SOSIS

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

D3.1 – Methodology for Product Line Testing and Test Optimization of Safety-Critical Systems

Submission date of deliverable: August 30, 2026

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

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	V1.0
Work package	WP3
Work package leader	Tim Schürmann (IOTIQ, Germany)
Dissemination level	Public
Description ***(max 5 lines)***	This document presents a methodology for product line testing and test optimization for critical systems with different variants. The approach examines test metrics, test methods, and techniques for optimizing automated test generation at the product line level. The goal is to measure test coverage, reuse tests, and reduce test effort by leveraging traceability between requirements, features, and variants. It also includes research on various layers of AI and open-source projects.

Executive Summary

Work Package 3 aims to develop a new methodology and toolset for product line testing and test optimization for critical systems with different variants. Since testing all configurations individually in systems with numerous variants is costly and time-consuming, this project aims to define meaningful and measurable test metrics at the product line level, analyse test coverage and quality indicators, and establish a framework to provide decision support for test engineers. Accordingly, an approach is developed that enables the evaluation of test performance across the entire product line using test data obtained from existing variants.

This document examines existing approaches to achieve these objectives and details the proposed methodology. The methodology is based on variability-aware test design and automated test generation, establishing traceability between tests and requirements and features, and mechanisms that support the prioritization of tests according to different criteria and the optimization of execution processes. In this way, the goal is to reduce testing effort, increase defect detection efficiency, and maintain the high level of quality required for critical systems.

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 Prioritization

1. Introduction

Software Product Line Engineering (SPLE) is built around developing a family of distinct products from a shared core infrastructure. The trouble is that as soon as a product line's planned variability starts to grow, the number of possible variants grows with it – and not in a gentle, linear way. This combinatorial explosion is what makes testing costs spiral out of control, particularly in safety-critical sectors like aerospace, healthcare, telecom and automotive, where every variant genuinely needs to be verified.

Variant Explosion, Testing Costs and Traceability

The ability to create several different products from a common set of base assets is one of the key selling points of Software Product Line Engineering (SPLE). However, this advantage also has a downside: as planned variability begins to increase, the number of possible product variants grows exponentially[1]. "Add a few more features, and the combinations are multiplied rather than added together" – this is the model researchers refer to as the variant explosion. Beyond a certain point, testing every variant becomes computationally or financially unfeasible, and testing costs rise accordingly. Furthermore, when something goes wrong, it is extremely difficult to trace the fault back to the specific feature interaction or underlying asset responsible; this makes traceability a clear challenge throughout the entire product line lifecycle.

Product Line Testing

Product line testing overcomes this problem by shifting the focus from individual products to the shared architecture. This test relies heavily on Feature Models, which formally capture the elements that are common and distinct across the product line; a product is then instantiated by selecting a specific set of features during variant derivation. Generally speaking, existing test methodologies fall into two groups:

[1] Pohl, K., Böckle, G., & van der Linden, F. J. (2005). Software Product Line Engineering: Foundations, Principles, and Techniques. Springer-Verlag. https://doi.org/10.1007/3-540-28901-1

product-based testing, which validates each derived variant separately as an independent system in its own right; and family-based testing, which examines the shared components of the product line (feature models, domain code, etc.) in one go without having to create each product individually[2]. Family-based methods appear superior on paper – in principle, they capture greater variability – but are difficult to implement at a real-world industrial scale.

Safety-Critical Testing

The variant explosion has the greatest impact in safety-critical sectors such as aviation, the automotive industry and healthcare; in these sectors, standards such as ISO 26262 for road vehicles and DO-178C for aircraft software leave very little room for shortcuts. Compliance with these standards requires rigorous verification and validation, including challenging coverage requirements such as Modified Conditions/Decision Covers (MC/DC). Demonstrating this level of coverage across thousands of variants is a major engineering undertaking in its own right, and if feature interactions are not tested comprehensively enough, certification may stall, which could delay the entire product line and put human safety at risk.

Limitations of Current Approaches

Despite the progress made so far, traditional SPLE tests still have real limitations. Many organisations still rely on variant-independent tests – tests created for a single product that cannot be seamlessly transferred to the rest of the product family – which leads to a great deal of unnecessary testing: the same common features are tested repeatedly across different variants, consuming time and processing power without delivering any real benefit. Measuring coverage at the product line level also remains a clear challenge. Measuring code coverage for a single variant is straightforward enough, but determining whether the full range of valid feature interactions is covered – product line coverage – is something that most industrial tools do not yet support. All of this points to the same conclusion: test orchestration needs to be smarter, more automated and more aware of variant differences.

The ITEA SOSIS project was set up to tackle exactly these problems.

2. Background and State of the Art

2.1. Product Line Testing

Software Product Line Engineering (SPLE) supports the systematic development of a family of related software products by managing their common and variable characteristics. Instead of developing each product independently, reusable requirements, architectural components, source-code assets, and test artefacts are maintained at the product-line level and configured to derive individual products. This approach can reduce development effort and improve consistency, but it also introduces significant testing challenges because the behavior of a product may depend on the selected features and their interactions.

Variability is commonly represented through feature models. A feature model describes the mandatory, optional, and alternative features of a product line, together with constraints that determine which feature combinations constitute valid configurations. Variant derivation resolves these variability decisions and

[2] Shuai Wang, David Buchmann, Shaukat Ali, Arnaud Gotlieb, Dipesh Pradhan, and Marius Liaaen. 2014. Multi-objective test prioritization in software product line testing: an industrial case study. In Proceedings of the 18th International Software Product Line Conference - Volume 1 (SPLC '14). Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/2648511.2648515

produces a specific product configuration. From a testing perspective, the feature model defines the configuration space that must be considered and provides a basis for relating requirements, implementation components, and test cases to individual features and variants.

Testing all valid configurations is generally impractical because the number of possible variants can grow exponentially as additional features and constraints are introduced. Even moderately sized feature models may represent thousands or millions of valid products. Product-line testing therefore seeks to achieve sufficient coverage without executing every available test against every possible variant.

Existing product-line testing approaches can broadly be classified as product-based, family-based, and feature-based. Product-based testing derives a variant and tests it as an independent product. Although this approach is compatible with conventional testing tools, it repeatedly verifies common functionality and does not fully benefit from product-line reuse. Family-based testing analyses the product line as a whole and attempts to reason about multiple configurations simultaneously. It can reduce duplicated work but may require specialized models and tools that are difficult to apply to large industrial systems. Feature-based testing associates reusable test assets with individual features or feature combinations and composes variant-specific test suites according to the selected configuration.

Configuration-sampling techniques are frequently used to control the size of the testing space. Pairwise and more general *t*-wise approaches select a subset of valid configurations so that interactions among a specified number of features are covered. Other strategies use risk, feature criticality, usage frequency, historical failures, or random and search-based techniques to focus testing on the most relevant parts of the configuration space. These approaches reduce testing effort, but their effectiveness depends on the quality of the feature model and the assumptions made about which interactions are most likely to produce faults.

Model-based testing is another important approach in which test cases are generated from behavioral or architectural models containing variability information. A product-specific test model can be derived by resolving the variability for a selected configuration. This enables the systematic generation of tests while maintaining consistency between the product-line specification and the resulting test artefacts. However, creating and maintaining sufficiently detailed behavioral models may require considerable engineering effort, particularly when the implementation and product configurations evolve frequently.

Regression testing within a product line must consider both changes to shared core assets and modifications that affect only particular variants. Change-impact analysis can use traceability among requirements, features, variants, code elements, and test cases to identify the tests potentially affected by a modification. Tests associated with unchanged common functionality may be reused, whereas tests covering modified features and their interactions must be selected or adapted. Historical execution results and defect information may further support regression-test selection and prioritization.

Despite substantial research, industrial adoption continues to face several limitations[3]. Traceability information is often incomplete or distributed across requirements-management systems, issue trackers, source-code repositories, and test-management tools. Product-line-level coverage measures are less mature than conventional product-level code coverage, and many testing tools remain focused on individual products. Effective product-line testing therefore requires an integrated, variability-aware approach combining feature models, traceability, coverage metrics, change-impact analysis, test reuse, and automated optimization.

[3] Engström, E., & Runeson, P. (2011). Software product line testing—A systematic mapping study. Information and Software Technology, 53(1), 2–13. https://doi.org/10.1016/j.infsof.2010.05.011

2.2. Safety & Mission Critical Testing

Safety-critical and mission-critical software operates in domains where a failure can lead to loss of life, injury, mission loss, or severe financial and environmental damage. In these domains, testing serves two purposes at once. It detects defects, as in any other software project, and it produces the objective evidence that a certification body, an assessment body, or an internal safety authority examines before a system is allowed to enter service. The second purpose constrains how testing is planned, executed, and documented, because evidence that cannot be traced, repeated, or justified has limited value in a certification dossier. Several domain-specific functional safety standards define what this evidence consists of and how much of it is expected.

At the cross-sector level, IEC 61508 provides the reference framework for the functional safety of electrical, electronic, and programmable electronic safety-related systems. Its third part addresses software and defines four Safety Integrity Levels, SIL 1 to SIL 4, which express the degree of risk reduction attributed to a safety function[4]. The standard associates each level with a set of techniques and measures for specification, design, and verification, and presents them with different degrees of recommendation as the level rises. Most sector standards in use today either derive from this framework or align with it, which is why the same underlying concepts reappear, under different names, across the domains addressed in the SOSIS project.

In the automotive domain, ISO 26262 governs the functional safety of road-vehicle electrical and electronic systems and classifies risk through Automotive Safety Integrity Levels, ASIL A to ASIL D[5]. For airborne software, RTCA DO-178C, published jointly with EUROCAE ED-12C, defines five Design Assurance Levels, DAL A to DAL E, and organises its requirements as verification objectives whose applicability and required independence increase with criticality[6]. In the railway domain, EN 50716:2023 has superseded EN 50128 and EN 50657, bringing the software requirements for signalling and on-board applications into a single standard across five software safety integrity levels[7]. For medical devices, IEC 62304 defines the software life cycle processes and a risk-based classification of software items into safety classes A, B, and C[8].

A property common to these standards is that the amount of verification is not fixed but proportional to the assessed criticality. The integrity level assigned to a function or a software item determines which activities are expected, how independent the verification has to be from the development work, and how much documentation accompanies the result. A consequence that matters for product lines is that the same piece of software may be assigned different integrity levels in different products, depending on the function it implements and the hazards of the surrounding system. The verification effort attached to a reused component is therefore not a fixed quantity either, and it cannot be established once and treated as settled for every context in which the component appears.

Structural coverage measurement is one of the mechanisms through which the standards make the completeness of testing observable. Statement coverage is expected at the lower integrity levels,

[4] IEC 61508:2010, Functional safety of electrical/electronic/programmable electronic safety-related systems (Edition 2.0). International Electrotechnical Commission, Geneva.
[5] ISO 26262:2018, Road vehicles: Functional safety (2nd ed.). International Organization for Standardization, Geneva.
[6] RTCA DO-178C / EUROCAE ED-12C (2011), Software Considerations in Airborne Systems and Equipment Certification. RTCA Inc., Washington, D.C.
[7] EN 50716:2023, Railway applications: Requirements for software development. CENELEC, Brussels.
[8] IEC 62304:2006+A1:2015, Medical device software: Software life cycle processes. International Electrotechnical Commission, Geneva.

decision or branch coverage at the intermediate levels, and Modified Condition/Decision Coverage at the highest levels, where each condition within a decision has to be shown to affect the outcome of that decision independently. DO-178C, for example, associates Modified Condition/Decision Coverage with Level A software. Coverage is not treated as an objective in itself but as a check on the adequacy of tests derived from requirements, which is why unreached code and unexercised conditions call for an explicit analysis rather than only a lower reported percentage.

Requirements-based testing, together with bidirectional traceability, is the second mechanism. The standards expect every requirement to be exercised by at least one test and every test to be attributable to a requirement, so that the verification argument can be followed in both directions. Traceability extends further, linking requirements to design, design to code, and all of them to the verification results. It is what allows an assessor to check that nothing has been verified without a reason and that nothing required has been left unverified. In practice this traceability is also the working instrument for change management, because it identifies what has to be re-examined when an artefact is modified.

These obligations were formulated primarily from the perspective of a single product undergoing assessment. A software product line changes the situation, because the same core assets are reused across a family of products whose configurations differ. The evidence that matters for certification is the evidence for the product actually placed in service, so coverage data, traceability, and verification results have to hold for each configuration that is delivered and not only for a representative one. Reuse of an implementation does not by itself carry over the argument built on top of it, and this is the point at which product line engineering and the assurance regimes described above come into tension.

The difficulty is one of scale. The number of configurations that a feature model admits can grow combinatorially with the number of features and their dependencies, so reproducing a complete body of verification evidence independently for every derivable variant becomes expensive and, beyond a certain size, impractical. Secondary studies on product line testing have identified this scalability problem, together with the absence of coverage notions defined at the level of the product line rather than at the level of an individual product, as recurring open issues in the field[9]. Independent mapping and review studies of the same area reached comparable conclusions about the state of the field and about the need for strategies that exploit commonality across variants[10][11].

Empirical work on industrial practice gives a consistent picture. A multiple-case study of twelve medium to large industrial cases, in domains including automotive, aerospace, and railway systems, reports that organisations have adopted parts of the available variability management techniques while continuing to face difficulties in adopting and evolving product lines, and that approaches taken directly from the literature are often considered not applicable or not efficient in their setting[12]. A systematic literature review of empirical software product line engineering examines the body of published evidence in the

[9] Engström, E., & Runeson, P. (2011). Software product line testing: a systematic mapping study. Information and Software Technology, 53(1), 2-13. https://doi.org/10.1016/j.infsof.2010.05.011
[10] da Mota Silveira Neto, P. A., do Carmo Machado, I., McGregor, J. D., de Almeida, E. S., & de Lemos Meira, S. R. (2011). A systematic mapping study of software product lines testing. Information and Software Technology, 53(5), 407-423. https://doi.org/10.1016/j.infsof.2010.12.003
[11] do Carmo Machado, I., McGregor, J. D., Cavalcanti, Y. C., & de Almeida, E. S. (2014). On strategies for testing software product lines: a systematic literature review. Information and Software Technology, 56(10), 1183-1199. https://doi.org/10.1016/j.infsof.2014.04.002
[12] Berger, T., Steghöfer, J.-P., Ziadi, T., Robin, J., & Martinez, J. (2020). The state of adoption and the challenges of systematic variability management in industry. Empirical Software Engineering, 25(3), 1755-1797. https://doi.org/10.1007/s10664-019-09787-6

area and how it has been obtained[13]. For regression testing specifically, a study of the state of practice in large-scale embedded software development examines how regression testing is organised in industrial settings[14].

More recent secondary studies indicate where the gaps lie. A systematic literature review of software product line testing covering studies published up to 2022 reports that the literature offers many techniques for some concerns, such as controlling the cost and effort of testing, while other elements, including regression testing, non-functional testing, and the preservation of traceability between test assets and other artefacts, remain less well covered, and that most proposals have been evaluated by a single empirical method, most often an academic one[15]. Research on the intersection of safety, security, and configurable systems has mapped what has been studied in that intersection and noted the absence of an overall picture of it[16]. Approaches directed at certification itself have also been proposed, for example an integrated method combining the decomposition of integrity levels, software process modelling, and variability management in support of the process-based certification of variant-intensive systems[17]. Taken together, these results frame the problem addressed by the present deliverable: reconciling the per-product rigour that safety certification requires with the reuse-driven economics that motivate product line engineering. The methodology described in the sections that follow is directed at that reconciliation.

2.3. Limitations of Existing Approaches

Despite advances in software product line testing, existing approaches still face significant limitations when applied to large-scale and safety-critical systems. Many industrial testing processes and tools remain product-oriented and treat each derived variant as an independent software product. Consequently, common features and components are repeatedly tested across multiple variants, even when their implementation and behaviour remain unchanged. This increases execution time and resource consumption while limiting the reuse of existing test artefacts and results.

A major challenge is the limited variability awareness of conventional test suites. Tests are generally associated with a particular product, requirement, or code component, but their relationships with features, feature interactions, configurations, and variants are not always explicitly represented. When a requirement, feature, or shared component changes, it can therefore be difficult to determine which

[13] Chacón-Luna, A. E., Gutiérrez, A. M., Galindo, J. A., & Benavides, D. (2020). Empirical software product line engineering: A systematic literature review. Information and Software Technology, 128, Article 106389. https://doi.org/10.1016/j.infsof.2020.106389

[14] Minhas, N. M., Petersen, K., Börstler, J., & Wnuk, K. (2020). Regression testing for large-scale embedded software development: Exploring the state of practice. Information and Software Technology, 120, Article 106254. https://doi.org/10.1016/j.infsof.2019.106254

[15] Agh, H., Azamnouri, A., & Wagner, S. (2024). Software product line testing: a systematic literature review. Empirical Software Engineering, 29, Article 146. https://doi.org/10.1007/s10664-024-10516-x

[16] Kenner, A., May, R., Krüger, J., Saake, G., & Leich, T. (2021). Safety, Security, and Configurable Software Systems: A Systematic Mapping Study. In 25th ACM International Systems and Software Product Line Conference (SPLC), Volume A (pp. 148-159). ACM. https://doi.org/10.1145/3461001.3471147

[17] Bressan, L., de Oliveira, A. L., Campos, F., Papadopoulos, Y., & Parker, D. (2020). An Integrated Approach to Support the Process-Based Certification of Variant-Intensive Systems. In Model-Based Safety and Assessment (IMBSA 2020) (pp. 179-193). Springer. https://doi.org/10.1007/978-3-030-58920-2_12

variants and test cases are affected. This frequently results in either excessive regression testing or insufficient testing of affected configurations.

Coverage measurement at the product-line level is another important limitation. Traditional metrics such as statement, branch, and requirement coverage provide useful information for an individual product but do not show whether the variability space has been adequately tested. Additional measures, including feature coverage, variant coverage, configuration coverage, and feature-interaction coverage, are required to assess the product line as a whole. However, these metrics are not consistently supported by existing industrial tools, and their calculation becomes increasingly difficult as the configuration space expands.

Repeatability and reuse also remain challenging. Test cases may depend on variant-specific data, environmental conditions, external services, hardware configurations, or manually prepared preconditions. Such dependencies can prevent a test developed for one variant from being executed directly on another. Differences in test environments and toolchains may also make it difficult to reproduce previous results. Without reusable test components and controlled execution environments, organisations must repeatedly adapt similar tests for different products.

Traceability information is often fragmented across requirements-management systems, issue trackers, source-code repositories, configuration models, and test-management platforms.[18] Incomplete or outdated links between these artefacts reduce the reliability of change-impact analysis and make it difficult to demonstrate that every relevant requirement and feature has been verified for a specific variant. This problem is particularly important for safety-critical systems, where verification evidence must be complete, reproducible, and attributable to the configuration being assessed.

Finally, many existing approaches optimize only a single testing objective, such as maximizing coverage or minimizing execution time. In practice, test decisions must balance several factors, including coverage, risk, feature criticality, historical failures, change relevance, execution cost, and available resources. Addressing these limitations requires an integrated and automated methodology that combines variability modelling, traceability, product-line-level metrics, reusable test assets, change-impact analysis, and multi-objective test optimization.

3. Research on Various Layers of AI and Open-source Test Projects

One of its main outputs in the SOSIS project is the open-source OPA4T (Open Platform for Automated AI-driven Testing) platform. Built by Orion Innovation, OPA4T takes raw test logs coming out of CI/CD pipelines (tools like Cucumber and Selenium) and runs them through AI agents, retrieval-augmented generation (RAG), and autonomous root-cause analysis, turning them into a single, intelligent test-management layer.

Between 2021 and 2026, researchers have put forward a number of methodologies that could feed directly into the mathematical and algorithmic core of testing frameworks like OPA4T. The sections that follow walk through how these optimization, selection, and prioritization methods actually work.

[18] Agh, H., Azamnouri, A., & Wagner, S. (2024). Software product line testing: A systematic literature review. *Empirical Software Engineering, 29*, Article 146. https://doi.org/10.1007/s10664-024-10516-x

3.1. Methodology Research on Various Layers of AI

To organise the latest academic findings, this report categorises testing and optimisation methodologies into four layers:

- **Layer 1 (Fundamental AI/ Optimisation):** Algorithms that solve combinatorial constraints, such as genetic programming, swarm intelligence and graph-based models.
- **Layer 2 (Machine Learning):** Supervised and unsupervised models that predict errors or reduce test suites by learning from historical test data.
- **Layer 3 (Deep Learning):** Neural networks designed to detect sequential patterns in execution logs or to learn dynamic test sequencing policies.
- **Layer 4 (Generative AI and Agentic AI):** Large language models that validate system designs or generate test scripts. Autonomous Agentic AI systems that coordinate these layers to execute test suites and repair simple errors in a closed loop.

3.1.1. Layer 1: Fundamental Artificial Intelligence / Optimisation Layer

This layer includes methods that model test optimisation as a classical search, scheduling or graph traversal problem and solve these using well-known algorithms.

Cost-Sensitive Evolutionary Genetic Algorithm Method

Traditional test prioritisation techniques generally assume, implicitly, that running all tests incurs the same cost and that all defects are equally serious. Bello and Alhassan[19] (2025) challenged this assumption with a cost-sensitive evolutionary regression test prioritisation method.

Mechanism and Steps

1. Feature Modelling and Path Extraction: System variations are converted into feature models using the Özellik IDE tool. All possible execution paths for integration testing are automatically derived from the programme's source code using the Java System Dependency Graph (JSDG).
2. Chromosome Representation: In the genetic algorithm, each chromosome represents a sequence of candidate test scenarios.
3. Cost-Sensitive Fitness Function: To evaluate candidate chromosomes, the algorithm uses the Average Percentage of Fault Detection per Cost (APFDc) metric—which takes into account both test execution time and fault severity—as the fitness value.

Results

When evaluated on the Video Store (VS1–VS4) product series, this method outperformed random sorting (RNDP) and classical approaches, increasing the APFDc score to 94.48% for VS1, 91.84% for VS2 and 90.93% for VS3, whilst also minimising costs.

Dragon Boat Optimization Algorithm

A recent development in meta-heuristic search is the Dragon Boat Optimization Algorithm (DBOA) proposed by Assiri[20] in 2025, a swarm-based optimization technique inspired by the synchronized teamwork of athletes in dragon boat races.

---

[19] Bello, A., & Alhassan, H. M. (2025). Cost-cognizant test case prioritization for software product line using genetic algorithm. FUDMA Journal of Sciences, 9(9), 129–138. https://fjs.fudutsinma.edu.ng/index.php/fjs/article/view/2901

[20] Assiri, M. (2025). Test case prioritization using Dragon Boat Optimization for software quality testing. Electronics, 14(8), 1524. https://www.mdpi.com/2079-9292/14/8/1524

Working Principle

- Agent Modeling: Each agent (boat) in the population carries position and velocity parameters representing a possible permutation of the test case sequence.
- Synchronization: Inspired by the drumbeat and oar synchronization in dragon boat races, the agents rapidly converge to the best local solution in the search space.
- The algorithm's optimization objective is to maximize APFD (which measures early fault detection rate) while minimizing test time. In multidimensional, constrained SPL search spaces, DBOA escapes local minimum traps much faster than classical Genetic Algorithms.

A Comparative Analysis of Meta-Heuristic Algorithms in Combinatorial Interaction Testing

Test efficiency is not solely dependent on the order in which tests are run; how the initial configurations are generated is equally important. Maidin et al[21]. (2025) compared four meta-heuristic algorithms for combinatorial interaction testing (CIT) — Genetic Algorithm (GA), Cuckoo Search, Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO)—by running them at interaction strengths (t-values) ranging from two to six.

GA outperformed the other three, and an interaction strength of t=5 proved to be the optimal point, covering a sufficient number of feature combinations without requiring extensive testing. This aligns with the results of Genetic Algorithms and Dragon Boat Optimization, and GA consistently emerges as a strong foundation. What sets this study apart from others is where it applies this strength. In other words, it is used not to rank the tests, but to create the configurations—the initial stage where everything on the process line is interconnected.

3.1.2. Layer 2: Machine Learning Layer

The machine learning layer works differently from the deterministic optimization methods described above: it uses supervised and unsupervised learning to identify predictive patterns from historical test execution data. In practice, this means learning from past runs to predict where errors are likely to occur in the next phase, determining how urgent a particular test is, or pruning unnecessary test cases by clustering them.

Data-Driven Test Case Prioritization (DD-TCP)[22]

This framework dynamically reorders test cases using gradient-enhanced decision trees and a probabilistic gain function, based on multidimensional attributes including code change frequency, dependency metrics, execution time, and historical error density. In evaluations on open-source projects and large-scale synthetic test suites, it achieved a 32.4% improvement in APFD score and a 27.1% reduction in execution overhead compared to baseline methods.

Aggregated Ensemble Learning + Deep Q-Learning

---

[21] Maidin, N. F., Hassan, S., Baharom, S., & Md. Sultan, A. B. (2025). A comparative study on testing optimization techniques with combinatorial interaction testing for optimizing software product line testing. Journal of Advanced Research in Applied Sciences and Engineering Technology, 49(1), 77–94. https://doi.org/10.37934/araset.49.1.7794

[22] Ramzan, H. A., Islam, K., Hussain, M. A., Monim, R. M., Asad, S. M., & Ramzan, S. (2026). Data-driven test case prioritization (DD-TCP): A machine learning framework for intelligent software quality assurance. Computers, Materials and Continua, 88(1). https://doi.org/10.32604/cmc.2026.077782

Zarrad et al[23]. (2025/2026) designed this as a two-stage architecture. The first stage is a stacked ensemble combining Gradient-Boosted, Support Vector Machine, Random Forest, and Naive Bayes classifiers, trained on features such as last run, loop, failure rate, and number of runs, which prioritizes test cases with 98.847% accuracy.

The second stage builds on this: a Deep Q-Learning framework paired with a Genetic Algorithm handles the fine-grained ranking of tests within the same priority group and achieves a ranking accuracy of 91.72%. What makes this combination work is the division of labor—a fast, lightweight classifier performs the coarse ranking, and a heavier but more precise reinforcement learning layer is activated only for fine-tuning.

Clustering-Based Test Reduction: CLUE

Vo and Nguyen[24] (2024) took a simpler approach with CLUE, a clustering-based test reduction method for software product lines: use K-means clustering to group test cases based on feature-interaction similarity, then retain only a few representative examples from each cluster, rather than running everything. It's a simple idea, but the numbers back it up—CLUE reduces testing effort by up to 88% while retaining most of the original defect detection capability.

3.1.3. Layer 3: Deep Learning Layer

The Deep Learning layer uses neural network architectures to learn complex sequential representations directly from raw data, such as test execution logs or tracking sequences.

Deep Q-Learning for Fine-Grained Test Sequencing

It is worth revisiting the second stage of Zarrad et al.'s architecture, this time from a deep learning perspective. The Deep Q-Network treats each test execution decision as an action in a Markov Decision Process, earns rewards proportional to the errors it captures at each step, and gradually learns a ranking policy from this. Achieving a ranking accuracy of 0.9172 in this manner is a good indication that, given sufficient interaction data, a deep network can outperform static heuristic rules.

Deep Learning for CI/CD Log-Based Fault Prediction

Hadadi et al. conducted a systematic comparison of deep learning models (RNN, LSTM, CNN, and Transformer variants, each paired with log-embedded vectors) for log-based fault prediction and found that a CNN-based encoder combined with Logkey2vec yielded the best results.

3.1.4. Layer 4: Generative AI and Agentic AI Layer

The Generative AI layer signals a real shift in what these systems do: rather than analyzing or reordering existing test artifacts, models at this level generate new artifacts from scratch. Large language models can work on semi-formal product line specifications to detect logical inconsistencies or convert plain, human-readable behavior definitions directly into executable test scripts.

LLM-Based Early-Stage Product Line Scope Definition and Variation Analysis

---

[23] Zarrad, A., Armstrong, T., & Jemai, J. (2025/2026). A hybrid approach to software testing efficiency: Stacked ensembles and deep Q-learning for test case prioritization and ranking. Computers, Materials & Continua, 86(3). https://doi.org/10.32604/cmc.2025.072768

[24] Vo, H. D., & Nguyen, T. T. (2024). CLUE: A clustering-based test reduction approach for software product lines. Journal of Computer Science and Cybernetics, 40(2), 165–185. https://doi.org/10.15625/1813-9663/19694

Le et al[25]. took this a step further with an LLM-based method that validates product line designs even before any code exists or a formal feature model has been created.

Methodology Flow (Draft Template-Based Early Validation)

1. Creation of a draft feature model: The features of the product line and the constraints between them are defined using semi-formal expressions that are close to natural language but machine-interpretable. Rules such as “A smartwatch always has the Screen feature” or “The GPS feature is available only when the Cellular Data feature is enabled” form the draft feature model.
2. LLM-based analysis operations (AOs): The draft text is fed directly into an LLM analysis engine for evaluation. Using specially designed command templates, the LLM performs 16 standard specialized tasks through logical inference, including conflict detection, dead feature detection, and configuration space size estimation.
3. Comparison with the FLAMA solver: The study showed that reasoning-focused models such as Grok 4 Fast Reasoning and Gemini 2.5 Pro produced results that matched those of the mathematical formal solver FLAMA with 88–89% accuracy.

This method allows for the early elimination of logical errors in a software product line before investing in complex formal methods infrastructure.

Transformer-Based Cucumber/Selenium Test Script Generation

Poth et al[26]. (2025) investigated whether LLMs can generate directly executable Selenium UI test scripts from Gherkin specifications and compared the generated scripts with human-written scripts in terms of expression and scope. They noted that while the scope was comparable, the generated scripts still required some manual adjustments.

3.2. Open-source Test Projects

In the previous section, while academic methods were classified according to a four-layer AI framework (Foundational AI, Machine Learning, Deep Learning, Generative AI and Agent-Based AI), the following open-source projects were examined according to six functional layers.

- ***Learning Layer:*** Stores and provides ML models, experience buffers, and policy control points. It roughly corresponds to Layers 2–3 of the academic framework (Machine Learning and Deep Learning).
- ***Planning Layer:*** Handles test planning, prioritization decisions, and routing test cases to appropriate execution agents. It leverages optimization algorithms from Layer 1 (Basic AI).
- ***Execution Layer:*** Performs actual test execution, generation, fuzz testing, and browser automation. It hosts tools that implement Layer 1 (optimization-based generators) and Layer 4 (Generative AI-based generators).

[25] Le, V.-M., Tran, T. N. T., Lubos, S., Felfernig, A., & Garber, D. (2026). Early-stage product line validation using LLMs: A study on semi-formal blueprint analysis. arXiv:2604.20523. https://arxiv.org/abs/2604.20523

[26] Poth, A., Rrjolli, O., Wang, H., & Schmid, K. (2025). Baseline Evaluation of LLM-Facilitated UI Test-Case Generation from Gherkin Specifications. In: Yilmaz, M., Clarke, P., Riel, A., Messnarz, R., Zelmenis, M., Buce, I.A. (eds) Systems, Software and Services Process Improvement. EuroSPI 2025. Communications in Computer and Information Science, vol 2657. Springer, Cham. https://doi.org/10.1007/978-3-032-04288-0_3

- ***Evaluation Layer:*** Evaluates test results, calculates quality metrics, performs visual regression analysis, and verifies compliance with the contract.
- ***Monitoring Layer:*** Provides real-time CI/CD observability, trend analysis, anomaly detection, and coverage tracking across the entire pipeline.
- ***Agent Orchestration Layer:*** Coordinates multi-agent workflows, manages agent lifecycles, and integrates heterogeneous test agents into end-to-end pipelines. This corresponds to the 4th layer (Agent-Based AI) of the academic framework.

RL4SE — Reinforcement Learning for Software Engineering

Overview

RL4SE is a research initiative and repository that systematically maps the application of Reinforcement Learning (RL) to Software Engineering (SE) tasks. The main output is the research paper by Lin et al[27]. titled "An Overview of Reinforcement Learning for Software Engineering"; this paper reviews 115 peer-reviewed studies from 22 SE domains published since the emergence of Deep Reinforcement Learning (around 2015).

The repository classifies SE tasks into design, development, quality assurance, and maintenance. In the quality assurance domain, RL4SE presents a taxonomy of RL algorithms applied to test case prioritization (TCP), test case selection, program repair, and vulnerability detection.

Industrial use cases: CI/CD test suite optimization, adaptive regression test prioritization, and automated program repair in continuous deployment pipelines.

RL4SE's Internal Architecture

RL4SE defines the canonical agent-environment interaction cycle for software engineering tasks:

- *State Encoder*: Converts raw software artifacts (such as code differences, test metadata, and historical error rates) into feature vectors. Common techniques include AST-based embedding, code2vec representations, and handcrafted CI features such as execution time, change frequency, and the timestamp of the last error.
- *Policy Network:* Maps state representations to actions. Common architectures include feedforward networks (DQN variants) for discrete action spaces and policy gradient networks (PPO, A2C) for continuous or large discrete spaces.
- *Reward Function:* Typically binary (test pass/fail) or continuous (APFD—Average Percentage of Detected Faults, coverage difference, or time to first failure).
- *Environment Wrapper:* Interfaces with the CI system to execute selected or prioritized tests and return results.

Core Algorithms

- *Deep Q-Network (DQN) and Variants:* Used when the action space is discrete (e.g., selecting one of N test cases). The Q-function $Q(s,a;\theta)$ estimates the expected cumulative reward of performing action a in state s. Dual DQN and Duel DQN reduce over-prediction bias and improve value decomposition.
- *Parallel Policy Optimization (PPO):* Applied specifically when the action space is a permutation—such as in test suite ranking—and the policy must be learned directly.

[27] Lin, K., Wang, D., You, H., et al. (2024). A survey of reinforcement learning for software engineering. arXiv preprint arXiv:2408.xxxxx. https://github.com/KaiWei-Lin-lanina/RL4SE

- *Multi-Armed Bandit (MAB):* Simpler formulations model each test case as an arm. Thompson Sampling and the Upper Confidence Bound (UCB) algorithm select tests that maximize the acquisition of information regarding the probability of failure.
- *RETECS (Reinforcement Learning for Test Case Selection):* Spieker et al. propose a CI-based TCP that uses reward signals derived from test execution results and time constraints after each CI cycle to update the agent's policy.

Artificial Intelligence/Machine Learning Techniques

- Learning Paradigm: Online reinforcement learning with episodic updates after each CI cycle.
- Models: DQN, Dual DQN, Double DQN, PPO, A2C, and REINFORCE (with a base model).
- State Representations: Hand-crafted features (execution time, past decisions, and code changes) and learned embedding vectors (code2vec or CodeBERT fine-tuning).
- Reward Shaping: Domain-specific metrics (APFD, Normalized APFD, and cost-focused APFD).

Advantages

- Provides a formal framework for modeling software engineering tasks as sequential decision processes with theoretical convergence guarantees.
- Online learning enables adaptation to project drift without requiring retraining from scratch.
- Reward functions can encode arbitrary business objectives (minimizing CI time, maximizing early defect detection, or enforcing budget constraints).
- Comprehensive empirical validation across numerous industrial CI datasets.

Limitations

- *Reward sparsity*: Test failures are rare events in mature projects, leading to sparse positive signals and slow convergence.
- *Cold-start problem:* Insufficient historical data in new projects reduces the initial policy quality.
- The state representation design is labor-intensive and project-specific.
- The computational burden of maintaining and updating RL policies within tight CI time budgets.

Midscene — Vision-Language Model UI Testing

Overview

Midscene.js is an AI-driven UI automation framework that replaces traditional selector-based testing (CSS, XPath) with Vision-Language Model (VLM) based element identification and interaction. The system interprets screenshots of the application UI using multimodal models to locate elements, extract data, and execute actions specified in natural language.

GitHub: https://github.com/web-infra-dev/midscene

Industrial use cases: Cross-platform UI testing (web, Android, iOS, desktop), visual regression testing, and canvas/WebGL application testing where DOM-based selectors are unavailable.

Internal Architecture

- SDK Layer: Provides JavaScript APIs (aiAction, aiQuery, aiAssert) and YAML-based scripting for test definitions, integrating with Playwright, Puppeteer, and native mobile drivers.
- VLM Integration: A model-agnostic design supporting GPT-4o, Qwen-VL, UI-TARS, Gemini, and GLM-4V. Models receive screenshots and natural language instructions, returning element coordinates or extracted data.
- Caching Layer: Records VLM responses for deterministic replay. "Instant Actions" bypass VLM inference for previously seen interactions, reducing latency and API costs.

- Debug Reporter: Generates HTML reports with annotated screenshots showing VLM reasoning traces and element identification results.

Core Algorithms

- Vision-Language Grounding: The VLM performs visual grounding—mapping natural language descriptions (like “the submit button”) to pixel coordinates in the screenshot. This is a multimodal attention mechanism where the model jointly attends to image patches and text tokens.
- Set-of-Mark Prompting: Some configurations annotate screenshots with numbered markers overlaid on interactive elements. The VLM references these markers rather than computing raw coordinates, improving precision.
- Action Planning: For multi-step instructions, the VLM decomposes the natural language goal into a sequence of atomic actions (click, type, scroll), each grounded to specific UI elements.
- Response Caching: A content-addressable cache keyed by screenshot hash and instruction text. Cache hits bypass VLM inference entirely.

AI/ML Techniques

- Models: Multimodal Vision-Language Models (GPT-4o, Qwen-VL, UI-TARS, and Gemini). UI-TARS is specifically trained for UI understanding tasks.
- Inference: Zero-shot or few-shot prompting of pre-trained VLMs. No fine-tuning is performed by default.
- Optimization: Caching reduces redundant VLM calls. Instant Actions provide near-zero latency for repeated interactions.

Advantages

- Reduces selector maintenance: Tests remain resilient to DOM changes as long as the visual layout remains consistent.
- Supports canvas, WebGL, native mobile, and desktop applications where DOM-based selectors fail.
- Natural language test specifications lower the authoring barrier for non-developers.
- A model-agnostic design enables self-hosted deployment with open-source VLMs for data privacy.
- The caching mechanism amortizes VLM inference costs over repeated test runs.

Limitations

- VLM inference latency (seconds per action) is significantly higher than selector-based approaches (milliseconds).
- VLM accuracy depends on model capability; small or ambiguous UI elements may be misidentified.
- API costs for cloud-hosted VLMs can be substantial at scale.
- Non-deterministic VLM outputs require caching or multiple retries for stability.
- A screenshot-based approach misses non-visual states (like JavaScript variables and hidden form fields).

Promptfoo — LLM Evaluation and Red Teaming

Overview

Promptfoo is a CLI-first framework for systematic LLM evaluation, prompt engineering, and adversarial red teaming. It enables developers, acting as LLM evaluators, to define evaluation matrices that compare multiple prompts, models, and test cases using configurable validation types, including

semantic similarity and structural validation. Promptfoo continues to operate as an active open-source project following its acquisition by OpenAI in March 2026.

GitHub: https://github.com/promptfoo/promptfoo

Industrial use cases: LLM prompt optimization, AI security red teaming, RAG pipeline evaluation, and CI/CD quality checks for LLM-powered features.

Internal Architecture

- Configuration Layer: YAML-based evaluation definitions that enable version control and CI/CD integration.
- Provider Abstraction: A unified interface for over 50 LLM providers (including OpenAI, Anthropic, Google, AWS Bedrock, and Ollama for on-premises models).
- Red Team Module: Automated generation of adversarial inputs targeting the OWASP LLM Top 10 vulnerabilities (e.g., prompt injection, jailbreak, PII leakage, and RAG document leakage).
- Validation Engine: Extensible to support deterministic checks (regex, JSON schema) and non-deterministic evaluation strategies (LLM as a referee, embedding similarity).

Core Algorithms

- As an LLM-referee: A secondary LLM evaluates the primary model's output according to the criteria defined in the validation configuration. The referee prompt includes the original input, the model's output, and the evaluation criteria. The referee returns a structured decision (pass/fail) with justification.
- Semantic Similarity: Outputs are converted into embedding vectors using models such as text-embedding-3-small. The cosine similarity between the output embedding and the expected output embedding is calculated, and a configurable threshold determines the pass/fail status.
- Adversarial Input Generation: The red team module uses an adversarial LLM to generate adversarial prompts targeting specific vulnerability categories. These generated inputs are executed against the target application to test successful exploits.
- Matrix Evaluation: Execution of the Cartesian product (prompts × providers × test cases) via parallel inference and result aggregation.

Artificial Intelligence/Machine Learning Techniques

- LLM-Based Evaluation: Model-graded evaluation using structured rubric prompts.
- Embedding-Based Similarity: Comparison of outputs in a vector space using text embedding models.
- Adversarial Generation: LLM-based red teaming for vulnerability scanning.
- No Training: Promptfoo does not train models; it leverages pre-trained LLMs for evaluation and adversarial generation.

Advantages

- Local-first execution ensures data privacy and compliance.
- Vendor-agnostic design enables model comparison without vendor lock-in.
- YAML configurations enable version-controlled, reproducible evaluations.
- The Red Team module provides automated OWASP LLM Top 10 vulnerability scanning.
- An extensible validation framework supports domain-specific evaluation criteria.

Limitations

- As an LLM-based approach, the evaluation model introduces bias and non-determinism.

- The scope of red teaming depends on the creativity of the attacker model; new attack vectors may be overlooked.
- There is no built-in experiment tracking feature (requires external integration for longitudinal analysis).
- Evaluation costs increase linearly with the size of the test matrix (prompts × providers × scenarios).

DeepEval — LLM Testing and RAG Evaluation

Overview

DeepEval is an LLM evaluation framework designed to perform unit testing of LLM applications. Offering a Pytest-like developer experience, DeepEval provides over 50 research-backed metrics to evaluate RAG pipelines, chatbots, and general LLM outputs using a paradigm that employs the LLM as a Referee, complete with self-explanatory score justifications.

GitHub: https://github.com/confident-ai/deepeval

Industrial use cases: RAG pipeline quality assurance, LLM output regression testing, hallucination detection, and CI/CD integration for LLM-powered applications.

Internal Architecture

- Test Case Abstraction: LLMTestCase and ConversationalTestCase encompass inputs, outputs, and contexts.
- Metrics Engine: Each metric class implements a measure() method that invokes the evaluated LLM using a metric-specific prompt template and returns a score in the [0,1] range along with a natural language explanation.
- Synthetic Data Generator: Generates “gold” (question-answer pairs) from a text corpus for bootstrap evaluation datasets.
- Trusted AI Platform: Optional cloud integration for experiment tracking, regression monitoring, and production-level observability.

Base Algorithms

- Loyalty Metrics: Breaks down the generated output into individual claims and verifies each claim by comparing it to the context obtained using the referee LLM.
- Contextual Accuracy: Evaluates the quality of the ranking of the retrieved segments. For each segment, the referee assesses its relevance to the query.
- Contextual Recall: Measures whether the received context contains all the information necessary to reconstruct the expected output. The evaluator breaks down the expected output into statements and compares each one with the context.
- Hallucination Detection: Identifies claims in the output that are not supported by the provided context. Similar to Faithfulness, it uses inference and verification at the claim level, but is applied in the opposite direction.
- Answer Relevance: The evaluator assesses whether the output directly addresses the input question, regardless of its factual accuracy.

AI/Machine Learning Techniques

- LLM as Referee: All metrics use an LLM as a referee for semantic evaluation. The default referee is GPT-4o, but custom models are supported via DeepEvalBaseLLM.
- Claim Parsing: Parsing outputs into atomic claims inspired by NLI for fine-grained verification.

- Synthetic Data Generation: Generation of LLM-driven evaluation datasets from document corpora.
- No Training: DeepEval does not train custom models; instead, it relies on pre-trained LLMs to act as evaluation judges.

Advantages

- Pytest-specific integration provides a familiar developer workflow for LLM testing.
- Self-explanatory metrics provide actionable debugging insights that go beyond numerical scores.
- Component-level RAG evaluation (receptor vs. generator) isolates the root causes of failures.
- A comprehensive metric library (50+) covering accuracy, relevance, toxicity, bias, and domain-specific criteria.
- Framework-agnostic: Works with LangChain, LlamaIndex, or raw API calls.

Limitations

- Judge LLM costs are multiplicative: each metric requires one or more judge inference calls per test case.
- Bias in the judge model may systematically over- or under-score certain output styles.
- Evaluation uncertainty requires multiple runs for statistical confidence.
- The quality of synthetic data generation depends on the source dataset and the generation model.

4. Methodology Overview and Core Principles

4.1. Methodology Overview

The proposed methodology provides an integrated process for testing and optimizing software product lines while explicitly considering the variability among product variants. The process consists of five interconnected stages: variability modelling, metric extraction, test generation, test execution, and optimization and feedback. Together, these stages support the systematic selection, generation, execution, and evaluation of tests across multiple product configurations.

The first stage establishes a variability model describing the common and variable features of the product line, their dependencies, and the constraints governing valid configurations. Requirements, features, variants, software components, and test cases are linked through traceability relations[28]. This information provides the basis for determining which parts of the product line may be affected by a requirement or code change and which test assets should consequently be selected.

In the metric-extraction stage, information is collected from requirements-management systems, source-code repositories, issue-tracking platforms, continuous-integration environments, and previous test executions. Product-level metrics, such as code and requirement coverage, are combined with product-line-level metrics, including feature, variant, configuration, and feature-interaction coverage. Change frequency, historical failures, execution time, component criticality, and risk information may also be used to support test selection and prioritization.

Test cases are subsequently generated or selected according to the features and configurations under evaluation. Existing tests are reused whenever their associated requirements, features, and

[28] Agh, H., Azamnouri, A., & Wagner, S. (2024). Software product line testing: A systematic literature review. *Empirical Software Engineering, 29*, Article 146. https://doi.org/10.1007/s10664-024-10516-x

components remain unchanged. For newly introduced or modified functionality, additional tests may be produced through model-based, rule-based, combinatorial, or AI-assisted generation techniques. The generated tests retain their links to the relevant requirements, features, variants, and source-code elements to maintain end-to-end traceability.

During test execution, the selected tests are run against the relevant product variants, and their results are recorded together with coverage, duration, failure, and configuration information. The collected results are then evaluated during the optimization and feedback stage. Change-impact information, risk scores, historical failures, coverage contributions, and execution costs are used to identify redundant tests, prioritize critical tests, and improve subsequent test cycles. The methodology therefore forms an iterative process in which each execution provides additional evidence for future test selection, generation, and optimization decisions.

4.2. Core Principles

The methodology is based on four core principles: variability awareness, traceability-driven testing, metric-based optimization, and automation-first execution.

Variability awareness means that testing decisions are made with explicit consideration of the differences and commonalities among product variants. Tests are not evaluated only at the level of an individual product; their relationships with features, feature interactions, configurations, and variants are also considered. This enables shared functionality to be tested efficiently while ensuring that variant-specific behaviour and feature combinations receive adequate attention.

Traceability-driven testing maintains bidirectional links among requirements, features, variants, software changes, and test cases. These links support change-impact analysis by showing which test cases may be affected by a modification. They also make it possible to determine which requirements and features are covered by the executed tests. Traceability is therefore treated as an operational element of the testing process rather than merely as documentation produced after development.

Metric-based optimization uses measurable evidence to support test-selection and prioritization decisions. Coverage, risk, execution time, historical failure information, change frequency, feature criticality, and configuration relevance may be evaluated together. Since these objectives can conflict, the methodology supports multi-objective optimization to achieve the highest practical coverage and risk reduction within the available time and resource constraints.

Automation-first execution aims to automate repetitive activities throughout the testing lifecycle, including data collection, impact analysis, test generation, test execution, result evaluation, issue creation, and reporting. Human review remains necessary, particularly for safety-related requirements, risk assessment, and AI-generated test assets. Accordingly, automation is used to reduce manual effort and improve consistency while keeping domain experts involved in decisions requiring engineering judgement.

5. Test Metrics Design and Analysis

5.1. Product vs Product Line Metrics

Testing metrics in software product lines must be considered at two complementary levels: the individual product level and the overall product-line level. Product-level metrics evaluate the verification status of

a specific derived variant, whereas product-line-level metrics assess how effectively the testing process covers the commonality and variability represented across the entire product family. Relying exclusively on one level may hide important gaps; therefore, both perspectives are required.

At the product level, **code coverage** measures the proportion of the implementation exercised by the test suite of a particular variant. Depending on the required assurance level, it may include statement, branch, condition, function, or Modified Condition/Decision Coverage. Code coverage helps identify implementation areas that have not been executed, but it does not independently demonstrate that all intended behaviours or requirements have been verified.

Requirement coverage measures the extent to which the functional, non-functional, safety, and quality requirements applicable to a variant are associated with and exercised by test cases. Bidirectional traceability should make it possible to identify both requirements without tests and tests without an associated requirement. Requirement coverage is especially important in safety-critical systems because verification evidence must correspond to the exact requirements applicable to the delivered configuration.

Product-line-level metrics extend this assessment beyond a single product. **Feature coverage** measures the proportion of product-line features exercised by at least one test in one or more valid variants.[29] It may be calculated for all features or separately for mandatory, optional, alternative, and safety-critical features. However, testing every feature individually does not guarantee that interactions between features have been sufficiently verified.

Variant coverage measures the proportion of defined, released, or operationally relevant variants included in the testing process. Since testing all theoretically possible variants is generally infeasible, the metric should be interpreted against a clearly defined target set. This set may include released products, representative variants, high-risk configurations, customer-specific variants, or variants selected through a sampling strategy.

Configuration coverage evaluates how effectively the executed tests represent the valid configuration space defined by the feature model. Unlike variant coverage, which focuses on identifiable products, configuration coverage considers the combinations of selected and excluded features. It can be supported by sampling criteria such as pairwise or *t*-wise coverage and may be weighted according to feature criticality, usage frequency, or risk.

Product-level and product-line-level metrics should be analysed together. High code and requirement coverage for a small number of products does not imply sufficient coverage of the overall product line. Similarly, broad feature or variant coverage does not guarantee adequate verification within each delivered product. The combined use of these metrics provides a more accurate view of testing completeness and supports the identification of under-tested variants, features, requirements, and implementation areas.

[29] Cmyrev, A., & Reissing, R. (2014). Efficient and effective testing of automotive software product lines. *Applied Science and Engineering Progress, 7*(2), 53–57. Article page

5.2. Variability-aware Coverage Metrics

Variability-aware coverage metrics evaluate whether testing adequately addresses the differences and interactions among product configurations. Conventional code and requirement coverage metrics are calculated for a specific product and do not indicate how thoroughly the variability represented by the feature model has been exercised. Variability-aware metrics complement these measures by considering feature selections, feature combinations, configuration constraints, and variant-specific behaviour.

Feature Interaction Coverage measures the extent to which relevant combinations of features are exercised by the test suite. Testing each feature independently is insufficient because failures may emerge only when two or more features are enabled together. These interactions may result from explicit dependencies in the feature model, shared implementation components, data exchanges, resource conflicts, or behavioural assumptions between features.

Feature interaction coverage can be calculated using *t*-wise criteria.[30] Pairwise coverage requires every valid combination of two feature selections to appear in at least one tested configuration, while higher-strength criteria consider combinations of three or more features. Increasing the interaction strength can improve fault-detection capability but also increases the number of configurations and tests required. The selected interaction strength should therefore reflect system criticality, known dependencies, historical failures, and available testing resources.

Not all feature interactions have equal importance. Combinations involving safety-critical, frequently changed, commonly used, or historically failure-prone features may be assigned higher weights. Explicit feature-model constraints and dependencies can also be used to identify interactions requiring particular attention. Invalid combinations must be excluded from the coverage calculation so that the metric reflects only configurations that can be derived and executed.

Configuration Space Coverage measures how effectively the tested configurations represent the complete set of valid configurations defined by the feature model. A simple ratio between tested and valid configurations may be used for small product lines. However, this measure becomes less informative for large configuration spaces, where exhaustive testing is impossible and the tested proportion may remain very small even when a carefully selected sample provides meaningful interaction and risk coverage.

For large product lines, configuration space coverage should therefore be assessed through multiple indicators, such as *t*-wise interaction coverage, feature selection frequency, inclusion of boundary configurations, and representation of high-risk or operationally important regions. A configuration can also be weighted according to its market relevance, deployment frequency, customer importance, system criticality, or difference from previously tested variants.

Variability-aware metrics should be calculated together with traceability and execution information. Each tested configuration must be linked to its selected features, applicable requirements, associated test cases, and results. This enables engineers to identify uncovered feature interactions, under-represented configuration regions, and variants requiring additional testing. The resulting information supports

[30] Johansen, M. F., Haugen, Ø., & Fleurey, F. (2012). An algorithm for generating *t*-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference (SPLC '12)*, 46–55. https://doi.org/10.1145/2362536.2362547

configuration sampling, regression-test selection, and test prioritization while avoiding the cost of exhaustive product-line testing.

5.3. Quality Metrics

Coverage metrics indicate which parts of a product or product line have been tested, but they do not independently demonstrate the quality or effectiveness of the testing process. Quality metrics complement coverage information by measuring safety assurance, reliability, performance, defect behaviour, and the risk associated with insufficiently tested features or variants.

Safety metrics evaluate whether identified hazards and their associated safety requirements have been adequately verified. Hazard coverage measures the proportion of recognised hazards addressed by one or more test cases. It may be extended by considering hazard severity, likelihood, detectability, and the safety integrity level assigned to the relevant function. Tests related to high-severity hazards should receive greater importance than tests addressing low-impact conditions.

Reliability metrics assess the ability of a product or variant to operate without failure under specified conditions. Failure density can be calculated as the number of detected failures relative to an appropriate unit, such as executed tests, operating time, transactions, requirements, features, or code size. Additional indicators may include failure frequency, mean time between failures, recurring defect rate, test pass rate, and failure trends across successive releases.

Reliability should also be evaluated across variants. A failure appearing in a shared core component may affect many products, while a failure in a variant-specific feature may have a more limited scope. Analysing failures according to module, feature, variant, configuration, and product release helps distinguish local defects from systematic problems affecting the wider product line.

Performance metrics measure whether the system satisfies its expected timing and resource-consumption requirements. Depending on the application, these may include response time, throughput, latency, CPU and memory utilisation, network usage, energy consumption, and execution time. Performance should be evaluated under relevant feature combinations and workloads because enabling or disabling particular features may significantly change system behaviour.

Since individual metrics represent different quality concerns, a **weighted metric model** may be used to create an aggregated assessment. Weights can be assigned according to safety criticality, business importance, feature usage, customer impact, or certification requirements. The weighting scheme must remain transparent so that engineers can understand how the resulting score was calculated and adjust it according to the context of a particular variant.

Risk-based scoring combines the probability and potential impact of failure with available testing evidence.[31] Factors may include hazard severity, component criticality, change frequency, historical failure density, code complexity, feature interaction exposure, test coverage, and the number of affected

[31] Felderer, M., & Schieferdecker, I. (2014). A taxonomy of risk-based testing. *International Journal on Software Tools for Technology Transfer, 16*(5), 559–568. https://doi.org/10.1007/s10009-014-0332-3

variants. Areas with high impact, frequent changes, poor coverage, or recurring failures receive higher risk scores and can consequently be prioritised for additional testing.

Quality metrics should not be interpreted in isolation. High coverage does not necessarily indicate low risk, while a low number of observed failures may result from insufficient testing. Combining safety, reliability, performance, coverage, and risk information provides a more balanced assessment and supports test selection, prioritisation, release decisions, and continuous improvement across the product line.

6. Variant based Testing and Test Generation

This section describes the variant-based testing and test generation approach adopted in the SOSIS methodology. It is organised as a sequence of four elements that build on one another. A traceability model, presented in Subsection 6.1, establishes the links between requirements, features, variants, and test cases, and provides the structure on which the remaining elements rely. Feature-based test design, in Subsection 6.2, organises test assets around features instead of around whole products, so that the logic used to exercise a feature travels with that feature. Automated test generation, in Subsection 6.3, produces configurations and test cases from those assets without enumerating every derivable product. Test reuse, in Subsection 6.4, carries test artefacts and results from one variant to the next and confines the retest effort to the parts affected by the differences between them.

The four elements respond to the difficulties set out in Section 2.2. Certification evidence has to hold per variant, while the number of variants makes independent re-verification of each one impractical, and the mechanisms on which the standards rely, namely requirements-based testing with measured coverage and bidirectional traceability, have to remain available in a variant-rich setting. The approach therefore treats traceability and reuse as primary concerns rather than as by-products of test execution. Figure 1 presents the traceability chain that underpins the section as a whole, and Figure 2 summarises the reuse strategy that closes it.

6.1. Traceability Model

Traceability is the mechanism that connects the problem space of a product line to the artefacts that realise and verify it. In the SOSIS methodology, test assets are linked along an explicit chain that runs from requirements, through the features that satisfy them, to the variants in which those features are bound, and finally to the individual test cases that exercise them. Figure 1 shows this chain together with the engineering context in which each element is produced.

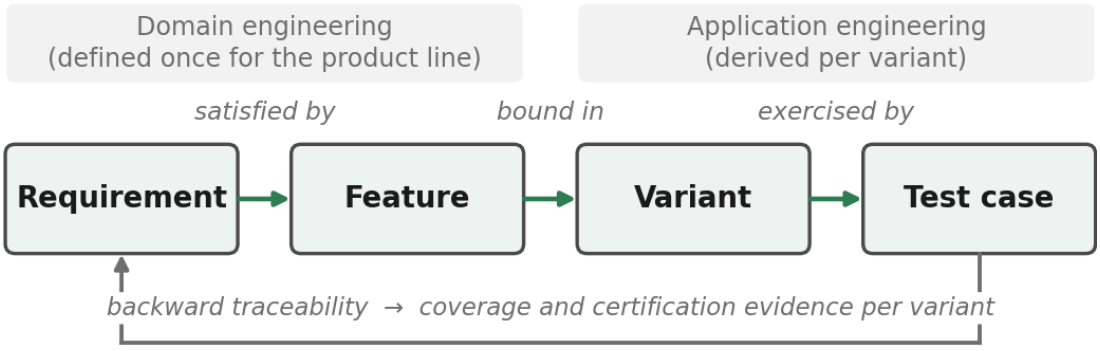

Figure 1. Traceability chain from requirements to test cases across domain and application engineering.

Each link in the chain carries a specific meaning. A requirement is satisfied by one or more features, which are the units in which the product line expresses its commonality and its variability. A feature is bound in a variant when a configuration selects it, so a variant is a resolved selection of features rather than an independently developed product. A variant is in turn exercised by the test cases assembled for it. Reading the chain from left to right moves from what the product line has to do towards the concrete artefacts that demonstrate that it does so.

The chain crosses the separation between domain engineering and application engineering that underpins software product line engineering[32]. Requirements and features belong to the domain level, where they are defined once for the whole product line, together with the reusable test components attached to them. Variants and their test cases belong to the application level, where they are derived for a specific product by resolving the variability. This division is what makes reuse possible, because work performed once at the domain level is drawn upon repeatedly at the application level.

Followed in the forward direction, the chain supports change impact analysis. When a requirement is modified, the features that satisfy it can be identified, then the variants in which those features are bound, and finally the test cases that exercise them. The result is a bounded set of artefacts to be re-examined, which is more useful operationally than a general instruction to retest, and it becomes the input to the reuse strategy described in Subsection 6.4.

Followed in the backward direction, shown by the lower path in Figure 1, the chain supports coverage and certification arguments. Each test case can be attributed to the feature and the requirement it addresses, so coverage can be reported for a specific variant rather than only for the product line as a whole. This corresponds to the expectation described in Section 2.2 that every requirement is exercised by a test and every test is attributable to a requirement, and it allows verification gaps to be located per configuration rather than in aggregate.

Maintaining the links in both directions is what distinguishes this model from a simple record of which tests exist. The secondary studies referenced in Section 2.2 identify the preservation of traceability between test assets and other artefacts as a concern that remains only partly addressed, which is consistent with the difficulty of keeping links current while a product line evolves. The model therefore treats the links as artefacts to be maintained alongside the items they connect, rather than as documentation assembled once verification is complete.

In practice the model is realised as a set of relations recorded together with the product line assets: requirement identifiers associated with features in the feature model, feature selections recorded in each variant configuration, and test components associated with features and instantiated per variant. The usefulness of any analysis built on these relations depends on the discipline with which they are captured, since links that are incomplete or out of date will narrow or misdirect an impact analysis. The methodology consequently assumes that traceability is recorded as part of the normal engineering workflow and kept under the same version control as the assets themselves.

6.2. Feature-based Test Design

Test design in the methodology is organised around features rather than around whole products. The starting point is that in a product line the same functionality recurs across many variants, so designing a test suite separately for each product repeats work that has already been done. Organising the design

[32] Pohl, K., Böckle, G., & van der Linden, F. J. (2005). Software Product Line Engineering: Foundations, Principles, and Techniques. Springer-Verlag. https://doi.org/10.1007/3-540-28901-1

around features moves the effort to the level at which commonality and variability are expressed, which is also the level at which the traceability model records its links.

Each feature, or a cohesive group of interacting features, is associated with reusable test components. A test component specifies how the feature is to be exercised, including the stimuli applied, the expected observable behaviour, and any preconditions on the configuration in which it is valid. It is written independently of the particular variant in which the feature will appear, and it is parameterised where the behaviour of a feature depends on choices made elsewhere in the configuration.

When a variant is configured, its test suite is assembled by composing the components of the features that the variant includes. The composition is driven by the configuration itself, so the suite for a variant follows from its feature selection rather than from a separate design activity performed per product. This is the point at which the links described in Subsection 6.1 are used constructively, since the relation between features and test components determines what enters the suite.

Composition alone does not account for behaviour that arises from combinations of features. Interactions, where the joint presence of two or more features produces behaviour that neither exhibits alone, require test components of their own, associated with the combination rather than with a single feature. Determining which combinations deserve attention is a question of sampling and generation, addressed in Subsection 6.3. The design concern here is that components covering the relevant interactions exist and are attributable to the combination they concern, so that interaction coverage can be reported as well as feature coverage.

The distinction between product-based, family-based, and feature-based approaches is established in the literature on analysis and verification strategies for product lines[33]. Product-based approaches treat each derived product separately, family-based approaches reason over the whole product line at once, and feature-based approaches operate on the artefacts of individual features. The design described here is feature-based in that sense, while the generation techniques in Subsection 6.3 introduce sampling over products and the reuse strategy in Subsection 6.4 exploits relations between them, so the methodology combines the strategies rather than adopting only one.

The feature model is central to this organisation, since it defines which features exist, how they may be combined, and which constraints restrict the configuration space. Work on learning software configuration spaces has reviewed how the properties of configurable systems are related to the options selected and which techniques have been used to reason about large configuration spaces from samples[34]. For test design the practical consequence is that the feature model is treated as a shared reference used by design, generation, and traceability alike, rather than as documentation maintained separately from the test assets.

The effort implication of feature-based design is that test-design work concentrates on what varies. A feature that appears in many variants carries its verification logic with it, so the incremental cost of an additional variant is driven by the features that are new to it and by the interactions it introduces, rather than by the total size of its configuration. This property is what the reuse strategy in Subsection 6.4 builds upon. It also makes the design sensitive to the quality of the feature model, because features that are poorly separated tend to produce test components that cannot be reused without modification.

---

[33] Thüm, T., Apel, S., Kästner, C., Schaefer, I., & Saake, G. (2014). A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Computing Surveys, 47(1), Article 6, 45 pp. https://doi.org/10.1145/2580950

[34] Pereira, J. A., Acher, M., Martin, H., Jézéquel, J.-M., Botterweck, G., & Ventresque, A. (2021). Learning software configuration spaces: A systematic literature review. Journal of Systems and Software, 182, Article 111044. https://doi.org/10.1016/j.jss.2021.111044

6.3. Automated Test Generation

Exhaustive testing of every derivable variant is not attainable in a product line of realistic size, so the methodology relies on generating configurations and test cases rather than enumerating products. Generation is applied at two levels: selecting which configurations to test, and producing the test cases that exercise a selected configuration. The techniques described below address these two levels and are complementary rather than mutually exclusive.

Combinatorial, or t-wise, sampling addresses the first level. Instead of testing all configurations, a subset is selected so that every interaction among any t features appears in at least one sampled configuration, on the rationale that faults are frequently triggered by interactions of low order. Constructing such samples in the presence of constraints has been studied as a combinatorial problem, and greedy constructions for highly configurable systems with constraints are established in the literature[35]. Comparative studies of pairwise testing for product lines have examined how different construction strategies behave when applied to feature models[36].

Because a feature model admits only configurations that satisfy its constraints, sampling has to operate over valid configurations, which connects it to satisfiability solving and model counting. Work on uniform and scalable sampling of highly configurable systems has addressed how to draw configurations so that the resulting sample is representative of the valid configuration space, and how far this scales on real models[37]. Adaptive and weighted sampling platforms have also been proposed, so that sampling can be biased towards regions of the configuration space considered more relevant instead of treating all configurations as equally interesting[38].

The trade-off between the coverage achieved and the number of configurations to be tested has received specific attention, since a higher value of t raises both the size of the sample and the time needed to construct it. Recent work has proposed relaxing uniform coverage across all features and assigning higher interaction strength only to selected subsets, so that smaller samples are obtained while higher-order coverage is retained where it is judged to matter[39]. Other work has examined how far uniform random sampling can be improved to reach high t-wise coverage, including completion and reduction steps applied to the samples obtained[40]. For a safety-related product line these results are directly relevant, because the size of the sample determines how much verification has to be executed for each release.

---

[35] Cohen, M. B., Dwyer, M. B., & Shi, J. (2008). Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. IEEE Transactions on Software Engineering, 34(5), 633-650. https://doi.org/10.1109/TSE.2008.50

[36] Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., & Le Traon, Y. (2012). Pairwise testing for software product lines: comparison of two approaches. Software Quality Journal, 20(3-4), 605-643. https://doi.org/10.1007/s11219-011-9160-9

[37] Heradio, R., Fernandez-Amoros, D., Galindo, J. A., Benavides, D., & Batory, D. (2022). Uniform and scalable sampling of highly configurable systems. Empirical Software Engineering, 27(2), Article 44. https://doi.org/10.1007/s10664-021-10102-5

[38] Baranov, E., & Legay, A. (2022). Baital: an adaptive weighted sampling platform for configurable systems. In 26th ACM International Systems and Software Product Line Conference (SPLC), Volume B (pp. 46-49). ACM. https://doi.org/10.1145/3503229.3547030

[39] Pett, T., Krieter, S., Thüm, T., & Schaefer, I. (2024). MulTi-Wise Sampling: Trading Uniform T-Wise Feature Interaction Coverage for Smaller Samples. In 28th ACM International Systems and Software Product Line Conference (SPLC) (pp. 47-53). ACM. https://doi.org/10.1145/3646548.3672589

[40] Heß, T., Schmidt, T. J., Ostheimer, L., Krieter, S., & Thüm, T. (2024). UnWise: High T-Wise Coverage from Uniform Sampling. In 18th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS) (pp. 37-45). ACM. https://doi.org/10.1145/3634713.3634716

Mutation-based generation addresses the adequacy of the tests rather than the selection of configurations. Systematic changes, called mutants, are introduced into the artefacts under test or into the feature model in order to represent potential faults, and test configurations are generated and then assessed by their ability to distinguish the mutants from the original. Applying this idea to product line test configurations yields a fault-oriented criterion that complements interaction coverage[41].

Model-based generation addresses the production of test cases for a configuration once it has been selected. Behavioural models annotated with variability describe the intended behaviour of the product line, and the test cases for a given configuration are derived from the model restricted to that configuration, so that the tests follow from a specification maintained once for the family. Surveys of model-based product line testing document the range of formalisms and derivation strategies that have been applied in this setting[42].

More recently, language models have been investigated as a further source of test cases. Empirical evaluations of large language models for automated unit test generation report results on the tests produced, including whether they compile and the coverage they achieve, alongside limitations such as generated tests that fail to compile or that encode incorrect expected behaviour[43]. Broader surveys map how these models have been applied across testing activities and identify open problems, among them the evaluation of the artefacts generated[44][45]. Reported industrial application has concentrated on improving existing test suites rather than replacing them, with generated tests filtered before adoption[46]. Within the present methodology such techniques are treated as candidate sources of test cases whose output enters the same traceability and coverage accounting as any other test, and whose use in a safety-related context depends on the review applied to what they produce.

6.4. Test Reuse Strategy

The final element of the methodology is a strategy for reusing test artefacts and results across variants, so that verifying an additional variant does not require repeating the whole test effort. The strategy combines reuse of what two variants share with a bounded treatment of what differs between them. Figure 2 summarises it, from the differences between two variants through to the evidence carried across the product line.

---

[41] Henard, C., Papadakis, M., & Le Traon, Y. (2014). Mutation-Based Generation of Software Product Line Test Configurations. In C. Le Goues & S. Yoo (Eds.), Search-Based Software Engineering (SSBSE 2014), LNCS 8636 (pp. 92-106). Springer. https://doi.org/10.1007/978-3-319-09940-8_7

[42] Oster, S., Wübbeke, A., Engels, G., & Schürr, A. (2011). A Survey of Model-Based Software Product Lines Testing. In J. Zander, I. Schieferdecker, & P. J. Mosterman (Eds.), Model-based Testing for Embedded Systems (pp. 339-381). CRC Press. https://doi.org/10.1201/b11321-14

[43] Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. IEEE Transactions on Software Engineering, 50(1), 85-105. https://doi.org/10.1109/TSE.2023.3334955

[44] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software Testing with Large Language Models: Survey, Landscape, and Vision. IEEE Transactions on Software Engineering, 50(4), 911-936. https://doi.org/10.1109/TSE.2024.3368208

[45] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large Language Models for Software Engineering: A Systematic Literature Review. ACM Transactions on Software Engineering and Methodology, 33(8), 1-79. https://doi.org/10.1145/3695988

[46] Alshahwan, N., Chheda, J., Finogenova, A., Gokkaya, B., Harman, M., Harper, I., Marginean, A., Sengupta, S., & Wang, E. (2024). Automated unit test improvement using large language models at Meta. In 32nd ACM International Conference on the Foundations of Software Engineering (FSE) (pp. 185-196). ACM. https://doi.org/10.1145/3663529.3663839

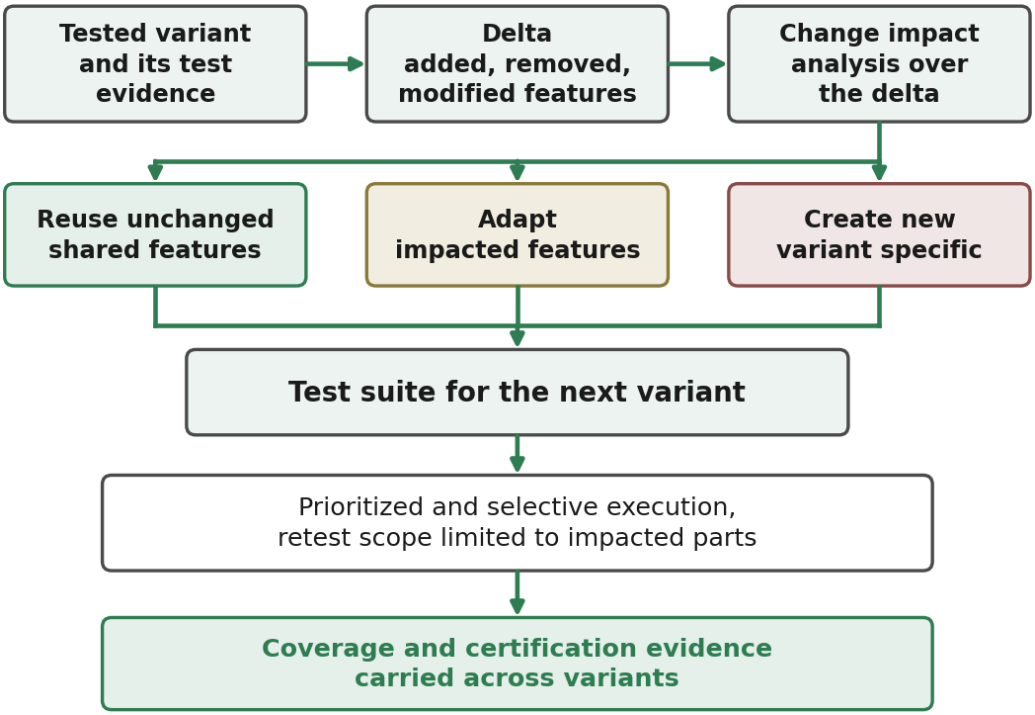

Figure 2. Summary of the test reuse strategy, from the delta between variants to the evidence carried across the product line.

Cross-variant reuse follows from the feature-based design and from the traceability model. Test components attached to features that two variants share are reused without modification, because each component was written against the feature rather than against a product. What remains is the work associated with features specific to the new variant and with the interactions that its particular configuration introduces, which is a smaller quantity than the full suite whenever the two variants have substantial commonality.

Delta-oriented, or incremental, testing addresses the relation between one variant and the next. Instead of treating each variant as an independent object of testing, the commonality and the differences between two variants, that is the delta, are made explicit, and the test artefacts for the second variant are constructed by reusing those of the first and adapting only what the delta affects. Incremental model-based testing of delta-oriented product lines has been developed along these lines[47].

The delta drives a change impact analysis that determines the retest obligations. As shown in Figure 2, the outcome partitions the test assets of the new variant into three classes: components reused unchanged, because the features they exercise are unaffected by the delta; components adapted, because the delta touches the features they exercise; and components created, because the variant introduces parts that were not present before. Confining execution to the second and third classes, together with the regression checks that the first class still requires, is what limits the retest scope for each additional variant.

In a continuous integration setting the same problem arises at a higher frequency, because changes are integrated often and executing every test at every integration is costly. A systematic mapping study of test case prioritization in continuous integration environments reports the characteristics of the

[47] Lochau, M., Schaefer, I., Kamischke, J., & Lity, S. (2012). Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In A. D. Brucker & J. Julliand (Eds.), Tests and Proofs (TAP 2012), LNCS 7305 (pp. 67-82). Springer. https://doi.org/10.1007/978-3-642-30473-6_7

approaches proposed and the way they have been evaluated, including the time constraints that distinguish this context from classical regression testing[48].

Selection and prioritization have increasingly been approached with learning-based techniques. A systematic literature review of machine learning for test case selection and prioritization analyses the techniques used, the information on which they operate, and how they have been evaluated[49]. Reinforcement learning has been applied to prioritization so that the ordering adapts to the outcomes observed over successive executions[50], and further work has addressed the scalability and the accuracy of prioritization in continuous integration contexts[51]. For highly configurable software specifically, learning-based prioritization has been studied in the continuous integration of product lines, where the configuration to which a test belongs forms part of the available information[52].

Taken together, these elements are what allow the assurance evidence discussed in Section 2.2 to be carried across a product line rather than reconstructed for each variant. Reuse of components preserves the traceability links on which the evidence rests, the delta analysis identifies what the existing evidence no longer covers, and prioritization determines the order in which the outstanding verification is executed. The residual obligation, which the methodology does not remove, is that the evidence presented for a delivered variant has to correspond to that variant, so the records produced by reuse and by incremental retesting are maintained per configuration.

7. Test Prioritization and Optimization

7.1. Problem Definition

Testing every valid variant of a software product line is generally infeasible because the number of possible configurations grows rapidly with the number of variable features and their permitted interactions. This challenge becomes more significant in safety-critical systems, where each delivered variant must be supported by sufficient verification evidence while testing time, computational capacity, and engineering resources remain limited.

Running the entire test suite after every requirement, source-code, or configuration change is also inefficient. Many tests may exercise unaffected functionality, whereas tests covering modified, critical, or historically failure-prone components may require earlier and more frequent execution. The testing process must therefore determine which variants and test cases are relevant to a particular change and how the available testing resources should be allocated.

---

[48] Prado Lima, J. A., & Vergilio, S. R. (2020). Test Case Prioritization in Continuous Integration environments: A systematic mapping study. Information and Software Technology, 121, Article 106268. https://doi.org/10.1016/j.infsof.2020.106268

[49] Pan, R., Bagherzadeh, M., Ghaleb, T. A., & Briand, L. (2022). Test case selection and prioritization using machine learning: a systematic literature review. Empirical Software Engineering, 27(2), Article 29. https://doi.org/10.1007/s10664-021-10066-6

[50] Bagherzadeh, M., Kahani, N., & Briand, L. (2022). Reinforcement learning for test case prioritization. IEEE Transactions on Software Engineering, 48(8), 2836-2856. https://doi.org/10.1109/TSE.2021.3070549

[51] Yaraghi, A. S., Bagherzadeh, M., Kahani, N., & Briand, L. C. (2023). Scalable and accurate test case prioritization in continuous integration contexts. IEEE Transactions on Software Engineering, 49(4), 1615-1639. https://doi.org/10.1109/TSE.2022.3184842

[52] Prado Lima, J. A., Mendonça, W. D. F., Vergilio, S. R., & Assunção, W. K. G. (2020). Learning-based prioritization of test cases in continuous integration of highly-configurable software. In 24th ACM International Systems and Software Product Line Conference (SPLC), Volume A (pp. 1-11). ACM. https://doi.org/10.1145/3382025.3414967

The optimization problem can consequently be expressed as selecting and ordering a subset of test cases that maximizes relevant coverage and fault-detection capability while minimizing execution time and cost.[53] The selection process must consider product-line variability, change impact, requirement and feature criticality, historical failures, test duration, and dependencies among test cases. In safety-related contexts, optimization must not remove mandatory verification activities; instead, it should help identify redundant effort and execute the most informative and critical tests as early as possible.

7.2. Optimization Techniques

The methodology combines change-impact analysis with multi-objective optimization. Change-impact analysis uses the traceability links among requirements, features, variants, issues, pull requests, commits, source-code files, and test cases to identify the software areas affected by a modification. Tests associated with unaffected components can be deprioritized or reused where appropriate, while tests linked to modified requirements, features, and components form the initial candidate set for regression testing.

Repositories and issue-tracking data may be represented as a relationship graph to support this analysis. Issue and pull-request clustering can reveal groups of changes associated with the same feature, module, defect category, or development activity. Graph-based analysis can also identify indirect relationships that may not be visible when commits, issues, and files are examined separately. These relationships provide additional evidence for determining the potential scope of a change and selecting relevant tests.

Following impact analysis, multi-objective optimization is applied to balance several competing objectives[54]. These objectives may include maximizing requirement, feature, variant, interaction, and risk coverage while minimizing the number of selected tests, total execution time, and computational cost. Each test case can be represented by attributes such as its coverage contribution, execution duration, historical failure rate, associated feature criticality, and relevance to the current change.

A weighted scoring or Pareto-based approach may be used depending on the testing context. Weighted scoring produces a single priority value by assigning relative importance to the selected objectives. Pareto-based optimization, by contrast, identifies alternative test sets for which no objective can be improved without negatively affecting another. Safety constraints, mandatory tests, configuration dependencies, and minimum coverage thresholds are treated as hard constraints that cannot be violated during optimization.

The optimization process is iterative. Test results, newly detected defects, execution durations, and updated traceability information are fed back into the system after each test cycle. This enables future selections to reflect the most recent behaviour of the product line and allows the optimization strategy to adapt as variants, requirements, and implementation components evolve.

[53] Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability, 22*(2), 67–120. https://doi.org/10.1002/stvr.430
[54] Ferrer, J., Chicano, F., & Alba, E. (2012). Evolutionary algorithms for the multi-objective test data generation problem. *Software: Practice and Experience, 42*(11), 1331–1362. https://doi.org/10.1002/spe.1135

7.3. Test Prioritization Strategies

After the relevant test cases have been selected, they are ordered so that tests with the highest expected value are executed first. Prioritization is particularly important when the available testing window is insufficient to execute the complete candidate set. The proposed approach combines risk-based prioritization, feature criticality, historical failure information, and change relevance[55].

Risk-based prioritization assigns higher priority to tests associated with components whose failure may have a greater technical, operational, safety, or business impact. Risk can be calculated by combining the probability of failure with its expected severity. Tests covering safety-related requirements, core components shared by multiple variants, or areas affected by recent and extensive changes are therefore placed earlier in the execution order.

Feature criticality reflects the importance and usage scope of a feature across the product line. A feature used by many variants may receive a higher priority because a defect could affect a large part of the product family. Variant-specific features may also be prioritized when they introduce new behaviour, complex feature interactions, or additional certification obligations. This prevents prioritization from focusing exclusively on commonly reused functionality.

Historical failure data provides empirical evidence about which tests, modules, and configurations have previously revealed defects. Tests with frequent or recent failures, as well as tests associated with recurring issue clusters, receive a higher priority. However, historical information is combined with change-impact and coverage data to prevent repeatedly failing tests from dominating the execution order while newly introduced risks remain untested.

The final priority score may combine change relevance, risk, feature criticality, historical failure rate, coverage contribution, and execution cost. Short tests with high risk and coverage contributions may be executed early to provide rapid feedback, while longer tests may be scheduled according to their criticality and available resources. The prioritization results should remain explainable by showing the factors contributing to each test's position, allowing engineers to review and override automated decisions when necessary.

8. Conclusion

This document presents the latest technology, findings, and recommended methodologies for product line testing and test optimization in multi-variant systems. The analysis concludes that, given the limitations of current testing approaches and the complexity of product lines, there is a need for strategies that are sensitive to variability, measurable, and scalable.

The examined AI and open-source testing approaches demonstrate the potential of AI-driven techniques for automating and optimizing testing activities, particularly for complex systems. The findings provide a foundation for incorporating appropriate AI and optimization techniques into the proposed framework, integrating them into the OPA4T implementation, and evaluating their effectiveness in subsequent project activities.

Overall, the proposed approaches establish a structured foundation for evaluating and improving test performance across the product line. The next steps will focus on the implementation and integration of

[55] Srikanth, H., Banerjee, S., Williams, L., & Osborne, J. (2014). Towards the prioritization of system test cases. *Software Testing, Verification and Reliability, 24*(4), 320–337. https://doi.org/10.1002/stvr.1500

the identified mechanisms, the definition and validation of the proposed metrics, and the evaluation of the approach in representative use cases.