IBM tools such as **Rational Quality Manager (RQM)**, **Rational Test Manager**, and **ClearCase**, supporting version control, test management, and reporting.

d. Data-Driven Testing Support

RFT allows the use of CSV, XML, Excel, and database sources for data-driven tests, enabling testers to run the same script with multiple input sets.

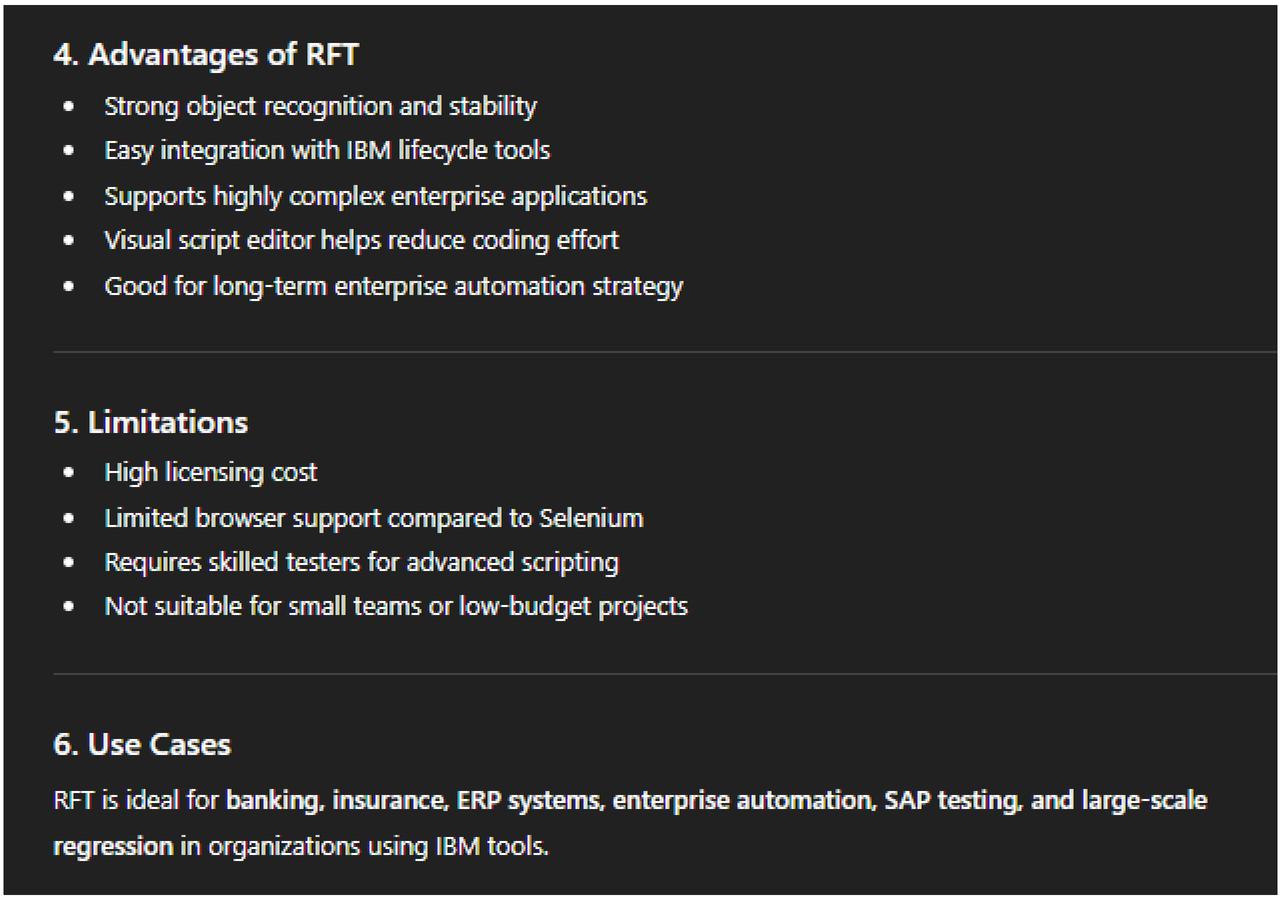
e. Visual Scripting (No-Code Automation)

RFT offers a visual editor for non-programmers. Test flows are shown as graphical steps, making it easy for beginners to understand and maintain scripts.

f. Cross-Technology Support

It supports:

- Web applications
- Java Swing, AWT apps
- .NET applications
- Terminal emulator testing
- SAP and ERP systems



4. Advantages of RFT

- Strong object recognition and stability
- Easy integration with IBM lifecycle tools
- Supports highly complex enterprise applications
- Visual script editor helps reduce coding effort
- Good for long-term enterprise automation strategy

5. Limitations

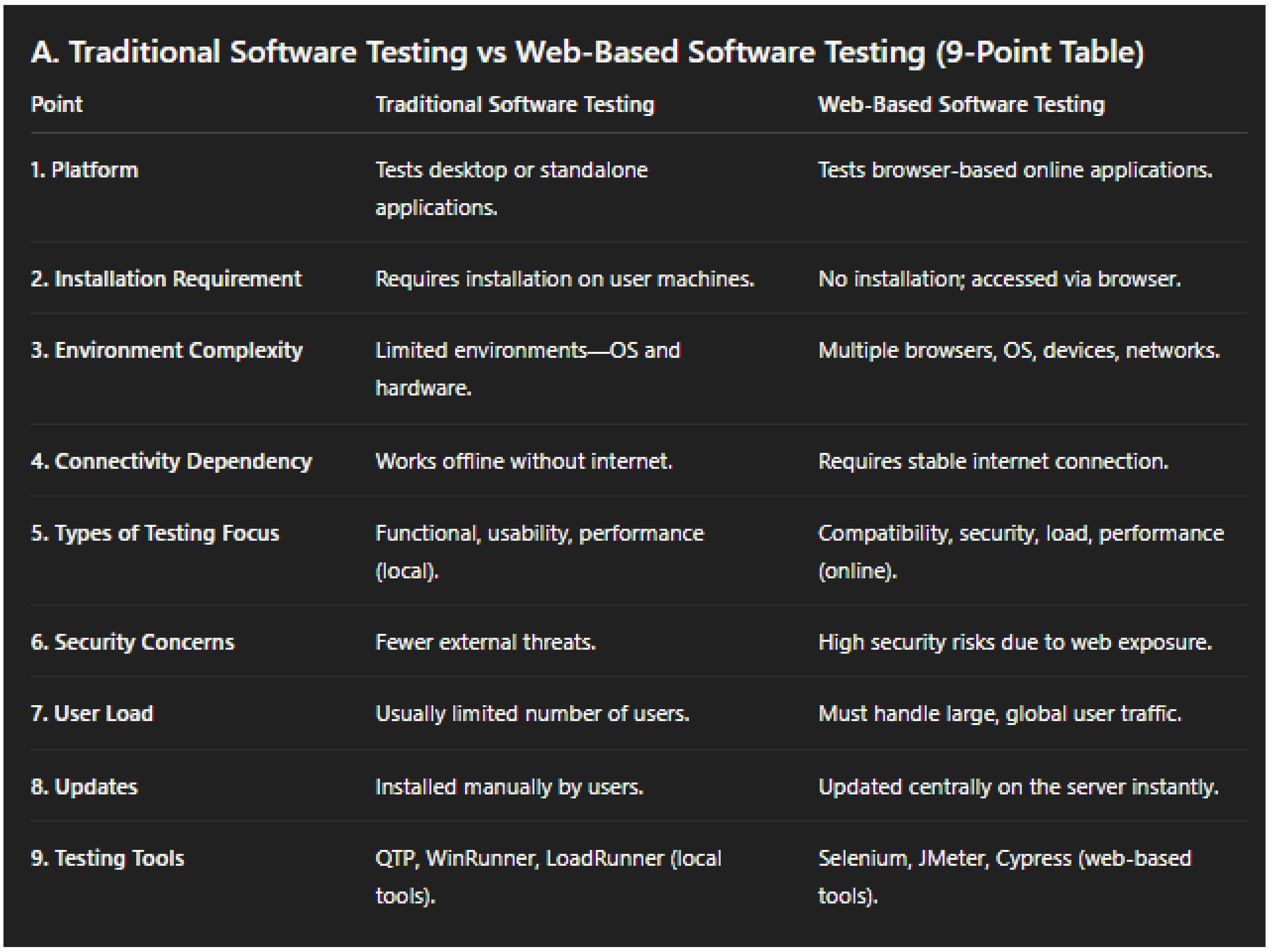
- High licensing cost
- Limited browser support compared to Selenium
- Requires skilled testers for advanced scripting
- Not suitable for small teams or low-budget projects

6. Use Cases

RFT is ideal for **banking, insurance, ERP systems, enterprise automation, SAP testing, and large-scale regression** in organizations using IBM tools.

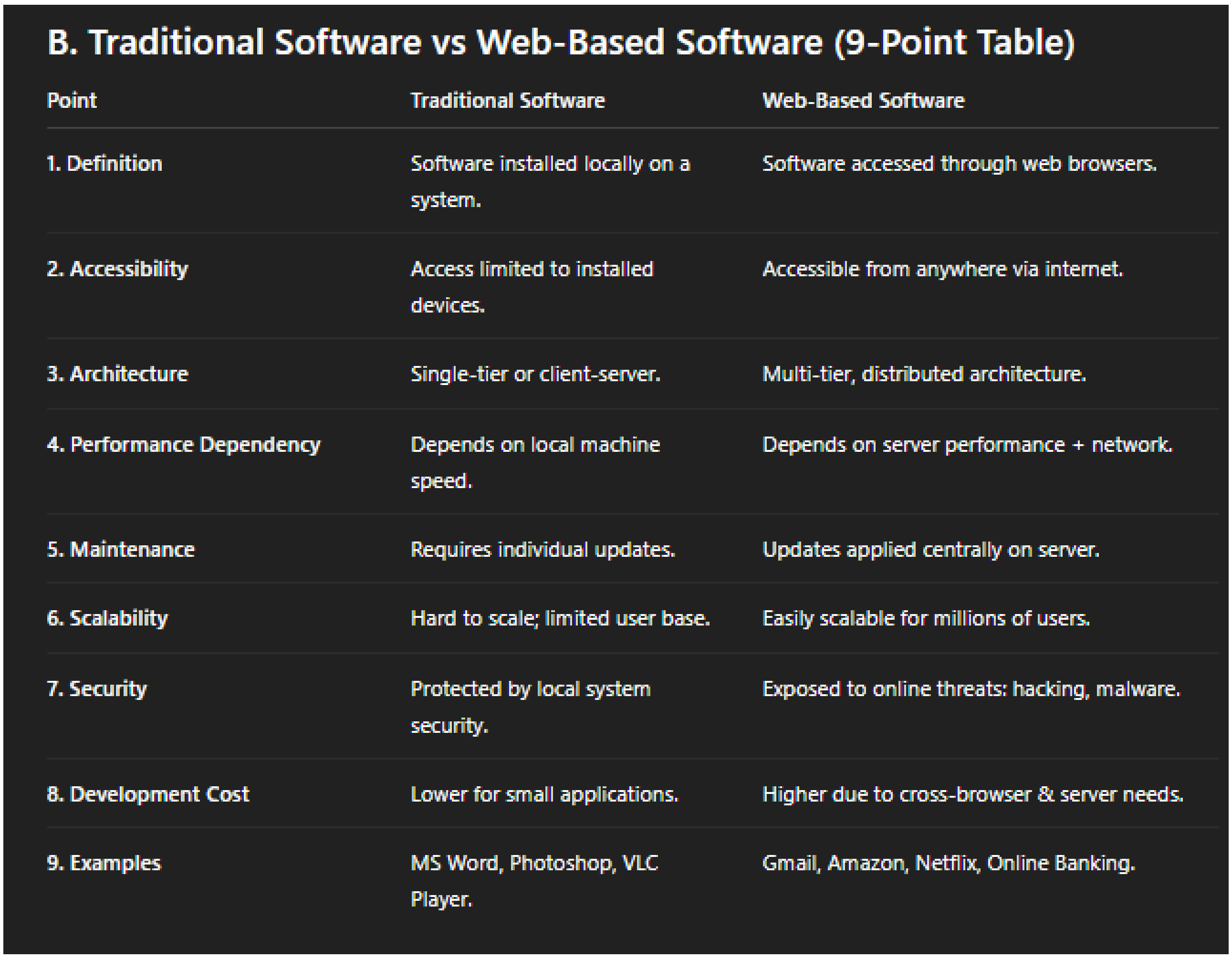
UNIT–V : Testing for Specialized Environments

36. Compare Traditional Software Testing and Web based Software Testing. Compare the Traditional software and Web based software.



A. Traditional Software Testing vs Web-Based Software Testing (9-Point Table)

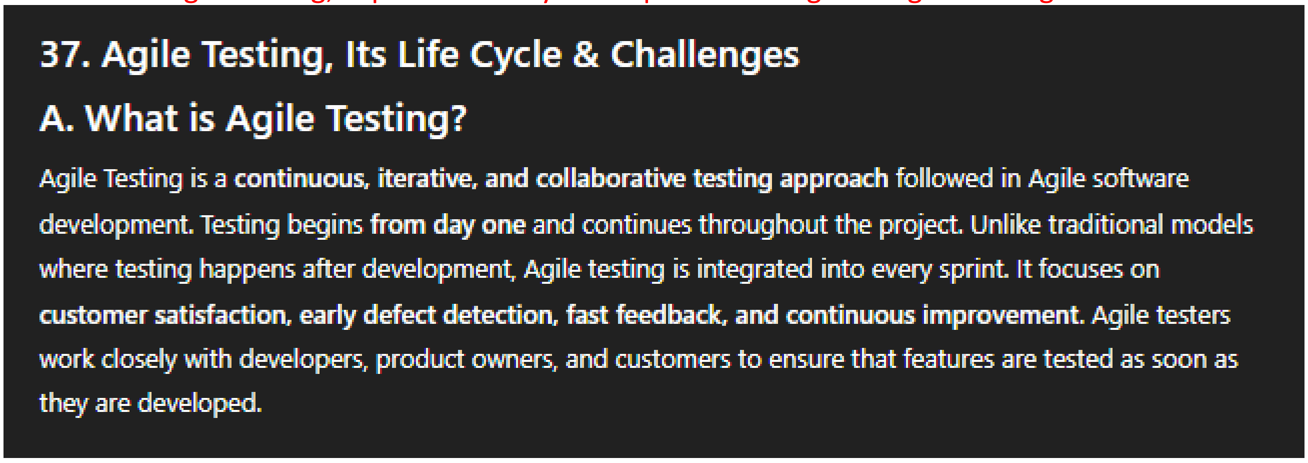
Point	Traditional Software Testing	Web-Based Software Testing
1. Platform	Tests desktop or standalone applications.	Tests browser-based online applications.
2. Installation Requirement	Requires installation on user machines.	No installation; accessed via browser.
3. Environment Complexity	Limited environments—OS and hardware.	Multiple browsers, OS, devices, networks.
4. Connectivity Dependency	Works offline without internet.	Requires stable internet connection.
5. Types of Testing Focus	Functional, usability, performance (local).	Compatibility, security, load, performance (online).
6. Security Concerns	Fewer external threats.	High security risks due to web exposure.
7. User Load	Usually limited number of users.	Must handle large, global user traffic.
8. Updates	Installed manually by users.	Updated centrally on the server instantly.
9. Testing Tools	QTP, WinRunner, LoadRunner (local tools).	Selenium, JMeter, Cypress (web-based tools).



B. Traditional Software vs Web-Based Software (9-Point Table)

Point	Traditional Software	Web-Based Software
1. Definition	Software installed locally on a system.	Software accessed through web browsers.
2. Accessibility	Access limited to installed devices.	Accessible from anywhere via internet.
3. Architecture	Single-tier or client-server.	Multi-tier, distributed architecture.
4. Performance Dependency	Depends on local machine speed.	Depends on server performance + network.
5. Maintenance	Requires individual updates.	Updates applied centrally on server.
6. Scalability	Hard to scale; limited user base.	Easily scalable for millions of users.
7. Security	Protected by local system security.	Exposed to online threats: hacking, malware.
8. Development Cost	Lower for small applications.	Higher due to cross-browser & server needs.
9. Examples	MS Word, Photoshop, VLC Player.	Gmail, Amazon, Netflix, Online Banking.

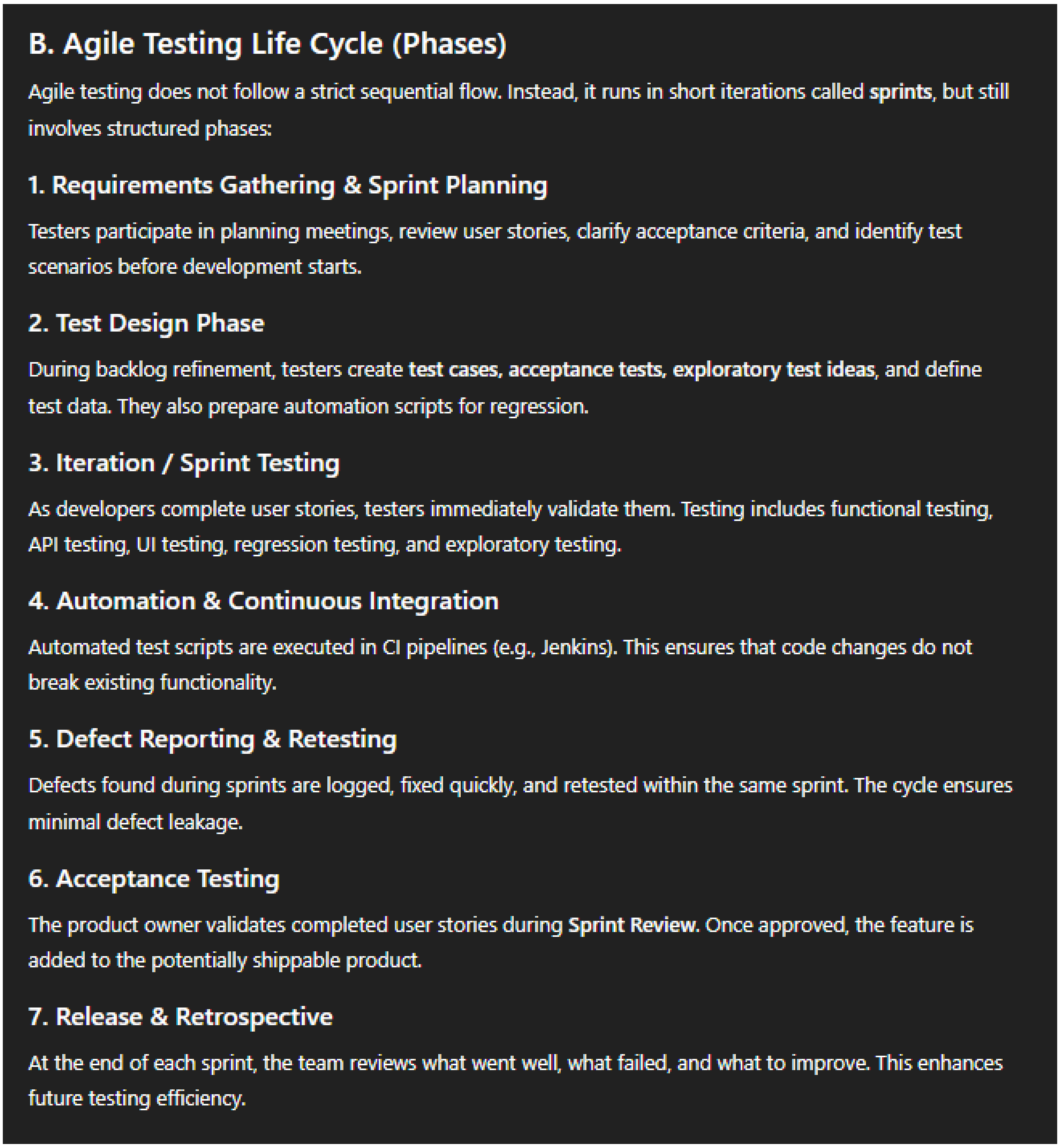
37. What is Agile testing, explain its life cycle? Explain challenges in Agile Testing.*



37. Agile Testing, Its Life Cycle & Challenges

A. What is Agile Testing?

Agile Testing is a **continuous, iterative, and collaborative testing approach** followed in Agile software development. Testing begins **from day one** and continues throughout the project. Unlike traditional models where testing happens after development, Agile testing is integrated into every sprint. It focuses on **customer satisfaction, early defect detection, fast feedback, and continuous improvement.** Agile testers work closely with developers, product owners, and customers to ensure that features are tested as soon as they are developed.



B. Agile Testing Life Cycle (Phases)

Agile testing does not follow a strict sequential flow. Instead, it runs in short iterations called **sprints**, but still involves structured phases:

1. Requirements Gathering & Sprint Planning

Testers participate in planning meetings, review user stories, clarify acceptance criteria, and identify test scenarios before development starts.

2. Test Design Phase

During backlog refinement, testers create **test cases, acceptance tests, exploratory test ideas**, and define test data. They also prepare automation scripts for regression.

3. Iteration / Sprint Testing

As developers complete user stories, testers immediately validate them. Testing includes functional testing, API testing, UI testing, regression testing, and exploratory testing.

4. Automation & Continuous Integration

Automated test scripts are executed in CI pipelines (e.g., Jenkins). This ensures that code changes do not break existing functionality.

5. Defect Reporting & Retesting

Defects found during sprints are logged, fixed quickly, and retested within the same sprint. The cycle ensures minimal defect leakage.

6. Acceptance Testing

The product owner validates completed user stories during **Sprint Review.** Once approved, the feature is added to the potentially shippable product.

7. Release & Retrospective

At the end of each sprint, the team reviews what went well, what failed, and what to improve. This enhances future testing efficiency.

C. Challenges in Agile Testing

1. Frequent Requirement Changes

User stories change often, making it difficult to maintain test cases and test data.

2. Limited Time for Testing

Sprints are short (1–3 weeks), so testers must complete testing quickly without compromising quality.

3. High Dependency on Communication

Miscommunication between developers, testers, and product owners can cause gaps in understanding requirements.

4. Continuous Regression Testing

Frequent builds require repeated regression testing; maintaining automation becomes challenging.

5. Incomplete or Evolving Requirements

User stories may lack details initially, making test planning difficult.

6. Tool and Automation Challenges

Automation must be stable and fast; slow or unreliable scripts cause delays.

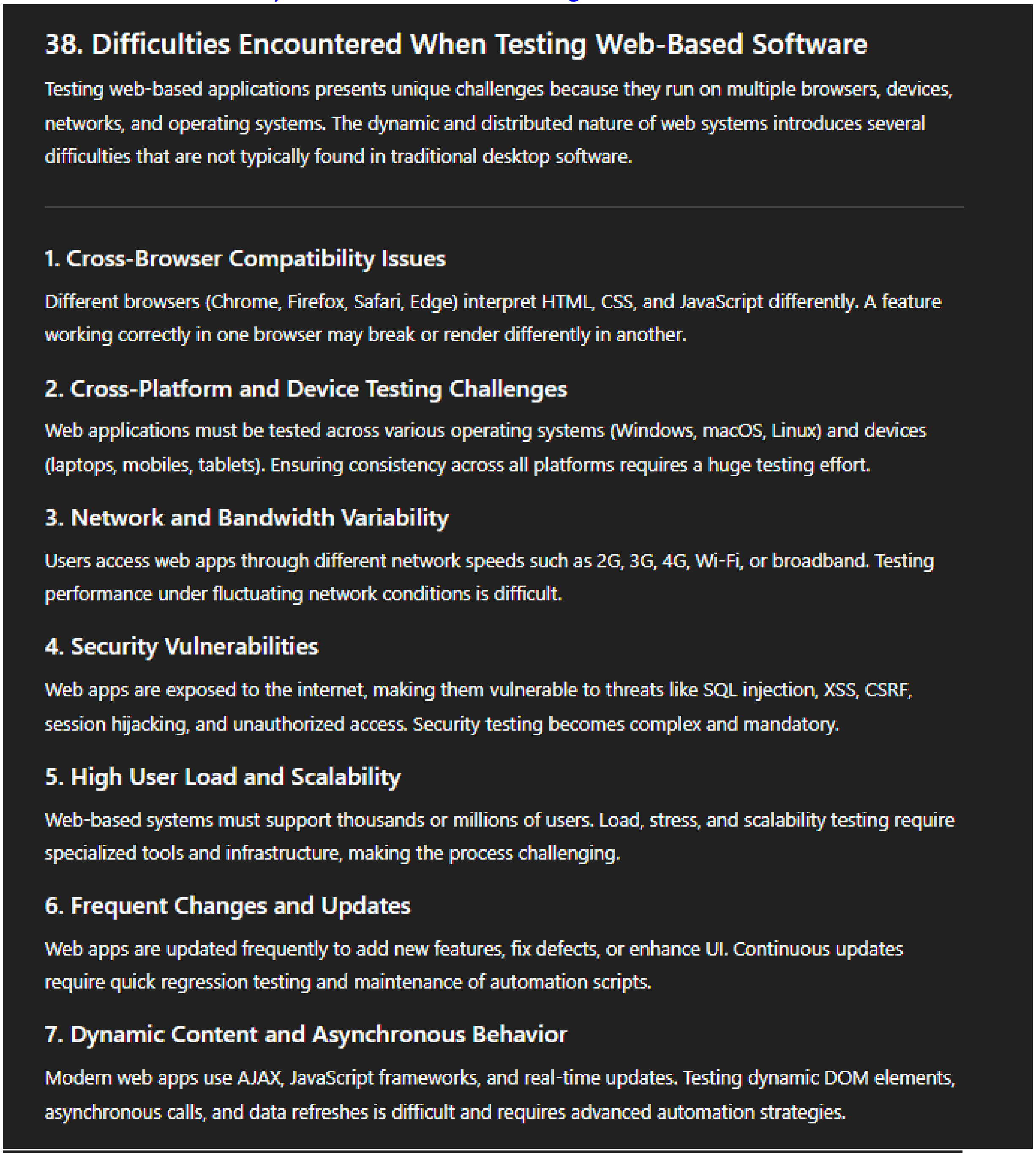
7. Skill Requirements

Agile testers need strong domain knowledge, technical skills, and communication ability.

8. Test Environment Issues

Shared environments may be unstable or frequently updated, impacting test execution.

38. What difficulties do you encounter when testing web-based software?



38. Difficulties Encountered When Testing Web-Based Software

Testing web-based applications presents unique challenges because they run on multiple browsers, devices, networks, and operating systems. The dynamic and distributed nature of web systems introduces several difficulties that are not typically found in traditional desktop software.

---

1. Cross-Browser Compatibility Issues

Different browsers (Chrome, Firefox, Safari, Edge) interpret HTML, CSS, and JavaScript differently. A feature working correctly in one browser may break or render differently in another.

2. Cross-Platform and Device Testing Challenges

Web applications must be tested across various operating systems (Windows, macOS, Linux) and devices (laptops, mobiles, tablets). Ensuring consistency across all platforms requires a huge testing effort.

3. Network and Bandwidth Variability

Users access web apps through different network speeds such as 2G, 3G, 4G, Wi-Fi, or broadband. Testing performance under fluctuating network conditions is difficult.

4. Security Vulnerabilities

Web apps are exposed to the internet, making them vulnerable to threats like SQL injection, XSS, CSRF, session hijacking, and unauthorized access. Security testing becomes complex and mandatory.

5. High User Load and Scalability

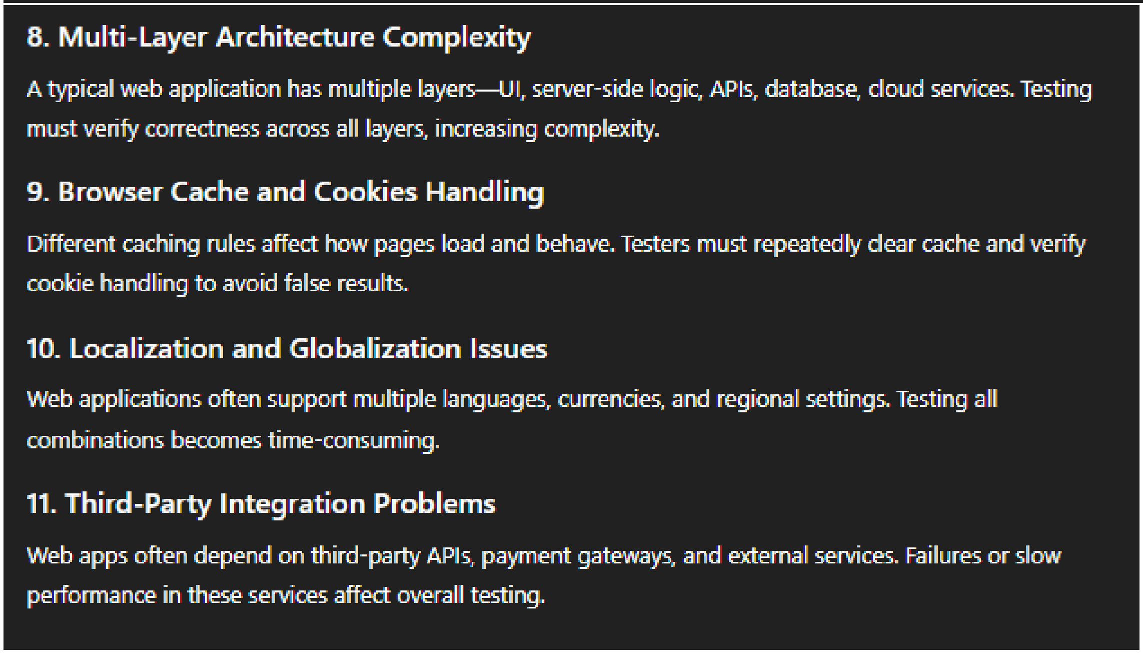
Web-based systems must support thousands or millions of users. Load, stress, and scalability testing require specialized tools and infrastructure, making the process challenging.

6. Frequent Changes and Updates

Web apps are updated frequently to add new features, fix defects, or enhance UI. Continuous updates require quick regression testing and maintenance of automation scripts.

7. Dynamic Content and Asynchronous Behavior

Modern web apps use AJAX, JavaScript frameworks, and real-time updates. Testing dynamic DOM elements, asynchronous calls, and data refreshes is difficult and requires advanced automation strategies.



8. Multi-Layer Architecture Complexity

A typical web application has multiple layers—UI, server-side logic, APIs, database, cloud services. Testing must verify correctness across all layers, increasing complexity.

9. Browser Cache and Cookies Handling

Different caching rules affect how pages load and behave. Testers must repeatedly clear cache and verify cookie handling to avoid false results.

10. Localization and Globalization Issues

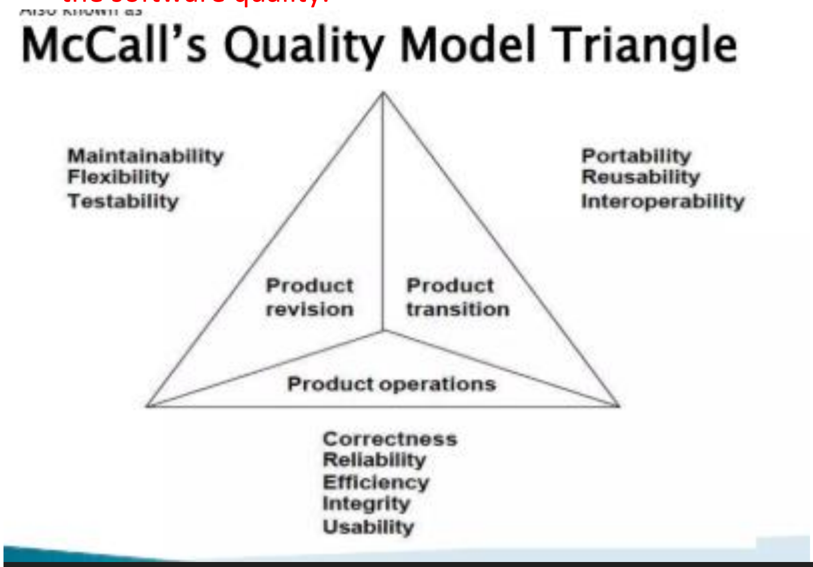
Web applications often support multiple languages, currencies, and regional settings. Testing all combinations becomes time-consuming.

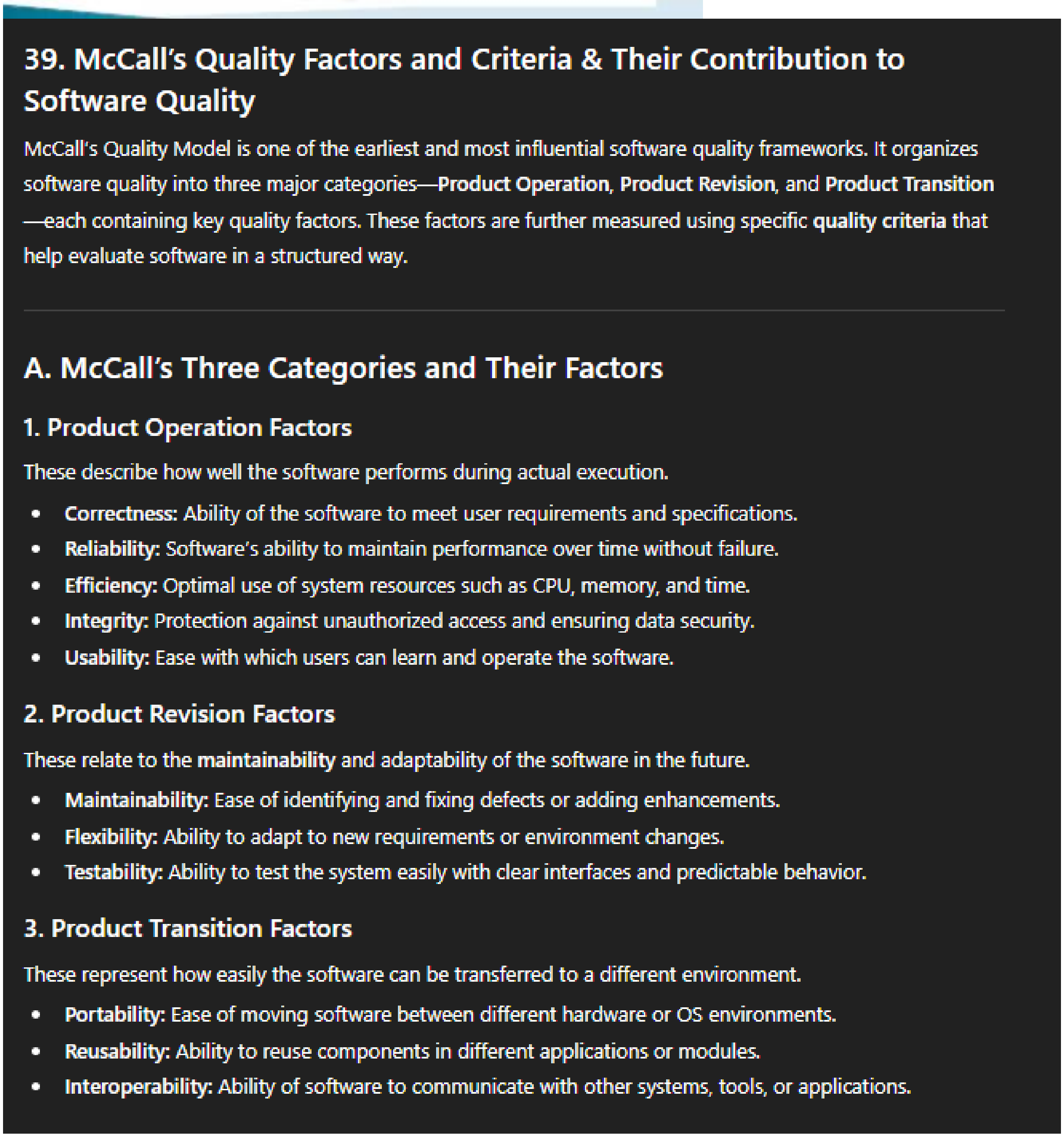
11. Third-Party Integration Problems

Web apps often depend on third-party APIs, payment gateways, and external services. Failures or slow performance in these services affect overall testing.

UNIT–VI : Quality Management

39. Explain McCall's quality factors and Criteria and how do they contribute the overall understanding of the software quality.





39. McCall's Quality Factors and Criteria & Their Contribution to Software Quality

McCall's Quality Model is one of the earliest and most influential software quality frameworks. It organizes software quality into three major categories—**Product Operation**, **Product Revision**, and **Product Transition**—each containing key quality factors. These factors are further measured using specific **quality criteria** that help evaluate software in a structured way.

A. McCall's Three Categories and Their Factors

1. Product Operation Factors

These describe how well the software performs during actual execution.

- **Correctness:** Ability of the software to meet user requirements and specifications.
- **Reliability:** Software's ability to maintain performance over time without failure.
- **Efficiency:** Optimal use of system resources such as CPU, memory, and time.
- **Integrity:** Protection against unauthorized access and ensuring data security.
- **Usability:** Ease with which users can learn and operate the software.

2. Product Revision Factors

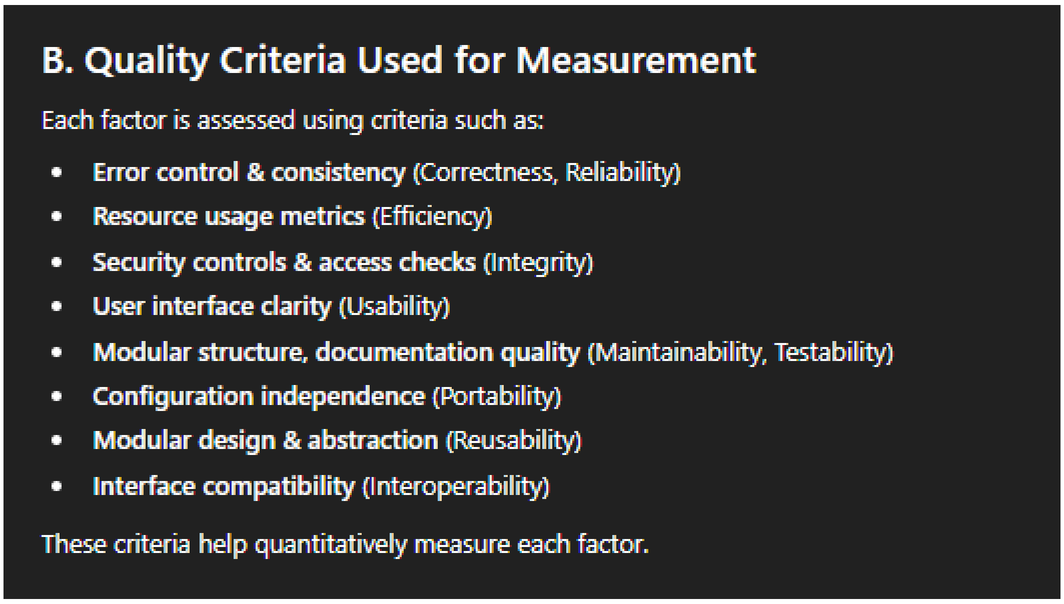
These relate to the **maintainability** and adaptability of the software in the future.

- **Maintainability:** Ease of identifying and fixing defects or adding enhancements.
- **Flexibility:** Ability to adapt to new requirements or environment changes.
- **Testability:** Ability to test the system easily with clear interfaces and predictable behavior.

3. Product Transition Factors

These represent how easily the software can be transferred to a different environment.

- **Portability:** Ease of moving software between different hardware or OS environments.
- **Reusability:** Ability to reuse components in different applications or modules.
- **Interoperability:** Ability of software to communicate with other systems, tools, or applications.

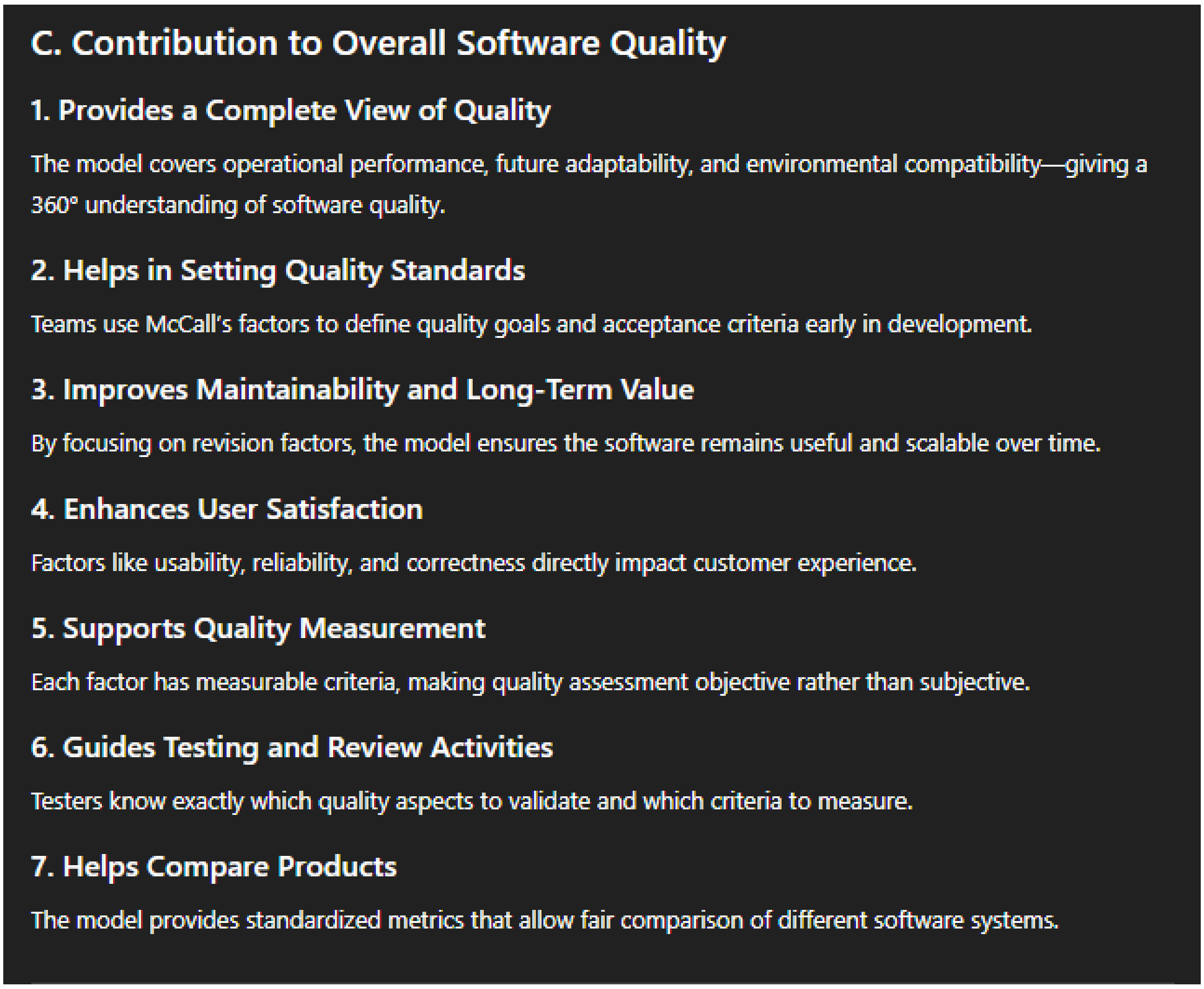


B. Quality Criteria Used for Measurement

Each factor is assessed using criteria such as:

- **Error control & consistency** (Correctness, Reliability)
- **Resource usage metrics** (Efficiency)
- **Security controls & access checks** (Integrity)
- **User interface clarity** (Usability)
- **Modular structure, documentation quality** (Maintainability, Testability)
- **Configuration independence** (Portability)
- **Modular design & abstraction** (Reusability)
- **Interface compatibility** (Interoperability)

These criteria help quantitatively measure each factor.



C. Contribution to Overall Software Quality

1. Provides a Complete View of Quality

The model covers operational performance, future adaptability, and environmental compatibility—giving a 360° understanding of software quality.

2. Helps in Setting Quality Standards

Teams use McCall's factors to define quality goals and acceptance criteria early in development.

3. Improves Maintainability and Long-Term Value

By focusing on revision factors, the model ensures the software remains useful and scalable over time.

4. Enhances User Satisfaction

Factors like usability, reliability, and correctness directly impact customer experience.

5. Supports Quality Measurement

Each factor has measurable criteria, making quality assessment objective rather than subjective.

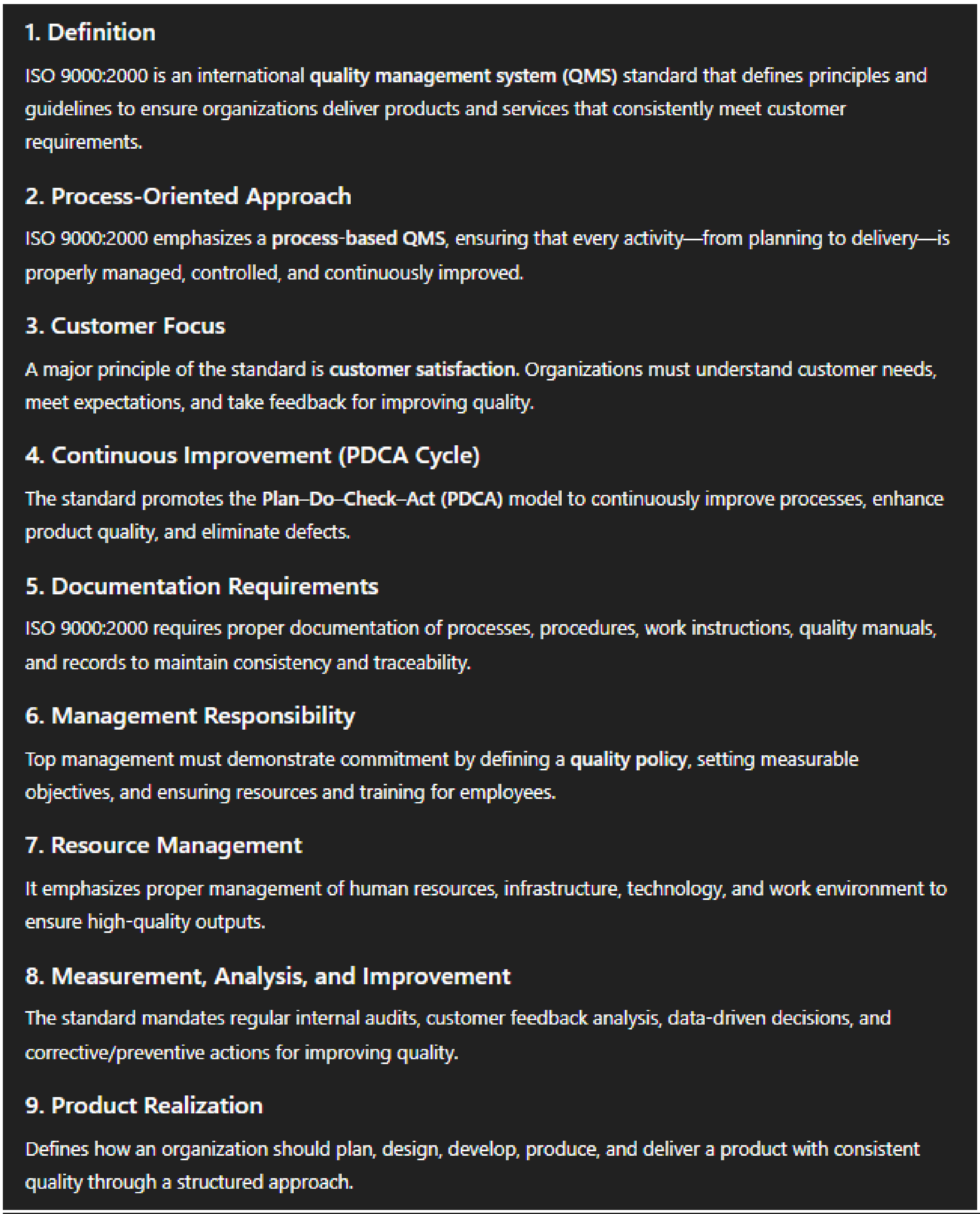
6. Guides Testing and Review Activities

Testers know exactly which quality aspects to validate and which criteria to measure.

7. Helps Compare Products

The model provides standardized metrics that allow fair comparison of different software systems.

40. Write a short note on ISO 9000:2000 standard.



1. Definition

ISO 9000:2000 is an international **quality management system (QMS)** standard that defines principles and guidelines to ensure organizations deliver products and services that consistently meet customer requirements.

2. Process-Oriented Approach

ISO 9000:2000 emphasizes a **process-based QMS**, ensuring that every activity—from planning to delivery—is properly managed, controlled, and continuously improved.

3. Customer Focus

A major principle of the standard is **customer satisfaction.** Organizations must understand customer needs, meet expectations, and take feedback for improving quality.

4. Continuous Improvement (PDCA Cycle)

The standard promotes the **Plan–Do–Check–Act (PDCA)** model to continuously improve processes, enhance product quality, and eliminate defects.

5. Documentation Requirements

ISO 9000:2000 requires proper documentation of processes, procedures, work instructions, quality manuals, and records to maintain consistency and traceability.

6. Management Responsibility

Top management must demonstrate commitment by defining a **quality policy**, setting measurable objectives, and ensuring resources and training for employees.

7. Resource Management

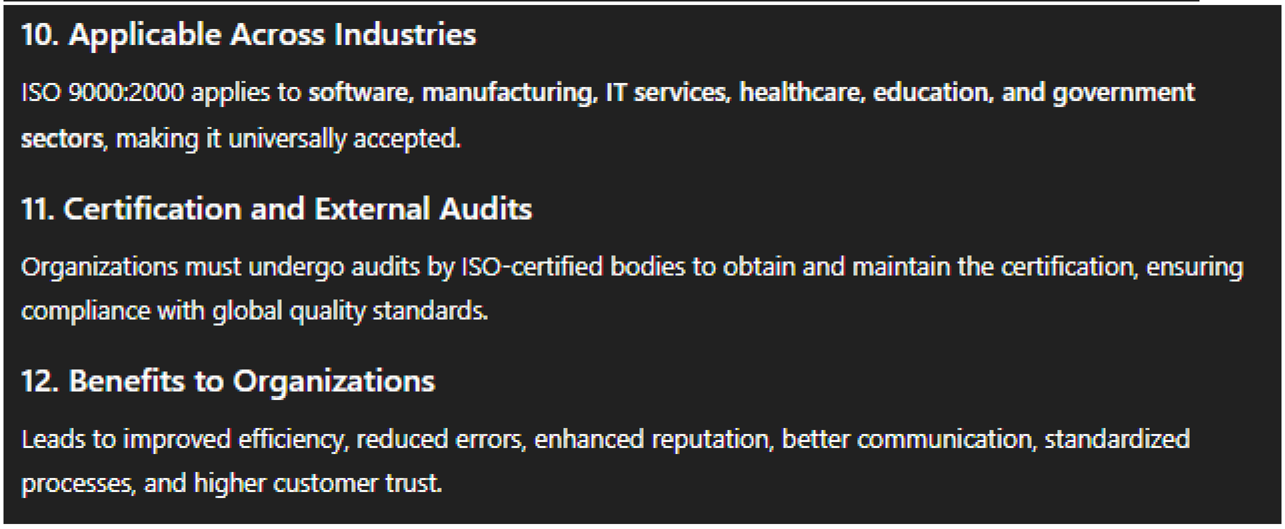
It emphasizes proper management of human resources, infrastructure, technology, and work environment to ensure high-quality outputs.

8. Measurement, Analysis, and Improvement

The standard mandates regular internal audits, customer feedback analysis, data-driven decisions, and corrective/preventive actions for improving quality.

9. Product Realization

Defines how an organization should plan, design, develop, produce, and deliver a product with consistent quality through a structured approach.



10. Applicable Across Industries

ISO 9000:2000 applies to **software, manufacturing, IT services, healthcare, education, and government sectors**, making it universally accepted.

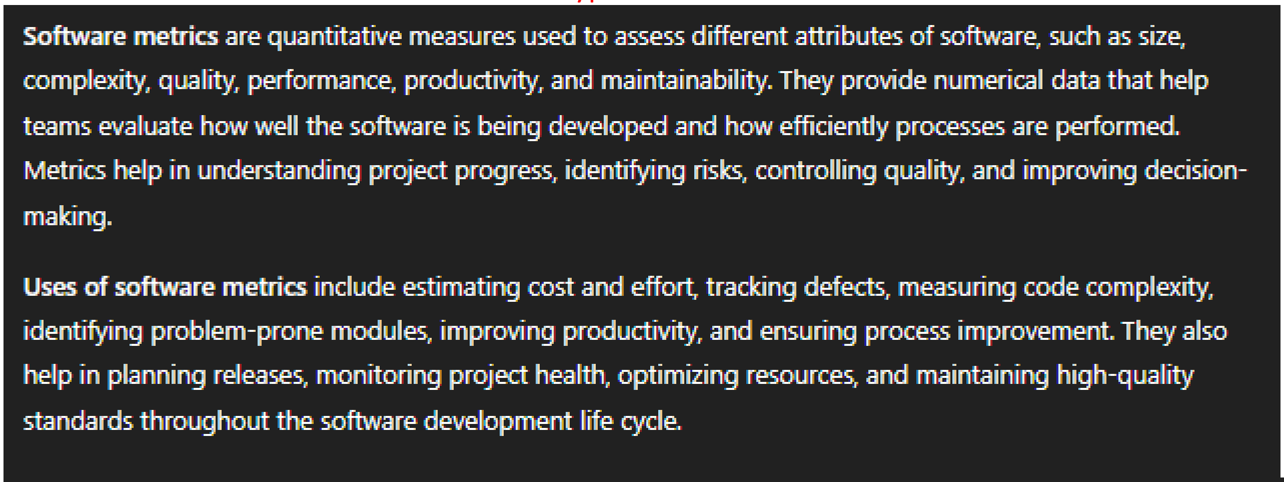
11. Certification and External Audits

Organizations must undergo audits by ISO-certified bodies to obtain and maintain the certification, ensuring compliance with global quality standards.

12. Benefits to Organizations

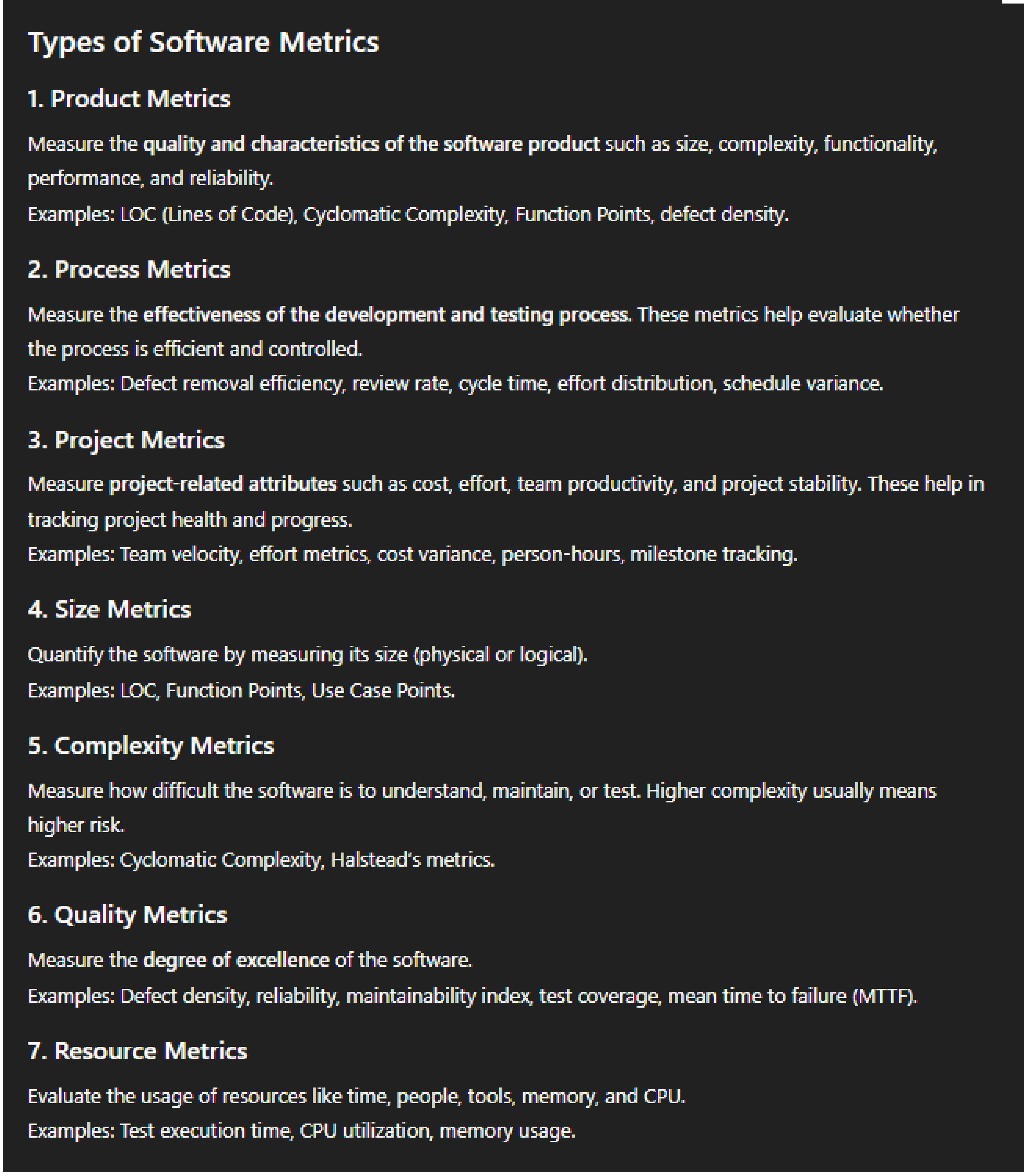
Leads to improved efficiency, reduced errors, enhanced reputation, better communication, standardized processes, and higher customer trust.

41. Define Software Metrics. List different types of Software metrics.*



Software metrics are quantitative measures used to assess different attributes of software, such as size, complexity, quality, performance, productivity, and maintainability. They provide numerical data that help teams evaluate how well the software is being developed and how efficiently processes are performed. Metrics help in understanding project progress, identifying risks, controlling quality, and improving decision-making.

Uses of software metrics include estimating cost and effort, tracking defects, measuring code complexity, identifying problem-prone modules, improving productivity, and ensuring process improvement. They also help in planning releases, monitoring project health, optimizing resources, and maintaining high-quality standards throughout the software development life cycle.



Types of Software Metrics

1. Product Metrics

Measure the **quality and characteristics of the software product** such as size, complexity, functionality, performance, and reliability.
Examples: LOC (Lines of Code), Cyclomatic Complexity, Function Points, defect density.

2. Process Metrics

Measure the **effectiveness of the development and testing process**. These metrics help evaluate whether the process is efficient and controlled.
Examples: Defect removal efficiency, review rate, cycle time, effort distribution, schedule variance.

3. Project Metrics

Measure **project-related attributes** such as cost, effort, team productivity, and project stability. These help in tracking project health and progress.
Examples: Team velocity, effort metrics, cost variance, person-hours, milestone tracking.

4. Size Metrics

Quantify the software by measuring its size (physical or logical).
Examples: LOC, Function Points, Use Case Points.

5. Complexity Metrics

Measure how difficult the software is to understand, maintain, or test. Higher complexity usually means higher risk.
Examples: Cyclomatic Complexity, Halstead's metrics.

6. Quality Metrics

Measure the **degree of excellence** of the software.
Examples: Defect density, reliability, maintainability index, test coverage, mean time to failure (MTTF).

7. Resource Metrics

Evaluate the usage of resources like time, people, tools, memory, and CPU.
Examples: Test execution time, CPU utilization, memory usage.