SOP RPACT GxP Solution

SOP-RPACT-004

Version 1.0.0

Contents

RPACT GmbH

Am Rodenkathen 11
23611 Sereetz
Germany
www.rpact.com

1 Standard Operating Procedure for Developing, Deploying, and Maintaining a GxP Solution

SOP-ID	SOP-RPACT-004
Title	SOP for Developing, Deploying, and Maintaining a GxP Solution
Description	Standard operating procedure (SOP) for the process of developing, deploying, and maintaining GxP-compliant software solutions
Author	Friedrich Pahlke
Version	1.0.0
Date	2024-09-24

This SOP describes the process for developing, deploying, and maintaining software solutions that must meet GxP (Good Practice) compliance standards. The document outlines the steps for ensuring that such solutions are developed in a controlled, validated environment, with a focus on automation and traceability.

1.1 Process Overview

1.1.1 Purpose

The purpose of this SOP is to establish a structured process for ensuring that all stages of software development, deployment, and maintenance follow GxP requirements. This ensures that the solutions provided are reliable, compliant, and validated.

1.1.2 Scope

This SOP applies to all software development activities aimed at creating GxP-compliant solutions. It covers the validation of software, the traceability between requirements and test cases, as well as ongoing maintenance and compliance checks.

1.2 Validation Plan

1.2.1 Defining the Validation Plan

The validation plan forms the core of the development process for GxP-compliant software. It is designed to ensure that all components of the solution are developed, tested, and validated in a manner consistent with regulatory standards. This includes:

- Validation of individual software components (functional units, interfaces, outputs)
- Automation of key validation tasks using the `rpact.validator` tool
- Traceability of requirements to corresponding test cases and validation results
- Risk management and minimization strategies (see @sec-risk-assessment for detailed risk assessment)

Our validation plan also addresses deviations from the plan, ensuring they are documented and corrected.

1.2.2 Automation with `rpact.validator`

To streamline the validation process, we use our custom-built `rpact.validator` tool. This tool automates many aspects of the validation process, including:

- Generating and updating validation documentation
- Creating traceability references between requirements and test scripts
- Automatically linking test results to specific requirements, ensuring full traceability

By automating these processes, we reduce human error and increase the efficiency and accuracy of our validation efforts.

1.2.3 Version Control and Traceability

Version control and traceability of all code changes are managed through GitHub. Each code change is tracked, and a detailed history of modifications is maintained. This ensures that all updates and releases can be reviewed and traced back to specific requirements or bug fixes.

1.3 Requirements Definition and Documentation

1.3.1 Define and Document Requirements

Requirements are gathered from stakeholders and documented through user stories and functional specifications. These requirements are the foundation of the software development process and are stored in a requirements management system.

1.3.2 Traceability References

Using the `rpact.validator`, we automatically generate traceability references that links each requirement to one or more test cases. This ensures that every requirement is verified through testing, providing full visibility into how each aspect of the software meets its requirements.

1.3.3 Bug Tracking

All issues, bugs, and feature requests are tracked using GitHub Issues. This ensures that the entire team can monitor the progress of bug resolution and feature development. Each issue is assigned a priority level and is linked to the relevant code changes and tests for traceability.

1.4 Development and Testing

1.4.1 Development Process

The software development follows an agile methodology, allowing for iterative development, testing, and review of features. Each feature undergoes:

- **Code Review**: Ensures adherence to coding standards and quality.
- **Unit Testing**: Verifies that individual modules of the software behave as expected.
- **Integration Testing**: Ensures that different modules work together correctly.

1.4.2 Writing and Executing Test Scripts

Test scripts are written based on the documented requirements (see @sec-appendix-testing-strategy for details on the testing strategy). These scripts are then executed to verify the functionality of the software. Test cases cover the following areas:

- **Unit Tests**: Validate individual components of the software.
- **Integration Tests**: Ensure that different parts of the system work together seamlessly.
- **System Tests**: Validate the entire system in a production-like environment.
- **User Acceptance Tests (UAT)**: Allow end users to confirm that the solution meets their needs.

All test results are automatically linked back to the corresponding requirements using `rpact.validator`, ensuring comprehensive traceability.

1.4.3 Test Automation

Where possible, tests are automated to ensure consistency and efficiency. Automated tests cover unit, integration, and system tests, reducing the time needed for manual validation and increasing reliability.

1.4.4 Test Coverage and Review

We aim for a test coverage of $> 80\%$ to ensure that the majority of the code is covered by automated tests. This helps identify any areas that may require additional testing or review. The test coverage report is reviewed as part of our quality assurance process.

1.5 Risk Management and Minimization

1.5.1 Risk Assessment Levels

We apply a risk-based approach to the development and validation of GxP solutions. The following risk levels are used to assess the potential impact of different functions within the software:

- **High Risk**: Functions and calculations that directly or indirectly affect the planning and analysis of clinical trials. Errors in these functions could result in incorrect conclusions about the efficacy or safety of drugs, potentially causing harm to patients or misguiding regulatory decisions.
- **Medium Risk**: Functions that support the main analytical procedures but do not directly influence the critical outcomes of clinical trial planning and analysis. These could include data preparation or intermediate calculations that inform but do not determine the final analysis.

- **Low Risk**: Utility functions for specific output formats (e.g., R Markdown) or user-specific customizations that do not impact the core analytical procedures or outcomes of clinical trials. These functions assist in presenting results but do not affect the main calculations.

1.5.2 Mitigation Strategies

For each risk level, the following mitigation strategies are applied:

- **High Risk**: Rigorous unit and integration testing, additional peer reviews, and validation in a production-like environment.
- **Medium Risk**: Standard testing procedures with automated checks and reviews.
- **Low Risk**: Basic functional testing and user acceptance testing (UAT).

1.6 Deployment Process

1.6.1 Deployment to GitHub

After all development and testing stages are completed, the source code is pushed to a secure GitHub repository. This repository can be public (open source software) or private (closed source software) to ensure the integrity and confidentiality of the code, and each release is tagged for traceability.

1.6.2 Deployment to Production Environment

Once the solution is fully tested and reviewed, it is deployed to the designated production environment. Deployment can include cloud-based or on-premise infrastructure, depending on the solution and client requirements. The deployment process includes:

- Automated Build and Deployment: Ensures consistency between development and production environments. Post-Deployment Testing: Final checks are performed to ensure that the software is functioning as expected in the live environment.

1.7 Ongoing Maintenance and Compliance

1.7.1 Regular Reviews and Updates

The validation plan, traceability references, and other key documentation are reviewed on a regular basis. This ensures that the software remains compliant with GxP standards, even as updates and changes are made.

1.7.2 Re-Validation After Updates

Any significant changes to the software, such as new features or major bug fixes, trigger a re-validation process. The `rpact.validator` tool is used to automate much of the re-validation, ensuring that any new functionality is fully tested and validated.

1.7.3 Documentation of Deviations

Any deviations from the validation plan are documented and reviewed. Corrective actions are taken to ensure that these deviations do not impact the overall compliance of the solution.

1.8 Confirmation of Execution and Review

By signing below, I confirm that all steps outlined in this SOP have been fully executed and reviewed. I certify that the procedures were followed as described, and that the results have been verified for accuracy and compliance with the relevant standards.

Name	Signature	Date

2 Appendix: Testing Requirements and Strategy

The testing strategy of software developed by RPACT is based on one major approach: Unit Tests. Unit tests will be used to compare the results of source code units with pre-defined expected results (control data). The control data for comparison will be generated in the following different ways:

1. An algorithm taken from the literature, e.g., the monograph by Wassmer and Brannath (2016) doi:10.1007/978-3-319-32562-0 will be used to generate it.
2. An alternative software package, e.g., the R package gsDesign (URL: cran.r-project.org/package=gsDesign), will be used to generate it.
3. Simulation algorithms will be used to generate it. The required algorithms will be implemented with R especially for that.

Unit tests will be performed using the R package testthat (URL: cran.rproject. org/package=testthat; project URL: github.com/r-lib/testthat). In `testthat` the test files live in the package folder `tests/testthat/`. The name of each test file must start with test.

Tests are organised hierarchically: expectations are grouped into tests which are organised in files:

- An expectation is the atom of testing. It describes the expected result of a computation: Does it have the right value and right class? Does it produce error messages when it should? An expectation automates visual checking of results in the console. Expectations are functions that start with `expect_`.

- A test groups together multiple expectations to test the output from a simple function, a range of possibilities for a single parameter from a more complicated function, or tightly related functionality from across multiple functions. This is why they are sometimes called unit as they test one unit of functionality. A test is created with test_that().
- A file groups together multiple related tests. Files are given a human readable name with context().

(Source: r-pkgs.org/testing-basics.html)

The strategy at RPACT for creating the subsequent test plan and the test protocol is to automate as much as possible with R. The following approaches will be used to do that:

- The test plan is a LaTeX document that will be generated automatically by running our custom-built `rpact.validator` tool that analyzes the code of all unit tests, parses the `expect_` definitions and builds LaTeX tables with test descriptions and control data (expected results).
- The test protocol for operational qualifications is a LaTeX report that will be generated automatically by running the `rpact.validator` tool that performs all unit tests, collects the results, builds LaTeX tables, result summaries and result statistics.
- The test protocol for installation qualification is based on the results of the win-builder service of the R project (win-builder.r-project.org). The results will be parsed by the `rpact.validator` tool and converted to a LaTeX report.

The big advantage of this approach is that the validation documentation for future releases of the software can be updated quite fast. Furthermore the method `testInstalledPackage()` included in the R core package tools can be used to perform all tests of the software on the locally installed R environment to guarantee that the software produces the same results on every machine with regard to the individual software and hardware configuration.