# SOSIS

Software product line Optimization for Safety-/mission-critical Industrial Systems

## D4.1 – Reuse Methodologies and Validation Techniques

Submission date of deliverable: February 28, 2026

Edited by: Hakan Kilinc (Orion, Türkiye), Ömercan Devran, Selin Şirin Aslangül (Beko, Türkiye) Eray Tüzün (Bilkent University, Türkiye), Daniel Esteban Villamil Sierra (UC3M, Panel Sistemas Spain), Juan Miguel Gomez (UC3M, Spain) Tamer Berk (Erste, Türkiye), Ural Sezer (IOTIQ, Germany)

| | |
|---|---|
| **Project start date** | May 1, 2024 |
| **Project duration** | 36 months |
| **Project coordinator** | Tamer Berk, ERSTE Software |
| **Project number & call** | 22029 - ITEA 4 |
| **Project website** | https://itea4.org/project/sosis.html |
| **Contributing partners** | See affiliations in “Edited by” above. |
| **Version number** | V1.2 |
| **Work package** | WP4 |
| **Work package leader** | David Barbachano (Panel, Spain) |
| **Dissemination level** | Public |
| **Description** ***(max 5 lines)*** | The purpose of document D4.1 is to address reuse methodologies and validation techniques. It systematically defines the questions: “How will we reuse? How will we verify? With which techniques? At what level?” It is the first output of Task 4.1. |

## Change Log

| Version | Date | Authors | Description of changes |
|---|---|---|---|
| 1.0 | 22.01.2026 | Hakan Kilinc (Orion) | First template and content are created. |
| 1.1 | 09.03.2026 | Tamer Berk | Updated the document |
| 1.2 | 10.03.2026 | Daniel Villamil | Final contribution and formatting |
| 1.3 | | | |
| 1.4 | | | |
| 1.5 | | | |
| 1.6 | | | |
| 1.7 | | | |
| 1.8 | | | |


## Executive Summary

The primary objective of WP4 is to define approaches that will enable certification and verification activities in safety-critical and mission-critical industrial systems to be made repeatable, systematic, and reusable. In this context, D4.1 presents a methodological framework for how reuse and verification processes should be planned and structured from the early development stages onwards. In doing so, it references the use cases defined in WP1. The approach aims to reduce verification costs in complex systems with an increasing number of variants and to increase reuse throughout the entire SDLC.


## Table of contents

## Document Glossary

| Acronym | Description |
|---|---|
| AI | Artificial Intelligence |
| APFD | Average Percentage of Fault Detection |
| AST | Abstract Syntax Tree |
| CIA | Change Impact Analysis |
| CI/CD | Continuous Integration/Continuous Delivery |
| CIT | Combinatorial interaction testing |
| ILP | Integer Linear Programming |
| IMS | Information Management System |
| LLM | Large Language Models |
| LSTM | Long Short-Term Memory |
| ML | Machine Learning (ML) |
| NLP | Nonlinear Programming |
| PBX | Private Branch Exchange |
| RNN | Recurrent Neural Network |
| SIP | Session Initiation Protocol |
| SoTA | State-of-the-Art |
| SPL | Software Product Line |
| SPLE | Software Product Line Engineering |
| TCP | Test Case Prioritisation |

# 1. Introduction

Work Package 4 focuses on defining methodologies and validation techniques that enable systematic reuse of software artefacts and verification evidence across Software Product Line (SPL) variants. As industrial systems increasingly rely on configurable architectures and multiple product variants, verification and certification activities often become repetitive and costly.

The objective of WP4 is therefore to define reusable validation strategies that allow verification artefacts, test results, and certification evidence to be reused across variants while maintaining compliance with safety and quality standards.

This deliverable presents the conceptual framework for reuse, including definitions of reusable components, classification strategies, validation techniques, and application to the project use cases.

# 2. Reusable Components: Definition and Properties

## 2.1. Definition of a Reusable Component

In the context of the **SOSIS project**, a *reusable component* is defined as a software, model, or test artefact that can be systematically integrated and reused across multiple variants within an **SPL** without compromising functional correctness, quality attributes, or certification requirements. A reusable component encapsulates well-defined functionality together with explicit interfaces, dependencies, operational assumptions, and associated validation artefacts. Beyond implementation code, reusable components may also include architectural models, configuration templates, test scenarios, and verification evidence. Each component is version-controlled and linked through traceability mechanisms to requirements, issues, commits, and feature models. This structured definition enables reuse decisions to be based on measurable criteria such as structural compatibility, validation status, and stability of quality metrics. Within SOSIS, reusable components serve as foundational building blocks that support scalable development, validation reuse, and consistent quality assurance across evolving software variants.

## 2.2. Classification of Reusable Components

Within the **SOSIS project**, reusable components are classified according to their role in the software development and validation lifecycle. This classification supports systematic reuse across **software product line (SPL)** variants by distinguishing between different types of artefacts that contribute to system functionality, architecture, and quality assurance.

First, **software components** represent the executable building blocks of the system, such as services, modules, libraries, or microservices that implement specific functionality. These components can be reused across variants when their interfaces, dependencies, and operational assumptions remain compatible.

Second, **model components** describe architectural and structural representations of the system, including feature models, system architecture models (e.g., C4 representations), configuration models, and dependency graphs. These artefacts enable understanding of system structure and variability and support automated reasoning about variant relationships and change impact analysis.

Third, **test components** include reusable test artefacts such as test cases, test scenarios, test data, and validation scripts. These components allow previously validated behaviours to be reused when the structural and functional properties of reused components remain unchanged.

By organising reusable artefacts into these categories, SOSIS enables a structured reuse strategy where functional implementations, architectural knowledge, and validation evidence can be systematically reused across product variants while maintaining traceability, consistency, and quality assurance.

## 2.3. Reuse Strategy and Methodology

The reuse strategy defined in this project is grounded in Software Product Line Engineering (SPLE) principles and aims to enable systematic, scalable, and auditable reuse across safety- and mission-critical industrial systems. Reuse is treated as a structured engineering practice rather than simple code sharing. It encompasses functional artefacts (software components and services), architectural artefacts (models and structural representations), test artefacts (test scenarios and validation suites), and verification evidence (risk assessments, dependability analyses, certification artefacts). Each reusable element is version-controlled, traceable to requirements and issues, and associated with both functional and non-functional properties.

The methodology adopts a multi-level abstraction approach. Reuse decisions are evaluated at the code, component, container, and system levels, ensuring alignment between architectural structure and feature variability. Reusable artefacts must explicitly document assumptions, operational constraints, quality attributes (e.g., reliability, maintainability, security), and validation status. This structured characterisation enables reuse not only in development activities but also in verification and certification processes.

To operationalise reuse, the methodology integrates automated artefact extraction and traceability mechanisms. Architectural information is derived from source code and dependency analysis, while issue tracking systems provide feature identifiers, change rationale, and risk metadata. These artefacts are connected through traceability graphs that support change impact analysis and variant delta analysis. Reuse eligibility is determined based on structural similarity, traceability stability, and quantitative metric thresholds, ensuring that reuse decisions remain evidence-based and auditable.

Through this strategy, reuse becomes measurable and governed by explicit rules rather than informal practice. By combining architectural modelling, traceability, and validation techniques, the methodology reduces redundant verification effort while maintaining compliance and quality standards across product variants. This approach directly supports the project's objective of making reuse and validation repeatable, systematic, and scalable in complex industrial software ecosystems.

# 3. Validation Techniques for Reusable Components

## 3.1. Formal Methods

Within the **SOSIS project**, formal methods are used as a rigorous validation technique to ensure that reusable components preserve their functional correctness and configuration consistency when reused across different software product line variants. Formal methods provide mathematically grounded techniques that allow system properties to be specified, analysed, and verified independently of implementation details. In the context of SOSIS, these techniques are primarily applied to **feature models, configuration constraints, and architectural relationships** to guarantee that variant configurations remain valid and consistent.

Model checking and constraint satisfaction techniques are employed to verify that feature combinations and configuration rules do not violate defined system constraints. Feature models representing variability across the product line are analysed to detect conflicts such as incompatible feature selections, missing dependencies, or invalid configurations. This ensures that reusable components can be safely integrated into different variants without introducing structural inconsistencies.

In addition, formal consistency checks are applied to architectural models and configuration rules derived from repository and dependency analysis. These checks allow the SOSIS framework to validate whether previously verified components remain valid when reused in a new variant configuration. By applying formal reasoning techniques to variability models and architectural representations, SOSIS ensures that reuse decisions are supported by **systematically verifiable constraints**, reducing the risk of configuration errors and improving the reliability of reusable validation artefacts across the software product line.

## 3.2. Fault Injection

Within the **SOSIS project**, fault injection is used as a validation technique to evaluate the robustness and resilience of reusable components when integrated into different software product line variants. Fault injection deliberately introduces controlled faults into a system to observe how components behave under abnormal or failure conditions. This approach helps verify whether reusable components preserve their reliability and error-handling capabilities across different configurations and operational contexts.

Fault injection can be applied at multiple levels of the system. **Software fault injection** targets application-level components by introducing faults such as invalid inputs, unexpected exceptions, communication failures, or resource limitations. **Hardware fault injection**, when relevant for embedded or mission-critical systems, simulates failures in hardware interfaces or external dependencies. These techniques allow the system to be evaluated under realistic failure scenarios.

In SOSIS, fault injection also supports **variant robustness analysis**. When a component is reused in a new variant configuration, fault injection tests can be executed to verify that changes in dependencies, feature combinations, or execution environments do not introduce unexpected failure behaviours. The results of these experiments become reusable validation artefacts, enabling organisations to reuse robustness evidence when the structural conditions of reuse remain

unchanged. This approach helps reduce redundant testing effort while ensuring that reused components maintain the required reliability and fault tolerance across the software product line.

### 3.3. Dependability Analysis

Dependability analysis ensures that reusable components preserve required levels of reliability, availability, and maintainability when integrated into different product variants. In safety- and mission-critical systems, reuse must not compromise system assurance arguments. Therefore, dependability properties are treated as first-class validation artefacts and are explicitly linked to architectural components, configuration parameters, and operational assumptions.

The analysis is performed at multiple abstraction levels. At the component and container levels, reliability metrics (e.g., failure rate, mean time between failures – MTBF), availability indicators (e.g., uptime ratio, mean time to repair – MTTR), and maintainability measures (e.g., change effort, defect density, modularity indicators) are evaluated. These quantitative indicators are derived from operational logs, CI/CD metrics, static analysis outputs, and historical defect data. When a component is reused in a new variant, dependability impact is assessed through structural delta analysis and change impact analysis to determine whether previously validated RAM properties remain valid.

To support systematic reuse, dependability models such as reliability block diagrams, fault trees, and Markov-based state models may be constructed for critical components. These models enable the evaluation of failure propagation across architectural boundaries and variant-specific configurations. If structural equivalence and operational assumptions remain unchanged, previously established dependability evidence can be reused. Otherwise, partial or full re-evaluation is triggered according to predefined reuse rules.

By integrating dependability analysis with architectural modelling and traceability mechanisms, the methodology ensures that reuse decisions remain evidence-based and auditable. This approach reduces redundant verification effort while preserving compliance with safety and mission-critical quality standards across product variants.

### 3.4. Risk Analysis

Within the **SOSIS project**, risk analysis is applied as a validation technique to assess whether reusable components can be safely integrated into different software product line variants without introducing unacceptable system risks. When a component is reused in a new architectural context or configuration, changes in dependencies, operational environments, or feature combinations may affect its reliability and behaviour. Risk analysis, therefore, evaluates the potential impact of reuse on key dependability attributes, particularly **Reliability, Availability, and Maintainability (RAM)**.

The validation process uses architectural traceability, historical defect data, and dependency analysis to identify components that may introduce higher integration risks. Metrics derived from version control repositories, issue tracking systems, and testing results help determine whether the operational context of a reused component remains consistent with previously validated conditions. If significant structural differences or risk indicators are detected, additional verification activities—such as extended testing, fault injection, or dependability analysis—are triggered. Through this approach, risk analysis supports **evidence-based reuse decisions** and ensures that reusable components maintain their required quality and dependability properties across software variants.

# 4. Use Case Mapping

## 4.1. UC1 – Enterprise Quality / Variant Analysis

UC1 aims to enable systematic and reusable variant analysis for enterprise-scale Software Product Lines (SPLs). As the number of product variants increases across countries, regulatory environments, and deployment contexts, verification and certification costs grow significantly. This use case addresses this challenge by defining a structured and automatable approach to architectural extraction, traceability construction, and quality-based variant validation.

The core validation artefact in UC1 is based on the C4 model, which structures the system into Context, Container, Component, and Code levels. Architectural models are automatically generated from source code using static analysis and dependency graph extraction, and are enriched with traceability links to issues, features, and quality metrics. Issue tracking systems are mined to extract feature identifiers, risk classifications, and variant labels, enabling the construction of a unified artefact traceability graph linking features, issues, commits, and architectural components.

Automated variant analysis is then performed along two dimensions: (i) structural delta analysis between variants at container and component levels, and (ii) feature-level traceability stability analysis. If structural equivalence and metric stability conditions are satisfied, validation artefacts (e.g., test evidence, dependability analysis results, risk classifications) can be reused across variants, reducing redundant verification effort.

Through formalised reuse rules and automated architectural modelling, UC1 transforms architecture from static documentation into a reusable validation artefact. This directly supports WP4's objective of making verification activities repeatable, traceable, and scalable in enterprise-level SPL environments.

## 4.2. UC2 – Test Case Optimisation

Creating reusable test scenarios not only increases testing efficiency in projects but also saves time and effort. Reusing test scenarios allows us to use existing tests for new features, applications, or projects and reduces unnecessary repetition.

Reusable test scenarios refer to test scenarios that can be used across multiple projects or features with minimal or no changes. These test scenarios are designed to be general, adaptable, and applicable to multiple contexts without compromising accuracy or effectiveness. With reusable test scenarios, we can optimise testing efforts, ensure consistency, and accelerate the delivery of high-quality software.

The principles to follow when creating reusable test scenarios can be summarised as follows[1].

**Modularity:** Test scenarios are divided into small, independent modules that control specific functions. To ensure they can be easily reused in different scenarios, test scenarios are created that include clear inputs, expected results, and actions to be performed. For example, a test suite can be prepared that performs user addition, deletion, and editing scenarios for users in different roles. Another example is

[1] Best Practices in Test Case Design, Software Testing Magazine, October 29, 2024, https://www.softwaretestingmagazine.com/knowledge/best-practices-in-test-case-design

an end-to-end journey scenario, where these small modules can be combined for multiple scenarios, such as payment method selection, pickup location selection, delivery point location selection, and fee estimation, enabling their use in different scenarios. This approach reduces code duplication and test writing time while also simplifying maintenance.

**Parameterised Tests:** This involves using parameterised inputs to ensure that a single test scenario covers multiple cases or can be adapted to different environments. For example, parameterising login credentials or search queries allows testing with various inputs without requiring separate test scenarios. In this example, the same test step is reused with different data sets for valid login and invalid login scenarios. In other words, a single script can run multiple tests simultaneously with different inputs. In the payment process test scenario, using the product name, quantity, and price parameters instead of specific product name details allows payment process testing to be performed with different products without repeating the test scenarios.

**Abstraction[2],[3]:** one of the key concepts in OOP (Object-Oriented Programming), it allows frequently used test steps or activities to be combined into reusable components or libraries. This enables the creation of general test cases that can be adapted or extended to specific test scenarios. For example, in a mobile banking application that supports multiple transaction types (money transfer, bill payment, deposit), instead of creating separate test scenarios for each transaction type, common actions such as logging in, navigating to the transaction page, and verifying transaction details are converted into reusable functions. These functions can then be called within separate test scenarios for different transaction types.

Thanks to the abstraction principle, tests become behavior-focused and technical changes do not affect them. The same workflow can be reused in many scenarios, making tests more readable and sustainable. As a result, abstraction is the backbone of reusable test design. Strategies for designing reusable test scenarios can be determined based on this main backbone.

The first step is to identify common functions. By examining the application, repetitive functions or processes are identified. It is determined which test scenarios can be reused across different modules, features, or projects. In the second step, test scenario templates are created for inputs, actions, expected results, and validation criteria. Parameters or variables that can be customised according to specific test scenarios are defined. This is particularly important for performing parameterised tests. In the third step, the test logic is separated into reusable functions, methods, or libraries. As a final step, reusable test scenarios are categorised, organised, and stored. Tagging or classifying test scenarios according to their functions, modules, or compatibility will speed up access and reuse.

A review of the literature reveals that the PC-TRT test tool developed by Guo et al.[4] is noteworthy. PC-TRT is a test scenario reuse tool designed primarily for software and programs written in the C language. PC-TRT reuses test scenarios from previous program versions and generates test data for uncovered paths, thereby obtaining a set of test scenarios with high path coverage. Its main functions include analysing test scenario path coverage information, selecting reusable scenarios from old test scenario sets based on path similarity, and generating test data for uncovered paths.

---

[2] Test Automation Frameworks: Types, Benefits & Easy Examples (2025 Guide), https://sourcebae.com/blog/test-automation-frameworks-types-benefits-easy-examples-2025-guide

[3] Best practices | TestCases | Repetitions and reuse, https://docs.tricentis.com/tosca-2024.1/en-us/content/best_practices/testcases_repetitions_and_reuse.htm

[4] Zhonghao Guo, Sinong Chen, Xinyue Xu, Xiangxian Chen, PC-TRT: A Test Case Reuse and generation Tool to achieve high path coverage for Unit Test, SoftwareX, Volume 28, 2024

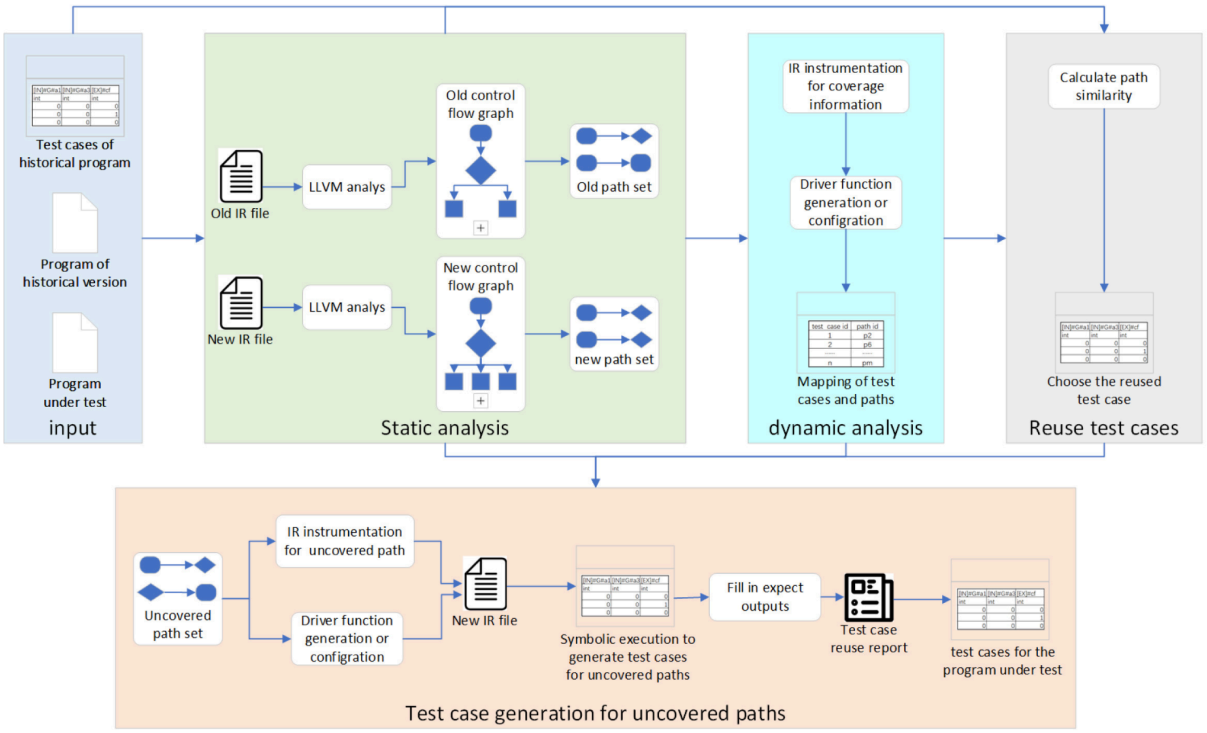


Figure 4.2.1: PC-TRT test tool architecture

Although test scenarios can be generated automatically, creating expected results (test predictions) is still largely manual and costly. This leads to inefficiency, especially with minor code changes. Although validated and reliable test scenarios from previous versions are valuable, it is unclear which ones are valid in the new version due to code changes, making test reuse difficult. Current research and methods, especially for unit tests, do not sufficiently consider code-level changes and fall short in providing high accuracy and coverage. A study by Xu et al.[5] proposes the Path-Similarity-Based Test Scenario Reuse (PSTR) method to overcome these issues. PSTR consists of three modules: static analysis, dynamic analysis, and reuse. It aims to reuse the most suitable test scenarios by comparing the execution paths in the old and new code versions.

[5] X. Xu, S. Chen, Z. Guo and X. Chen, "PSTR: A Test Case Reuse Method Based on Path Similarity," in IEEE Access, vol. 13, pp. 3175-3187, 2025

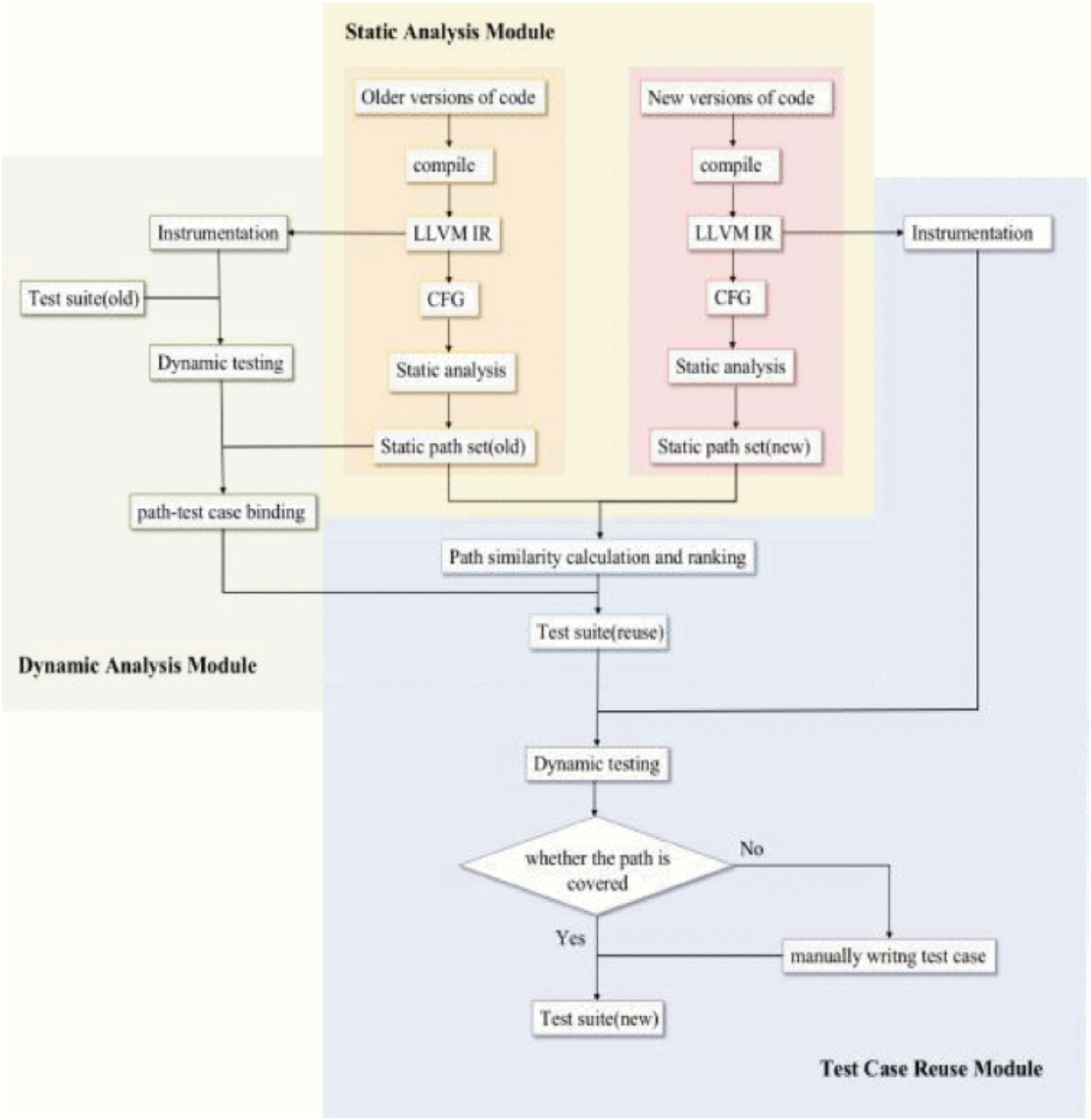


Figure 4.2.2: PSTR test tool workflow

## 4.3. UC3 – Software Production Line Quality Analysis

Product Lines (SPLs) provide a strategic approach for developing families of related products that share core assets while supporting controlled variability. While this approach increases development efficiency and accelerates market entry, it also introduces significant quality management challenges.

As variants evolve independently—through feature additions, configuration changes, and market-specific adaptations—divergences emerge across codebases, dependencies, and deployment environments. Without systematic cross-variant governance, these divergences create:

- Inconsistent quality levels

- Increased regression risk
- Redundant validation effort
- Knowledge concentration risks
- Security patch propagation gaps

UC3 addresses these challenges by introducing a cross-variant quality intelligence framework that evaluates the entire SPL as a unified ecosystem rather than as isolated products.

The objective of UC3 is to:

- Systematically assess quality across variants
- Reuse validation artefacts across the production line
- Detect knowledge distribution risks
- Prioritise testing based on structural and historical data
- Provide decision-level insights for sustainable SPL evolution

**Methodology**

The core methodology in UC3 treats Quality Data as a Reusable Artefact. Instead of treating each variant as a "black box" that requires full re-validation, the framework maps quality attributes directly to the Feature Model and the shared core assets. This enables systematic reuse of validation knowledge and prevents redundant quality assessment.

**Variant-Specific Quality & Metric Reuse**

The framework automatically calculates and compares quality metrics, such as technical debt, maintainability scores, and cyclomatic complexity across variants.

- Metric Reuse: Standardised "Quality Profiles" are defined at the core asset level and inherited by all variants. Using tools like SonarQube and Trivy, we ensure that a "Security Pass" or "Maintainability Grade" is measured consistently across the entire line, preventing the need to redefine validation criteria for every new market entry.
- Refactoring Identification: By identifying "Outlier Variants" that deviate from the core quality baseline, the system triggers targeted refactoring rather than redundant blanket testing.

**Developer Knowledge Insights & Risk Validation**

Using DETANGLE and version control analysis (Git/Jira), UC3 maps developer contributions to specific modules to identify "Knowledge Islands."

- Knowledge Distribution: In an SPL, a shared component is only as reusable as the team's shared understanding of it. The framework detects modules with limited contributor diversity.
- Validation Impact: If a critical shared module is identified as a "Knowledge Island," the framework flags it as a high-risk area for reuse, recommending cross-training to ensure that the validation of that component remains sustainable as the product line evolves.

**Test Coverage & Trend Validation Reuse**

To optimise the validation effort, UC3 utilises Constraint-Based Test Prioritisation and Trend Analysis.

- Test Coverage Prioritisation: The framework identifies gaps in unit and integration tests, focusing resources on high-impact areas where variants differ most from the core.
- Trend Validation Reuse: Historical data from long-lived variants is used to predict the "Quality Decay" of newer variants. If a new variant's complexity begins to follow a known negative trend observed in a previous version, the system triggers proactive alerts based on historical "lessons learned."

**Expected Outcomes and Impact**

Identifying these historical patterns allows the system to flag potential risks in a new variant long before they result in critical failures. This predictive capability works in tandem with constraint-based test prioritisation. By focusing testing resources specifically on the areas where a new variant diverges from the core assets, we eliminate the redundant verification of shared components that have already been proven stable.

Integrating these practices into the standard DevOps toolchain creates a unified traceability graph that links quality issues directly to specific variants and developers. Moving from reactive testing to predictive, line-wide quality analysis is essential for keeping the software product line competitive and robust. Ultimately, reusing validation insights, standardised metrics, and historical trends allows organisations to scale their software production without suffering the linear increase in quality assurance overhead that typically impacts large engineering projects

## 4.4. UC4 – AI-driven Requirements

The UC4 initiative transforms traditional, labour-intensive manual requirement engineering into a comprehensive suite of reusable automated processes. As illustrated in the UC4 Flow Chart, the requirements lifecycle initiates when an AI-powered voice agent establishes a direct interaction with stakeholders to gather primary project needs through conversational interviews. This systematic workflow establishes a structured transition from raw stakeholder input to organised, project-ready documentation, allowing the project to capture data in a way that minimises the risk of early-phase errors. By integrating these automated steps, the system provides a framework where the initial data gathering phase serves as the foundation for all subsequent development activities, maintaining a high degree of technical coherence throughout the project duration.

The voice agent conducts these stakeholder interviews by dynamically adapting its questions based on real-time responses to clarify any ambiguous needs or missing details that could hinder progress later. This interactive process facilitates the immediate structuring of narrative data into predefined categories such as functional requirements, technical constraints, or specific operational environments for the software. The SOSIS platform utilises these inputs to produce standardised documents through preformatted templates, which directly supports the goal of standardisation of requirements collection, prominently noted in the UC4 Table. This methodology reduces the time analysts spend on initial drafts, allowing them to focus on high-level strategic alignment instead of basic documentation assembly while supporting the scalability of the engineering process.

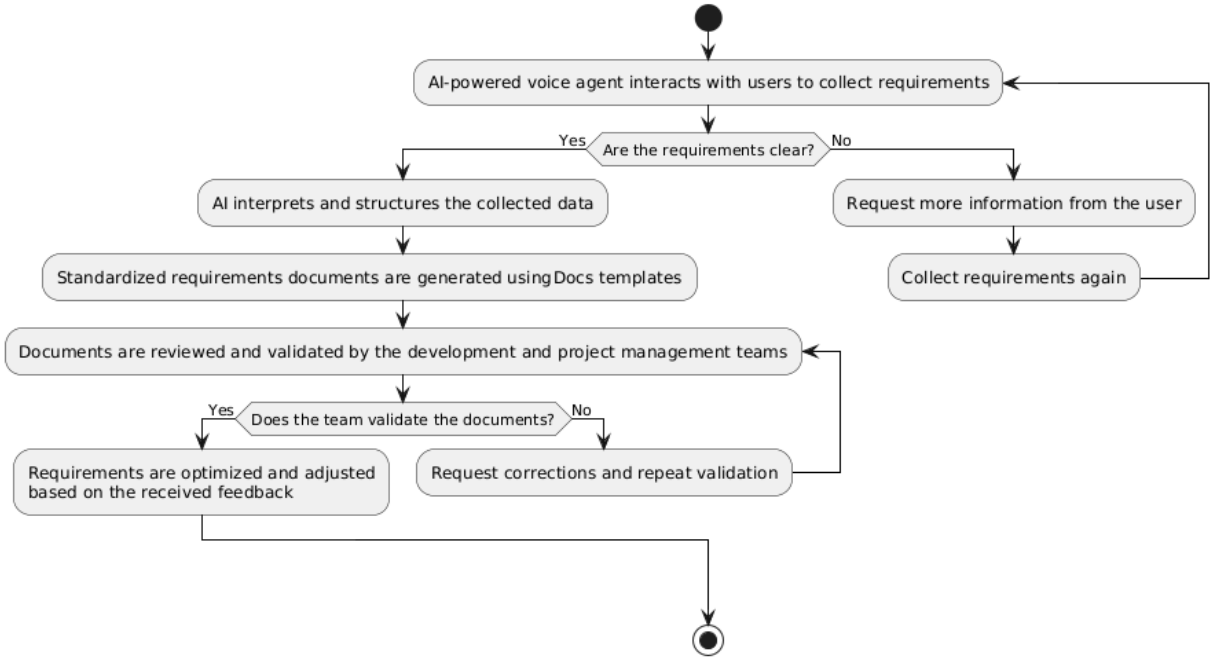


Figure 4.2.3: UC4 Flow Chart

Table 4.2.1: UC4 Components

| ID | UC-URSD |
|---|---|
| **Objective** | Automate the collection and documentation of user requirements to optimize efficiency and accuracy in software projects. |
| **Actor(s)** | Developers, Requirements Analysts, Project Managers. |
| **Point of Contact** | AI-powered voice agent. |
| **Data Sources** | User inputs, automated interviews, Document templates, AI language models. |
| **Trigger** | Start of a new software project requiring requirements collection. |
| **Steps** | 1. The AI-powered voice agent interacts with users to collect requirements.<br>2. The AI interprets and structures the collected data.<br>3. Standardized requirements documents are generated using Google Docs templates.<br>4. The documents are reviewed and validated by development and project management teams.<br>5. Requirements are optimized and adjusted based on the feedback received. |
| **Improvements** | Standardization of requirements collection, reduction of manual effort, and increased accuracy in documentation. |
| **Results** | More efficient processes, fewer errors in documentation, better organization, and improved scalability in software projects. |

A critical technical milestone within this framework involves the automated validation layer defined in Scenario 02, which intentionally treats validation logic as a repeatable and highly valuable asset for the development team. According to the UC4 Flow Chart, once the standardised documents are generated, they undergo a formal review where the automated engine identifies structural gaps, formatting errors, or misalignments with the core project goals. This automated validation process creates a reusable feedback loop that allows teams to apply the exact same evaluation criteria across multiple iterations of a single project document or even across different project variants. By utilising this repeatable logic, the organisation maintains a consistent quality baseline without having to

re-develop validation scripts or manual checklists for every new requirement update, thereby increasing the reliability of the output.

In situations where stakeholder inputs remain unclear or the development team does not accept the initially generated drafts, the system triggers a predefined iterative cycle to resolve these issues. The UC4 Flow Chart provides a clear visualisation of this loop, where the AI proactively requests more information or specific corrections from the users before repeating the entire validation sequence to confirm the updates. This capability supports the progressive refinement of documentation accuracy by allowing the system to learn from previous corrections and apply those insights to the current document version. This iterative approach helps the project reach a state of documentation readiness faster by focusing only on the elements that need correction rather than requiring a complete manual overhaul of the requirements set, making the verification of needs much more efficient.

The platform also prioritises consistency check reuse through Scenario 03, which establishes a sophisticated harmonisation layer designed to manage several parallel projects within an industrial environment. This layer imports diverse requirements from various heterogeneous sources and unifies their terminology, naming conventions, and formatting into a single standardised model that is easy to navigate and analyse. By applying consistent semantic rules across different software variants, the system provides a robust foundation for traceability and better organisation, as highlighted in the UC4 Table. This harmonisation allows the project to reuse the same organisational logic across a diverse portfolio, which facilitates better governance and more efficient resource allocation for managers overseeing multiple development teams.

Automated engines embedded within this framework perform continuous checks to detect repeated requirements, low-confidence inputs, or contradictory statements that could lead to technical debt if left unaddressed. These logical consistency patterns are specifically designed for application across various project variants to check that new configurations or feature selections do not violate any established system constraints or architectural rules. The direct result of this automation is a massive reduction of the manual labour typically associated with identifying structural conflicts or logic errors in high-volume documentation sets. By reusing these consistency logic patterns, the SOSIS platform maintains the integrity of the requirements model even as the complexity of the software product line grows over time, helping to prevent the propagation of defects.

The full integration of these reusable mechanisms leads to a substantial reduction of manual effort and increased overall scalability for complex software projects within the mission-critical domain. According to the UC4 Table, these automated validation and consistency workflows reduce documentation errors and significantly shorten response times for all stakeholders, from developers to project managers. These processes ultimately deliver the reusable certified artifacts that are necessary for subsequent safety-critical development stages and formal certification activities, which can be reused in future audits. This approach helps the organisation achieve its target KPIs by making verification and validation activities a systematic, repeatable, and highly efficient part of the software engineering lifecycle.

## 5. Expected Outputs and Conclusion

This deliverable establishes the methodological foundation for reuse and validation within the SOSIS framework. It defines the concept of reusable components, classifies them across software, model, and test artefacts, and introduces a structured reuse methodology grounded in Software Product Line Engineering principles. In addition, the report presents a set of validation techniques—including formal methods, fault injection, dependability analysis, and risk analysis—to ensure that reusable components maintain their correctness, reliability, and quality when integrated into different product variants. The methodologies are mapped to the SOSIS industrial use cases, demonstrating how validation artefacts and verification evidence can be systematically reused across variants. The expected outcome is a reusable validation framework that reduces redundant verification effort, strengthens traceability between development artefacts, and supports scalable quality assurance in complex industrial software ecosystems. The results of this deliverable provide the conceptual basis for the implementation and evaluation activities in the subsequent work packages of the SOSIS project.