SOSIS

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

D1.5 – Academic and Technology SoTA Report

Submission date of deliverable: June 30, 2025

Edited by: Hakan Kilinc (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 (IOTIQ, Germany), Hasan Sözer, Leyla Isabalayeva, Gürsel Cesur, Tanay Alpkonur (Özyeğin Üniversity, 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	V1.8
Work package	WP1
Work package leader	Hakan Kilinc (Orion, Türkiye)
Dissemination level	Public
Description *(max 5 lines)*	The goal of D1.5 is to conduct a comprehensive analysis of the current academic and technological aspects to identify scenario-specific requirements and create innovative solutions.

Change Log

Versio n	Date	Authors	Description of changes
1.0	09.04.2025	Hakan Kilinc (Orion)	First template and content are created.
1.1	03.06.2025	Ömercan Devran (Beko)	Variant Quality Analysis Techniques section is updated.
1.2	09.06.2025	Eray Tüzün (Bilkent)	Variant Quality Analysis Techniques section is updated.
1.3	12.06.2025	Hakan Kilinc (Orion)	Test Case Optimization & Prioritization and Test Log Analysis section is updated.
1.4	09.07.2025	Hakan Kilinc (Orion), Hasan Sözer, Leyla Isabalayeva, Gürsel Cesur, Tanay Alpkonur (Özyeğin)	Test Case Optimization & Prioritization and Test Log Analysis section is updated and finalized.
1.5	16.07.2025	Ömercan Devran, Selin Şirin Aslangül (Beko) Eray Tüzün (Bilkent)	Variant Quality Analysis Techniques section is updated and finalized.
1.6	24.07.2025	Tamer Berk (Erste), Ural Sezer (IOTIQ)	Software Quality Analysis for Software Product Line is updated
1.7	29.07.2025	Tamer Berk (Erste)	Software Quality Analysis for Software Product Line is updated and finalized.
1.8	08.08.2025	Hakan Kilinc (Orion)	Document is finalized

Executive Summary

Software product line Optimization for Safety-/mission-critical Industrial Systems requires a multi-layered approach that combines technology, training, and best practices. Using these approaches, organizations can significantly reduce effort and costs. The SOSIS project aims to develop solutions by focusing on these approaches.

"D1.5 Academic and Technology SoTA Report" deliverable is the output of WP1. This task defines a comprehensive analysis of the current academic and technological aspects to create innovative approaches and solutions.

This deliverable D1.5 reports on the technological and academic state of the art on the identified focus areas. The outputs of this deliverable will enable it to initiate the other WPs.

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

The increasing complexity of safety and mission-critical industrial systems has led to a growing need for robust, scalable, and intelligent methods for software product line (SPL) engineering. These systems, used in a wide range of fields from telecommunications to home appliances, must not only meet functional requirements but also ensure high quality and traceability.

The SOSIS project aims to address these challenges by proposing innovative solutions and methodologies that enable efficient variant management, test optimization & analysis, and quality assurance. In this context, this state-of-the-art (SoTA) report provides a comprehensive analysis of the latest technologies in five key focus areas: variable quality analysis, test scenario optimization and test log analysis, software quality assessment for SPLs, AI-driven automation for user requirements, and the application of AI in these areas.

This document identifies the key gaps, opportunities, and practical methods that will shape the development of the SOSIS framework and serves as a foundational knowledge base for other work packages (WPs).

2. Variant Quality Analysis Techniques

The growing complexity of industrial-scale software systems—especially in domains such as telecommunications, home appliances, and safety-critical applications—has necessitated systematic methods for analysing and maintaining the quality of software variants. In large-scale deployments, software must often be customized for regional, regulatory, or client-specific requirements, resulting in a proliferation of variants that share core functionality but diverge in feature sets, configurations, and behaviour. Without a robust mechanism for variant quality analysis, organizations face significant challenges in ensuring consistency, traceability, and maintainability across these software families.

The SOSIS project addresses these challenges through a variant-aware quality analysis framework. This framework is designed to support software product line (SPL) practices by introducing scalable techniques for metric collection, duplication detection, feature-traceability, and intelligent testing. Four industrial use cases (UC1–UC4) serve as anchors for applying and validating these methods in real environments.

This section reviews the state of the art in variant quality analysis techniques and presents how SOSIS advances current practices across the following dimensions:

2.1. Feature Model Analysis

Feature models give an explicit, hierarchical view of a product line's variation points, dating back to Kang *et al. 's FODA* study in 1990[1]. Classic notations focus on *what* can vary; they say little about *how* that variability influences quality once the software ships. Subsequent surveys, therefore, called for ways to attach evidence—tests, metrics, and artefact links—to individual features so that analysts can reason about cost, risk, and certification impact[2]. Riebisch[3] showed that many product-line defects arise when changes in code or requirements are not reflected back into the feature model.

Feature modelling is foundational to SPL engineering, providing a structured view of variation points and configuration options. However, traditional models focus on *defining* variability, not *analysing its quality implications*. In SOSIS (notably in UC1 and UC3), feature models are extended by linking features to software artifacts, test results, and quality metrics.

For example, in the Connecta platform, variants deployed in different countries introduce distinct feature combinations. These are mapped using structured metadata (e.g., Json variant files), allowing teams to trace quality degradation or divergence at the feature level. This approach enhances the expressiveness of feature models, allowing them to inform testing, refactoring, and certification decisions.

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Nowak, S. A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," CMU/SEI-90-TR-21, 1990.
[2] D. Benavides, P. Trinidad, A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, 35 (6), pp. 615-636, 2010
[3] H. Riebisch, "Supporting Evolutionary Development by Feature Models and Traceability Links," Proc. SPLC 2, pp. 85-92, 2002.

2.2. Change Impact Analysis

Bohner & Arnold's seminal text framed change-impact analysis (CIA) as a traceability problem spanning requirements to code [4] . Zimmermann *et al.* [5] later showed that mining evolutionary coupling from version histories could guide safe code changes.

In a previous ITEA project, SmartDelta, we produced one of the most extensive surveys of Change Impact Analysis (CIA) to date, cataloguing more than thirty academic approaches and over a dozen commercial-grade tools, and classifying them into *traceability-based*, *dependency-based* and *hybrid* techniques while also outlining unresolved issues such as "impact explosion," scalability, and context-aware prioritisation[6]. Building on these insights, our recent article "*Enhanced code reviews using pull-request-based change impact analysis*" introduces a combined call-graph and history-mining method that assigns a quantitative risk score to every pull request; two industrial focus-group studies demonstrated that this fine-grained CIA noticeably improves reviewers' ability to anticipate side effects[7].

Understanding the impact of a change in one part of the codebase on other modules or variants is critical for avoiding regressions and ensuring system stability. While traditional change impact tools often analyse dependencies at a structural level (e.g., class or module relationships)[8], SOSIS extends this by incorporating traceability links between commits, issues, features, and variants.

Using enriched Git metadata and issue tracking integration (e.g., Git–Jira links), SOSIS offers bidirectional impact analysis:

- *Forward tracing* identifies which modules, tests, or deployments are likely affected by a code change.
- *Backward tracing* reveals the origins of a defect, identifying the change or variant responsible.

In CI/CD pipelines, this enables gated merges that are variant-aware, reducing late-stage regressions without slowing delivery. This capability is especially impactful in CI/CD environments, where rapid iterations require high confidence in the effect of each change. UC2 demonstrates this in a telecom application server environment with hundreds of services and configurations.

[4] Robert S. Arnold. 1996. Software Change Impact Analysis. IEEE Computer Society Press, Washington, DC, USA.
[5] T. Zimmermann, P. Weibgerber, S. Diehl and A. Zeller, "Mining version histories to guide software changes," Proceedings. 26th International Conference on Software Engineering, Edinburgh, UK, 2004, pp. 563-572, doi: 10.1109/ICSE.2004.1317478.
[6] SmartDelta Consortium, "D4.5 – SmartDelta Quality Optimization and Recommendation Methodology," Project Deliverable, ITEA-3 Project 20023 SmartDelta, Version 1.0, 30 Nov 2024. [Online]. Available: https://itea4.org/project/workpackage/deliverable/document/download/413/D4.5-SmartDelta-Quality-Optimization-and-Recommendation-Methodology.pdf.
[7] Göçmen, I.S., Cezayir, A.S. & Tüzün, E. Enhanced code reviews using pull request-based change impact analysis. Empir Software Eng 30, 64 (2025). https://doi.org/10.1007/s10664-024-10600-2
[8] SmartDelta Consortium, "D4.5 – SmartDelta Quality Optimization and Recommendation Methodology," Project Deliverable, ITEA-3 Project 20023 SmartDelta, Version 1.0, 30 Nov 2024. [Online]. Available: https://itea4.org/project/workpackage/deliverable/document/download/413/D4.5-SmartDelta-Quality-Optimization-and-Recommendation-Methodology.pdf.

2.3. Feature Interactions and Dependency Analysis

Variants frequently trigger *feature interactions*—unexpected behaviours that arise only when particular features, timing constraints, or configuration parameters are combined. Calder *et al.* [9] first characterised this phenomenon in telecommunication services and showed why purely formal models cannot keep pace with industrial variability. Subsequent SPL studies demonstrated that a relatively small set of interacting features can dominate failure rates and performance regressions[10]. These findings motivated systematic detection and mitigation techniques.

Combinatorial interaction testing (CIT)

CIT generates a test suite in which every *t-wise* combination of features (typically *t = 2* or *3*) appears at least once. Kuhn *et al.* quantified how most software faults are triggered by the interaction of a few parameters and formalised the t-wise principle[11]. Tools such as NIST's ACTS, based on the IPOG algorithm[12], automatically construct minimal t-wise suites and have been adopted in avionics, telecom, and medical software. When full t-wise coverage is still too large, adaptive and similarity-based prioritisation heuristics rank the combinations most likely to reveal defects, cutting execution time by an order of magnitude while preserving fault-detection power[13].

Even exhaustive CIT cannot expose all timing- or environment-dependent interactions. Large-scale systems, therefore, mine execution logs to discover recurring failure signatures linked to specific feature sets. Xu *et al.* pioneered log-sequence mining for problem detection in distributed clusters [14]; later work refined anomaly- and failure-prediction models for modern cloud stacks[15].

SOSIS addresses these challenges through combinatorial test case generation and prioritization, supported by runtime analysis of test logs and defect trends. In UC2, for example, parameterized tests are generated to explore possible combinations of service configurations (e.g., SIP, IMS, PBX deployments), identifying high-risk intersections. Analysis of historical logs further allows for pattern-based detection of failures tied to specific combinations, supporting proactive test planning.

This data-driven approach enables teams to shift from reactive bug-fixing to preventive quality assurance, especially in safety-/mission-critical contexts.

[9] M. Calder, M. Kolberg, E. Magill, S. Reiff-Marganiec, "Feature interaction: A critical review and considered forecast," Computer Networks, 41 (1), 115-141, 2003.
[10] S. Apel, T. Thüm, C. Kästner, "Detecting reliable feature interactions in product line implementations," Proc. SPLC, 165-174, 2012.
[11] D. R. Kuhn, D. R. Wallace, A. M. Gallo Jr., "Software fault interactions and implications for software testing," IEEE TSE, 30 (6), 418-421, 2004.
[12] Y. Lei, R. N. Kacker, D. R. Kuhn, V. Okun, J. Lawrence, "IPOG: A general strategy for *t-*way software testing," Proc. ISSRE, 549-558, 2007.
[13] M. Chen, P. McMinn, M. Stevenson, "Adaptive combinatorial testing for effective fault detection," IEEE TSE, 45 (4), 382-412, 2019.
[14] W. Xu, L. Huang, A. Fox, D. Patterson, M. Jordan, "Detecting large-scale system problems by mining console logs," Proc. SOSP, 117-132, 2009.
[15] D. Duan, J. Li, G. Xu, S. Lu, M. Chen, "Understanding and analysing application logs for system reliability," Proc. OSDI, 1-16, 2014.

2.4. Machine Learning and Artificial Intelligence Applications

Highly configurable product lines generate enormous streams of evidence—feature selections, build metrics, test outcomes, and field logs. Manual inspection no longer scales, so recent work relies on machine learning (ML) and artificial intelligence (AI) techniques that *learn* how variant choices affect quality attributes.

- Influence modelling. Performance-influence models learned from a *small, carefully sampled* set of configurations can predict how individual options—and their interactions—affect throughput and latency [16].

- Cost-efficient sampling. Progressive and projective sampling strategies reduce the number of measured variants needed to train accurate predictors, trading laboratory effort for model precision [17].

- Sampling/learning interplay. Empirical guidelines show that the *choice* of sampling method conditions ML success; some learners thrive on random samples, others on stratified or interaction-covering sets[18].

- Adversarial configurations. Adversarial ML reveals blind spots in classifiers that label variants as "good" or "defective," underscoring the need for robustness checks when ML guards product-line quality gates[19].

3. Test Case Optimization & Prioritization and Test Log Analysis

3.1. Introduction

Software testing is an important activity in the software development lifecycle that ensures product quality, reliability, and user satisfaction. However, as modern software systems grow in complexity and scale, the cost and effort required for comprehensive testing increases significantly. In this context, two important topics are emerging to address these challenges: test case optimization and prioritization, and test log analysis.

Test case optimization and prioritization aim to reduce the size of test suites and execute the most valuable tests early. These techniques aim to improve defect detection rates, minimize test time, and optimize resource utilization, especially in continuous integration/continuous delivery (CI/CD) environments. Test engineers and researchers have proposed strategies ranging from static code-

[16] N. Siegmund, A. Grebhahn, S. Apel, C. Kästner, "Performance-Influence Models for Highly Configurable Systems," *Proc. ESEC/FSE 2015*, pp. 284–294.
[17] A. Sarkar, J. Guo, N. Siegmund, S. Apel, K. Czarnecki, "Cost-Efficient Sampling for Performance Prediction of Configurable Systems," *Proc. ASE 2015*, pp. 342–352.
[18] C. Kaltenecker, A. Grebhahn, N. Siegmund, S. Apel, "The Interplay of Sampling and Machine Learning for Software Performance Prediction," *IEEE Software*, 37 (3), 46–53, 2020.
[19] P. Temple, M. Acher, G. Perrouin, B. Biggio, J.-M. Jézéquel, F. Roli, "Towards Quality Assurance of Software Product Lines with Adversarial Configurations," *Proc. SPLC 2019*, pp. 177–181.

based heuristics to dynamic, search-based, and machine learning approaches to identify and plan the most effective test cases.

In modern software systems, large-scale test suites can often lead to high execution costs and long runtimes, making efficient regression testing a critical challenge. Regression testing[20] is an important procedure in modern software development lifecycle, especially in large-scale and continuously improving systems. It ensures that new changes do not unintentionally disrupt the preexisting functionalities. As test suites' size and complexity grow, with new features, bug fixes, and performance improvements added, the execution of entire tests becomes more expensive in terms of time and computational resources. For example, during the development process of a complex system, Windows 8.1, which lasted approximately 11 months, more than 30 million tests were executed[21]. This inefficiency makes it impractical and costly to run regression suites entirely and frequently. Thus, creating the need for test case optimization and prioritization techniques that can ensure effective testing while reducing execution costs.

Test log analysis has become a critical component of modern software testing, especially in large-scale distributed and cloud systems. Logs generated during test execution provide a wealth of information that can reveal failure patterns, performance issues, and hidden anomalies. However, due to the large volume and unstructured nature of these logs, manual analysis is time-consuming and error-prone. As a result, recent developments leverage log parsing, generative AI, natural language processing (NLP), and deep learning to automate log interpretation and extract actionable insights.

3.2. Test Case Optimization & Prioritization

Test case prioritization is a critical step in improving the efficiency of software testing processes and optimizing resource utilization. In the fast-paced world of software engineering, time is the most precious commodity, and testing, one of the most critical phases of the software development lifecycle, can be costly in terms of time and resources. With the pressure to deliver high-quality software faster, test case prioritization (TCP) becomes critical to the software testing process, especially in large and complex software, as it is not possible to run all test cases with the same priority. With TCP, the defect catch rate can be increased, and significant savings in test time can be achieved[22].

TCP means prioritizing which test cases to run based on their importance, functionality, and potential impact on the software. Prioritizing test cases and executing them in the most efficient order is essential to detect bugs in your software at the earliest possible stage of the development process. However, there are many parameters in test case prioritization; it may be necessary to determine the order in which test cases should be executed based on **requirement, risk, importance, coverage, release, cost, or defect probability. These prioritization techniques vary according to the project.**

[20] E. Engström and P. Runeson, Industry Practices of Regression Testing – An Empirical Investigation of 41 Companies, Lund University, Sweden, 2010. [Online]. Available: https://fileadmin.cs.lth.se/cs/Personal/Emelie_Engstrom/Papers/Regression_testing_practices.pdf

[21] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The Art of Testing Less without Sacrificing Quality." Accessed: Jun. 02, 2025. [Online]. Available: https://www.michaelagreiler.com/wp-content/uploads/2019/02/The-Artof-Testing-Less-without-Sacrificing-Quality.pdf

[22] H. Son, "Test case prioritization techniques and metrics," 2023. [Online]. Available: "https://www.testrail.com/blog/test-case-prioritization/"

- **Requirements-Based Prioritization**[23]**:** Test cases are prioritized based on the importance or risk level of the requirement. Requirements with higher criticality or business value are tested first.
- **Risk-Based Prioritization**[24]**:** Test cases are prioritized based on the perceived risk of failure associated with the features.
- **Code Complexity Based Prioritization:** Metrics such as cyclomatic complexity or call graph depth help identify error-prone code regions, and test cases are prioritized based on these metrics.
- **Coverage-Based Prioritization:** Test cases that cover more (or unique) code are prioritized.
- **History-Based Prioritization:** Historical execution data (e.g., failed test cases, failed tests) is used for prioritization.
- **Time Sensitive Prioritization:** Prioritization considers test execution time to balance defect detection and runtime efficiency.

In addition, there are machine learning approaches such as reinforcement learning[25] and hybrid approaches, which we detail below.

TCP is a long-standing topic in the software testing literature. Machine learning and artificial intelligence-driven approaches have made remarkable advances in TCP[26],[27],[28]. Recently, the usability of large language models (LLMs) in software engineering tasks such as bug detection, test case generation, requirements analysis, code completion, and refactoring has increased significantly[29],[30]. Thanks to their natural language understanding capabilities, LLMs are becoming supporting actors in decision-making processes by test case generation, prioritization, optimization, analysis of test logs, and interpretation of defect reports or test descriptions[31],[32]. In addition, the

[23] R. Krishnamoorthi and S. A. Sahaaya Arul Mary. 2009. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. Inf. Softw. Technol. 51, 4 (April, 2009), 799–808.
[24] Bogdan Korel, George Koutsogiannakis, and Luay H. Tahat. 2007. Model-based test prioritization heuristic methods and their evaluation. In Proceedings of the 3rd international workshop on Advances in model-based testing (A-MOST '07). Association for Computing Machinery, New York, NY, USA, 34–43.
[25] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2022. Reinforcement Learning for Test Case Prioritization. IEEE Trans. Softw. Eng. 48, 8 (Aug. 2022), 2836–2856.
[26] V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. P. Guimaraes, “Machine learning applied to software testing: A systematic mapping study,” IEEE Transactions on Reliability, vol. 68, no. 3, pp. 1189–1212, 2019.
[27] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, “Reinforcement learning for automatic test case prioritization and selection in continuous integration,” in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 12–22.
[28] A. Rafael Lenz, A. Pozo, and S. Regina Vergilio, “Linking software testing results with a machine learning approach,” Engineering Applications of Artificial Intelligence, vol. 26, no. 5, pp. 1631–1640, 2013.
[29] A. E. Hassan, D. Lin, G. K. Rajbahadur, K. Gallaba, F. R. Cogo, B. Chen, H. Zhang, K. Thangarajah, G. Oliva, J. J. Lin, W. M. Abdullah, and Z. M. J. Jiang, “Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy fmware,” in Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 294–305.
[30] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” ACM Trans. Softw. Eng. Methodol., vol. 33, no. 8, Dec. 2024. [Online]. Available: https://doi.org/10.1145/3695988
[31] M. S. Bouafif, M. Hamdaqa, and E. Zulkoski, “Primg : Efficient llm-driven test generation using mutant prioritization,” 2025. [Online]. Available: https://arxiv.org/abs/2505.05584
[32] I. Zosimadis and I. Stamelos, “Llm-enhanced test case prioritization for complex software systems,” in Proceedings of the 28th Pan-Hellenic Conference on Progress in Computing and Informatics, ser. PCI ’24. New York, NY, USA: Association for Computing Machinery, 2025, p. 46–50. [Online]. Available: https://doi.org/10.1145/3716554.3716561

performance of models such as ChatGPT in the context of software testing has been examined in some studies[33],[34].

In the test case optimization and prioritization techniques, there are two approaches: multi-criteria and single-criteria approaches. In the multi-criteria test case optimization and prioritization approaches, where the goal is to minimize the number of test cases, select a smaller subset of tests that still satisfy the required coverage criteria, and prioritize reorder test cases based on multiple factors such as statement coverage and execution time. The single-criteria approaches, such as maximizing only the coverage or minimizing the number of tests, usually achieve results far from the optimal due to leaving out fault detection and runtime from the calculations[35]. There are former studies[36] on this topic with optimization methods[37],[38] such as Integer Linear Programming (ILP) and Nonlinear Programming (NLP). However, these approaches are usually impractical for real-time or large-scale use due to the complex calculations involved.

Although integer or nonlinear programming, which are solutions focused on optimization, can calculate global optimal solutions also with multiple criteria, in terms of computation, they are too complex and are not suitable for use with large test suites without making heavy alterations to the framework. For example, in the Nemo framework[36], the multi-criteria test suite minimization is constructed as an integer nonlinear (INP) and then changed into the linear form so ILP solvers can be used. However, our project uses a greedy, heuristic-based approach that executes in O(n2) time and ensures modular design that allows for easier integration and adaptation to future improvements.

Nemo framework[36] is one of the most well-known papers in the area of multi-criteria test case optimization. The Nemo framework models the test prioritization and minimization problem as a nonlinear optimization method. Nemo takes into account a number of factors, such as fault detection probability, execution time, and statement coverage, with the goal of ultimately arriving at an optimal ordering of the test cases. The use of solvers and the nonlinear inner objective function makes Nemo cumbersome and adds practical difficulties in utilizing the framework with multiple software development processes, such as continuous integration and regression testing. So, inspired by some of the objectives of Nemo, our proposed project is a modular framework in Java that implements greedy heuristics to achieve a multi-criteria solution for the test case minimization and prioritization task, without a solver. For we do not intend to solve a global optimization problem, but to offer a lightweight, effective solution to identifying a suitable balance of fault detection and runtime, as evidenced by the APFDc scores.

[33] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," Proc. ACM Softw. Eng., vol. 1, no. FSE, Jul. 2024.
[34] H.-F. Chang and M. Shokrolah Shirazi, "A systematic approach for assessing large language models' test case generation capability," Software, vol. 4, no. 1, 2025.
[35] G. Rothermel, Mary Jean Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," Nov. 2002, doi: https://doi.org/10.1109/icsm.1998.738487.
[36] J. W. Lin, J. Garcia, R. Jabbarvand, and S. Malek, "Nemo: Multi-criteria test-suite minimization with integer nonlinear programming," in Proc. 40th Int. Conf. Softw. Eng. (ICSE), Gothenburg, Sweden, May 2018, pp. 1033–1044.
[37] Y. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in Proc. Int. Symp. Softw. Test. Anal. (ISSTA), 2007, pp. 140–150.
[38] L. Zhang, J. H. R. Jiang, and S. Zhang, "Multi-objective test case prioritization with preference learning," IEEE Trans. Softw. Eng., vol. 36, no. 3, pp. 345–362, 2009.

Hsu and Orso's MINTS tool[39] provides a general framework for test-suite minimization that is capable of optimizing multiple criteria by using Integer Linear Programming (ILP). MINTS formulates minimization problems as ILP and, when solved to get optimal results, produces minimal test subsets that satisfy all coverage constraints. Their results demonstrate that modern ILP solvers can efficiently handle a range of multi-criteria minimization demands, producing exact solutions where single-criterion approaches aren't as effective.

Yoo and Harman's[40] survey provides common techniques that are categorized for test selection, test minimization, and test prioritization, and highlights the rising cost of regression testing. They differentiate between test minimization (removing redundant tests), test selection (choosing tests impacted by recent changes), and test prioritization (reordering tests to detect faults earlier). The paper identifies some challenges that occur even today, key techniques in each case, and identifies open problems and potential research.

Di Nardo et al.[41] conduct a case study investigating coverage-based regression testing strategies on a real-world industrial software system. This case study compares four test prioritization methods, one test selection strategy, one minimization strategy, and a hybrid selection-minimization variant on different code coverage criteria. The authors present evidence that fine-grained prioritizing based on added coverage provides significantly better fault detection rates. However, the use of change information did not show measurable improvement. The authors also claim that selection only would not produce significant savings in execution cost (<2%) while minimizing some run-time based on detailed coverage, saving up to 79.5% of run-time and retaining over 70% fault detection capabilities, which demonstrates a clear trade-off with cost and effectiveness. The hybrid condition did not outperform classical minimization.

Jabbarvand et al.[42] deal with minimizing test suites for Android applications with a strong focus on energy consumption – a significant concern in mobile environments. They define a new energy coverage metric for the purpose of locating the "energy-greedy" locations in code, and so, minimize the number of tests needed to identify energy-related bugs. They use two complementary algorithms, one being a computationally complex but optimal ILP formulation and the other being a faster, near-optimal greedy alternative, just like in our project. Their experimental results, based on real-world Android test suites, reported an average reduction of 84% of tests but were still able to identify the majority of energy faults.

Özener and Sözer[43] suggest a different linear formulation for the multi-criteria test suite minimization problem, which overcomes the limitations of existing models. According to them, previous literature utilizing heuristic or ILP-based methods is typically focused on one or two objectives and usually produces suboptimal solutions due to overlaps in coverage or fault detection from test cases. In the

[39] H.-Y. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," Proc. 31st Int. Conf. Softw. Eng. (ICSE), 2009, pp. 419–429.
[40] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," Softw. Test. Verif. Reliab., vol. 22, no. 2, pp. 67–120, 2012.
[41] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system," Softw. Test. Verif. Reliab., vol. 25, no. 4, pp. 371–396, 2015.
[42] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware Test-suite Minimization for Android Apps," Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA), 2016, pp. 425–436.
[43] Ö. Ö. Özener and H. Sözer, "An effective formulation of the multi-criteria test suite minimization problem," J. Syst. Softw., vol. 168, 110632, 2020.

case of Nemo, the authors do indeed avoid some overlap from coverage and fault detection, but rely on a sophisticated non-linear model, which significantly increases its computational complexity and risk of sub-optimality. To provide an alternative method for solving multi-criteria test suite minimization, Özener and Sözer reformulated the problem into a new linear model that considers coverage, fault-detection capability, and overall test duration, all at once. They verified this model against recent literature, including Nemo, under the same objective function, and report experimental results that prove their formulation mostly returned better (or at least equivalent) results under the linear constraint, while needing much less CPU time.

Most recently, Pan, Ghaleb, and Briand propose a black-box test suite reduction process called ATM[44], which is a similarity-based test suite reduction method that they presented at ICSE 2023. ATM uses an abstract syntax tree (AST) to represent test cases and to remove redundant test cases in a test suite. ATM then removes redundant test cases using four different tree-based similarity metrics combined with evolutionary search methods, such as genetic algorithms. ATM was evaluated on the DEFECTS4J dataset and has a mean fault detection level of around 82% and does so in a reasonable time (~1.2 hours). The best-performing configuration uses multiple AST similarity measures, which gave an approximately 80% fault sensitivity level. The Nemo study directly referenced ATM to show that effective reductions can still be done even if coverage information is not drawn from the source code. In cases where white-box metrics like coverage are more costly to obtain, similarity-based methods such as ATM may yield a better ratio of effect to time. ATM also directly notes in its paper that the high cost of running test suites with very large numbers of test suites is a type of scalability issue.

[44] R. Pan, T. A. Ghaleb, and L. C. Briand, "ATM: Black-box test case minimization based on test code similarity and evolutionary search," Proc. 45th Int. Conf. Softw. Eng. (ICSE), 2023, pp. 1700–1711.

3.3. Test Log Analysis

As software systems increase in size and complexity, the number of test cases and the volume and variety of logs generated during their execution grow exponentially. These logs contain invaluable information for understanding system behaviour, diagnosing failures, and improving test effectiveness. However, the unstructured nature of logs, their size, and the dynamic environments in which they are generated pose significant challenges for manual review. Moreover, a holistic view, combined with prioritization approaches, is useful for quickly identifying the root cause of many problems.

Test logs are analysed using parsing, pattern mining, anomaly detection, and machine learning-based modelling. The challenges and state-of-the-art techniques that these methods bring should not be overlooked.

Performing test log analysis has many goals. These goals range from automated detection of incorrect system behaviour to root cause analysis, from capturing control and data flow between components during test execution to analysing behavioural changes between versions and builds. There are also challenges to overcome to achieve the goals. There are challenges such as semi-structured or unstructured logs, different formats of logs, incomplete or noisy logs, hiding meaningful patterns due to high data volume[45].

To perform the analysis, it is necessary to transform the raw log data into structured representations using log parsers. Log parsers such as Drain[46], Spell[47], and LogMine[48] extract event patterns from log lines, abstracting variables while preserving semantic structure. Parsed logs enable goal-oriented tasks such as anomaly detection and root cause tracing to be performed.

After the logs are parsed, different log analysis techniques are used.

- Rule-based correlation and regular expression matching: It is a simple log analysis and involves predefined rules or regular expressions to search for known error patterns. Usually, domain-specific heuristics such as "login failure → check database → retry request" are used.
- Machine learning approaches: Supervised and unsupervised ML models are used to detect patterns, sequence modeling, clustering logs, or classifying anomalies.
- Deep learning approaches[26]: Recurrent models such as RNN and LSTM, autoencoders, transformer-based models such as LogBERT[49] are used to model sequential patterns and semantic relationships in logs.

In recent years, generative AI and large language model approaches[50] have become popular.

3.4. Evaluation Metrics

Several metrics are used to evaluate the effectiveness of optimization and prioritization strategies:

[45] Max Landauer, Sebastian Onder, Florian Skopik, Markus Wurzenberger, Deep learning for anomaly detection in log data: A survey, Machine Learning with Applications, Volume 12, 2023.
[46] Pinjia He; Jieming Zhu; Zibin Zheng; Michael R. Lyu, Drain: An Online Log Parsing Approach with Fixed Depth Tree, 2017 IEEE International Conference on Web Services (ICWS), 25-30 June 2017

- APFD (Average Percentage of Fault Detection): Measures how quickly defects are detected.
- Code/Requirement Coverage: The percentage of lines of code or requirements implemented by test cases.
- Execution Time: Total time taken to run selected or prioritized test cases.
- Mutation Score: Indicates the ability of a test suite to detect artificially injected bugs (mutants).

For test log analysis, log anomaly detection, and log classification, precision, recall, and F1 score as standard metrics are used:

- Precision: Proportion of detected anomalies that are truly anomalous.
- Recall: Proportion of actual anomalies that were detected.
- F1 Score: Harmonic mean of precision and recall.

Average Percentage of Fault Detected (APFD)

To calculate the effectiveness of a prioritization technique, the Average Percentage of Faults Detected (APFD)[51] metric can be used. APFD measures how quickly faults are detected over the course of test execution. It calculates the area under the curve that plots "cumulative percentage of faults detected" on one axis, and "fraction of the test suite executed" on the other. A higher APFD score shows that faults are detected earlier in the testing process, which is the desirable outcome. However, since the standard APFD does not consider differences in test execution cost, we employ the cost-cognizant version, known as APFDc[52], which modifies the formula to incorporate the runtime and fault weight of each test. This adjustment is crucial in scenarios where some tests are significantly more expensive than others.

Average Percentage of Fault Detected (APFD) is defined as follows:

- m: Total number of faults.
- n: Total number of test cases.
- T = Test Suite containing n test cases
- F = Set of m faults revealed by T

TFi = The first test case in ordering $T0$ of T, which reveals fault i.

[47] Min Du; Feifei Li, Spell: Streaming Parsing of System Event Logs, 2016 IEEE 16th International Conference on Data Mining (ICDM), 12-15 December 2016
[48] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast Pattern Recognition for Log Analytics. In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM '16)
[49] Haixuan Guo, Shuhan Yuan, Xintao Wu,"LogBERT: Log Anomaly Detection via BERT." 2021, https://arxiv.org/abs/2103.04475.
[50] Karlsen, E., Luo, X., Zincir-Heywood, N. *et al.* Benchmarking Large Language Models for Log Analysis, Security, and Interpretation. *J Netw Syst Manage* 32, 59 (2024).
[51] Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," IEEE Trans. Softw. Eng., vol. 28, no. 2, pp. 159–182, Feb. 2002.
[52] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," IEEE Trans. Softw. Eng., vol. 27, no. 10, pp. 929–948, 2001.

$$APFD = 1 - \frac{TF1 + TF2 + \cdots + TFm}{nm} + \frac{1}{2n}$$

Higher APFD value, closer to 1, indicates that the faults are detected faster in the test execution process. This standard version assumes that the test cost is the same, so each test is counted the same, without paying attention to its runtime. However, in real-world issues, the runtimes for each test differ, thus the original APFD metric can be an unreliable evaluation metric. For example, a test that takes 30 minutes and detects multiple faults would be considered the same as a test that runs for only a second.

To avoid unreliable calculation, the cost-cognizant version of the original APFD metric can be chosen: the APFDc metric. It simply takes into account the test runtime and fault weight when calculating the score.

Average Percentage of Fault Detected with Cost (APFDc) is defined as follows:

The (cost-cognizant) weighted average percentage of faults detected during the execution of test suite T' is given by the equation:

- *m:* Total number of faults.
- *n:* Total number of test cases.
- *fi:* Fault weight, the importance or severity of fault i.
- *TFi:* Test failure index.
- *tj:* Test execution time, or cost of test j.

$$APFDc = \frac{\sum_{i=1}^{m} \left(f_i \times \left(\sum_{j=TF_i}^{n} t_f - \frac{1}{2} t_{TF_i} \right) \right)}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i}$$

When all the fault weights are the same, the APFDc function becomes a sum of detection times relative to the total test runtime. A higher APFDc, again, closer to 1, means earlier detection of critical faults per unit time.

To better understand the function of these metrics, below is an example:

Assume we have three tests T1, T2, T3 with runtimes 2, 1, 3 seconds, respectively.

- T1 detects F1, F2;
- T2 detects F2, F3;
- T3 detects F1, F3.

If we choose to order tests in T2, T1, T3 order, then

- F1 is first found by T1 at 2+1=3 seconds;
- F2 is first found by T1 at 1 second;
- F3 is first found by T2 at 1 second.

If we assume all fault weights to be 1, we get

$\sum_{i=1}^{3} f_i TFi = 3 + 1 + 1 = 5, \ \sum_{j=1}^{3} t_j = 2 + 1 + 3 = 6.$

So, the final

$APFDc = 1 - \frac{5}{6 \times 3} + \frac{1}{2 \times 6} = 0.8056.$

The classic APFD would be

$APFD = 1 - \frac{TF_1 + TF_2 + TF_3}{3 \times 3} + \frac{1}{2 \times 3} = 0.8333$

As it can be seen from the results of a small example, the classic APFD score ignores the fact that T3 is more expensive than the others.

3.5. Summary

In this section, we present a comprehensive state-of-the-art survey on test case optimization and prioritization and test log analysis, two critical components of modern software testing. As software systems increase in complexity and scale, effective and intelligent test management has become essential to ensure quality without compromising agility.

We presented a detailed literature survey on test case prioritization, reviewing static and dynamic techniques. We then examined the evolution of test log analysis from rule-based heuristics to advanced machine learning and deep learning models, including log parsing, anomaly detection, and sequence modelling. We reviewed evaluation metrics for these two critical components and presented a detailed review of the APFD metric.

We plan to focus more on Generative AI and LLM models, which are quite new in the project. All the methods examined will pave the way for a promising transition from reactive diagnostics to proactive, context-sensitive and adaptive test management, especially with respect to CI/CD and DevOps environments.

4. Software Quality Analysis for Software Product Line

Software Product Line Engineering (SPLE) enables the systematic reuse of software assets by managing variability across product families. While this paradigm increases productivity and reduces time-to-market, it also introduces new challenges for software quality assurance. Traditional quality analysis techniques fall short in SPL contexts due to the combinatorial explosion of variants and the intricate interdependencies between features, code, and configuration artifacts[53].

Evaluating the quality of a software product line is more complex than analyzing a single software system. Each product variant may differ in its features, behavior, and performance. Therefore, quality analysis must go beyond general-purpose software metrics and focus on specific aspects of variability and evolution[54].

To address this, software quality analysis is broken down into several focused areas. These include feature quality analysis, which examines the correctness and consistency of features; code quality analysis, which looks at the structure and maintainability of variant-specific code; and code

[53] Elbaum, A. G., Malishevsky, A. G., & Rothermel, G. *Test case prioritization: A family of empirical studies*. *IEEE Trans. Softw. Eng.*, 28(2), 159–182, Feb. 2002.
[54] Elbaum, A. G., Malishevsky, A. G., & Rothermel, G. *Test case prioritization: A family of empirical studies*. *IEEE Trans. Softw. Eng.*, 28(2), 159–182, Feb. 2002.

duplication detection, which identifies repeated or redundant code introduced by variability mechanisms[55].In addition, commit and contribution analysis helps track developer activity and collaboration patterns, which affect long-term maintainability[55].

It is also important to evaluate the overall quality of each variant, especially in large-scale product lines where hundreds of configurations may exist. Finally, machine learning (ML) and artificial intelligence (AI) techniques are increasingly being applied to support tasks such as defect prediction, test prioritization, and variant recommendation[55].

This section reviews the current research and techniques in these sub-areas of software quality analysis, aiming to provide a comprehensive view of how to maintain high quality across evolving and diverse product lines.

4.1. Feature Quality Analysis

Feature quality analysis is an essential component within Software Product Line (SPL) engineering, focusing specifically on evaluating the quality and effectiveness of individual software features. As software systems expand and diversify, variations in the implementation of features can significantly affect the overall quality, reliability, and performance of software products. An accurate and detailed evaluation of feature quality helps identify problematic areas and supports informed decisions about software design and development[56].

In the SOSIS project, feature quality analysis includes several approaches to systematically assess the quality of features within the SPL environment:

- **Feature Completeness Analysis**: Evaluates whether a feature meets all specified functional and non-functional requirements across different software variants, ensuring no crucial aspects are overlooked[56].
- **Feature Stability and Reliability Assessment**: Measures how consistently and predictably a feature behaves across various deployments, configurations, and usage scenarios, identifying potential stability concerns[56].
- **Feature Performance Evaluation**: Quantitatively assesses feature performance metrics such as response times, resource usage, and scalability, pinpointing performance bottlenecks and optimization opportunities[56].
- **Feature Traceability and Documentation Quality**: Ensures comprehensive documentation and clear traceability between features, requirements, implementation, and test cases, simplifying maintenance and compliance activities[57].
- **Feature Security Analysis**: Identifies vulnerabilities and evaluates security compliance at the feature level, minimizing the risks associated with each feature in the product line[58].

[55] Karlsen, E., Luo, X., Zincir-Heywood, N. et al. *Benchmarking Large Language Models for Log Analysis, Security, and Interpretation*. *J Netw Syst Manage* 32, 59 (2024)
[56] Riebisch, M. *Supporting Evolutionary Development by Feature Models and Traceability Links*. In *Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems* (ECBS'04), 2004.
[57] Riebisch, M. *Supporting Evolutionary Development by Feature Models and Traceability Links*. In *Proceedings of ECBS'04*, IEEE, 2004
[58] Guo, J., Siegmund, N., Apel, S., & Kästner, C. *Learning to Predict Performance Properties of Software Configurations*. *Empirical Software Engineering*, 20(3), 2015.

- **Cross-Feature Interaction Analysis**: Examines how individual features interact with one another within different software variants, identifying negative interactions that could compromise functionality, performance, or security[59].

Additionally, SOSIS integrates these analyses with automated tools and machine learning techniques. Historical data related to feature usage, defect frequency, and variant-specific performance is utilized to generate predictive models, thus enabling proactive identification and resolution of feature-related quality issues.

4.2. Code Quality Analysis

Code quality analysis is a fundamental practice in software product line (SPL) development, essential for maintaining consistency, reducing complexity, and ensuring the reliability of software variants. Effective code quality analysis ensures the detection and correction of common software issues such as:

- **Coding Errors**: Logical or syntactical mistakes within the code that prevent software from functioning as intended[60].
- **Poor Readability**: Difficulty in understanding the code due to improper naming conventions, insufficient documentation, or overly complex logic, hindering collaboration and maintainability[60].
- **Security Vulnerabilities**: Weaknesses or flaws in software that could be exploited by malicious actors, potentially leading to unauthorized access or data breaches[59].
- **Maintainability Challenges**: Issues arising from overly complex or rigid code structures, making modifications, enhancements, and debugging more challenging and time-consuming[61].
- **Inefficient Programming Practices**: Suboptimal coding patterns and algorithms that degrade performance, waste computational resources, and increase operational costs.
- **Code Duplication (Redundancy)**: Repeated code segments that increase the software complexity, risk of inconsistencies, and maintenance efforts[62].
- **Inadequate Error Handling**: Failure to properly handle exceptions, errors, or unexpected conditions, leading to software crashes, unresponsive behaviors, or undefined states[63].
- **Testability Issues**: Writing code that is difficult or impossible to test due to tight coupling, insufficient modularity, or lack of clear interfaces, thus increasing the risk of undetected defects[64].

In the SOSIS project, code quality analysis integrates static code analysis tools such as SonarQube, PMD, Checkstyle, and FindBugs. These tools enable the automated examination of source code for

[59]Nguyen, C. D., & Massacci, F. *Security Metrics for Software Systems: A Survey and Classification*. *Computing Surveys (CSUR)*, 50(4), 2017..
[60]McConnell, S. *Code Complete: A Practical Handbook of Software Construction*, 2nd Edition. Microsoft Press, 2004.
[61]ISO/IEC 25010:2011. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*.
[62]Spinellis, D. *Code Quality: The Open Source Perspective*. Addison-Wesley, 2006.
[63]Han, J., et al. *Understanding and Classifying Software Exception Handling Code*. *Proceedings of the ICSE*, 2012.
[64]ISO/IEC 25010:2011. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*.

defects, compliance with coding standards, and potential improvements in quality[65]. SOSIS further enriches standard static analysis practices by combining analysis results with variant-specific metadata. Through integration with continuous integration/continuous delivery (CI/CD) environments and issue tracking tools such as Jira, quality indicators can be tracked historically and linked to specific software variants and feature implementations[66].

Additionally, machine learning-based techniques are employed within the SOSIS framework to analyse historical data from code repositories, identifying patterns that correlate specific coding practices with software defects and variant-specific quality degradation. This predictive capability facilitates proactive quality management, assisting developers in prioritizing refactoring efforts and quality improvements based on variant-specific contexts and historical trends[67].

[65]Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., & Penix, J. *Using FindBugs on Production Software*. *Communications of the ACM*, 50(2), 34–40, 2007.
[66]Zaidman, A., van Rompaey, B., Demeyer, S., & van Deursen, A. *Mining Software Repositories to Study Co-Evolution of Production & Test Code*. *Software Quality Journal*, 2008.
[67]Guo, J., Siegmund, N., Apel, S., & Kästner, C. *Learning to Predict Performance Properties of Software Configurations*. *Empirical Software Engineering*, 20(3), 2015.

4.3. Code Duplication Detection

Code duplication is a common symptom of unmanaged variability, especially in organizations using copy-and-modify strategies to deploy software variants. Such duplications significantly degrade software quality by increasing maintenance effort, causing inconsistencies, and complicating bug fixing and enhancement processes. Traditional static analysis tools (e.g., PMD, SonarQube) identify duplicated logic but lack awareness of which variant a duplication belongs to, making it difficult to differentiate intentional reuse from accidental duplication[68].

In the SOSIS project, duplication detection is enhanced through variant context integration. By coupling static code analysis results with version control metadata, Jira work items, and structured variant-specific metadata (e.g., JSON variant definition files), the framework distinguishes variant-specific duplication from intended shared code reuse. SOSIS employs comprehensive duplication detection techniques, identifying exact (Type-1), parameterized (Type-2), and near-miss (Type-3) clones, thus precisely categorizing different duplication types[69]:

- **Exact Clones (Type-1)**: Completely identical code segments, differing only in minor aspects like whitespace or comments.
- **Parameterized Clones (Type-2)**: Segments with identical logic but minor syntactic differences, such as renamed variables or methods.
- **Near-Miss Clones (Type-3)**: Similar code fragments with inserted, modified, or deleted lines, representing partial duplication.

Integration with version control systems and issue trackers enables historical traceability of duplications. By associating duplication detection outcomes with specific commits, pull requests, and Jira issues, SOSIS facilitates effective identification of duplication sources, timing, and contributors, allowing targeted remediation[69].

Additionally, SOSIS utilizes visualization and reporting tools such as heatmaps and duplication graphs, enabling teams to intuitively grasp the extent, frequency, and impact of duplications across software variants. Advanced machine learning algorithms analyze historical duplication data and predict code regions susceptible to future duplication, proactively suggesting preventive measures[70].

This multi-faceted approach enables organizations to focus their refactoring and maintenance efforts precisely on duplicated modules unnecessarily diverging between variants.

Consequently, SOSIS significantly improves software maintainability, reliability, and overall product-line quality, without compromising variant-specific behavior.

4.4. Commit and Contribution Analysis

Commit and contribution analysis involves systematically examining software repository commits, developer contributions, and historical code changes to ensure software product line (SPL) quality.

[68]Roy, C. K., & Cordy, J. R. *A Survey on Software Clone Detection Research*. *Queen's University at Kingston Technical Report*, 2007.
[69]Koschke, R. *Survey of Research on Software Clones*. *Dagstuhl Seminar Proceedings*, 2007.
[70]Zaidman, A., van Rompaey, B., Demeyer, S., & van Deursen, A. *Mining Software Repositories to Study Co-Evolution of Production & Test Code*. *Software Quality Journal*, 2008.

This analysis helps identify trends, measure developer effectiveness, and detect potential risks associated with specific commits or contributor patterns[71].

For example:

- **Developer Efficiency**: By analyzing commit frequency, size, and complexity, it is possible to detect scenarios where a developer consistently submits excessively large or overly complex changes. Such behaviors might indicate inadequate modularity or insufficient understanding of project guidelines[71].
- **Commit Risk Assessment**: Historical commit analysis can reveal certain commits frequently associated with subsequent defect reports. For instance, commits involving critical modules such as authentication or encryption libraries can be flagged automatically as high-risk and subjected to more rigorous peer reviews and testing procedures[72].
- **Impact of Commit Timing**: Commits made very close to release deadlines can be flagged for additional scrutiny, as hurried changes often introduce regressions or incomplete features[73].

In the SOSIS framework, commit and contribution analysis utilizes repository mining techniques to extract meaningful insights from version control systems (e.g., Git). By integrating commit history with variant-specific metadata and issue-tracking systems such as Jira, SOSIS provides comprehensive traceability of software changes to corresponding requirements, variants, and feature implementations[74]. This allows project teams to evaluate the impact of individual commits and understand how developer contributions affect software quality over time.

Moreover, advanced analysis techniques, including machine learning-based classification and anomaly detection, are applied to identify problematic commits that could introduce defects or vulnerabilities into the software product line. Historical commit data is analyzed to identify frequently modified components, assess developer workloads, and pinpoint risky code segments that require additional reviews or testing.

Through these practices, the SOSIS project ensures proactive and context-aware management of software contributions, significantly enhancing the overall reliability, security, and maintainability of the software variants within product lines.

4.5. Variant Quality Metrics

In software product lines (SPL), it is essential to define and apply variant-specific quality metrics to assess the internal and comparative quality of each product variant. These metrics enable organizations to systematically evaluate the quality impacts of variability and guide improvements[75].

Commonly Used Metrics:

- **Code Quality Metrics**:

[71]Mockus, A., & Herbsleb, J. D. *Expertise Browser: A Quantitative Approach to Identifying Expertise*. *ICSE*, 2002.
[72]Kim, S., Whitehead Jr., E. J., & Zhang, Y. *Classifying Software Changes: Clean or Buggy?*. *IEEE Transactions on Software Engineering*, 34(2), 2008.
[73]Rahman, F., & Devanbu, P. *How, and Why, Process Metrics Are Better*. *ICSE*, 2013.
[74]Zaidman, A., van Rompaey, B., Demeyer, S., & van Deursen, A. *Mining Software Repositories to Study Co-Evolution of Production & Test Code*. *Software Quality Journal*, 2008.
[75]Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

- ○ Defect Density: Measures the number of defects relative to code size in each variant[76].
  - ○ Cyclomatic Complexity: Higher complexity often correlates with higher defect proneness[77].
  - ○ Code Reuse Ratio: Indicates how much of the variant relies on shared vs. custom components[78].
- **Performance Metrics**:
  - Response Time / Latency: Critical for real-time or mission-critical systems[79].
  - Resource Usage: Tracks memory, CPU, and I/O consumption per variant[79].
- **Test Coverage Metrics**: Measures how thoroughly each variant has been tested. Includes code coverage percentages and the number of failed test cases[76].
- **Quality Drift Due to Variability**: Analyzes quality differences among variants derived from the same core assets[80].

Variability-induced defects may arise due to configuration errors, platform dependencies, or integration mismatches.

In the SOSIS project, variant quality metrics are enriched and contextualized through:

- Traceability via Git/Jira links to associate quality issues with specific variants[81],
- Log analysis to identify variant-specific performance degradations[82],
- AI-driven prediction models that evaluate metric trends over time[83].

These metrics support both reactive quality control and proactive optimization strategies across the software product line.

4.6. Machine Learning and Artificial Intelligence Applications

Software Product Lines (SPL) aim to manage variability across a family of products that share common core assets but differ in features, configurations, or deployment environments.

However, as the number of variants and their dependencies grow, ensuring consistent software quality becomes a significant challenge. Manual evaluation of these diverse variants is impractical due to scale, time, and cost constraints[84].

To overcome these limitations, Machine Learning (ML) and Artificial Intelligence (AI) methods are increasingly leveraged for predicting, assessing, and improving software quality in SPL contexts.

- **Influence Modelling**: Influence models learn how individual features or combinations thereof affect key quality metrics such as response time, latency, or resource utilization.

[76]Nagappan, N., Ball, T., & Zeller, A. *Mining Metrics to Predict Component Failures*. *ICSE*, 2006.
[77]McCabe, T. J. *A Complexity Measure*. *IEEE Transactions on Software Engineering*, 1976.
[78]Kästner, C., et al. *Variability-Aware Module Metrics for Software Product Lines*. *SPLC*, 2011.
[79]Siegmund, N., et al. *Performance-Influence Models for Highly Configurable Systems*. *ESEC/FSE*, 2012.
[80]Thüm, T., et al. *Measuring and Predicting Quality in Product Lines: A Feature-Based Perspective*. *Journal of Systems and Software*, 2014.
[81]Zaidman, A., van Rompaey, B., Demeyer, S., & van Deursen, A. *Mining Software Repositories*
[82]Karlsen, E., Luo, X., Zincir-Heywood, N. et al. *Benchmarking Large Language Models for Log Analysis, Security, and Interpretation*. *J Netw Syst Manage*, 32, 59 (2024).
[83]Guo, J., Siegmund, N., Apel, S., & Kästner, C. *Learning to Predict Performance Properties of Software Configurations*. *Empirical Software Engineering*, 20(3), 2015.
[84]Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

Rather than testing every possible configuration, a well-selected sample set can be used to train predictive models with high accuracy[85].

- **Cost-Effective Sampling Strategies**: Sampling techniques—either random or stratified—are used to select representative configurations for training ML models. These strategies reduce the testing effort while maintaining predictive reliability[86].
- **Adversarial Machine Learning**: Adversarial ML generates edge-case configurations to test the robustness of quality classifiers. These configurations intentionally target the weaknesses of predictive models and help in identifying blind spots in variant validation[87].
- **CI/CD Integration**: By integrating ML models into CI/CD pipelines, the impact of each code change on variant quality can be assessed in real time. This supports gated deployments, reduces late-stage regressions, and enhances test prioritization strategies[84].

These AI/ML-based approaches empower development teams to shift from reactive fault detection to proactive and adaptive quality assurance across large-scale software ecosystems.

4.7. Summary

The SOSIS project introduces a comprehensive framework for quality analysis in Software Product Line (SPL) engineering, addressing the complexities of feature variability and code diversity across software variants. It emphasizes systematic assessment through six core pillars:

- **Feature Quality Analysis**, evaluating completeness, stability, performance, documentation, security, and inter-feature interactions.
- **Code Quality Analysis**, targeting issues such as coding errors, security vulnerabilities, poor readability, and inefficient patterns through static analysis tools and historical context.
- **Code Duplication Detection**, enhanced with variant-aware context and advanced clone detection techniques to distinguish between reuse and redundancy.
- **Commit and Contribution Analysis**, leveraging repository mining and machine learning to assess developer behavior, commit risk, and the impact of changes on variant quality.
- **Variant Quality Metrics**, providing tailored indicators such as defect density, complexity, performance benchmarks, and test coverage to evaluate and compare individual product variants.
- **AI/ML Applications**, enabling predictive modeling, intelligent sampling, adversarial configuration testing, and CI/CD integration to automate and optimize quality assurance at scale.

Through the integration of static and dynamic analyses, version control metadata, and intelligent prediction systems, SOSIS enhances traceability, early defect detection, and proactive quality management. The result is a robust methodology that supports scalable, maintainable, and high-quality software product lines.

[85]Siegmund, N., et al. *Performance-Influence Models for Highly Configurable Systems*. *ESEC/FSE*, 2012.
[86]Guo, J., Siegmund, N., Apel, S., & Kästner, C. *Learning to Predict Performance Properties of Software Configurations*. *Empirical Software Engineering*, 20(3), 2015.
[87]Xie, X., et al. *Testing Machine Learning Programs with Adversarial Examples: A White-Box Approach*. *IEEE Transactions on Software Engineering*, 2022.

5. AI-Driven Automation for User Requirements

AI-driven automation is transforming the landscape of user requirements engineering by integrating machine learning (ML), natural language processing (NLP), and multi-agent systems into every phase of requirements management. AI can manage data from interviews, documents, emails, and even voice conversations—interpreting, synthesizing, and formalizing the results as high-quality requirements[88]. This facilitates more adaptive, error-resistant workflows in safety- and mission-critical systems, such as those within the SOSIS platform. Recent studies show that these technologies reduce manual effort, improve clarity, and enhance traceability, leading to fewer project failures and more agile development cycle[89, 90].

However, AI introduces its own challenges, particularly regarding validation and practical deployment: bias in ML models, the need for explainable decision-making, and ensuring compatibility with standards (like ISO 26262) remain open research problems[91]. Despite these, increasingly sophisticated NLP and autonomous agents have demonstrated substantial gains in requirements completeness and relevance—a recurring challenge in both traditional and agile lifecycles. As the volume and complexity of projects grow, these capabilities will be critical for delivering high-quality software at scale[92].

AI's benefits are especially apparent in collaborative, multi-project environments. Cross-team feedback, automated requirement consolidation, and real-time conflict detection support more informed project decisions—making AI a catalyst not just for automation, but also for enhanced organizational learning and process improvement[93]. Nonetheless, robust and continual validation is necessary to sustain quality over time, especially as requirements evolve.

AI-driven approaches are moving requirements engineering toward a model where data-driven insights fuel every decision, from user-story generation to final validation. These advances benefit not only project managers and analysts, but also stakeholders across the lifecycle, fulfilling the goals of efficiency, accuracy, and adaptability set out in SOSIS and similar initiatives.

5.1. Automated Requirement Gathering

Automated requirement gathering leverages AI—especially NLP and conversational agents—to capture requirements from a variety of sources, such as client interviews, emails, and historical

[88] Sami, M. A., Waseem, M., Zhang, Z., Rasheed, Z., Systä, K., & Abrahamsson, P. (2024). AI based Multiagent Approach for Requirements Elicitation and Analysis. Software Engineering. https://arxiv.org/abs/2409.00038

[89] Papapanos, K., & Pfeifer, J. (2023). A Literature Review on the Impact of Artificial Intelligence in Requirements Elicitation and Analysis. Stockholm University, Department of Computer and Systems Sciences. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1784322&dswid=8050

[90] Alenezi, M., et al. (2025). AI-Driven Innovations in Software Engineering: A Review of Current Practices and Future Directions. Applied Sciences, 15(3), 1344. https://doi.org/10.3390/app15031344

[91] Di Thommazo, A., Rovina, R., Riveiro, T., Olivato, G., Hernandes, E., Werneck, V., & Fabbri, S. (2014). Using Artificial Intelligence Techniques to Enhance Traceability Links. In Proceedings of the 16th International Conference on Enterprise Information Systems (ICEIS-2014), pp. 26–38. https://doi.org/10.5220/0004879600260038

[92] Guo, J., Cheng, J. & Cleland-Huang, J. (2017). Semantically Enhanced Software Traceability Using Deep Learning Techniques. 39th International Conference on Software Engineering (ICSE), 3–14.https://doi.org/10.1109/ICSE.2017.9

[93] Benitez, C. D., & Serrano, M. (2023). The Integration and Impact of Artificial Intelligence in Software Engineering. International Journal of Advanced Research in Science Communication and Technology, 3, 279–293

documents. Recent academic reviews note that AI-powered agents can generate structured requirements from raw stakeholder input, reducing bias, ambiguity, and omissions[94]. For example, multi-agent systems built on large language models (LLMs) systematically elicit, refine, and validate requirements through iterative conversations, far surpassing the throughput and accuracy of manual elicitation.

These agents extract patterns and domain terminology, ensuring uniformity in requirements documentation and automatic translation into standard formats suitable for subsequent analysis or development phases. Systematic evaluations demonstrate that integrating AI into the early stages of requirements engineering significantly reduces the time required for initial gathering while increasing completeness and structure quality.

However, automated gathering is not a panacea. It must be tailored to each project's context and regularly calibrated with human oversight to prevent misunderstanding nuanced stakeholder needs or propagating systemic biases inherent in training datasets. In regulated or high-stakes environments, these systems supplement rather than replace traditional interviews—enabling auditability and traceability by logging every interaction.

Advancements also include automatic summarization and duplicate detection across stakeholder inputs, ensuring that critical requirements are not missed and redundant information does not confuse the development team. As conversational AI technology continues to mature, automated requirement gathering is poised to become a mainstay in agile, distributed, and high-complexity projects[95].

5.2. Requirement Analysis and Prioritization

In the requirement analysis and prioritization phase, AI models process large, diverse stakeholder inputs, historical project data, and external documentation to discover high-priority needs, predict dependencies, and model the impact of each requirement. Studies have shown that ML-based prioritization approaches outperform traditional techniques—improving adaptability and responsiveness by rapidly recalculating priorities as project goals shift or dependencies are discovered[96].

ML and fuzzy-logic algorithms map requirements to project constraints, automatically flagging conflicts or missing information that needs clarification. In agile and continuous delivery scenarios, this allows dynamic backlog management and real-time reprioritization based on changing user feedback or evolving technical risks.

Systematic literature reviews document a range of AI-driven techniques: multi-criteria decision analysis, historical project-outcome modelling, and clustering to group related requirements for

[94] Kadri, S., Aouag, S., & Hedjazi, D. (2021). MS-QuAAF: A generic evaluation framework for monitoring software architecture quality. Information and Software Technology, 140, 106713. https://doi.org/10.1016/j.infsof.2021.106713

[95] Prasetyo, M. L., Peranginangin, R.A., Martinovic, N., Icshan, M. & Wicaknoso, H. (2025). Artificial intelligence in open innovation project management: A systematic literature review on technologies, applications, and integration requirements. Journal of Open Innovation: Technology, Market, and Complexity, 11(1), 100445. https://doi.org/10.1016/j.joitmc.2024.100445

[96] Habiba, U., Haug, M., Bogner, J., & Wagner, S. (2024). How mature is requirements engineering for AI-based systems? A systematic mapping study on practices, challenges, and future research directions. Requirements Eng, 29, 567-600. https://doi.org/10.1007/s00766-024-00432-3

batch analysis. These methods increase transparency and reduce the likelihood of costly mistakes—such as over-valuing low-impact features or neglecting critical dependencies—by grounding decisions in project data rather than intuition alone.

Potential risks include algorithmic bias and lack of interpretability, underscoring the need for domain experts to remain involved in the calibration and oversight of AI-driven prioritization systems. Nonetheless, empirical results suggest that these approaches dramatically reduce waste and rework, leading to more predictable project delivery and greater stakeholder satisfaction[97].

5.3. Requirement Traceability

AI-enhanced requirement traceability automates the generation and continuous updating of traceability matrices, linking requirements with design, implementation, and testing artifacts. Academic research demonstrates that combining NLP, neural networks, and fuzzy logic considerably improves the accuracy and efficiency of establishing these links, which are critical for audit, compliance, and change management.

Several studies confirm that AI-based traceability systems can analyze evolving project documentation at scale, automatically generating suggestions and flagging broken or missing links for review by human analysts. As requirements change, AI algorithms can rapidly perform impact analysis, predicting downstream effects and informing stakeholders of potential risks or required mitigation steps[98].

A key advantage cited in the literature is the reduction in time and effort needed for comprehensive traceability, particularly in large or regulated projects where manual trace matrix management is prohibitive. Further benefits include enhanced documentation quality and more robust foundations for safety certification processes.

However, a systematic literature review suggests that achieving full automation remains a challenge: domain adaptation, explainability, and integration with legacy systems are ongoing concerns. Overall, evidence supports that AI-driven traceability dramatically improves the reliability and auditability of requirements management processes[99].

5.4. Multiple Project Requirements Validation and Consistency

Validation and consistency checking across multiple projects harnesses advanced AI—especially NLP, ML classifiers, and cross-project analytics—to automatically detect ambiguities, inconsistencies, and non-conformities in requirements documentation. AI tools compare

[97] Ahmad, K., Abdelrazek, M., Arora, C., Bano, M., & Grundy, J. (2023). Requirements Engineering for Artificial Intelligence Systems: A Systematic Mapping Study. Information and Software Technology, 158, 107176. https://doi.org/10.1016/j.infsof.2023.107176

[98] Anwar, R., & Bahir, M. B. (2023). A Systematic Literature Review of AI-Based Software Requirements Prioritization Techniques. IEEE Access, 11, 143815-143860. 10.1109/ACCESS.2023.3343252

[99] Tasneem, N., Zulzalil, H. B., & Hassan, S. (2025). Enhancing Agile Software Development: A Systematic Literature Review of Requirement Prioritization and Reprioritization Techniques. IEEE Access, 13, 32993-33034. 10.1109/ACCESS.2025.3539357

contemporary and historical project specifications, highlighting patterns and discrepancies that manual review would likely overlook[100].

Studies emphasize the role of AI in real-time validation and remediation, reducing costly rework and improving requirements maturity before further development or resource allocation. By enforcing internal standards and normalization, organizations can consolidate requirements for multi-project coordination, resource allocation, and cross-disciplinary analysis.

Recent research confirms significant increases in requirement quality and a corresponding reduction in project overruns or failures when AI-based validation systems are deployed. These systems support continuous quality improvement, automatically enforcing best practices, standard terminology, and domain conventions across distributed project teams[101].

Despite these gains, maintaining explainability and aligning AI-driven validation with complex regulatory constraints remains challenging. Academic evidence, however, supports the substantial ROI of implementing AI validation—often detecting errors and omissions invisible to traditional rules-based verification.

5.5. Summary

AI-driven user requirements management demonstrates transformative potential throughout the requirements lifecycle. Academic research highlights substantial efficiency gains in requirement elicitation, analysis, prioritization, traceability, and validation when leveraging modern AI methods. Tailoring and harmonizing these tools to project- and domain-specific constraints, while ensuring transparency and ongoing validation, are essential for realizing AI's full benefits. Projects like SOSIS exemplify how integrating AI-based automation at every stage can optimize workflow, reduce errors, and support large-scale, cross-team software development, especially in safety- and mission-critical environments.

6. Conclusion

This deliverable provides a consolidated overview of the technological and academic landscape surrounding the key components of the SOSIS project.

By examining the latest techniques in variant-focused quality analysis, test case optimization and prioritization, and software quality metrics, and addressing AI-enabled requirements engineering, the report outlines both the challenges and opportunities in developing adaptable and reliable SPL systems.

A key insight from this deliverable is the growing importance of artificial intelligence and machine learning in addressing scalability, automation, and decision support needs throughout the software lifecycle. The integration of intelligent prioritization strategies and generative models into SPL environments signifies a significant shift from traditional reactive approaches to proactive and context-aware quality assurance approaches.

[100] Cleland-Huang, J., Gotel, O. C. Z., Hayes, J. H., Mäder, P., & Zisman, A. (2014). Software traceability: trends and future directions. FOSE 2014: Future of Software Engineering Proceedings, 55 -69. https://doi.org/10.1145/2593882.2593891
[101] Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K. J., Ajagbe, M. A., Chioasca,E., & Batista-Navarro, R. T. (2021). Natural Language Processing for Requirements Engineering: A Systematic Mapping Study. ACM Computing Surveys (CSUR), 54(3). https://doi.org/10.1145/3444689

The SOSIS project aims not only to advance the latest technology in SPL engineering through this structured approach but also to provide practical, industry-ready solutions for quality-focused, diverse software ecosystems.