Testing 1

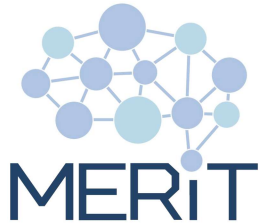
An Introduction

Introduction to Testing

Definition

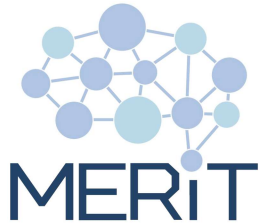
- The design is the process of evaluating a software system or its components to determine **whether it meets the specified requirements** and **is free of defects**.
- It ensures that the software **behaves as expected**, **under expected and unexpected conditions**.

Purpose



- Verify that the software **works correctly** according to requirements.
- Validate that the software **solves the right problem** for the user.
- Identify and eliminate **bugs**, **performance issues**, and **regressions**.
- **Reduce the risk of failure** in production.

Goals

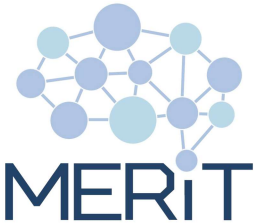


- **Correctness**: Does it do what it's supposed to do?
- **Reliability**: Does it behave consistently over time and conditions?
- **Performance**: Does it respond and scale acceptably under load?
- **Security**: Is it protected against unauthorized access or misuse?

Goals

- **Usability**: Does it provide an intuitive and error-resistant user experience?
- **Compliance**: Does it meet legal, safety, and domain-specific standards?

Testing vs. Debugging



- Detects what's wrong
- Usually automated or scripted
- Performed by QAs or Developers
- Identifies defects

- Finds why it's wrong
- Often manual and exploratory
- Performed by Developers
- Locates and fixes defects

Key Activities

- Plan Tests
- Design Test Cases
- Configure Test Environment
- Execute Tests
- Report and Track Defects
- Run Regression Tests

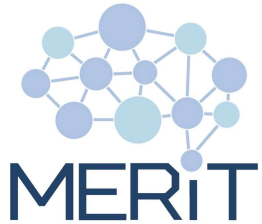
Key Deliverables

- Test Plan
- Test Cases
- Test Data
- Test Environment Configuration Documents
- Test Execution Reports
- Defect Reports and Bug Logs
- Test Summary Report

Key Roles

- Test Lead / QA Lead: Manages test strategy, team coordination, reporting, and closure.
- Tester / QA Engineer: Designs and executes tests, reports bugs, contributes to automation
- Automation Engineer: Designs and maintains test scripts for regression, CI/CD, and performance.
- DevOps Engineer: Supports environment setup, CI pipelines, monitoring, and test deploys.

Key Roles



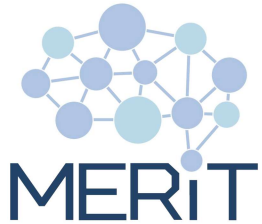
- Developer / Software Engineer: Fixes defects, writes unit tests, and collaborates on integration and regression.
- Product Owner / Business Analyst: Confirms test coverage matches requirements and business logic.
- End User: Participates in acceptance testing to validate system behavior against real needs.

Bug

Bug (Literally)

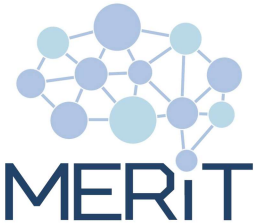
- In 1947, engineers working on the Harvard Mark II computer found that the system was **malfunctioning**.
- Upon investigation, they discovered **a moth trapped in a relay**.
- They taped the moth into the **logbook** with the note: “First actual case of bug being found”.
- This humorous log entry popularized the term “bug” to mean **a technical glitch** or **fault** in a system.

Definition



- A bug is a **mismatch** between **the actual behavior** and **the expected behavior** of the software, as **defined** by its requirements, specifications, or user expectations.
- It is any defect, flaw, or error in code or logic.
- Bugs can be introduced during any phase of software development.

Bugs in SDLC



- Analysis: Incomplete or incorrect requirements
- Design: flawed architecture or logic
- Implementation: coding mistakes or incorrect logic
- Testing: incorrect or incomplete test cases (yes, test code can have bugs too)
- Deployment: misconfiguration, integration issues

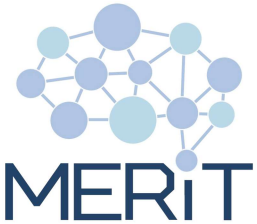
Testing

- The primary goal of testing is to:
  - ✔ Identify bugs **early**
  - ✔ **Report** them clearly
  - ✔ Ensure they are **fixed**
  - ✔ Retest to **confirm** resolution

Bug Types by Nature

- Functional: A feature doesn’t work as specified.
- Logical: Incorrect logic or flow.
- Syntax: Invalid code that won’t compile/run.
- Validation: Missing or improper input validation.
- Calculation: Wrong math operations or formulas.

Bug Types by Layer

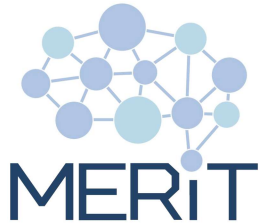


- UI/UX: Visual errors, alignment issues, accessibility.
- Database: Incorrect queries, constraint violations.
- API: Broken endpoints, incorrect responses, or status codes.
- Integration: Miscommunication between modules.
- Configuration: Wrong settings, missing files.

Bug Types by Behavior

- Crash: System freezes or crashes.
- Performance: Slowness, memory leaks.
- Security: Data leakage, unauthorized access.
- Concurrency: Race conditions, deadlocks.
- Load-related: Fails under high stress.

Bug Types by Fun



- Regression: Reappears after being fixed.
- Heisenbug: Disappears when debugged.
- Bohrbug: Reproducible and stable.
- Mandelbug: Chaotic, hard to reproduce.
- Schrodinbug: Only breaks after reading or noticing it.

Bug Lifecycle

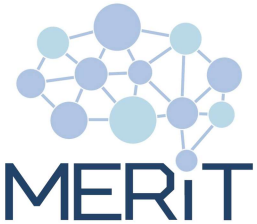
- A tester finds a bug during test execution.
- The bug is documented in a bug tracking system.
- The team evaluates severity, priority, and assigns ownership.
- Developers resolve the bug.
- Testers retest to confirm it’s fixed
- If fixed and verified, the bug is closed.

Bug Report

- Steps to Reproduce
- Expected Result
- Actual Result
- Screenshots/Logs
- Severity (impact on functionality)
- Priority (urgency to fix)
- Status (open, in progress, resolved, etc.)

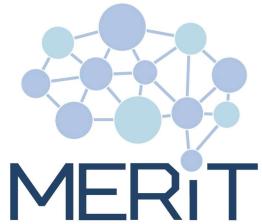
Test Types by Scope / Level

Unit Testing



- Testing individual units or components (e.g., functions, methods) in isolation to ensure they perform as expected.
- It is performed by developers.

Integration Testing

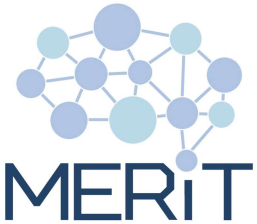


- Testing the interaction between multiple units or components to verify that they work together correctly and exchange data properly.
- It is performed by developers and testers.

System Testing

- Testing the system against business requirements and end-user needs to determine whether it is acceptable for delivery.
- It is performed by testers.

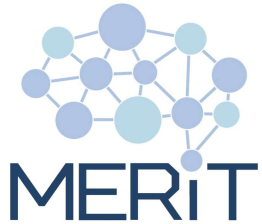
Acceptance Testing



- Acceptance testing determines whether the software meets business requirements and is ready for release.
- It is performed by the product owner, the business analyst, and the end user.

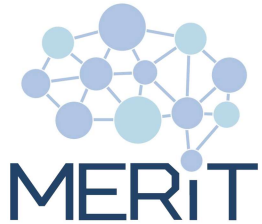
Test Types by Approach

Black-Box Testing



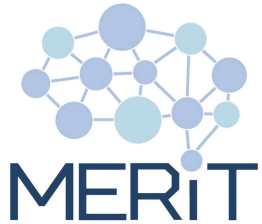
- Testing without any knowledge of the internal code or structure.
- It focuses purely on inputs and expected outputs.
- It validates functional correctness against specifications.
- It ensures that the system handles normal, edge, and invalid inputs appropriately.

White-Box Testing



- Testing with full knowledge of the internal logic and structure of the application.
- It examines control flow, loops, and paths.
- It ensures all code paths, branches, and logic conditions are exercised.
- It detects bugs such as off-by-one errors, unreachable code, infinite loops, and logic flaws.

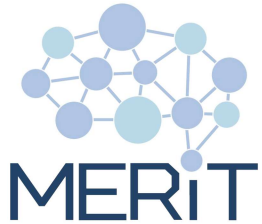
Gray-Box Testing



- Testing with partial knowledge of the internal workings of the system.
- It is a hybrid of black-box and white-box methods.
- It leverages internal knowledge to test external behavior more effectively.
- It simulates scenarios that a user or attacker with limited system insight might try.

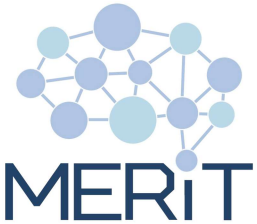
Test Types by Execution

Manual Testing



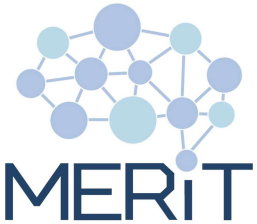
- Human testers execute test cases and explore the application manually without using automation scripts.

Automated Testing



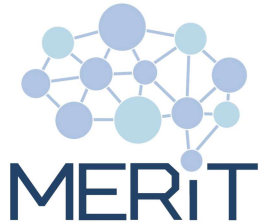
- Testing is performed using scripts, tools, or frameworks to execute predefined test cases automatically and repeatedly.
- It reduces human effort and increases test execution speed, ensuring consistent and repeatable validation of software behavior.
- It supports continuous testing in modern DevOps workflows.

Exploratory Testing



- Simultaneous learning, test design, and execution by a tester.
- It relies on domain expertise and intuition rather than formal test cases.

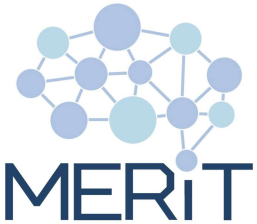
Ad-Hoc Testing



- Informal, unplanned testing without documentation.
- It is usually performed once to quickly find obvious bugs.

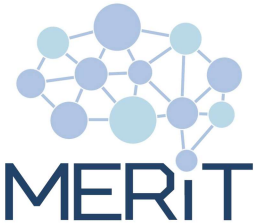
Test Types by Purpose

Functional Testing



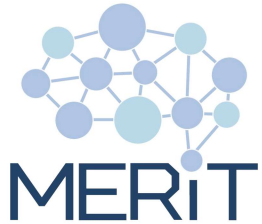
- Verifies that the software behaves according to the functional specifications.
- It focuses on user-facing features and business logic.

Non-Functional Testing



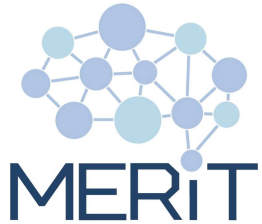
- Validates aspects such as performance, scalability, security, usability, and reliability.
- It focuses on how the system behaves.

Regression Testing



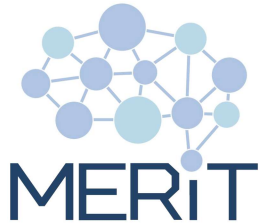
- Re-running previously passed tests to ensure that recent code changes have not broken existing functionality.

Smoke Testing



- A quick set of basic tests to check whether the major functionalities of a build are working well enough for further testing.

Sanity Testing

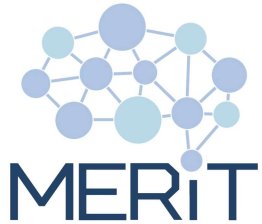


- Focused testing of a specific functionality or bug fix to verify that it works and hasn't caused new issues.

Performance Testing

- Evaluates how the system performs under expected and stress conditions , such as response time, throughput, and resource usage.

Security Testing



- Identifies vulnerabilities, risks, and security flaws in the software to protect data and resources.

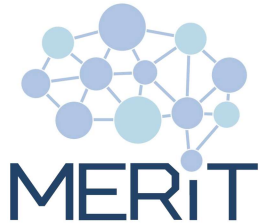
Compatibility Testing

- Checks how the software performs across different devices, operating systems, browsers, and environments.

Usability Testing

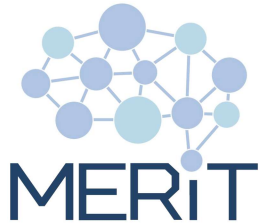
- Measures how intuitive, user-friendly, and accessible the software is from the end-user's perspective.

Recovery Testing



- Tests how well the system recovers from crashes, failures, or other catastrophic problems.

Compliance Testing



- Ensures that the software conforms to regulatory, legal, or organizational standards (e.g., GDPR, HIPAA, ISO).

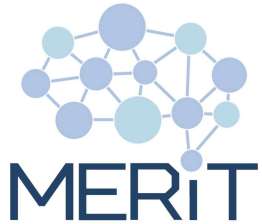
Test Pyramid

Definition

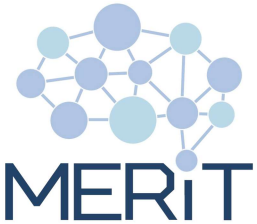
- The Test Pyramid is a conceptual framework that guides the **organization and prioritization** of different types of software tests within a **testing strategy**.
- It emphasizes writing **a large number of low-level**, fast, and reliable tests (like unit tests), and **fewer high-level**, slower, more complex tests (like UI or end-to-end tests).
- It promotes **cost-effective** and **maintainable** test cases.

Layers

- 1. Unit Tests (Base Layer)
- 2. Integration Tests (Middle Layer)
- 3. End-to-End (E2E) / UI Tests (Top Layer)

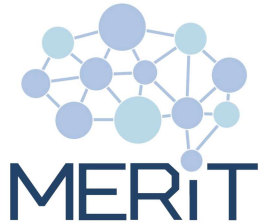


Layer 1



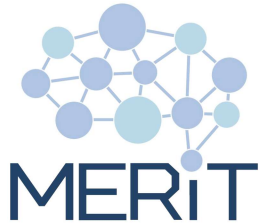
- Scope: Test individual functions or components in isolation.
- Fast, low cost, and high coverage.
- Examples: Testing a login validator, a math function, or a single class.

Layer 2



- Scope: Test interactions between components or systems.
- Slower, more complex setup than unit tests.
- Examples: Testing service → DB interaction, or API → logic → database.

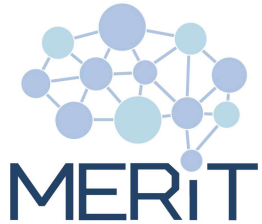
Layer 3



- Scope: Test the entire application flow, including UI, backend, and database.
- Slow, flaky, expensive to maintain.
- Examples: Simulating a user registering, logging in, buying a product.

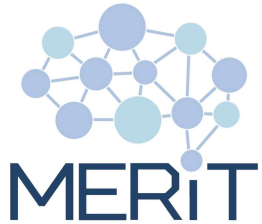
Test Trophy by Kent C. Dodds

Definition



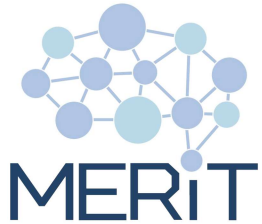
- The Test Trophy is a **modern testing model** proposed by Kent C. Dodds that emphasizes **writing more integration tests** than unit or end-to-end tests, while also promoting static tests (like linting and type checks) as the foundation.
- It encourages realistic testing of component interactions over excessive isolation.

Key Ideas



- **Integration tests** (especially **with real components**) are more valuable than numerous isolated unit tests.
- Static analysis (like TypeScript or ESLint) should be foundational.
- Don’t over-isolate logic; real component integration gives better confidence.

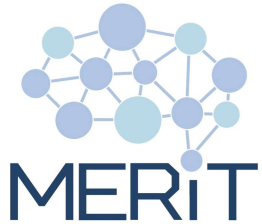
Used In



- Modern JavaScript/React apps
- Full-stack apps with shared components

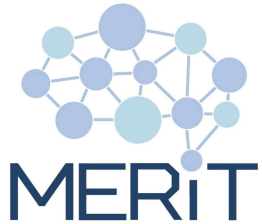
Test Honeycomb by Spotify

Definition



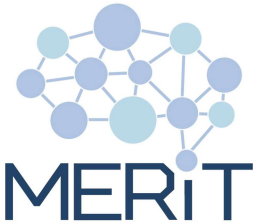
- The Test Honeycomb is a **multidimensional testing model** created by Spotify, replacing the traditional pyramid hierarchy with a distributed and flexible approach.
- It highlights the importance of **diverse test types**, including performance, chaos, security, and exploratory tests **across a wide system surface**.

Key Ideas



- Emphasizes **non-functional testing** (resilience, performance, observability).
- Supports **microservices** and **distributed systems**.
- Promotes **horizontal thinking**: many small, fast, purpose-driven tests across disciplines.

Used In



- Large-scale, cloud-native, microservices-heavy systems.