IT Project Guidance

Baseline Development View (Testing)

Version:

1.0

## Purpose

The purpose of this document is to summarise expectations for the development of automated tests for a system, documented in a reusable manner such that it can be distributed as is to relevant stakeholders, and being referenced from a project’s SAD.

## Synopsis

As covered within *IT Project Guidance - Resourcing IT Developers and Architects* [1] test development is distinct from system service (backend), front-end development, infrastructure & delivery pipeline development, with its own standards, best practice & patterns to improve quality and outcomes.

## Contents

[Purpose 1](#_Toc192006490)

[Synopsis 1](#_Toc192006491)

[Contents 2](#_Toc192006492)

[When to Test 4](#_Toc192006493)

[Where to Test 4](#_Toc192006494)

[What to Test For 5](#_Toc192006495)

[Before testing anything, it is essential to understand what needs to be validated. This includes both the system’s initial contractual System Requirements and the evolving User Stories that extend or refine functionality over time. 5](#_Toc192006496)

[System Requirements 5](#_Toc192006497)

[Functional Requirements 5](#_Toc192006498)

[Non-Functional Requirements 6](#_Toc192006499)

[How to Test - Testing Approaches 6](#_Toc192006500)

[Manual Testing 7](#_Toc192006501)

[Exploratory Testing 7](#_Toc192006502)

[Automated Testing 7](#_Toc192006503)

[Static Testing 8](#_Toc192006504)

[Dynamic Testing 8](#_Toc192006505)

[Standards 9](#_Toc192006506)

[ISO-25010 9](#_Toc192006507)

[ISO-25012 9](#_Toc192006508)

[ISO-25022 9](#_Toc192006509)

[ISO/IEC/IEEE 29119 (Software Testing Standard) 9](#_Toc192006510)

[IEEE Standard 1671 (Automatic Test Markup Language (ATML)) 10](#_Toc192006511)

[Gherkin 10](#_Toc192006512)

[Best Practice 10](#_Toc192006513)

[Testing Patterns 10](#_Toc192006514)

[No Direct Data Access 10](#_Toc192006515)

[Test Data Management 10](#_Toc192006516)

[Automated Testing 10](#_Toc192006517)

[Short 11](#_Toc192006518)

[Arrange Act Assert 11](#_Toc192006519)

[Gherkin for Test Definition 11](#_Toc192006520)

[Sequential Test Development 11](#_Toc192006521)

[Service Testing Patterns 11](#_Toc192006522)

[Static Testing of the Application Layer APIs 11](#_Toc192006523)

[Dynamic Tests of the Application Layer APIs 12](#_Toc192006524)

[Service Client Testing Patterns 12](#_Toc192006525)

[User Interface Testing 12](#_Toc192006526)

[Appendices 13](#_Toc192006527)

[Appendix A - Document Information 13](#_Toc192006528)

[Versions 13](#_Toc192006529)

[Images 13](#_Toc192006530)

[Tables 13](#_Toc192006531)

[Review Distribution 14](#_Toc192006532)

[Audience 14](#_Toc192006533)

[Structure 14](#_Toc192006534)

[Diagrams 14](#_Toc192006535)

[Standards 14](#_Toc192006536)

[Acronyms 14](#_Toc192006537)

[Terms 15](#_Toc192006538)

## Why Test

Testing is the foundation for ensuring software quality, system reliability, and long-term maintainability. It serves two critical purposes: ensuring that the system delivers value to end users and preserving institutional knowledge long after the original developers have left. A well-structured test framework not only validates that the service meets its intended purpose but also encodes the intellectual property behind its functionality, allowing future developers to iteratively fix and improve with confidence.

## When to Test

Testing is not just a step in initial delivery but a continuous process that safeguards the system throughout its lifecycle. Once system and frontend developers have completed their work and moved on, testing remains the only viable mechanism for protecting the system from unintended regressions introduced by support and operations developers making changes without access to those who developed the original logic.

## Where to Test

Modern best practices no longer rely on manual testing but instead use automation to achieve the same outcomes and more. Automated tests are integrated into the delivery pipeline, ensuring consistent validation at every stage of development and deployment. These pipelines, which incorporate test execution alongside compilation, packaging, deployment, and provisioning, are described in detail within *ICT Project Guidance - Design - SAD - Baseline Development View (Deployment) [1].*

Automated tests must be integrated into Continuous Integration/Continuous Deployment (CI/CD) pipelines, preventing Traditionally, system testing (ST) was performed manually, but this approach is now considered an anti-pattern. Instead, manual exploratory testing should be conducted in the non-production ST environment, not as a primary testing method, but as a means to identify scenarios that require automated tests. Once identified, these tests are formally documented and provided as work items for test developers, who implement them as automated test cases.

The developed automated tests are then committed to source control and integrated into the development pipeline, ensuring they are executed consistently as part of the delivery process.

For further guidance on integrating tests into the pipeline, refer to *IT Project Guidance – Design – SAD – Baseline Development View (Delivery)* [3]

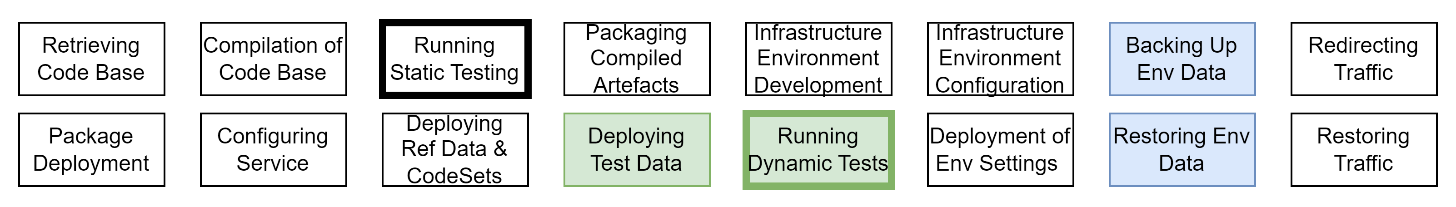


Figure 2: Default Deployment Pipeline Steps

## What to Test For

### Before testing anything, it is essential to understand what needs to be validated. This includes both the system’s initial contractual System Requirements and the evolving User Stories that extend or refine functionality over time.

### System Requirements

The System Requirements are a logical set composed of both Functional and Non-Functional Requirements – along with later added User Story functional acceptance criteria.

#### Functional Requirements

The goal of functional testing is to confirm that the system delivers the capabilities defined in the original system’s **Functional Requirements**, while also meeting the acceptance criteria of **User Stories** that were introduced later in a more Agile manner due to them either not being fully realised at the time of the initial system design, or a change in priorities, feedback, or refinements made during production.

Functionality testing is best organized in a way that reflects the logical dependencies of system capabilities:

##### Baseline Capabilities Tests

These tests ensure the system has stable foundational capabilities upon which all other services rely. Examples of baseline functionality includes Diagnostics Tracing, Error Logging, Session Management, Session Operation Audit Management, Permissions Management, Roles Development Management, Groups Management, User Management, and Group and Resource Role Management. For a detailed list of baseline system capabilities, refer to *IT Guidance – Design – SAD – Capabilities View (Service) [3].*

##### Business Service Capabilities

The second layer of functional tests the stakeholder and user capabilities built on top of the baseline capabilities, built upon the baseline capabilities..

Since business functionality is often not sufficiently defined in detail at the outset, these tests are often sourced from the acceptance tests captured in User Stories that emerged afterwards.

#### Non-Functional Requirements

Mature Non-Functional Requirements (NFRs) are typically structured around ISO-25010 System Quality Requirements categories, ensuring that tests assess system performance, security, maintainability, usability, and other quality attributes.

Testing these categories validates that the service is being made available within agreed Quality of Service constraints (security, configurability, maintainability, performance, etc).

Even if the Non-Functional Requirements are incomplete or not explicitly structured under ISO-25010, best practice is to develop tests that measure achievement in each of these areas. Any Non-Functional Requirements that are later defined will align with one or more of these categories, ensuring that test coverage is forward-compatible with evolving system needs.

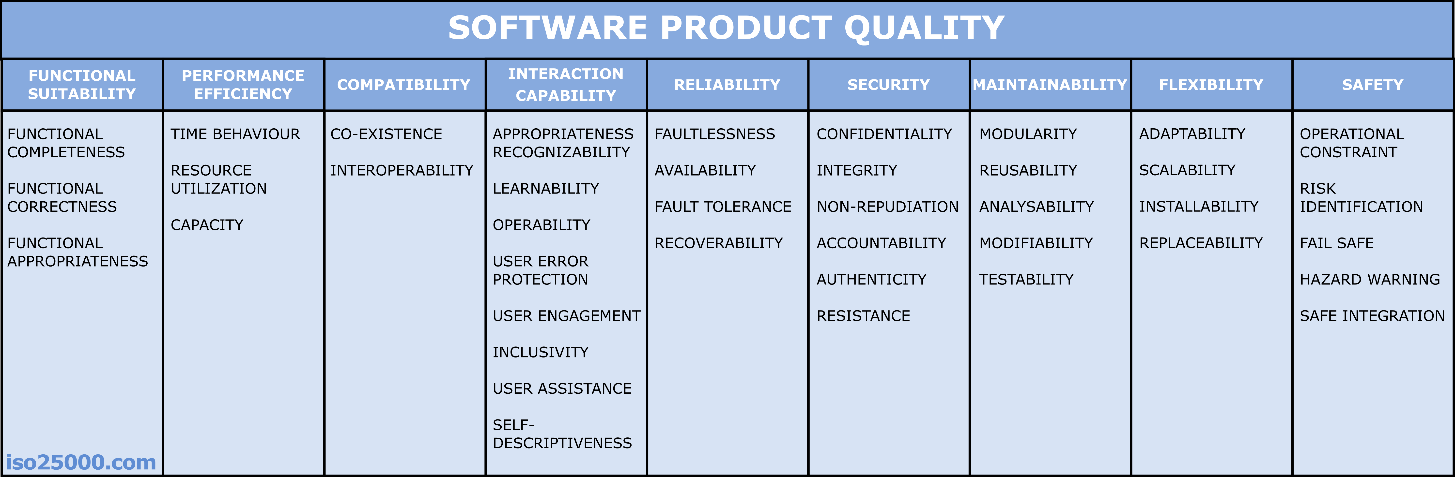


Figure 1: Non-Functional Quality Requirements (ISO-25010)

## How to Test - Testing Approaches

Beyond understanding what to test, it is essential to determine which types of tests are best suited to evaluate how well each aspect of the system is delivered. Different types of testing serve different purposes, with some being more applicable to specific validation needs than others.

Each testing approach has advantages that make it advantageous to measure particular aspects of a system’s functionality, or non-functional qualities. Some tests focus on preventing defects early in development, while others validate real-world behaviour in deployed environments. Choosing the right approach ensures that testing is both effective and efficient, providing meaningful insights at each stage of the software lifecycle.

The following sections outline the key types of testing used to ensure system quality and stability.

#### Manual Testing

Traditionally, system testing was performed manually by executing predefined test scripts. However, this approach is now considered an **anti-pattern** due to its inefficiency, brittleness, and inconsistency. Manual testing is slow, prone to human error, and difficult to scale, making it unsuitable for modern development practices.

Manual execution of scripted tests should not be introduced into any current project. Instead, testing must be automated to ensure repeatability, reliability, and integration within the development pipeline. Testing resources should be selected based on their ability to implement and maintain automated testing. For guidance on selecting appropriate resources, refer to *IT Project Guidance – Resourcing – Developers and Architects* [2]

#### Exploratory Testing

The term ‘exploratory testing’ is misleading as it not a structured testing method but rather an *investigative* process used to design better tests.

Unlike manual testing, which follows predefined scripts, exploratory testing involves actively probing the system to uncover edge cases, unexpected behaviours, and gaps in automation coverage. It is a flexible approach that helps identify scenarios that might not be anticipated in standard test cases.

As exploratory testing does not produce consistent, repeatable results on its own, it should not be relied upon as a primary testing method. Instead, its main purpose is to inform the creation of automated test cases, ensuring broader coverage of complex or unpredictable scenarios. The findings from exploratory testing are then formalized into automated tests, which are integrated into the development pipeline for continuous execution.

#### Automated Testing

Instead of *manual testing*, all testing - bar exploratory testing - should be fully automated and executed within the pipeline, enabling rapid feedback and minimizing reliance on manual effort. For further details on integration of testing into pipelines, refer to IT Project Guidance – Design – SAD – Baseline Development View (Delivery) [3].

bar ‘exploratory testing, Automated Testing should cover all testing types outlined below.

#### Static Testing

Static Tests are typically run as part of the build process, after compilation, executed on the build server itself, catching issues early. .

Static testing tests individual functions and for anything more complex, leverages mocking or similar techniques.

Note that it is possible to concentrate on only static testing the Application Layer APIs as it ensures that all used code in lower layers is tested while avoiding unnecessary validation of code paths that are never executed in practice. However, while it reduces the amount of tests required, it does require mocking of most underlying infrastructure services so as to not require actual service deployment.

If all static tests are successful, the code can be packaged and deployed to a target environment, which is prerequisite for dynamic tests.

#### Dynamic Testing

Dynamic tests are fully automated and executed within the delivery pipeline. Dynamic tests don’t use mocks and instead use configured, integrated services directly. Dynamic testing also relies on test data, designed specifically for testing, having been loaded into the target environment.

**Critical:**Production data must never be used, as doing so introduces sever security risks as well as makes tests brittle and/or unreliable due to inevitable changes in live data over time.

##### [Dynamic] Smoke Testing

Dynamic testing usually starts with one or more smoke tests. Smoke testing is a preliminary check to ensure that a deployed service is functioning at a basic level before deeper tests are executed. It verifies that the system starts correctly, critical paths are operational, and the most essential functionalities do not fail immediately. Typically automated, smoke tests are the first tests run in a development pipeline after deployment to a test environment. If these tests fail, further testing is halted, as it indicates that fundamental issues prevent meaningful execution of other tests.

##### Dynamic UX API Testing of the Application Layer APIs

These tests validate the APIs that back the services client. By careful synchronisation of calls whole workflows can be simulated without reliance on automating a service client, making the tests far more enduring.

Tests should include normal flows under realistic conditions while ensuring that services handle expected and edge-case scenarios correctly.

##### Dynamic Contract [Integration API] Testing

Distinct from internal UX APIs are the integration APIs made available to other services and service consumers. Contract Testing verifies that services adhere to predefined and published API specifications, ensuring that updates or modifications do not break compatibility for consumers that depend on them.

Although microservices are generally not recommended for most system designs due to complexity and operational overhead, in cases where they are used, contract testing helps ensure that service dependencies remain stable. By defining explicit contracts between services, contract tests confirm that each service continues to function correctly when integrated with others.

Dynamic User Interface Testing

Testing of user interfaces remains contentious. While they are the most brittle form of tests – as interfaces are the most prone to change – they have limited value, and Best Practice is to prefer Dynamic UX API Testing.

However, traditional Testers without development experience, prefer their use as it is the most intuitive to them, or at least the closest to manual testing.

##### Additional Dynamic Testing Aspects

Other forms of dynamic testing to consider are Mutation, Chaos, Canary and Shadow testing, but these are beyond the scope of this summary document.

## Standards

It is a recommendation – and depending on context, an obligation - that development of any kind adhere to standards.

#### ISO-25010

The ISO standard for System Qualities. Essential information for Non-Functional Testing.

#### ISO-25012

The ISO standard for System *Data* Qualities. Essential information for Non-Functional Testing.

#### ISO-25022

The ISO standard for System *User Experience* Qualities. Essential information for Non-Functional Testing.

#### ISO/IEC/IEEE 29119 (Software Testing Standard)

ISO 29119 specifies general concepts in software testing and presents key concepts for the ISO/IEC/IEEE 29119 series.

#### IEEE Standard 1671 (Automatic Test Markup Language (ATML))

IEEE Standard 1671, more widely known as Automatic Test Markup Language, is an XM-based standard originally developed for electronic systems.   
*At present there isn’t a use case for software testing, however this may change in time.*

#### Gherkin

Not technically a standard, but it is a well-known and widely used testing Domain Specific Language (DSL).

## Best Practice

The following are key patterns to implement when developing testing of the service and service client components of IT services.

### Testing Patterns

The following are patterns applicable to delivering best practice outcomes.

#### No Direct Data Access

Testing must not require direct database access, which always introduces a large security issue. All testing should instead be conducted by automation via application interfaces (e.g. APIs that support the UI) or be exploratory via the interface, to inform later automated testing.

Best practice is to use “test data” developed specifically for tests.

#### Test Data Management

Develop data for testing; under NO circumstances should production data be used, even in anonymized or otherwise scrambled form.   
Just providing direct access to production data to a maintenance person (or worse, a System Support Specialist or Business Analysts) to extract production data to make test data is a complete sidestep of security controls.

#### Automated Testing

Testing is not done manually - instead automate tests as code that runs in the pipeline.

Note:  
An environment (usually ‘ST’) is still accessed by manual testers, but it is only for exploratory manual testing in order to developer instructions to provide to a test developer to develop the automation that is then run in a pipeline.

#### Short

Like general coding best practice, Automated Test development should be short.

#### Arrange Act Assert

Each tests should be organised into three conceptual sections:   
- Arrange [the dependencies and arguments],   
- Act [perform the actual work] ,   
- Assert [that the results are as expected].

#### Gherkin for Test Definition

While Gherkin is not a formal standard, it is a structured, human-readable language for defining tests in a format that closely aligns with the development thought process: **inputs, function, output** or **Arrange, Act, Assert.** Defining tests in Gherkin before implementation ensures clarity and avoids misinterpretation between Business Analysts (BAs), Test Analysts (TAs), and Developers.

#### Sequential Test Development

Historically, work items have been assigned in parallel to both testers and developers, leading to discrepancies in interpretation, rework, and delays. A sequential approach ensures consistency:

1. **BAs and SMEs** define requirements.
2. **TAs define Gherkin tests** that explicitly capture expected behaviour.
3. **Developers implement automated test cases** before writing system code.
4. The system code is developed iteratively until all tests pass.

This method ensures that system code only enters the shared codebase once it meets defined test expectations.

### Service Testing Patterns

The following are patterns specific to testing Services, as opposed to Service Clients (ie Browsers).

#### Static Testing of the Application Layer APIs

Service testing patterns focus on verifying the functionality, reliability, and performance of services rather than their clients, such as browsers or other user-facing interfaces.

Static testing involves leveraging mocking or similar techniques to achieve comprehensive test coverage of APIs without requiring actual service deployment.

Leverage mocking or other approaches to provide complete coverage of the Application Layer’s APIs.

Note:  
Covering the application layer essentially equates to 100% coverage of used code in lower layers, while not requiring testing of code that never comes into play, therefore not an issue.

#### Dynamic Tests of the Application Layer APIs

Deploy the service to a Build Test (BT) environment for automated testing of the Application Layer’s APIs using data specific developed specifically for the tests.

Do NOT use production data as the basis of tests: not only is it introduces a large security issue, but it also makes tests ‘brittle’ as production data changes over time.

### Service Client Testing Patterns

The following are patterns specific to testing Service Clients – i.e., Browsers in the case of SPAs and API end points

#### User Interface Testing

The User Interface is the most volatile part of a system, frequently changing in ways that break existing test suites. To avoid excessive maintenance, direct UI testing should be limited to a few high-level smoke tests covering business-critical workflows for each major stakeholder group—users, service providers, support, and operations. These tests should validate end-to-end functionality rather than individual views or operations.

Beyond this, prioritise tests that chain multiple operations together—such as searching, selecting, viewing, editing, and verifying outcomes. These sequences naturally exercise multiple API endpoints in a single run, reducing redundancy while improving coverage.

Complement this with isolated automated tests for each API operation to close any gaps. Avoid unnecessary validation of visual placements—real users interact with the system organically, not by comparing screens side by side. Instead, focus on ensuring that workflows remain functional and reliable as the system evolves.

Appendices

Appendix A - Document Information

Author & Contributors:

Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft
  2. Additions, Corrections, Reformatting

1.0 First Release

### Images

[Figure 1: Non-Functional Quality Requirements (ISO-25010) 4](#_Toc191988943)

[Figure 2: Default Deployment Pipeline Steps 10](#_Toc191988944)

### Tables

**No table of figures entries found.**

References

|  |  |
| --- | --- |
| [1] | Sigal, IT Project Guidance - Resourcing - IT Developers and Architects. |
| [2] | Sigal, IT Project Guidance - Design - SAD - Baseline Development View (Deployment). |
| [3] | Sigal, IT Project Guidance - Design - SAD - Baseline Development View (Delivery). |
| [4] | Sigal, IT Project Guidance - Design - SAD - Baseline Development View (Infrastructure). |
| [5] | Sigal, IT Project Guidance - Design - SAD - Baseline Development View (Testing). |
| [6] | Sigal, IT Project Guidance - Design - SAD - Baseline Development View (Service Client). |
| [7] | Sigal, IT Project Guidance - Resourcing - IT Developers and Architects. |
| [8] | Sigal, “IT Project Guidance - Design - SAD - Baseline Capabilities View (Service)”. |
| [9] | Sigal, “IT Project Guidance - Design - SAD - Baseline Development View”. |
| [10] | Sigal, “IT Project Guidance - Discovery - Obligations, Agreements, Commitments, Standards, Principles (New Zealand Government Agencies)”. |

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
| Russell Campbell, Project Manager |  |
| Rebecca Clark, Business Analyst |  |
| Vincent Wierdsma, Lead Developer |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Standards

ISO-25010

: …

ISO-25012

: …

ISO-25022

: …

### Acronyms

Refer to the project’s Glossary.

ATML

: acronym for [Automatic Test Markup Language](#_Automatic_Test_Markup)

DSL

: acronym for [Domain Specific Language](#_Domain_Specific_Language).

IT

: acronym for Information, using Technology to automate and facilitate its management.

##### ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. IT is a subset of ICT.

### Terms

##### Automatic Test Markup Language (ATML)

: an XML based description of Tests.

##### Domain Specific Language (DSL)

: a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem.